Delay-Insensitive Processes: A Formal Approach to the Design of Asynchronous Circuits

by
Hemangee K. Kapoor

A thesis submitted in the partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in
Computer Science

Centre for Concurrent Systems and VLSI
Faculty of Business, Computing and Information Management
London South Bank University

Abstract

With the proliferation of electronic devices in our day-to-day existence, the quality of the underlying circuits is becoming increasingly important. The devices are expected to run robustly under different operating conditions. Asynchronous circuits are promising as compared to synchronous approach, in achieving low power, low noise and high speed circuits which can be developed in a modular way. However, the absence of a global clock in these circuits comes at the cost of added concurrency. Therefore, it is important to have a better understanding of such highly concurrent systems in order to have confidence in the resultant devices.

A formalism known as delay-insensitive (DI) processes is used to reason about a special class of asynchronous circuits that make no assumptions about delays in any of its components or wires. The formalism is shown to be useful in verification of such circuits using existing verification tools. DI processes can be easily integrated into such tools and existing equivalence checking techniques applied to them, instead of starting from scratch. In particular, the application of the Concurrency Workbench (CWB) to the verification of DI processes is shown by modelling them in CCS and using MUST-testing for equivalence and refinement checking.

DI processes interact with their environment to form closed systems. A new restriction operator is defined to obtain the effective behaviour of a process in a given environment, which eases specification and facilitates implementation. This operator is also shown to be more general than the alternation operator defined previously by Mallon.

Building on the work of Josephs and Udding, the algebraic semantics of DI processes is investigated and transformations are automated using the term rewriting system Maude. A canonical form is defined and is further used for equivalence and refinement checking based on syntactic comparison.

The formalism is useful not only in verification, but also in the decomposition of certain forms of processes that have been found difficult for logic synthesis tool, such as Petrify, to handle. The decompositions introduce Wires and Fork elements that preserve the delay-insensitive behaviour of asynchronous controllers. The proposed heuristics are applied on benchmark examples to help the tool Petrify to synthesise area-efficient circuits rapidly.

Besides using the CWB, Maude and Petrify, the experiments reported here have involved tools developed in-house, namely, Furey's translation tool di2pn, verification tool diana and the authors translation tool di2ccs.

The thesis demonstrates that (i) DI processes can be verified by adopting existing verification tools and techniques; (ii) consideration of the environment of a process leads to simpler specifications; and (iii) decomposing specifications leads to area efficient implementations.

Acknowledgements

ॐ भूर्भुवः स्वः तत्सवितुर्वरेण्यं
भर्गो देवस्य धीमहि धियो योनः प्रचोदयात् ।
Om̃ bhūrbhuva: sva: tatsaviturvareṇyaṃ
bhargo devasya dhīmahi dhiyo yona: pracodayāt |

[*Throughout the experience of Life* **That** *essential nature illuminating existence is the adorable One. May all beings perceive through subtle and meditative intellect the brilliance of enlightened awareness.*]
– Gayatri Mantra

I would like to thank Prof. Mark B. Josephs, Director of Studies, who took great efforts in providing good guidance and research training and was enthusiastic about my work. I learnt from him the skills of writing technical papers, and also the patience and persistence required to analyse one's own work and address peer review comments.

I would also like to thank my second supervisor, Prof. Jonathan P. Bowen, who apart from giving information about technical talks and conferences, never failed to keep me abreast of places of interest, like museums and parks, which are necessary in days of stress and pressure during a PhD. Prof. Nimal Nissanke was also encouraging and enthusiastic about my progress.

I have been fortunate to be able to attend many events including a summer school in Asynchronous Circuit Design, the International Symposium on Asynchronous Circuits and Systems, ACiD-WG workshops, PhD Forums at EPSRC PREP, the Design Automation and Test in Europe 2003 conference, exhibiting tools at the DATE University Booth, presenting a paper at the International Conference on Application of Concurrency to System Design and last but not the least the half yearly UK Asynchronous Forums, which have been the most encouraging factor. I would like to acknowledge the support of the European Commission under FP5 contract IST-1999-29119 (ACiD-WG) to attend all these events.

Special thanks are due to Prof. Steven Nowick from Columbia University for providing the burst-mode specifications for the benchmark examples, Prof. Jordi Cortadella from University Politecnica de Catalunya for providing the latest build of Petrify for synthesising the benchmarks, and Dr. Dennis Furey for the development of the tool di2pn which was supported in part by the UK EPSRC grant reference M51567 and the tool diana. The developers of JAVACC (Java Compiler Compiler) version 3.0 are acknowledged.

I am thankful to the university administration staff, especially Chung Lam, the resources office staff, and the technicians. These acknowledgements would be incomplete without thanking my fellow PhD students sharing offices with me,

Huibiao Zhu, Vinh Cong Phan and Kalpesh Kapoor for always encouraging each other to work hard.

Finally, I am indebted to my parents who have built my educational foundation, encouraged me throughout my studies and given me the choice and chance to pursue what I desired. My husband, Kalpesh, has always been at my side in all the ups and downs of my PhD life. Without his support the thesis would have been incomplete. My in-laws were encouraging towards my PhD. My sister Trupti and friend Nadia have always been very helpful in giving me confidence.

Hemangee K. Kapoor

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why Asynchronous Circuits?

The semiconductor industry uses computer-aided engineering tools for the verification of electronic system designs and for the synthesis of digital logic that implements those designs. Typically, designs are entered in a hardware description language (HDL) and simulated. The two most popular HDLs, VHDL [VHD00] and Verilog [Ver01], and the tools that support them, are geared towards implementing designs in which components operate in lock step, synchronised to a global clock.

As device sizes shrink and systems become more complex, it becomes harder for engineers to achieve timing closure [Ful03], avoid problems with noise and stay within power budgets. This opens up opportunities for globally asynchronous design, in which components synchronise by some form of handshaking. Furthermore, the components themselves can be locally asynchronous if desired, by controlling them with sequential circuits, provided that care is taken to avoid hazards and races. By this means, designers may be able to realise one or more of the following benefits [BJN99, BS94, Hau95, SF01]:

- Low power consumption and zero standby power – none wasted on a global

clock, each component operating at its optimal frequency.

- High performance – no waiting for an edge of a global clock.
- Low electro-magnetic radiation – no synchronisation to a global clock.
- Robustness towards variations in supply voltage, temperature and fabrication process parameters – no absolute timing constraints.
- Ease of composition and reuse – handshaking interfaces.
- No clock distribution and clock skew problems.

Interestingly, computer-aided engineering tools in the telecommunications industry already target globally asynchronous software designs. Semiconductor industries are also adopting tools that support formal verification (particularly by state-space exploration [Hol03]) and compilation (synthesising an implementation from a high-level description).

1.2 Why Process Algebra?

Signal transitions (changes in the logic level of wires) control the operation of asynchronous circuits. Variants of Petri nets, such as I-Nets [MFR85] and Signal Transition Graphs [Chu87], are popular formalisms for specifying dependencies (causality) between signal transitions. The tools Petrify [CKK+97a] and ATACS [Mye95] have been used successfully to (automatically) synthesise asynchronous circuits from Petri nets. The Burst-Mode approach, another graphical notation and associated tools [FNT+99, YD92], is the main alternative to the Petri net approach.

Asynchronous circuits are often packaged into modules that communicate according to a delay-insensitive signalling scheme, such as four-phase handshaking [Sei80, Mar87, Ber93]. Complex VLSI systems can then be constructed by hierarchical composition of such modules. Unfortunately, Petri nets and Burst-Mode

specifications are not so well suited to hierarchical composition, nor are they able to handle parameterised specifications.

Process algebras are the main alternative to these graphical notations as a means of specifying concurrent systems. Process algebras (in common with the hardware description languages VHDL and Verilog) are textual and permit an algorithmic style of specification. Delay-Insensitive (DI) Algebra [JU90a, JU93], in particular, was created in order to allow hierarchical specification and verification of modules that communicate by delay-insensitive signalling. The events or actions of a DI process can be interpreted as signal transitions and algebraic laws capture the possibilities of reordering and interference as those transitions are propagated along wires [Udd86].

The lockstep operation of synchronous circuits simplifies their analysis. Asynchronous circuits, on the other hand, are nondeterministic in the timing of signal transitions. This can lead to a large number of possible execution paths, each corresponding to a different order of synchronisation of components depending on their relative delays. Therefore, simulation of asynchronous circuits cannot guarantee functional correctness, as it would cover only a fraction of all possible executions. The solution is to *verify* that the design is correct.

In the past, language-based proof techniques have been developed by Ebergen [Ebe91] and Josephs and Udding [JU90a] underpinned by a trace-theoretic model of *delay-insensitive* processes. Dill [Dil89] also worked within trace theory, but his emphasis was on automated verification. The general-purpose process algebra CCS [CPS93, Mol91] has itself been applied to the verification of asynchronous systems [Liu95, Ste94, TB98]. Structured verification, design and test of asynchronous circuits based on process spaces was proposed in [Neg98] and implemented in the FIREMAPS tool.

1.3 Scope of this Thesis

This thesis is concerned solely with the concept of a delay-insensitive process. It is used here to model, verify and synthesise asynchronous circuits. Such circuits control on-chip modules that interact through DI interfaces (such as handshaking ports [Ber93]). They are becoming attractive to system designers for several reasons:

1. *Necessity*. With multiple clock domains or clockless logic, there need not be a common clock on which modules can synchronise.

2. *Convenience*. Re-use of modules in different designs and in different implementation technologies is facilitated by the removal of timing constraints.

3. *Robustness*. In deep sub-micron CMOS technology wire delays dominate over gate delays.

A large number of verification tools exist supporting different kinds of verification, such as theorem proving, model-checking, bisimulation and testing-equivalences. The thesis develops and implements various methods for the verification of DI control circuits by applying available verification tools and techniques. This work involves development of methods to model the required behaviours in the formal framework supported by the tool. It thus shows their applicability in verifying such circuits, saving us from building complete verifiers from scratch.

The applicability of the general-purpose Concurrency Workbench in the verification of DI circuits is shown. A specialist tool, diana [Fur02], recently became available and was found to be particularly useful in the verification of circuits in conjunction with their environments. Yet another approach to automated verification transforms processes into a canonical form for equivalence and refinement checking. This has been implemented in the term rewriting system Maude [MOM00, Mes96].

Apart from verification, the thesis addresses asynchronous logic synthesis. The CAD tools di2pn [JF02] and Petrify [CKK+97a] are used to obtain net-lists from DI processes. The contribution here is a method of decomposing processes to obtain area-efficient circuits.

In the remainder of this chapter, we consider the applicability of formal methods to the verification of digital circuits in general. Classification of the various types of asynchronous circuits and overview of some synthesis techniques is given next. CSP-based formalisms that specifically support delay-insensitive processes, and their associated tools, are described in detail. The chapter ends with a description of the thesis structure.

1.4 Formal Methods

Given a design, one needs to validate it. This is traditionally accomplished by functional testing based upon simulation. However simulation may leave many errors undetected.

The likelihood of design error increases with the size and complexity of circuits, and exhaustive testing becomes infeasible. Moreover, as production is expensive and time-consuming, it is essential to detect errors as early as possible during the design process. Thus, there is a demand for *formal verification* [KG99] based upon a proof that an implementation satisfies a specification (i.e. a more abstract design or a set of the properties). All cases must be considered, but this is often feasible in practice [WC96].

Other methods like *synthesis* and *correctness-preserving transformations* can also be applied [Eve87]. These support first-time right design, eliminating costly design iterations.

1.4.1 Formalisms

Formal verification requires that both the specification and the implementation are expressed using an appropriate formalism. Sometimes different formalisms are used for specification and implementation, but this hampers stepwise refinement.

Some formalisms such as Z [Spi88], VDM [Jon86], and Larch [GH93] focus on specifying the behaviour of a sequential system. Such specifications are defined in terms of set theory and predicate calculus. Other formalisms such as CSP [Hoa85], CCS [Mil89], Petri nets [Pet62, Mur89], Statecharts [Har87], Temporal Logic [MP91, Lam84] and I/O automata [LT87, LT89], focus on specifying the behaviour of concurrent systems in terms of events.

1.4.2 Automated verification

The two main approaches to automated verification are model checking and theorem proving.

Model checking is a technique that relies on building a model of a system and checking that desired properties (often expressed in temporal logic) hold in that model. These checks are performed as an exhaustive state space search which guarantees to terminate when the model is a finite automaton. Research into model checking is constantly improving the algorithms and data structures so that ever larger state spaces can be searched. Model checking has been used primarily in hardware and protocol verification.

Actually it is possible to model both a specification and an implementation as finite automata, and to use model checking to prove that the latter conforms to [HK90, Kur94] (refines) the former.

Some popular model checking tools include temporal logic checkers like EMC [CE81, CES86, BCDM86], SMV [McM93], Spin [Hol91, Hol03], and CWB [CPS93]; and behaviour conformance checkers like Cospan/FormalCheck [HK90], FDR [Ros98] and CWB [CPS93, CS96].

In **theorem proving** both the design and its desired properties are expressed as formulas in some mathematical logic and the properties must be shown to be implied by the design. This logic is presented as a formal system consisting of axioms and inference rules. Theorem provers range from entirely automated general-purpose programs, to interactive special-purpose programs.

In contrast to model-checking, theorem proving can deal directly with parameterised specifications and even with infinite state spaces. Some popular theorem provers include deduction systems like ACL2 [KMM00], Eves [CKM+88], LP [GG88], Nqthm [BM79], RRL [KZ95] and Maude [MOM00]; proof checkers like HOL [GM93] and Nuprl [CAB+86]; and combination of model checking and theorem proving tools like PVS [ORS92], VIS [BHSV+96] and STeP [BBC+96].

1.5 Asynchronous Circuits

Digital circuits are made up of combinational or sequential logic blocks. Sequential blocks are those that depend on the history of the signals and therefore need some sequencing. The method used to sequence these blocks distinguishes the design style of digital circuits into *synchronous* and *asynchronous*.

Synchronous circuits assume time divided into discrete intervals by a global clock. The data and control signals are stored and forwarded at fixed intervals determined by the clock. The circuits are easy to design because one does not need to consider about hazards and races.

Asynchronous circuits have no common and discrete time as there is no clock. Instead, these circuits use handshaking between components to synchronise for control and data transfer and sequencing of operations. This handshaking helps to break the system into hierarchical modules. Building of large complex systems is made easier as one does not need to consider the distribution of the global clock. This ease comes at the cost of difficult module design which needs to be free of glitches and hazards.

1.5.1 Classification of asynchronous circuits

At the gate-level, asynchronous circuits can be classified as being self-timed, speed-independent or delay-insensitive depending on the delay assumptions that are made [SF01].

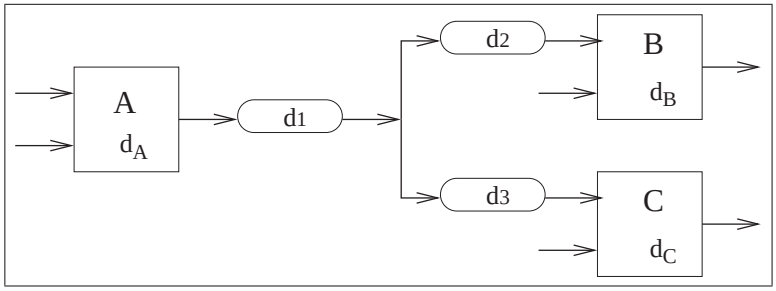


Figure 1.1: Fragment of circuit showing gate and wire delays [SF01].

A *speed-independent* (SI) circuit is a circuit that operates "correctly" assuming positive, bounded but unknown delays in gates and ideal zero-delay wires. In Figure 1.1 this means the gates A,B,C have arbitrary delays d_A, d_B, d_C, respectively while the wire delays are zero, i.e. $d_1 = d_2 = d_3 = 0$.

A circuit that operates "correctly" with positive, bounded but unknown delays in wires as well as gates is *delay-insensitive* (DI). In Figure 1.1 this refers to d_A, d_B, d_C, d_1, d_2 and d_3 to have arbitrary values. Such circuits are obviously extremely robust. Unfortunately the class of delay-insensitive circuits is rather small. Such circuits with the exception of some carefully identified wire forks where $d_2 = d_3$ are called *quasi-delay-insensitive* (QDI). Such wire forks, where signal transitions occur at the same time at all end-points, are called isochronic. These forks are generally used at gate-level implementations where the delays can be controlled by the designer. At the higher levels of abstraction, the composition of building blocks is delay-insensitive.

Circuits whose correct operation relies on more elaborate and/or engineering timing assumptions are called *self-timed*.

1.5.2 Synthesis techniques

In everyday usage, the term "synthesis" refers to the combination of various elements to make a complete system. But in the case of circuit designs, synthesis stands for stepwise refinement of circuit descriptions from high levels of abstractions to actual logic-gate level implementation. Generally, various optimisations can be applied at various levels of the synthesis process.

Many circuits are designed by hand. The rapid technological developments of the past few decades has lead to design and fabrication of larger and larger electronic systems. This is not possible without the development of the design methodologies and automation to get faster and more reliable circuits [MLD92]. Correct circuit design means the implementation satisfies the specifications in a formal mathematical sense.

Several techniques exist for synthesis of digital circuits in general, but those specific to asynchronous circuits are considered here.

Graph based methods

Signal Transition Graphs (STG) is the most popular formalism used in the design of SI circuits. An STG is a directed graph representing the events in a process. Another form of graphical representation used in the design of concurrent systems is Petri Net (PN) [DJ01]. Starting from such a graphical specification a reachability graph is constructed which is bisimilar [Ros98] to the original STG or PN. A state assignment is then performed by solving the Complete State Coding (CSC) problem [CKK+96, CKK+97b, YKSK96]. State assignment is coupled with Boolean logic minimisation, logic decomposition and technology mapping to obtain SI asynchronous circuits. A tool Petrify [CKK+97a] implements the whole procedure.

Another widely used formalism is Burst-mode specification, which is used in the design of Huffman type Mealy finite state machines. Burst-mode circuits re-

quire the fundamental mode assumption that input bursts should not change/occur when previous are not yet consumed. These are also extended to XBM (extended burst mode) to overcome some limitations. A complete synthesis path has been developed for XBM and implemented in the tool MINIMALIST [FNT+99].

Language based methods

High level approaches that include datapath and are mainly syntax directed
The first language based design approach was due to Alain Martin. The formal synthesis approach begins with a sequential description of the specification to be implemented in a language called Communicating Hardware Processes (CHP) a slight modification of Hoare's Communicating Sequential Processes (CSP) [Hoa85]. Semantic preserving transformations are applied to the sequential description to get a complex concurrent system which is a valid implementation of the given specification [Mar87]. This concurrent representation is then mapped to logic gates [Mar87] by means of production rules.

A synthesis technique for design of finely pipelined asynchronous systems was developed in [MLM99]. This work was based on the concept of projecting a CHP program onto different variables used in the text of the program. Conditions were given under which such a transformation can be applied, which relied on certain processes being locally slack elastic [MM98].

Another attempt was made by Philips in the CSP like Tangram [Ber93] language. This language has complete support for synthesis, simulation and test-vector generation and gives QDI circuits using Handshake Expansion. The Balsa [BE97] language was developed by University of Manchester and closely follows the Tangram approach. University of Utah [BS89] used Occam as the specification language, which uses two-phase protocol (non return to zero) and therefore has an expensive implementation.

As seen above, several CSP-like hardware description languages exist for asynchronous design. The advantages of these languages are their support of concur-

rency and synchronous message passing, as well as a limited and well-defined set of language constructs that makes syntax-directed compilation a relatively simple task.

The designers can still however choose to use one of the industry standard languages VHDL [VHD00] and Verilog [Ver01] for the design of asynchronous circuits. But VHDL lacks built-in primitives for synchronous message passing on channels and statement-level concurrency within a process. On the other hand the advantages are those of available CAD frameworks for simulation, pre-designed modules, mixed-mode simulation and tools for synthesis, layout and the back annotation of timing information. Separate VHDL packages exist that implement channel based communication which support manual top-down stepwise-refinement design flow where the same test bench can be used to simulate the design throughout the entire design process from high-level specification to low-level circuit implementation [SF01].

Low level approach mainly for control flow synthesis using STGs

Recently a new language Delay-Insensitive Sequential Processes (DISP) was developed to synthesise delay-insensitive circuits. It is a variant of CSP [Hoa85] and DI-Algebra [JU93]. It subsumes both CHP and Handshaking Expansion (HSE) by using named processes to represent handshakes, whether passive or active. It is similar to handshake expansions, but more uniform in its treatment of signals. It uses the names of signals to designate transitions. Thus, a given specification consists of event transitions for the module and the environment with which the module interacts. This pair of processes is given to the tools di2pn [JF02] and Petrify [CKK+97a] to synthesise asynchronous circuits. The synthesis techniques used by Martin use reshuffling of HSE signals to obtain the final circuit implementation. This arbitrary reshuffling might introduce deadlock as pointed out by Manohar in [Man01]. But DISP helps to detect deadlock at the time of specification itself [KJ02].

1.6 Delay Insensitive Processes

A circuit is connected to its environment by wires. If there are no timing assumptions about the delays in these wires for the correct circuit operation, then the circuit is said to be delay-insensitive. The design of delay-insensitive circuits is difficult because one needs to consider the scenario in which the signal has been transmitted at one end of the wire but has not yet reached the other end.

Delay-insensitive processes are attractive because they can be designed in a modular way; as no timing constraints have to be satisfied in connecting such circuits together. As no clock signal is used, sequencing is enforced entirely by communication mechanisms. They provide ideal separation of concerns in mathematical and physical aspects of circuit design making them more elegant, versatile and robust.

As DI processes make no assumptions on the speed of signal traversal along wires, the synchronisation must be implemented by some form of handshaking. Various basic theories [Hoa85, Mil89] exist to describe the behaviour of communicating components. In [Sne85], trace theory, the theory of finite sequences of communications was used as a mathematical basis for the design of hardware. In [Udd84], the structural properties of trace sets that describe delay-insensitive communicating processes were investigated. In [Ebe87], composition of basic building blocks to create networks was investigated. Receptive process theory (RPT) was introduced in [Jos92]. This gave a trace based theory to describe networks of communicating processes that cannot refuse inputs.

The specification and safety properties of DI circuits can be given in an algebra called DI-Algebra [JU93]. This algebra is based on Hoare's CSP [Hoa85] notation. The denotational semantics for DI-Algebra is compatible with the failure-divergences model of CSP. The algebra is also shown to be complete in [GJLU93] by giving a set of algebraic laws to transform a given process to its normal form.

DI-Algebra consists of series of operators and rewrite rules to describe pro-

cesses that communicate delay-insensitively [JU90a, JU93]. The algebra allows a more abstract view of processes and is easier to work with than trace sets. In [Luc94], it is shown that the algebraic rewrite rules are sound and consistent with the trace set approach.

In [Ver94] another trace based model was presented that was expressive enough to compute rather than invent parts of a design. In [Mal00b] a mathematically sound support is given along with computer-aided tool support for the designer of DI-circuits. Methods and tools support were given to transform DI-Algebra expressions into finite automata and also for decomposition of DI processes.

DI-Algebra has also been successfully applied in decomposition and verification [Jos02, JU90a, JU90a, JU90b, JMU+92, Luc94, JLUV94, LU96].

1.6.1 DI-Algebra

A process described in DI-Algebra is associated with an input alphabet $\mathcal{A}$ and an output alphabet $\mathcal{B}$, which are disjoint sets of signals denoting the input and output channels of the process, respectively. A process $a?;P$ describes a module that must "absorb" a transition on $a \in \mathcal{A}$ before it can behave like P. This is called input-prefixing. Similarly, there is output-prefixing, denoted by $c!;P$, where the module must "produce" a transition on $c \in \mathcal{B}$ before it can behave like P. As transitions are subjected to unbounded delays as they propagate along wires, the order in which a module absorbs or produces transitions cannot be observed.

$$a?;b?;P = b?;a?;P$$
$$c!;d!;P = d!;c!;P$$

An unexpected or unsafe input leads to undesirable behaviour denoted by the process $\perp$. In particular, pulses cannot be reliably transmitted:

$$a?;a?;P = a?;a?;\perp$$
$$c!;c!;P = \perp$$

A guarded choice describes a module that must first select an action to perform. For example, the guarded choice $[c! \rightarrow P \,\square\, a? \rightarrow Q]$ describes a module that must produce a transition on c or it can absorb one on a. The module then behaves like P or like Q, as appropriate. A `skip` statement that does not perform any action and transfers control to following statements can also be used as a guard.

A process that does not perform any action can be modelled as the *stop* process which is equivalent to a choice with no alternatives.

$$stop = [\,]$$

A process P evolves to $P/a?$ after a transition on a has been sent to the module, and to $P/c!$ after a transition from the module on c has been received.

The non-deterministic choice between two processes P and Q is denoted by $P \sqcap Q$.

Cyclic behaviour can be defined using recursion. For example, a wire that alternates between absorbing a transition on a and producing a transition on c can be specified as

$$W = a?; c!; W$$

Formally, the meaning of a recursion $P = f(P)$ is the least fixed point $\mu X.f(X)$ of f. Its successive approximations are $\bot$, $f(\bot)$, $f(f(\bot))$, etc.

Parallel composition is denoted by the infix binary operator $||$. In the parallel composition $P \,||\, Q$, the input (output) alphabet of P should be disjoint from that of Q. The input (output) alphabet of $P \,||\, Q$ then consists of those input (output) signals of P and of Q that are not output (input) signals of the other. This operator is helpful in hierarchical circuit design. The processes composed in parallel describe modules connected by a wire for every signal that the processes have in common. Transitions on these wires are not observable.

Verification of such DI-processes by modelling them in CCS is demonstrated in chapter 2. Verification of these processes in restrictive environments is shown in

chapter 3.

1.6.2 Delay-Insensitive Sequential Processes

Delay-Insensitive Sequential Processes (DISP) [JF02] is a variant of Communicating Sequential Processes (CSP) [Hoa85] and DI-Algebra [JU93]. It can be used to specify input-output bursts in a very simple manner and thus can be very useful in the design of asynchronous control circuits. The representations of handshake protocols used exhaustively for control specification are very straightforward in DISP as IO bursts.

DISP is similar to handshaking expansion but gives uniform treatment to the signals. It simply uses the name of the signals and treats up-going and down-going transitions as transitions alone. A DISP specification consists of a pair of programs, one describing the module under consideration and the other describes the environment in which it will operate. These specifications are given to CAD tools like di2pn [JF02] and Petrify [CKK+97a] to perform validation of the specifications and automatically synthesise asynchronous logic from them.

Verification of these processes by converting them into their canonical representations is given in chapter 4. Decomposition of such processes to help in synthesis is the topic of chapter 5. Translation of processes expressed in DI-Algebra to DISP is described in section 4.8.

1.6.3 DI tools

di2pn

The tool di2pn can be used to translate DI-Algebra [JF00] and DISP [JF02] specifications into Petri nets. One can also input specifications in the form of a pair of processes, representing the module under consideration and the environment it will be operating in. In this case di2pn generates a closed Petri net which can

be input into Petrify to obtain the netlist for an SI implementation. The algorithm to translate from DISP to Petri nets as described in [JF02] is given for reference in Appendix D.

The output generated by di2pn uses the same text-file format as Petrify, an enhanced version of the ASTG format devised for SIS [SSL+92]. It is most closely related in function to the digg tool [MU98], which translates terms in DI-Algebra into state-graphs rather than Petri nets. An alpha release of di2pn adopted the same input format as digg. Compatibility with digg was abandoned in the beta release [JF00] because it was considered desirable to adopt input/output-bursts in place of the individual signal transitions of DI-Algebra. The current release of di2pn accepts DISP rather than DI-Algebra. It also now performs peep-hole optimisations. Consequently, the Petri nets it produces are simpler, making them more readable and requiring less work to be performed by Petrify.

di2pn and Petrify have together been successfully applied to the design of a number of interesting asynchronous logic blocks that can be found in the literature. Some real-world design examples include (1) asynchronous controllers for a micropipeline stage, of the kind used in the ARM-compatible asynchronous processor core of the AMULET2e embedded system chip [FGR+99]; (2) a self-timed adder cell of the kind used in the arithmetic units of the Caltech asynchronous microprocessors [MBL+89] and in the dual-rail cell library of the Tangram silicon compiler [KvBB+92] (which is used for product design by Philips Semiconductors); (3) asynchronous controllers for an analog-to-digital (A/D) converter [CCP01] and for a token ring [Man01]; (4) simple SCSI controllers [NYD92]; (5) pipelined SCSI controllers [YD92]; (6) loadable counter [Jos02]; (7) high-performance SCSI controller [YD95].

diana

The analysis tool diana [Fur02] can be used for automated verification using Petri net representations of DI processes. Two DI processes can be checked for equiv-

alence and refinement against each other or in conjunction with a given environment. DI processes can be converted into Petri nets using the tool di2pn and then input into diana for verification. Apart from equivalence checking, diana can (i) check if the circuits ever deadlock, (ii) if there are possible glitches or hazards, (iii) what one specification can do which the other cannot, i.e. finding the extra features of one compared to the other, (iv) building different kind of traces including quiescent, divergent, failure, deadlock and traces showing extra features of one process against the other.

diana generates Petri net reachability graphs using a method similar to stubborn sets [Val88c]. In an ordinary reachability graph, each node represents a Petri net marking, and an edge connects one node to another if and only if the firing of a single transition will transform the origin to the terminus. The method used by diana is based on an abbreviated form of the ordinary graph wherein one marking can be adjacent to another whenever the collective firing of a set of independent transitions effects the transformation. Markings reached only by firing individual members of such sets need never be constructed, which reduces the necessary size of the graph and permits verification of slightly larger specifications than would otherwise be feasible.

The tool diana has successfully verified the equivalence and refinement of many decompositions against their original specifications, including large Decision-Wait elements, a DME controller, a loadable counter, and a Call element.

1.7 Thesis Structure

Chapter 2 starts with a brief introduction to Milner's process calculus CCS and Hennessey's MUST-testing preorder. The Concurrency Workbench is described along with some references of its application in other areas of verification. A method to model DI processes in CCS is presented and the translation procedure from DI-Algebra to CCS is described. Finally the chapter shows some case studies

related to this work. A translation tool, di2ccs, was built as part of this work to translate DI processes into CCS. The details of this tool are given in Appendix A.

Chapter 3 describes the verification of DI processes in restrictive environments by introducing the restriction operator to DI-Algebra. As this operator is similar to Mallon's alternation operator, the chapter demonstrates this with the help of the Nacking arbiter example. With a brief introduction to trace semantics and alternation, the chapter gives several advantages of the restriction operator in the motivation section. Trace theoretic semantics of this operator are given which include proofs for healthiness conditions and some important properties. The chapter also demonstrates how to represent alternations using the restriction operator by giving a definition for constructing a suitable environment. A way of eliminating alternation is also provided. The use of two existing tools, di2pn and diana, in applying these results in practice is demonstrated.

Equivalence checking of finite processes expressed in DISP is the topic of Chapter 4. Here a detailed list of algebraic elimination laws for DISP is given to reduce process expressions to a normal form. Some semantic functions and a further conversion procedure are given to convert the normal form to a canonical form. This canonical form can then be used to compare two processes expressed in DISP. These laws are given for only finite DISP processes, though they can be used to compare iterative processes by comparison of the loop bodies. A brief description of the Maude rewriting system is given next, along with some verification case studies. Translation from DI-Algebra to DISP is also discussed. The complete description of the implementation of laws in Maude can be found in Appendix B. Certain properties satisfied by the DISP operators are stated and proved in section 4.6.1.

Chapter 5 describes the decomposition heuristics that can be applied to DISP specifications to help Petrify solve CSC conflicts. It starts with showing how concurrent outputs and self-contained blocks in the specification can be a cause of CSC conflicts and then describes the decomposition heuristics in detail. Decomposition

of small example circuits using these heuristics is demonstrated. Experimental evidence of their applicability is shown in the results obtained by decomposing a set of benchmark examples. The details of these benchmark circuits are given in Appendix C. The tool di2pn was used to obtain Petri nets from DISP specifications before using Petrify for synthesis.

Chapter 2

Verification using CCS and the Concurrency Workbench

2.1 Introduction

To verify processes using existing formalisms one needs to model them correctly and identify equivalence definitions appropriate to their semantics. Modelling of DI processes in CCS and verification of them using an existing tool, the Edinburgh Concurrency Workbench, is discussed in this chapter.

DI circuits are represented by infinite/iterative processes expressed in DI-Algebra. Syntactic comparison of such processes based on their normal-forms is not feasible due to recursively defined process expressions. Another approach would be to build a Label Transition System (LTS) for the process expressions and then compare them under bisimulation equivalence. Since there are existing tools available that perform this task, the chapter shows the method of applying such tools to DI circuits. The Edinburgh Concurrency Workbench (CWB) is such a tool and accepts processes described in the process calculus CCS [Mil89]. To use this tool for verification we would have to model DI processes using CCS and also identify an appropriate verification technique.

CCS has found direct applications in the verification of complex protocols [Bre93, Bur92, Par88]. Since asynchronous circuits communicate via handshaking protocols, their correct interaction can be viewed as a form of protocol verification. However, the constraints of hardware implementation require some modifications to CCS to be useful [Ste94]. As an example, a wire process defined in CCS represents a one-place buffer (which is valid), while two wires in series form a two-place buffer (instead of a single wire as might be expected for DI wire processes). Thus, processes expressed in DI-Algebra do not have the same meaning in CCS and therefore one needs a special method to model them.

Once we have appropriately modelled DI processes in CCS, we need to find the correct verification technique supported by the CWB to be used for equivalence checking. The CWB has many types of equivalence checking techniques, viz., bisimulation, model checking and testing equivalence. According to [Hen88], the MUST-testing preorder is consistent with the failures/divergences model of CSP [Hoa85]. The characterisation of delay-insensitivity in that and in related semantic models has been explored in [HJH90, Jos92, Luc94, Mal00b, Ver94]. Therefore, we can choose MUST-testing as the verification technique.

Before we discuss the modelling of DI processes in CCS and some verification case studies; the chapter first describes the syntax and transition rules for CCS in section 2.3. Section 2.4 gives the features of the CWB and the various verification techniques supported by it. The definition of MUST-testing preorder and how it can be performed using bisimulation equivalence [CH93] of CCS is described in section 2.5. Modelling of DI processes in CCS and the implementation of the translation procedure in the tool di2ccs is described in section 2.6. Section 2.7 gives some verification case studies.

2.2 Related Work

The CWB is an automated tool that helps in manipulation and analysis of concurrent systems [CPS89, CPS93, MS]. The CWB has been applied in modelling and verification of asynchronous circuits and microprocessors [Liu95, Ste94] before, but the property of delay-insensitivity has not been considered.

In Ying Liu's thesis [Liu95] application of the CWB is shown in the verification of asynchronous microprocessor the AMULET1 developed at Manchester University. It was an attempt to apply formal techniques to full-scale, practical and industrial strength asynchronous designs. CCS was shown to be an appropriate and efficient notation for modelling complex designs. Property checking was performed on the specification using the CWB and some specification flaws were identified during the verification.

In the *Analyze* verification tool developed as part of Kenneth Stevens's PhD thesis [Ste94], the CCS labelled transition system was modified by adding extra transition rules and meta evaluation rules based on computational interference. This allowed direct representation of asynchronous modules as CCS processes. Fixed point calculations required by the Analyze tool were performed by the CWB.

The CWB was also applied in the verification of a reduced complexity two-phase micropipeline latch controller [TB98]. The design was proved by observational equivalence against its specification.

Verification of DI circuits has been attempted in the past by Willem Mallon. Where the operational semantics were given for DI-Algebra. To check process equivalence and refinement, the descriptions were converted into finite state automata and then compared.

2.3 Process Calculus CCS

Hoare's CSP [Hoa85] and Milner's CCS [Mil89] are *process algebras* or mathematical systems that help in modelling and analysing concurrent systems. As CCS is the main modelling language of the CWB, its syntax and semantics is described here.

Communication and Concurrency are notions applied to describe systems. These systems might have a single component or could be made up of a set of interacting components. Such components are called *agents* in CCS. (Note that throughout this chapter agent and process are used interchangeably to denote a process defined in CCS.) These agents communicate with their neighbouring agents using *actions*. The actions are represented by a set of symbols called *labels*. Actions occurring without interaction between agents are termed as *concurrent*. The set of actions (Act) is partitioned using "complementation" to obtain a set of co-names ($\overline{Act}$) such that if a is an action then $\bar{\bar{a}} = a$. All throughout this chapter 'a is used instead of $\bar{a}$ as is the convention adopted by the CWB.

An agent performs actions and changes its state. For example, a process P can perform an action a and change to process P'.

$$P \xrightarrow{\mathtt{a}} P'$$

Such *transition* relations given for all possible actions of a process help in describing its behaviour.

A communication takes place over an action and its compliment. When two processes synchronise over such a pair of actions they evolve into transformed versions using an atomic internal event τ. In Figure 2.1 the two processes P and Q can communicate with each other over action c. The remaining actions a and b need other agents for synchronisation.

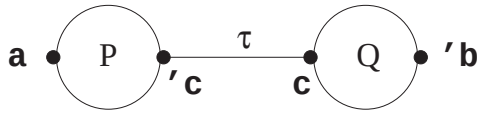


Figure 2.1: Process P and Q communicate over action c

2.3.1 Syntax and semantics of CCS

The syntax of CCS language is given below. It follows the notation used by the CWB.

P	$::=$	0	*Nil (deadlock)*
	$\mid$	X	*agent variable*
	$\mid$	$\mathtt{a}.P$	*action prefix*
	$\mid$	$P_1 + P_2$	*Summation (choice)*
	$\mid$	$P_1 \mid P_2$	*Composition (parallel)*
	$\mid$	$P\backslash\mathtt{a}$	*restriction of a single action*
	$\mid$	$P\backslash S$	*restriction of a set of actions*
	$\mid$	$P[R]$	*relabelling*
	$\mid$	(P)	*bracketed agent*
	$\mid$	$@$	*divergence*

The set of actions that a process can perform is known as its *sort*. The 0 (Nil) process performs no actions representing deadlock and thus has an empty sort. An agent having input action $\mathtt{a}$ and output action $\mathtt{b}$ has sort $\{\mathtt{a},'\mathtt{b}\}$. The semantics for CCS is given by a *labelled transition system* as follows:

$$(S, T, \{\xrightarrow{t} : t \in T\})$$

where S represents the set of states, T a set of transition labels, a transition relation $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$.

The transition semantics are defined by structural induction over process expression of the language above. Figure 2.2 defines these transition rules and each rule has a *conclusion* and zero or more *hypotheses*. In a rule associated with a

combinator, the conclusion will be a transition of an agent expression consisting of the combinator applied to one or more components, and the hypotheses will be the transitions of some of the components.

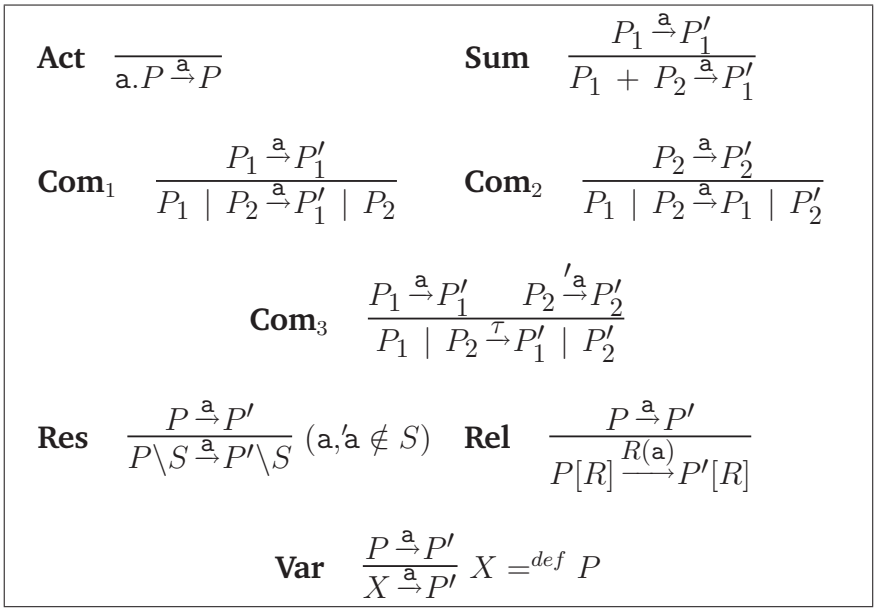


Figure 2.2: CCS transition rules

The **Act** rule syntactically using the '.' operator called prefixing is the basic building block for sequential operations. For example, the agent `a . 'b . 0` can do an action `a` followed by an action `'b` and nothing more.

The rule **Sum** states that if any one agent has an action, then the whole summation has that action. This is mainly used to specify guarded choice processes where any one of them will be invoked depending on what actions are performed. For example the process

$$\mathtt{P} = \mathtt{a} \,.\, '\mathtt{c} \,.\, \mathtt{P} + \mathtt{b} \,.\, '\mathtt{d} \,.\, \mathtt{P}$$

behaves like `'c . P` if action `a` is supplied or it behaves like `'d . P` if action `b` is supplied.

Concurrent processes can be defined using the **Com** rules with the | operator. Processes can communicate independently as in **Com1** and **Com2** rules or can synchronise over common actions as in **Com3** rule.

Restriction (**Res**) is defined by the set S where the process is restricted with the set of actions in S. This is mainly used to specify internal actions (denoted by τ) of processes composed by $|$, where they can synchronise on them and evolve using the τ action.

Applying the relabelling (**Rel**) function R to a process P helps to redefine the process which behaves like P but has its actions relabelled as specified by the function R. The relabelling function is defined as $[new/old]$ where all the occurrences of label old are replaced by the label new. This is typically applied to library elements or while reusing the definitions of a process already defined.

The rule **Var** defines references to processes so that one can defined recursive/iterative processes. For example a Wire process is defined as

$$\texttt{Wire} = \texttt{a} \,.\, \texttt{'b} \,.\, \texttt{Wire}$$

which says that after an input on `a` the process will produce an output on `b` and then repeat the behaviour.

2.4 Concurrency Workbench

Concurrency Workbench is an automated tool designed to analyse network of finite-state processes expressed in Milner's CCS. The key feature of the CWB is the breadth of verification techniques provided [CPS93], including equivalence checking, preorder checking and model checking. This versatility helps to support mixed verification strategies, facilitates comparison between many formal verification techniques and makes the system extensible. To obtain this flexibility, the architecture of the workbench is divided into three layers. The first layer interacts with the user and also builds labelled transition graphs. The second layer provides transformations on these graphs depending on the semantic model required by the user. The third layer includes the basic analysis algorithms for establishing

whether the process meets its specification.

The CWB has been successfully applied to the verification of the CSMA/CD protocol [Par88], the communication layer of the BHIVE multiprocessor [Goy91] and mutual exclusion algorithms [Wal89] and several other case studies [CT97, Bur97].

2.4.1 Verification techniques in the CWB

The CWB builds *transition graphs* (i.e. rooted labelled transition systems) to model processes. These graphs represent the observable behaviour of the processes. Such a graph consists of a set of *nodes*, a root node, and a set of edges labelled with an action. Each node additionally has an *information* field, the contents of which vary according to the computations being performed on the graph.

The CWB provides three main methods for proving that processes meet their specifications [CPS93]. These are briefly described below:

- *Equivalence checking*: Two processes are equivalent if they have the same behaviour. This is checked by performing node matching, i.e. if their corresponding nodes have the same information field, they have the same set of actions possible from each node and their root nodes are also matched. In other words two processes are equivalent if their transition graphs are bisimilar.

- *Preorder*: Here specifications are treated as minimal requirements to be met by implementations. The method relies on an ordering relation, or preorder, between processes: a process P is "more defined than" a process Q if P has the same behaviour as Q except for certain points in Q termed as don't care conditions. In other words,

  - If a node in the transition graph of Q is matched with that of P then the information field of Q should be subset of that of P.

- If a node in Q is matched to a node in P and P has a valid a-transition from that state, then each a-transition of Q must be matched by some a-transition of P.
- The start node of Q is matched to the start node of P.

The preorder can thus be considered as a *specification-implementation* relation in which the more-defined process is considered to be closer to the implementation than the less-defined one. This interpretation is based upon regarding divergent states as being *under-specified*, i.e. the @ can be seen as the totally unspecified state that allows any process as a correct implementation.

Section 2.5.2 gives the method which CWB employs to perform MUST-testing using bisimulation equivalence [CH93].

- *Model checking*: involves the use of propositional (modal) mu-calculus [Koz83]. Assertions are formulated in this logic such as "there are no deadlocks" or "every action of type a is followed by an action of type b". The system is then verified for satisfaction of these properties.

2.5 Verification Using the Testing Preorder

2.5.1 MUST-testing

One method for comparing transition systems is based on the observation of interactions between the system and an *experimenter* [Hen88]. The system and the experimenter are modelled as parallel processes. A particular run is said to be successful if the test reaches a designated success state; and the process "guarantees the test" if every run is successful. The experimenter and the process interact by communicating or synchronising with each other.

For a transition system T and an experimenter E, an experiment x is an execution $T \mid\mid E$ which is infinite or ends in a deadlocked state. The experimenter synchronises with the system on all external actions except w which denotes a success action. An experiment is said to be *successful* if w is enabled in at least one state. We say that T MUST E if each experiment of $T \mid\mid E$ is successful.

Definition

After a process p has engaged in a trace s, it may be in one of several possible states, in each of which a set of actions is enabled. $\mathcal{A}(p, s)$ denotes the set of such so-called Acceptance sets. We can write $\mathcal{A} \subset\subset \mathcal{B}$ if for every Acceptance set $X \in \mathcal{A}$ there exists some Acceptance set $Y \in \mathcal{B}$ such that $Y \subseteq X$. After s, if process p has infinite internal computation, it is said to be *divergent*; otherwise it is said to be *convergent*, denoted by $p \downarrow s$. The MUST testing preorder is then defined [Hen88] as follows:

$\texttt{mustpre}(p, q)$ if, for every sequence s of actions, $p \downarrow s$ implies

i) $q \downarrow s$, and

ii) $\mathcal{A}(q, s) \subset\subset \mathcal{A}(p, s)$

Note that, $\texttt{musteq}(p, q)$ iff $\texttt{mustpre}(p, q)$ and $\texttt{mustpre}(q, p)$.

Example:

Consider the two finite processes described as follows:

$P1 = a \,.\, (b + c)$ and $P2 = a \,.\, b + a \,.\, c$

The acceptance trees for $P1$ and $P2$ are shown in Figure 2.3, where each node is labelled with the set of its possible actions. Note that for every action a, every node in the tree has at most one successor branch labelled by a.

Clearly, the acceptance sets of $P2$ are subsets of those of $P1$ for each node in $P1$, and there are no divergent nodes. Therefore, as per the definition of MUST-testing, $\texttt{mustpre}(P2, P1) = true$, and $\texttt{mustpre}(P1, P2) = false$.

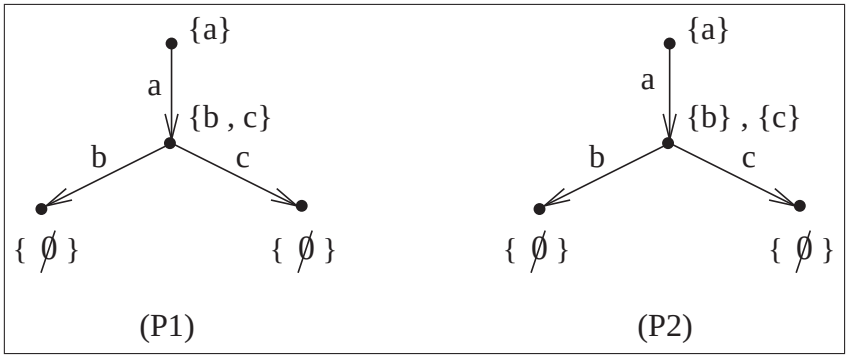


Figure 2.3: Acceptance trees for $P1$ and $P2$

To understand this, consider an experimenter process e that satisfies $P1$. This experimenter can perform the success action w initially, or after executing some set of its own internal actions can perform an action a and evolve to e'. Whatever e' is, it must eventually (after performing some internal actions) be capable of performing a w; or either b or c to become e'', which must eventually perform a w. Clearly such an experimenter after evolving to e' cannot satisfy $P2$, as $P2$ will be in a state where it can accept either a b or a c but not both. Therefore $\texttt{mustpre}(P1, P2) = false$. But another experimenter that satisfies $P2$ can satisfy $P1$ and therefore $\texttt{mustpre}(P2, P1) = true$. ♦

2.5.2 MUST testing as a bisimulation equivalence

The CWB performs analysis using a labelled-transition system representation of given processes by using bisimulation equivalence. To use these LTSs and bisimulation equivalence to perform MUST-testing, the CWB interprets the LTSs as instances of generalised bisimulation and pre-bisimulation preorders [CH93]. Transformations are done on the transition system in such a way that the testing relations on the original systems correspond to the bisimulation relations on the altered system.

Bisimulation equivalence is a behavioural equivalence on states that is defined in terms of relations called bisimulations. In other words, it can be seen as a

matching between states that has the property that if two states are matched then each a-derivative [1] of a state must be matched by some a-derivative of the other. To generalise this equivalence in order to use it for testing equivalences and preorders, Π-bisimulation and $\langle \Pi, \Psi \rangle$-prebisimulation, respectively, are defined [CH93]. Where Π is a binary relation between states and Ψ is a binary relation between states and actions.

Π relates states based on the type of information they contain (in the case of testing equivalences this defines the relation between acceptance sets of the states). To relate states based on preorders (refinement relationship) the matching conditions are relaxed to allow the possibility of matching an under-defined state to a more defined one. In other words, if a state p of one transition system is divergent on action a, then a prebisimulation relating the state p to state the q, of another transition system, need not match every a-transition of q with some a-transition of p. The binary relation Ψ (relating states to actions) is used to relax this matching requirement.

Graphs for MUST testing in the CWB

In the context of the CWB a labelled transition system is converted into *Acceptance Graphs*, or *Agraphs* for short, for performing testing equivalences [CH93]. The testing equivalence of our interest, viz. the MUST-testing, uses a variant of the Agraph called a *strong* Agraph (or SAgraph). An LTS $\langle S, Act, \rightarrow \rangle$, where S is a set of process states, Act is a set of actions including the internal τ action, and $\rightarrow \subseteq S \times Act \times S$ is the transition relation, is a SAgraph if $\rightarrow$ is deterministic and the following hold for each $t \in S$.

1. t is labelled with two pieces of information, $t.acc$ which is a set of actions possible from the current state and $t.closed$ a boolean stating whether the state is convergent.

[1] An a-derivative of a node p is the node connected to p by a transition labelled a, i.e. if $p \xrightarrow{a} p'$ then p' is the a-derivative of p.

2. $t.acc$ is finite, and each set $X \in t.acc$ is finite.

3. The set $t.acc$ is the minimised version of acceptance sets, i.e. elements of the set have no proper subset that is also in the set.
   Formally, $min\ A = \{X \in A | \neg \exists X' \in A, X' \subset X\}$

4. $t.closed = true \iff t.acc \neq \emptyset$

5. $t.closed = false$ implies there are no outgoing transitions from this node (denoting divergence).

To build Agraphs from an LTS, automata-theoretic notion of ϵ-closure is used on the set of states of the LTS. All states having τ transitions and more than one transition on a certain signal are combined to form a single new state in the transformed graph. The boolean associated with this new state is $true$ if all the states forming the new state are convergent. Combination of all possible actions (excluding the τ action) of the combining states form the acceptance sets of the new state. This set is empty if the new state is divergent. The transitions from the new states are inherited from the original LTS provided they are not τ and the new state is not divergent.

As an example, consider the LTS and its corresponding SAgraph shown in Figure 2.4 where each state (represented by a node of the graph) in the SAgraph is marked with its acceptance set. Convergent states are denoted with nodes having filled circles while divergent ones are left open.

MUST-testing preorder, $\texttt{mustpre}(p, q)$, is then defined in the CWB using the $\langle \Pi, \emptyset \rangle$-prebisimulation relation R defined as follows:

$$R = \{ \langle t, u \rangle \mid \forall s \in (Act - \{\tau\})^* \,.\, t \downarrow s \Rightarrow (u \downarrow s \wedge (s \in L(u) \Rightarrow s \in L(t) \wedge D(u,s).acc \subset\subset D(t,s).acc))\}$$

where,

$L(p) = \{s \in (Act - \{\tau\})^* \mid \exists p' \,.\, p \stackrel{s}{\Rightarrow} p'\}$ is the language of state p, and,

$D(p, s) = \{p' \mid p \stackrel{s}{\Rightarrow} p'\}$ is the destination node of p reached after executing the sequence of actions in s.

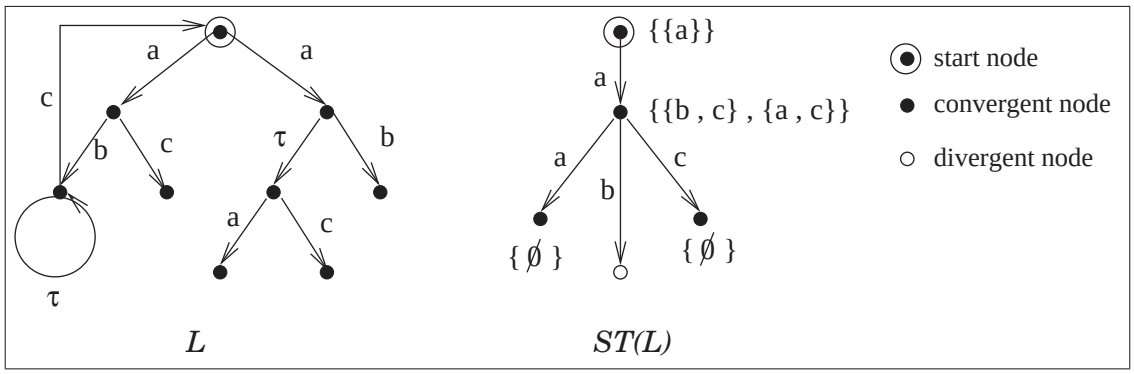


Figure 2.4: Transition system $\mathcal{L}$ and its corresponding SAgraph $\mathcal{ST}(\mathcal{L})$

Clearly, the $\langle \Pi, \emptyset \rangle$-prebisimulation relation defined above is consistent with the definition of MUST-testing preorder.

2.6 Modelling Delay-Insensitivity in CCS

After the background information about CCS and MUST-testing the formalisation of DI processes in CCS and its the verification can be easily described.

David Dill in his thesis [Dil89] has given a definition of delay-insensitivity and a simple procedure to test it. A process is delay-insensitive if adding delays to its inputs and outputs does not change its behaviour. He defined an operator DI on trace structures, which finds the least delay-insensitive specification that the original specification conforms to. A delay is equivalent to a non-inverting buffer (identity gate). The operator DI attaches such non-inverting buffers on all inputs and outputs of a trace structure, then hides the original wires and renames the new inputs and outputs to the original names. The overall effect of DI is to remove some ordering relations between signals. Let $\mathcal{T}$ denote the trace of a process. If $\mathcal{T}$ outputs ab but not ba, $\mathcal{T}' = DI(\mathcal{T})$ outputs both ab and ba, because signals may emerge from delays in different order than they entered. This is due to the Foam Rubber Wrapper [MFR85, Udd86] postulate. Hence, for all trace structures, $\mathcal{T}$ *conforms* to $DI(\mathcal{T})$.

Composing two delays and hiding the connections between them yields a trace structure that is conformation-equivalent to the original. Hence, DI is idempotent: for every $\mathcal{T}$, $DI(DI(\mathcal{T})) = DI(\mathcal{T})$. Adding more delays to the inputs and outputs of a delay-insensitive circuit has no effect on its operation.

This idea of making a process delay-insensitive by surrounding it with delays and hiding the internal signals, has been adopted in this chapter to model DI processes using CCS.

2.6.1 Defining DI processes in CCS

To model a process as delay-insensitive the delays to be connected to the inputs and outputs of the process need to be modelled. These are modelled as CCS wire processes. A wire can be defined in CCS as follows (in CCS a wire process represents a one-place buffer):

$$\mathtt{CW} = \mathtt{a} \,.\, {'\mathtt{b}} \,.\, \mathtt{CW}$$

In the case of DI processes two wires in series forms a single wire. If we connect two CCS wires, as defined above, in series we get a two place buffer instead of a one-place buffer. Figure 2.5 shows the transition system and the SAgraph formed by the serial connection (SC) of the two wires CW1 and CW2 given below:

$\mathtt{CW1} = \mathtt{a} \,.\, \mathtt{CW1'}$

$\mathtt{CW1'} = {'\mathtt{b}} \,.\, \mathtt{CW1}$

$\mathtt{CW2} = \mathtt{b} \,.\, \mathtt{CW2'}$

$\mathtt{CW2'} = {'\mathtt{c}} \,.\, \mathtt{CW2}$

$\mathtt{SC} = (\mathtt{CW1} \mid \mathtt{CW2}) \backslash \mathtt{b}$

Clearly SC depicts a two-place buffer instead of a one place buffer, i.e., a single wire.

Apart from forming a two-place buffer, by the serial connection of two wires, the CCS wire has no support for transmission interference [Sne85, Udd86] which

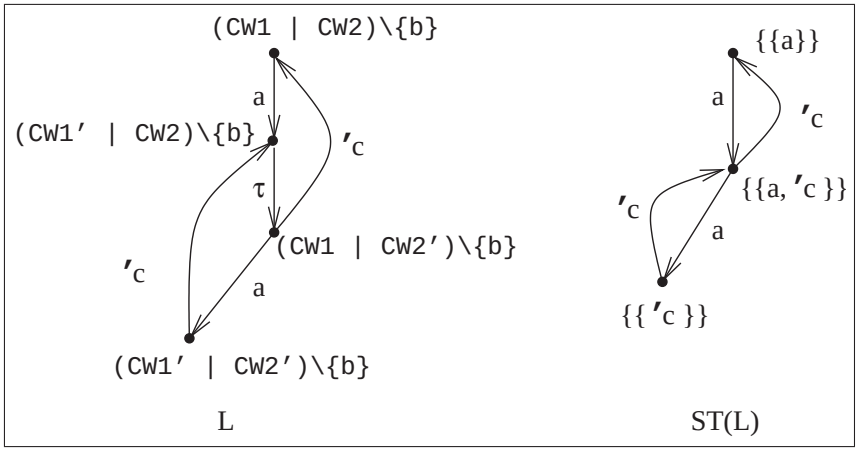


Figure 2.5: Two CCS wires in series form a two-place buffer

can be caused by two transitions propagating along the wire at the same time; as well as the unbounded delay requirement. As defined by Dill surrounding a process by such wires can handle the requirement of unbounded delay. For the transmission interference part, the CCS wire process needs to be modified such that two consecutive transitions on it lead to an erroneous state (@ in CCS). Such a wire (which is called a DI-wire here) can be modelled in CCS as follows:

$$\texttt{Di\_W} = \texttt{x} \,.\, (\texttt{'y} \,.\, \texttt{Di\_W} \;+\; \texttt{x} \,.\, @)$$

where actions x and $'y$ model the input and output of transitions at the corresponding terminals, and divergent process @ models interference, Figure 2.6. Thus, safe usage of a wire requires that input and output alternate. Under this definition of a DI wire process, the serial connection of two DI wires forms a single DI wire.

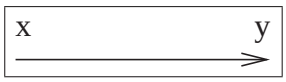


Figure 2.6: Wire with input terminal x and output terminal y

Unfortunately, this is not the case under the standard equivalence (bisimulation) of CCS. Therefore a semantic model is adopted in which this equivalence does hold, namely, the failures/divergences model of CSP [Hoa85] or, equiva-

lently, the MUST-testing preorder [Hen88], where the behaviour of a divergent process is considered to be undefined. (Note that the divergent process `@` of CCS is called `CHAOS` in CSP.) The following Figure 2.7 shows that two DI wires in series is equivalent to a single DI wire under MUST-testing equivalence.

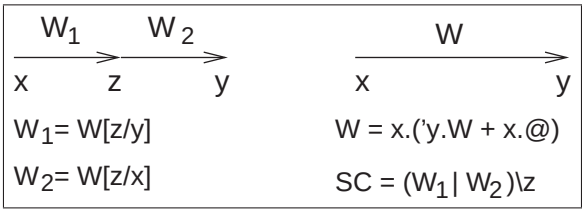


Figure 2.7: The serial connection of W_1 and W_2 is equivalent to W. Formally this can be checked as: $\texttt{musteq}(\texttt{SC}, \texttt{W}) = \texttt{true}$.

Using this definition of a DI-wire, one can then attach a DI-wire to each terminal of a process P, to construct a DI version Di_P of P, Figure 2.8. Formally,

$$\begin{aligned}\texttt{Di\_P} = \quad & (\ \texttt{P}[\texttt{ai}/\texttt{a}, \texttt{bi}/\texttt{b}, \texttt{co}/\texttt{c}, \texttt{do}/\texttt{d}]\ \mid \\ & \texttt{W}[\texttt{a}/\texttt{x}, \texttt{ai}/\texttt{y}]\ \mid \texttt{W}[\texttt{b}/\texttt{x}, \texttt{bi}/\texttt{y}]\ \mid \\ & \texttt{W}[\texttt{co}/\texttt{x}, \texttt{c}/\texttt{y}]\ \mid \texttt{W}[\texttt{do}/\texttt{x}, \texttt{d}/\texttt{y}]\ )\backslash\{\texttt{ai}, \texttt{bi}, \texttt{co}, \texttt{do}\}\end{aligned}$$

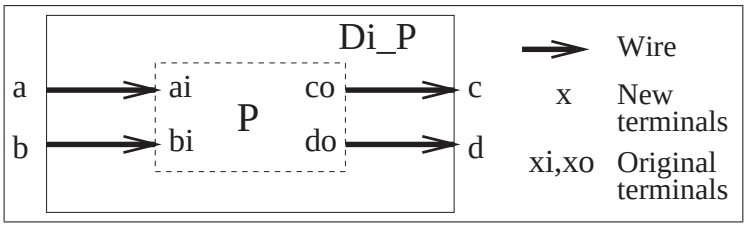


Figure 2.8: Delay-insensitised version Di_P of P. The original actions of P have been renamed and hidden.

In general, a process P is defined to be delay-insensitive if Di_P is equivalent to P. In particular, Di_P is delay-insensitive since Di_Di_P is equivalent to Di_P.

Example:

Consider the following two processes in CCS:

$\texttt{P} = \texttt{a} \,.\, \texttt{b} \,.\, \texttt{'c} \,.\, \texttt{P}$

$\texttt{Q} = \texttt{b} \,.\, \texttt{a} \,.\, \texttt{'c} \,.\, \texttt{Q}$

The two processes P and Q are not equivalent, but their delay-insensitised versions Di_P and Di_Q are equivalent under MUST-testing, as can be seen using the CWB:

$\texttt{musteq(P, Q)} = \texttt{false}$

$\texttt{musteq(Di\_P, Di\_Q)} = \texttt{true}$ ♦

Having thus defined a DI-wire process in CCS and constructed a DI version of any given CCS process, DI processes can now be safely converted into CCS preserving the semantics. Before we discuss the verification of DI processes, the procedure for making a process delay-insensitive in CCS is given in detail.

2.6.2 DI-Algebra syntax

One approach to the modelling of a DI module is to describe it as a process P in CCS and then verify that P is delay-insensitive, as defined above. An alternative is to use a language in which only DI processes can be described; DI-Algebra [JU93], a variant of CSP, is such a language. Moreover, the denotational semantics [Jos92, Luc94] of DI-Algebra is compatible with the failures/divergences model of CSP, so processes can still be characterised by MUST-testing. The algebra also has a complete set of algebraic laws [GJLU93].

The concrete syntax for DI-Algebra used in this work is as follows:

$$
\begin{array}{lcl}
\mathit{declaration} & ::= & \mathit{id} = \mathit{lowproc} \\
\mathit{proc} & ::= & \mathit{highproc} \mid \mathit{lowproc} \\
\mathit{highproc} & ::= & \mathit{id}\ \texttt{[}\ ||\ \mathit{id}\texttt{]}* \\
\mathit{lowproc} & ::= & \mathit{inputs}\ ,\ \mathit{outputs}\ \mathit{stmt} \\
\mathit{guard} & ::= & \mathit{sig}? \mid \mathit{sig}! \mid \texttt{skip} \\
\mathit{stmt} & ::= & \texttt{CHAOS} \mid \texttt{stop} \mid \mathit{id} \mid \mathit{stmt}\ /\ \mathit{sig}? \\
 & & \mid [\ \mathit{guard} \rightarrow \mathit{stmt}\ \texttt{[}\#\ \mathit{guard} \rightarrow \mathit{stmt}\ \texttt{]}*\ ] \\
 & & \mid \mathit{guard}\ ;\ \mathit{stmt} \mid \mathit{stmt}\ \texttt{ND}\ \mathit{stmt} \mid (\mathit{stmt})
\end{array}
$$

Here $inputs$ and $outputs$ are input($\mathcal{A}$) and output($\mathcal{B}$) alphabets, respectively.

In the parallel composition ($P \mid\mid Q$) of two processes P and Q, the input alphabet of P should be disjoint from that of Q; likewise the output alphabet of P should be disjoint from that of Q. Note that P `ND` Q denotes a non-deterministic choice between P and Q, whereas $[\ g_1 \rightarrow P_1 \ \#\ g_2 \rightarrow P_2\ ]$ denotes a guarded choice.

The advantages of using DI-Algebra rather than CCS for modelling DI modules are as follows:

1. There is no need to verify that a process is DI.

2. Point to point connection is directly modelled by the parallel composition operator "||" of DI-Algebra, rather than by the combination of "|" and "\" required by CCS.

3. The after-input operator of DI-Algebra is convenient for defining the behaviour of modules, especially their initial state.

4. There is a simple translation from processes in DI-Algebra into Petri nets [JF00] from which asynchronous logic can be synthesised using the tool Petrify [CKK+97a].

Of course, in order to verify designs modelled in DI-Algebra using the Concurrency Workbench, we first need to translate them into CCS. This translation procedure has been automated by the tool *di2ccs* developed as part of this work. The following subsections describe the translation procedure.

2.6.3 The translation tool di2ccs

The translation of processes from DI-Algebra into CCS has been automated in a tool *di2ccs* (implemented in Java).

The major tasks done by the tool are:

- Parsing of process declarations in DI-Algebra.

- Application of transformation rules given below to generate corresponding declarations in CCS.
- Declaration of DI versions of the above processes.

Syntactic transformation rules on guards and statements

- $x? \Rightarrow x$
- $x! \Rightarrow \ 'x$
- `skip` $\Rightarrow$ `tau`
- `CHAOS` $\Rightarrow @$
- `stop` $\Rightarrow 0$
- $g \ ; \ P \Rightarrow g \ . \ P$
- $[\ g_1 \rightarrow P_1 \ \# \ \ldots \ \# \ g_n \rightarrow P_n \ ] \Rightarrow g_1 \ . \ P_1 \ + \ \ldots \ + \ g_n \ . \ P_n$
- P `ND` $Q \Rightarrow$ `tau` $. \ P \ +$ `tau` $. \ Q$

This just leaves us to consider after-input operator and parallel composition. Before that a few examples are given below showing the application of the simple translation rules.

Example:
Consider the process P declared in DI-Algebra by
$P = \{a, b\}, \{c\} \ a? \ ; \ b? \ ; \ c! \ ; \ P$
Applying the translation for input and output event and for sequential composition, we get the following CCS transformed version of the process
$P = a \ . \ b \ . \ 'c \ . \ P$ ♦

Example:

Consider the following guarded choice process:

$P = \{a, b\}, \{c\}\ [\ a? \rightarrow c! \ \#\ b? \rightarrow \texttt{CHAOS}\ ]$

The transformed version in CCS is:

$P = a\ .\ 'c\ +\ b\ .\ @$ ♦

Translation of after-input $P/x?$

The after-input operator describes the behaviour of the process after an input signal has been sent to it. In other words this can be seen as the input being available on the input wire. To model this, the process itself can output a signal on its input terminal. Since the input terminal is used by the environment to make inputs to the process, we need to use some other terminal for modelling this. Hence a pushback wire is used that merges the inputs from the environment and the process. The after-input operator $P/x?$ is thus translated into *'px* . P. Instead of attaching a wire to x, a "pushback" wire is used, Figure 2.9, defined by

$\texttt{PBW} = \texttt{x}\ .\ \texttt{PBW}' + \texttt{px}\ .\ \texttt{PBW}'$

$\texttt{PBW}' = \ 'y\ .\ \texttt{PBW} + \texttt{x}\ .\ @ + \texttt{px}\ .\ @$

That is, actual input x and pushed back input px are merged.

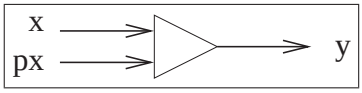


Figure 2.9: Pushback wire with input terminals x and px and output terminal y

Effectively what happens can be understood by the following Figure 2.10.

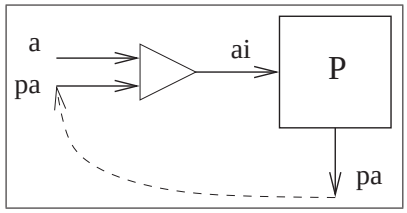


Figure 2.10: Process outputting on the pushback wire to model after-input operator $P/a?$, where ai is the renamed internal input terminal.

After defining the pushback wire, we also need to show that a single wire in series with a pushback wire is equivalent to a pushback wire. This result is proved using the CWB as shown in the Figure 2.11.

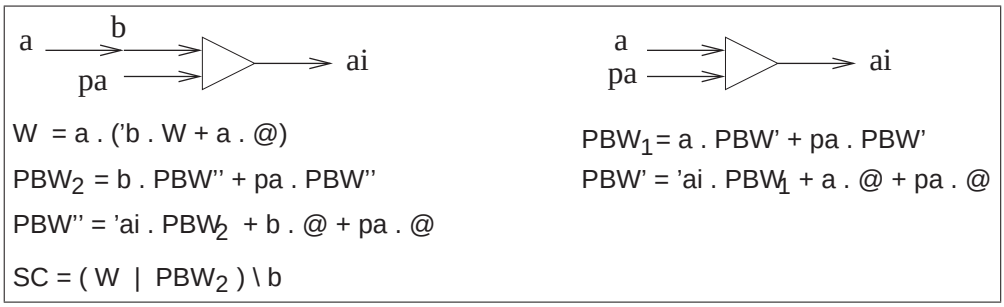


Figure 2.11: The serial connection of W and PBW_2 is equivalent to PBW_1. Formally, $\texttt{musteq}(\texttt{SC}, \texttt{PBW}_1) = \texttt{true}$.

Example:

Let the process P be declared in DI-Algebra as follows

$P = \{a, b\}, \{c\}\ a?\ ;\ b?\ ;\ c!\ ;\ P.$

The process $P/a?$ is syntactically transformed to the CCS process

$P = \ 'pa\ .\ a\ .\ b\ .\ 'c\ .\ P$

The output made on *pa* will be input by the pushback wire and fed back to the process P, thus achieving the effect of a pushed back input. ♦

Making the translated process delay-insensitive

Having translated the process described in DI-Algebra into its CCS representation, to preserve semantics we need to attach DI wires to the input and output terminals of the process. Thus the delay-insensitised version is the CCS representation composed with DI wires for each output terminal and pushback wires for each input terminal, with the necessary renaming applied.

Example:

The process P declared in DI-Algebra by

$P = \{a, b\}, \{c\}\ a?\ ;\ b?\ ;\ c!\ ;\ P$

is syntactically transformed to the CCS process

$P = a \,.\, b \,.\, 'c \,.\, P$

The delay-insensitised version is then obtained by attaching wires:

$$\begin{aligned} Di\_P = (\ & PBW[a/x, pa/px, ai/y] \mid \\ & PBW[b/x, pb/px, bi/y] \mid \\ & W[co/x, c/y] \mid \\ & P[ai/a, bi/b, co/c]\ )\backslash\{ai, pa, bi, pb, co\} \end{aligned}$$

Note that the signals in P are renamed to new signals and the surrounding wire processes use the original signal names. ♦

Translation of parallel composition

In the case of parallel composition, hiding of internal signals is done by collecting and computing them from the composed processes.

Let P and Q be two processes composed in parallel with input (output) alphabets $\mathcal{A}1$ ($\mathcal{B}1$) and $\mathcal{A}2$ ($\mathcal{B}2$) respectively. Let *internals* be defined as the set of shared signals between P and Q, as follows: $internals = (\mathcal{A}1 \cap \mathcal{B}2) \cup (\mathcal{A}2 \cap \mathcal{B}1)$. This gives us the translation

$P \mid\mid Q \Rightarrow (Di\_P \mid Di\_Q)\backslash internals$

where Di_P and Di_Q are delay-insensitised CCS translations of processes P and Q respectively.

Optimisation

The CWB builds and analyses a transition system representation of a process. We can observe from the above translations that, if the size (number of states in the corresponding transition system) of a process P is S_P, then the size of Di_P has an upper bound of $S_P \times 3^n$, where n is the total number of signals in the alphabet of the process P. Note that the size of both W and PBW is 3, so far as the CWB is concerned.

If we want to verify a possible implementation involving several components composed in parallel, the size of the implementation increases multiplicatively with the number of components. To reduce this increase in the size (and make large circuits verifiable), the di2ccs tool is optimised to generate fewer wires connecting shared signal terminals. In the case of parallel composition of two processes P and Q, instead of composing Di_P with Di_Q which would have a W and a PBW per internal signal, a single PBW is generated for each internal signal. Thus there is a reduction in the size by a factor of 3^m, where m is the number of internal signals.

Example:

Consider the parallel composition (M) of two processes P and Q described in DI-Algebra as follows:

$P = \{a, b\}, \{c\}\ a?\ ;\ b?\ ;\ c!\ ;\ P$

$Q = \{c\}, \{d, e\}\ c?\ ;\ d!\ ;\ e!\ ;\ Q$

$M = P\ ||\ Q$

Syntactic transformation of processes P and Q into CCS gives:

$P = a\ .\ b\ .\ 'c\ .\ P$

$Q = c\ .\ 'd\ .\ 'e\ .\ Q$

The process M is then transformed using the parallel composition of P and Q along with the attached wires. Note that only one pushback wire is generated for the internal signal c. The size of the delay-insensitive version of process M is 6642, while after optimisation it reduces to 2196. The optimised version is shown below:

$$
\begin{aligned}
Di\_M = (\ & PBW[a/x, pa/px, ai/y]\ | \\
& PBW[b/x, pb/px, bi/y]\ | \\
& PBW[co/x, pc/px, ci/y]\ | \\
& W[do/x, d/y]\ |\ W[eo/x, e/y]\ | \\
& P[ai/a, bi/b, co/c]\ |\ Q[ci/c, do/d, eo/e] \\
&)\backslash\{ai, pa, bi, pb, ci, pc, co, do, eo\}
\end{aligned}
$$

♦

2.7 Verification of DI-decompositions

A number of circuit decompositions were verified against their specifications. The descriptions are written in DI-Algebra and then translated into CCS using the tool di2ccs.

For example, considering handshake components [Ber93], a Connector was shown to be equivalent to the composition of an Or-element and a Mixer. A Non-receptive Mixer composed with a Join was also proved equivalent to a Latch.

Actually, to verify a DI-decomposition I against its specification S, we need only check that I refines S, i.e., $\texttt{mustpre}(\texttt{S}, \texttt{I})$. (Note that $\texttt{musteq}(\texttt{S}, \texttt{I})$ if and only if $\texttt{mustpre}(\texttt{S}, \texttt{I})$ and $\texttt{mustpre}(\texttt{I}, \texttt{S})$.) In this way a Toggle composed with a Merge element was shown to refine a 2-Phase-to-4-Phase Converter.

Table 2.1 shows verification results for these and some more circuits given in http://edis.win.tue.nl/edis.html. A detailed verification of a delay-insensitive Call element is shown in the next subsection.

Another existing verification and analysis tool diana [Fur02] performs verification of DI circuits based on their Petri net representations. Thus one can convert the given DI specification into a Petri net using the tool di2pn [JF00] and then use diana for verifying the two Petri nets. The circuits given in Table 2.1 were also verified using diana. The last column shows the time taken by diana to perform verification. As can be observed, diana takes more time even for smaller size circuits, but it is found to be scalable than the CWB; i.e. the 2-Call element was compared with its specification in 80 seconds without any hierarchical approach (whereas the CWB took more than 30 minutes to complete the verification).

Specification (S)	Size of S	Implementation components (I)	Size of I	Time (s)	Time (s) (diana)
RZ-Merge	190	Merge	82	0.084	2.4
Connector	325	OR, Mixer	1143	0.445	5
Connector	325	PAR, Join	1143	0.452	4.4
Mod-3 Counter	163	Mod-1 Counter, Join, Forks, Merge, Toggle	1596	0.775	4.4
Duplicator	487	PAR, Mixer	2358	0.827	8.1
2-to-4 Converter	568	Merge, Toggle	1953	0.851	4.3
Sequencer	973	Mixer, Join	6588	2.828	114
Latch	1459	Non-Receptive Mixer, Join	6102	2.844	12.6

Table 2.1: Performance of the CWB and Diana on various DI circuits

2.7.1 Verification of call element

A call element can be declared in DI-Algebra using the concrete syntax as follows:

$$
\begin{aligned}
Call = \quad & \{a0, a1, b\}, \{d0, d1, c\} \\
& [\ a0? \rightarrow c!; C0 \ \# \ a1? \rightarrow c!; C1\ ] \\
C0 = \quad & \{a0, a1, b\}, \{d0, d1, c\} \\
& [\ b? \rightarrow d0!; Call \ \# \ a0? \rightarrow \texttt{CHAOS} \ \# \ a1? \rightarrow \texttt{CHAOS}\ ] \\
C1 = \quad & \{a0, a1, b\}, \{d0, d1, c\} \\
& [\ b? \rightarrow d1!; Call \ \# \ a0? \rightarrow \texttt{CHAOS} \ \# \ a1? \rightarrow \texttt{CHAOS}\ ]
\end{aligned}
$$

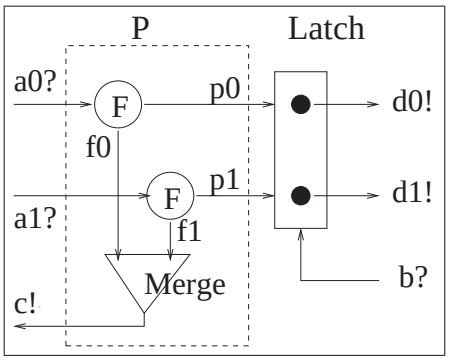


Figure 2.12: DI-Decomposition of a $Call$ element into $Fork$, $Merge$ and $Latch$ components

An implementation [Ver03] of a Call element is shown in Figure 2.12. The transition system built by the CWB for this decomposition had almost 10^7 states.

As it was not possible to verify this directly on the given machine (a 1.4 GHz Pentium 4 with 256 MB RAM), a hierarchical approach is adopted. The circuit is divided into two components, an abstract description P (of the component shown dotted in the figure) and the Latch element.

$$
\begin{aligned}
P = \quad & \{a0, a1\}, \{c, p0, p1\} \\
& [\ a0? \rightarrow c!; p0!; P \ \# \ a1? \rightarrow c!; p1!; P\ ] \\
Latch = \quad & \{p0, p1, b\}, \{d0, d1\} \\
& [\quad p0? \rightarrow [\ p1? \rightarrow \texttt{CHAOS} \ \# \ b? \rightarrow d0!; Latch\ ] \\
& \# \ p1? \rightarrow [\ p0? \rightarrow \texttt{CHAOS} \ \# \ b? \rightarrow d1!; Latch\ ]\ ] \\
M = \quad & (Di\_P \mid Di\_Latch) \backslash \{p0, p1\}
\end{aligned}
$$

Here M is the composition of Di_P and Di_Latch each of which includes wires for the internal signals $p0$ and $p1$. This is optimised by keeping just one wire for each of $p0$ and $p1$. M is verified against the specification of Di_Call element, as follows:

Size of Di_CALL = 5833, Size of M = 196857

$\texttt{mustpre}(\texttt{Di\_CALL}, \texttt{M}) = \texttt{true}$

Time taken = 30 min, 36 sec

P is implemented as two forks and a merge as shown below:

$$
\begin{aligned}
Fork0 = \quad & \{a0\}, \{p0, f0\} \ \ a0?; p0!; f0!; Fork0 \\
Fork1 = \quad & \{a1\}, \{p1, f1\} \ \ a1?; p1!; f1!; Fork1 \\
Merge = \quad & \{f1, f0\}, \{c\} \ \ [\ f1? \rightarrow c!; Merge \ \# \ f0? \rightarrow c!; Merge\ ] \\
N = \quad & (Di\_Fork0 \mid Di\_Fork1 \mid Di\_Merge) \backslash \{f0, f1\}
\end{aligned}
$$

Then N is verified against Di_P.

Size of Di_P = 1216, Size of N = 39375

$\texttt{mustpre}(\texttt{Di\_P}, \texttt{N}) = \texttt{true}$

Time taken = 19.596 sec

2.8 Conclusion

A method is defined for modelling delay-insensitive circuits using a general-purpose process calculus, CCS. The method requires one to model explicitly the wires that form a Foam Rubber Wrapper, as in [Dil89]. In contrast, [JU93] introduced a special-purpose calculus, DI-Algebra, in which every process is implicitly DI.

The property of delay-insensitivity for CCS processes has been defined with respect to the MUST-testing preorder (rather than bisimulation). Moreover, as in DI-Algebra, the MUST-testing preorder captures the refinement relationship between specification and implementation.

DI-modules (and indeed any module that can treat input as unsafe, but cannot block it) are modelled in [Dil89] by prefix-closed trace structures, rather than by CCS processes. A trace structure defines when an input can safely be received and when an output can safely be sent by a module. The refinement relationship (called "conformation" in [Dil89]) between trace structures allows more inputs and fewer outputs to be guaranteed as safe. Consequently, a trace structure corresponding to a "universal do-nothing" module refines any trace structure defined over the same alphabet of inputs and outputs. In contrast, CCS processes capture not only the above safety properties, but also the progress property that output must occur. In this semantically-richer framework, the MUST-testing preorder preserves all guarantees of progress, whilst allowing the elimination of nondeterministic choice between outputs.

The CWB supports verification based upon MUST-testing. As observed from experiments using this tool, response times can be reasonable for small circuits. As the number of wires increases, however, the state-explosion problem can become severe. Hierarchical verification can help here, but alternative tools are also worth investigating. Another problem with the CWB is that its facilities (such as the `dftrace` command) for generating counter-examples are available for bisimulation, but not for MUST-testing.

Chapter 3

Verification in Restrictive Environments

3.1 Introduction

The previous chapter discussed the verification of processes described in DI-Algebra by applying the Concurrency Workbench. This comparison did not assume any particular environment in which the process would be operating, and hence their equivalence was very strong.

Since a specification of a process describes its behaviour over communication actions with its environment, the environment can veto the occurrence of certain events. An obligation on the environment not to provide a particular input after a particular trace can be expressed in DI-Algebra, but it is often tedious to do so. It can be more convenient to describe the behaviour of a module separately from the behaviour of its environment. Both descriptions should then be taken into account when implementing the module. Verification of such processes in conjunction with their environments is the topic of this chapter.

Two processes which are not generally equivalent may have the same behaviour in a certain environment, and thus are indistinguishable when operating

in that environment. If the environment under which it is used does not demand for certain tasks, then such tasks become extra or unnecessary specification in the process. These will never be invoked and they need not be implemented. For example, consider a specification for a vending machine which delivers both tea and coffee. If we know that the customers will always ask for tea and never request coffee from the machine, then we can weaken our specification to a tea vending machine. In this situation a tea vending machine cannot be distinguished from a tea and coffee vending machine in the given environment.

To help verification of processes in conjunction with their environments, one needs to derive the effective behaviour of the process running in that environment. This chapter introduces the concept of restricting one DI process by another in order to weaken the former, where the two processes together form a closed system [1] [Ver94]. Restriction is formalised as an operator (denoted by the symbol $\restriction$) that performs a *directed transformation* which is contracting [Mal00a], i.e., $(P \restriction Q) \sqsubseteq P$. Apart from removing the extra specifications this operator is useful to get the correct behaviour of a process, i.e. to lead certain sequence of events to $\perp$ if the environment does not demand them. The usefulness of this requirement is illustrated by the following example.

Example:

Consider the specification of a Nacking-Arbiter [JU90a, Ver03]

$$
\begin{aligned}
N &= [\ r_0? \rightarrow a_0!\ ;\ B_0 \ \Box\ r_1? \rightarrow a_1!\ ;\ B_1\ ] \\
B_0 &= [\ r_0? \rightarrow a_0!\ ;\ N \ \Box\ r_1? \rightarrow n_1!\ ;\ B_0\ ] \\
B_1 &= [\ r_1? \rightarrow a_1!\ ;\ N \ \Box\ r_0? \rightarrow n_0!\ ;\ B_1\ ]
\end{aligned}
$$

Where the arbiter N has input alphabet $\mathcal{A} = \{r_0, r_1\}$ and output alphabet

[1] Closed systems formed by the interconnection of a module and its environment are also used in Compositional Model Checking [CLM89]; and supervisory control of Discrete Event Systems [RW89].

$\mathcal{B} = \{a_0, n_0, a_1, n_1\}$ and communicates with environment components E_0 and E_1 (Figure 3.1). The environment process E_0 can send signals over r_0 and receive signals over a_0 or n_0, similarly environment process E_1 can send signals over r_1 and receive signals over a_1 or n_1.

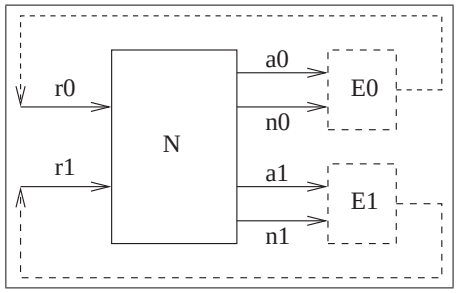


Figure 3.1: Nacking Arbiter (N) and its environment (shown dotted)

Suppose E_0 issues a request on r_0 and receives an acknowledgement on a_0. Subsequently, it sends another request r_0 as part of return-to-zero signalling. In response to this N sends an acknowledgement on a_0. Now E_1 sends a request on r_1 and receives an acknowledgement on a_1. Note that the process N is now in state B_1. If E_0 sends another request on r_0 (without absorbing the a_0 which is on its way) then N sends a nack to it on n_0. This is harmless as it does not cause interference on a_0. In other words, according to the specification, $N/r_0?/a_0!/r_0?/r_1?/a_1!/r_0? \neq \perp$. However this sequence of events is unsafe, since E_0 observes a sequence $r_0?; a_0!; r0?; r0?$, that is it produces two r_0 signals in succession violating the delay-insensitive behaviour. But all the processes are DI and therefore such a situation cannot arise. The nacking arbiter specification therefore need not guarantee the safety of the above sequence. The specification is therefore too strong.

For correct operation of the arbiter protocol the signal transitions on r_i must alternate with either a_i or n_i. Mallon specifies this using the alternation $\langle\{r_0\}, \{a_0, n_0\}\rangle, \langle\{r_1\}, \{a_1, n_1\}\rangle$. To obtain the same effect using the restriction operator, one can model an appropriate environment $E_0 \mid\mid E_1$ where E_0 and E_1 are described as follows and achieve the same result.

$E_0 = r_0!; [\ a_0? \rightarrow E_0 \ \Box\ n_0? \rightarrow E_0\ ]$

$E_1 = r_1!; [\ a_1? \rightarrow E_1 \ \Box\ n_1? \rightarrow E_1\ ]$

Note that this description of the environment does not send another r_i transition before it has absorbed a transition on either a_i or n_i. ♦

The restriction operator can thus help us to obtain the exact behaviour required of a process. This chapter gives trace theoretic semantics for the same. Application of restriction in synthesis and verification is shown using the existing tools di2pn [JF00] and diana [Fur02]. Basic concepts in trace theoretic semantics are given in section 3.2 followed by a brief background on semantics of DI-Algebra. Alternation is discussed in section 3.3 and motivation for the restriction operator if given in section 3.4. Section 3.5 gives the trace theoretic semantics of the restriction operator and proves its healthiness conditions. A way of constructing a suitable environment to obtain the effect of alternation is given in section 3.6. Also a method to eliminate alternation is given in section 3.7.

3.2 Trace Theoretic Semantics for DI-Algebra

Suppose a process has engaged in a finite sequence (trace) of events. As a result it may be in an undesirable state, in which case the subsequent behaviour of the process is undefined. Such a trace is called a divergence of the process. Another possibility is that the process is quiescent, i.e. it will not output until further input is supplied. In either case, such a trace is called a failure of the process, and so by definition every divergence is also a failure and every failure is also a trace [Jos92].

A denotational semantics is given in [Luc94, Mal00b] to DI-Algebra. This associates a set $F[\![P]\!]$ of failures with each process P. For example, an erroneous process ($\perp$) can do anything whatsoever and therefore $F[\![\perp]\!] = (\mathcal{A} \cup \mathcal{B})^*$. Also, $F[\![P \sqcap Q]\!] = F[\![P]\!] \cup F[\![Q]\!]$, so that $P \sqsubseteq Q$ iff $F[\![P]\!] \supseteq F[\![Q]\!]$. The set $T[\![P]\!]$ of traces and the set $D[\![P]\!]$ of divergences can in fact be calculated from $F[\![P]\!]$ [Jos92].

Example:

Consider the specification of a Merge module (Figure 3.2) given below:

$$M = [\ a? \rightarrow c!\ ;\ M \ \Box\ b? \rightarrow c!\ ;\ M\ ]$$

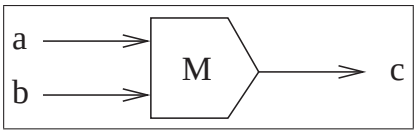


Figure 3.2: Merge module M with inputs a, b and output c

Divergences of M : $aa, bb, ab, ba, acbcaa, \ldots$

Traces of M : $\varepsilon, a, b, ac, bc, acb, aca, bca, \ldots$ + Divergences

Failures of M : $ac, bc, acac, acbc, \ldots$ + Divergences

These can be expressed using regular-expressions as follows:

$D[\![M]\!] = (ac + bc)^*(aa + bb + ab + ba)(a + b + c)^*$

$T[\![M]\!] = (ac + bc)^*(\varepsilon + a + b) \cup D[\![M]\!]$

$F[\![M]\!] = (ac + bc)^* \cup D[\![M]\!]$

♦

3.2.1 Trace reordering

A process expressed in DI-Algebra is invariant when composed with a Foam Rubber Wrapper (FRW) [MFR85, Udd86] modelling wires of unbounded delay. When communicating through them, the process and its environment may observe different traces. Suppose the environment observed trace x, the process observed trace y, and there are no signals currently in transit. Then x and y are related. They contain the same symbols, but the environment may have observed inputs earlier, whereas the process may have observed outputs earlier. These traces are

captured by the reordering relation $\ltimes$ between traces [JHH89] which allows

- input events to be moved in front of other events
- output events to be moved behind other events.

Formally, $a \in \mathcal{A} \vee b \in \mathcal{B} \Rightarrow s\,a\,b\,t \ltimes s\,b\,a\,t$
According to [Luc94] the failure set of a DI process is closed under reordering.
Therefore, $s \ltimes t \ \wedge \ t \in F[\![P]\!] \Rightarrow s \in F[\![P]\!]$
Note that moving input events in front makes the process less deterministic, i.e. if $s \ltimes t$ and $t \in T[\![P]\!]$, then $P/s \sqsubseteq P/t$.

3.2.2 Failure sets

Let $\sigma : Procvar \rightarrow F$, be the valuation function (the interpretation of the process variables). The meaning of a process expression P, is denoted by $F[\![P]\!]\sigma$. When valuation is clear from the context it is not shown. The semantic definitions of process expression as mentioned in [MU98] are given below.

1. $F[\![P]\!]\sigma = \sigma.P$,if $P \in Procvar$
2. $F[\![\bot]\!]\sigma = (\mathcal{A} \cup \mathcal{B})^*$
3. $F[\![P \sqcap Q]\!]\sigma = F[\![P]\!]\sigma \cup F[\![Q]\!]\sigma$
4. $F[\![P/a]\!]\sigma = \{s : a\,s \in F[\![P]\!]\sigma : s\}$
5. The set SD (sure divergences) contains all traces in which the environment causes interference by sending two signals on the same channel without waiting for an acknowledgement in between. As shown in [Luc94], SD is a subset of all failure sets, because DI processes are receptive [Jos92].
   $SD = \{a, s, t : a \in \mathcal{A} \wedge s \ltimes a\,a\,t : s\}$

6. $F[\![(i :: a_i \rightarrow P_i)]\!]\sigma =$

$$\cup \left(\begin{array}{l} \{s,t,i : s \ltimes a_i\, t \wedge t \in F[\![P_i]\!]\sigma : s\} \\ \{y,s,i : y\, a_i \in T[\![P_i]\!]\sigma \wedge a_i \in \mathcal{B} : ys\} \\ \{s,i : a_i = skip \wedge s \in F[\![P_i]\!]\sigma : s\} \\ SD \\ (\{b : b \in \mathcal{A} \wedge (\forall i :: a_i \neq b \wedge a_i \in \mathcal{A}) : b\})^+ \\ \{: (\forall i :: a_i \in \mathcal{A}) : \varepsilon\} \end{array} \right)$$

7. Recursive DI specification are of the form $X_i = E_i$, where X_i is a process variable ($Procvar$) and E_i is a process expression. If $\sqsubseteq$ on valuations is defined as

$$\sigma \sqsubseteq \tau \equiv (\forall X : X \in Procvar : \sigma.X \supseteq \tau.X)$$

then a DI specification specifies the least valuation μ (w.r.t. $\sqsubseteq$) such that for any i we have:

$$F[\![X_i]\!]\mu = F[\![E_i]\!]\mu$$

It was shown in [Luc94] that such a μ exists. Thus, by successive approximations we define the failure set of a recursive process as follows:

$$F[\![X_i]\!]\beta^0 = F[\![\bot]\!] = (\mathcal{A} \cup \mathcal{B})^*$$
$$F[\![X_i]\!]\beta^{k+1} = F[\![E_i]\!]\beta^k$$
$$F[\![X_i]\!]\mu = (\sqcup k :: F[\![E_i]\!]\beta^k), \text{ for all } i$$

3.3 Alternation

To force a process P to follow a certain sequence of events Mallon [Mal00a] introduced the alternation operator to consist of a pair of sets of events $\langle S, T\rangle$. Here S and T are disjoint subsets of $\mathcal{A} \cup \mathcal{B}$, and its application to the process P is denoted by $\langle S, T\rangle P$. The idea is that transitions on signals in S and T must alternate, beginning with a signal from S. A violation of this protocol leads to $\bot$.

Example:

Consider a process P defined as follows with $\mathcal{A} = \{ld, rd\}$ and $\mathcal{B} = \{ls, rs\}$.

$$P = ld? \ ; \ rd? \ ; \ ls! \ ; \ rs! \ ; \ P$$

Process P communicates with a left hand client (L) and a right hand client (R) (Figure 3.3). L can send signals to P over ld and receive signals over ls. R can communicate over rd and rs. L sends a transition on ld if it has absorbed a transition on ls. R behaves similarly.

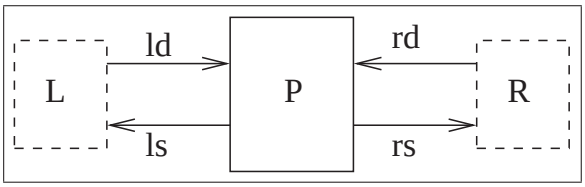


Figure 3.3: Process P communicating with environment components L and R

According to DI-Algebra once an output is generated by the process all inputs become safe again. This however is not true for the above process P. For example, after the process outputs on ls it is not safe to receive input on rd since R has still to absorb the output on rs. Therefore to satisfy this requirement one needs to modify the specification of P as follows:

$$\begin{aligned} P' = ld? \ ; \ rd? \ ; \ (\quad & (ls! \ ; \ [\, rs! \rightarrow P' \ \Box \ rd? \rightarrow \bot \,]) \\ \sqcap \quad & (rs! \ ; \ [\, ls! \rightarrow P' \ \Box \ ld? \rightarrow \bot \,])) \end{aligned}$$

The non-deterministic choice ascertains that the environment does not know which of the two expressions is responsible for generating the output. Thus, upon reception of ls it is unsafe to send rd, since the first argument may have been chosen. Likewise, upon reception of rs it is unsafe to send ld since the second argument may have been chosen. Such a requirement can be specified using al-

ternation as follows:

$$\langle\{ld?\},\{ls!\}\rangle\ \langle\{rd?\},\{rs!\}\rangle\ \ ld?\ ;\ rd?\ ;\ ls!\ ;\ rs!\ ;\ P$$

The use of $\langle\{ld?\},\{ls!\}\rangle$ operator amounts to specifying that in communicating with P, ld and ls must alternate starting with ld. Violation of this protocol is considered a transgression by the environment, and causes the process to enter the error state. ♦

In DI-Algebra, by default once a module absorbs (produces) a transition, it becomes safe for it to produce (absorb) any transition again. Often, however one wants to partition the input and output signals of a process into handshake ports. Alternation facilitates this. (Another approach is to raise the level of abstraction and adopt Handshake Algebra [JLUV94, JUY93].) Mallon claimed that it was not easy to eliminate the alternation operator. This chapter however shows the method of doing this by way of parallel composition in Section 3.7.

3.4 Motivation for the Restriction Operator

3.4.1 Alternation can be expressed by restriction

The natural way to specify a module in DI-Algebra may fail to capture the requirement for handshaking on a number of independent ports. Mallon [Mal00a] observed this phenomenon for the Nacking Arbiter [JU90a, Ver03] and used this example to motivate the introduction of his alternation operator. Here restriction can be used to achieve the same effect as was demonstrated by the example in the beginning of this chapter. It was shown that, $N/r_i?/a_i!/r_i?/r_{\bar{i}}?/a_{\bar{i}}!/r_i? \neq \perp$ even though the environment has yet to receive an acknowledgement (a transition on a_i) of the release of the arbiter, where $\bar{i} = 1 - i$. The specification is therefore too strong.

We can use the tool diana to analyse that the nacking arbiter specification N does not diverge on the above trace as follows. This is valid for the restriction operator as its semantics are consistent with that of diana.
To obtain the divergences we first translate N into its Petri net fragment (N.pn) using di2pn and then run the following command:

```
$> diana  N.pn  --traces  --plots=divergences
```

This gives all the divergences of N, without considering its environment. The result does not include the traces

```
r0? a0! r0? r1? a1! r0?
r1? a1! r1? r0? a0! r1?
```

On the other hand, restricting N by the environment $E = E_0 \,||\, E_1$ induces a weaker specification. This can be verified by translating E into a Petri net fragment and generating the divergences of N when restricted by the environment E.

```
$> diana  N.pn  E.pn  --traces  --plots=divergences
```

This time the desired traces are included amongst the divergences. Figure 3.4 shows all the divergent traces of N in its restrictive environment E.

3.4.2 Restriction is more general than alternation

The alternation $\langle S, T \rangle P$ enforces the requirement that transitions on signals in S and T must alternate in all safe traces of P. It does not support other requirements, such as a restriction to return-to-zero signalling.
Example:

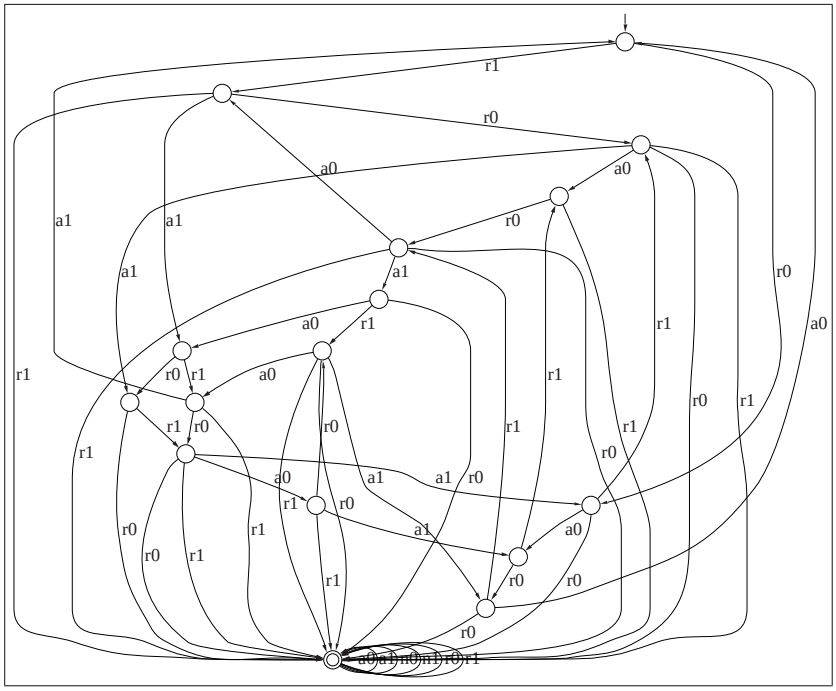


Figure 3.4: Divergent traces of $N \upharpoonright E$

Consider the process P describing a Merge module with input alphabet $\{a, b\}$ and output alphabet $\{c\}$.

$$P = [a? \rightarrow c!; P \,\Box\, b? \rightarrow c!; P]$$

The following process, R, describes a restrictive environment for P.

$$R = [a! \rightarrow c?; a!; c?; R \,\Box\, b! \rightarrow c?; b!; c?; R]$$

As the environment enforces a return-to-zero protocol, the safe traces to which P is restricted will be prefixes of $((acac) + (bcbc))^*$. That is, $P \upharpoonright R$ does not expect an odd number of handshakes involving a to be followed by a handshake involving b (or vice versa). This cannot be expressed using alternation with set $S = \{a, b\}$ and set $T = \{c\}$ or any other combination. The effective behaviour would be as

given by Q, below.

$$\begin{aligned} Q = \quad & [\quad a? \rightarrow c!; [a? \rightarrow c!; Q \,\Box\, b? \rightarrow \bot\ ] \\ & \Box\ b? \rightarrow c!; [b? \rightarrow c!; Q \,\Box\, a? \rightarrow \bot\ ]\ ] \end{aligned}$$

The equivalence of P and Q when restricted to R can be verified using diana as follows.

```
$> diana  P.pn  Q.pn  R.pn  --tests=equivalence

specification: P.pn
8 places 8 transitions 11 markings 6 reduced markings

implementation: Q.pn
13 places 12 transitions 15 markings 6 reduced markings

equivalence: yes
```

As the environment is restrictive, the original specification, P, has extra quiescent traces in comparison with the restricted one. These extra traces can be generated using diana.

```
$> diana  Q.pn  P.pn --plots=extras --traces
```

The graph generated (Figure 3.5) shows the traces that are quiescent for P, arising only after at least one input transition that is prohibited according to Q. These are traces of P that will remain unexplored in its restrictive environment. ♦

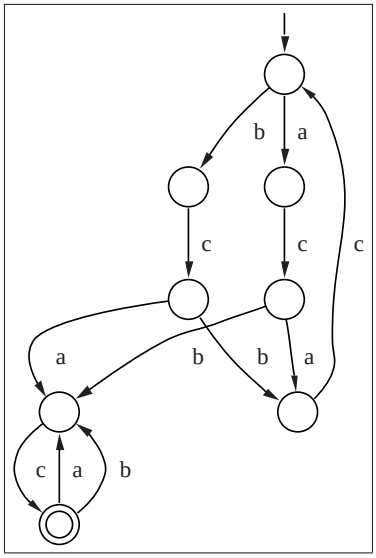


Figure 3.5: Extra traces of P in comparison with $P \upharpoonright R$

3.4.3 Restriction facilitates synthesis

As the effective behaviour of a process in a restrictive environment is weaker than the original process, the designer has more freedom in refinement and in logic synthesis. Sometimes a costly circuit implementation can be avoided, as the following example demonstrates.

Example:
Consider the process M describing a Mutual Exclusion element that is required to operate in the restrictive environment E_0, as follows:

$$M = [\ (j : 0 \leq j < 2 : r_j? \rightarrow g_j!\ ;\ r_j?\ ;\ g_j!\ ;\ M)\ ]$$
$$E_i = r_i!\ ;\ g_i?\ ;\ r_i!\ ;\ g_i?\ ;\ E_{\bar{i}}$$

where $\bar{i} = 1 - i,\ 0 \leq i < 2$.

In this environment, M can be implemented by two wires! That is, $M \upharpoonright E_0$ can be implemented by $W_0 \mid\mid W_1$, where

$$W_i = r_i?\ ;\ g_i!\ ;\ W_i, \quad 0 \leq i < 2.$$

To verify this, Petri net fragments M.pn, Impl.pn and E0.pn of M, $W_0 \mid\mid W_1$ and E_0, respectively, generated by di2pn are used as inputs to diana.

```
$> diana  M.pn  Impl.pn  E0.pn  --tests=equivalence

specification: M.pn
15 places 12 transitions 16 markings 8 reduced markings

implementation: Impl.pn
10 places 8 transitions 12 markings 8 reduced markings

equivalence: yes
```

Alternatively this implementation can be automatically synthesised from M and E_0, as shown below. ♦

Synthesis using di2pn and Petrify

Consider the specification of mutual exclusion element (M) and the restrictive environment (E_0) from the above example. To synthesise this mutex in an environment imposing serialisation of requests the tool di2pn is used as follows:

```
$> di2pn mutex.di
```

The generated Petri net file, M.pn, shown in Figure 3.6, is then input to Petrify to synthesise the circuit using a generalised C-element implementation.

```
$> petrify -gc -eqn Mutex.gc M.pn
```

The circuit synthesised by Petrify consists of two wires, as expected.

```
$> cat Mutex.gc
# EQN file for model Mutex
# Generated by petrify 4.0 (compiled 22-Dec-98 at 8:44 AM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
```

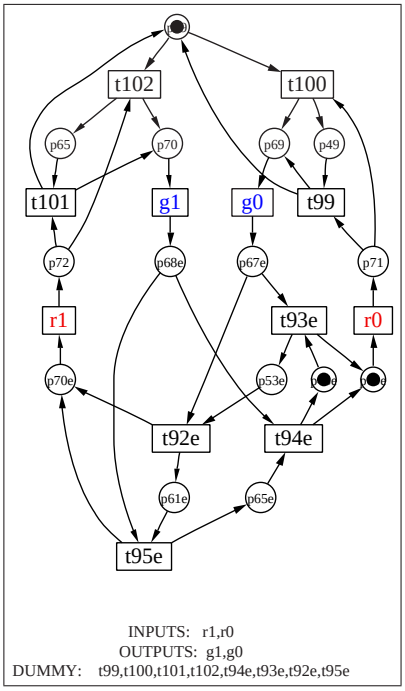


Figure 3.6: Petri net for the specification of a mutex in the environment $E0$

Estimated area = 2.00

INORDER = r1 r0 g1 g0;

OUTORDER = [g1] [g0];

[g1] = r1;

[g0] = r0;

Thus the specification of environment helps to obtain the exact specification of the process and thus simple circuit implementations. The verification of this result using diana was shown in the previous example.

3.5 Trace Theoretic Semantics of Restriction

Till now we have seen the various advantages and applications of restricting a process to its environment. The available tools di2pn and diana can be used for implementation and automatic verification, respectively, of processes in restrictive

environments. This section formally defines the restriction operator and states some of its important properties (such as $P \restriction Q \sqsubseteq P$).

3.5.1 Definition

Consider a process P (with input alphabet $\mathcal{A}$ and output alphabet $\mathcal{B}$) that communicates with its environment Q (with input alphabet $\mathcal{B}$ and output alphabet $\mathcal{A}$) by synchronising over the input and output of transitions. Let x be a trace, failure or divergence of P. If x is also a trace of Q then it is indeed a trace, failure or divergence, respectively, of $P \restriction Q$. If Q cannot engage in x, then the behaviour of P after x will never be explored. It is therefore harmless to model x and all its extensions as divergences of $P \restriction Q$.

Let the extra divergences ED be defined by

$$\{s, t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\}$$

Then,

(R1) $F[\![P \restriction Q]\!] = F[\![P]\!] \cup ED$

(R2) $T[\![P \restriction Q]\!] = T[\![P]\!] \cup ED$

(R3) $D[\![P \restriction Q]\!] = D[\![P]\!] \cup ED$

In fact, (R2) and (R3) follow from (R1) [Jos92]. Note that the definition of Mallon's alternation operator is far more complex.

Example:

Consider the process P and its environment Q defined by

$P = [a? \rightarrow c!; P \,\Box\, b? \rightarrow c!; P]$ and $Q = a!; c?; Q$. It follows that

$D[\![P]\!] = (ac + bc)^*(ab + aa + bb + ba)(a + b + c)^*$

$F[\![P]\!] = (ac + bc)^* \cup D[\![P]\!]$

$T[\![P]\!] = (ac + bc)^*(\varepsilon + a + b) \cup D[\![P]\!]$

$D[\![Q]\!] = (ac)^* c\ (a + b + c)^*$

$T[\![Q]\!] = (ac)^*(\varepsilon + a) \cup D[\![Q]\!]$

As can be seen, Q never produces an output on b and therefore $P \upharpoonright Q$ is allowed to diverge after an input on b. We can calculate that

$F[\![P \upharpoonright Q]\!] = (ac)^*(\varepsilon + (b + aa + ab)\ (a + b + c)^*)$

This can be expressed more succinctly in DI-Algebra by the process

$P' = [a? \rightarrow c!; P' \ \Box\ b? \rightarrow \bot]$. ♦

Relation to the *dive* operator

Mallon's *dive* operator [Mal00a] determines those traces in the process that do not comply to the alternation protocol and therefore lead to error. He defined the *dive* (divergence extension) on the failure set F of a process P and a trace set V as follows:

$$dive(V, F[\![P]\!]) = (\ltimes)((((\ltimes)V \cap T[\![P]\!]) \ominus \mathcal{B}^*)(\mathcal{A} \cup \mathcal{B})^*)$$

This set contains those traces which should be treated unsafe. The trace set V is the set of traces for which the projections are not alternations. Formally,

$$x \in V = x \uparrow (S \cup T) \notin (ST)^*$$

The reduction operator $\ominus$ is defined as follows: (here G and H are trace sets)

$$x \in G \ominus H = (\exists r : r \in H : x\, r \in G)$$

Relating *dive* to the restriction operator, we see that

$$ED = dive(V, F[\![P]\!]) \text{ with } V = T[\![P]\!] \setminus T[\![Q]\!]$$

Note that the definition of restriction operator is more general than *dive* and not limited to the sets S and T.

3.5.2 Healthiness conditions

The semantics of a process must satisfy certain "healthiness" conditions [Jos92]. We need to check these conditions to ensure that the restriction operator is well defined. The following are a few examples of such conditions satisfied by the failures (F) and divergences (D) of $P \upharpoonright Q$.

(H1) D is extension closed.

$\{s, t : s \in D \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \subseteq D$

Proof:

Follows from R3, given that both $D[\![P]\!]$ and ED are extension closed. ∎

(H2) Every divergence is a failure.

$D \subseteq F$

Proof:

Follows directly from R1 and R3. ∎

(H3) D is closed under curtailment of output.

$\{s, t : st \in D \wedge t \in \mathcal{B}^* : s\} \subseteq D$

Proof:

Either $st \in D[\![P]\!]$ or $st \in ED$ by R3.

If $st \in D[\![P]\!]$, then $s \in D$ follows from closure of $D[\![P]\!]$.

Otherwise, $st \in ED \setminus D[\![P]\!]$. There are then two cases.

(i) $(\exists q, r : s = q\,r : q \in T[\![P]\!] \setminus T[\![Q]\!])$, and so $s \in ED \subseteq D$.

(ii) Let $u \in \mathcal{B}^+$ be the shortest sequence such that

$(\exists v : t = u\,v : s\,u \in T[\![P]\!] \setminus T[\![Q]\!])$.

Let $u = u'\,b$.

$s\,u' \in T[\![Q]\!] \Rightarrow s\,u'\,b \in T[\![Q]\!]$ since Q is receptive to input.

But $s\,u'\,b = s\,u \notin T[\![Q]\!]$, so $s\,u' \notin T[\![Q]\!]$.

This contradicts the assumption that the shortest sequence is non-empty, since $T[\![P]\!]$ is prefix closed. ∎

(H4) F is closed under reordering.

$\{s, s' : s' \ltimes s \wedge s \in F : s'\} \subseteq F$

Proof:

Let $\ltimes_{\mathcal{A},\mathcal{B}}$ denote reordering with respect to input alphabet $\mathcal{A}$ and output alphabet $\mathcal{B}$.

We have the property, $s' \ltimes_{\mathcal{A},\mathcal{B}} s \equiv s \ltimes_{\mathcal{B},\mathcal{A}} s'$.

It suffices to show that $r \ltimes_{\mathcal{A},\mathcal{B}} st \wedge s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* \Rightarrow r \in ED$.

We proceed by induction on t.

Base case: $t = \varepsilon$.

Assume that $r \ltimes_{\mathcal{A},\mathcal{B}} s \ \wedge \ s \in T[\![P]\!] \setminus T[\![Q]\!]$.

$r \ltimes_{\mathcal{A},\mathcal{B}} s \ \wedge \ s \in T[\![P]\!]$

$\Rightarrow$ { $T[\![P]\!]$ is closed under $\ltimes_{\mathcal{A},\mathcal{B}}$ }

$r \in T[\![P]\!]$

$r \ltimes_{\mathcal{A},\mathcal{B}} s \ \wedge \ r \in T[\![Q]\!]$

$\Rightarrow$ { Property of $\ltimes$ }

$s \ltimes_{\mathcal{B},\mathcal{A}} r \ \wedge \ r \in T[\![Q]\!]$

$\Rightarrow$ { $T[\![Q]\!]$ is closed under $\ltimes_{\mathcal{B},\mathcal{A}}$ }

$s \in T[\![Q]\!]$

$\Rightarrow$ { $s \notin T[\![Q]\!]$ }

$false$

Therefore, $r \in T[\![P]\!] \setminus T[\![Q]\!]$ and so $r \in ED$.

Inductive step: Assuming that

$r' \ltimes_{\mathcal{A},\mathcal{B}} st' \ \wedge \ s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t' \in (\mathcal{A} \cup \mathcal{B})^* \Rightarrow r' \in ED$, for all r',

and $r \ltimes_{\mathcal{A},\mathcal{B}} st'c \ \wedge \ s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t'c \in (\mathcal{A} \cup \mathcal{B})^*$. We show that $r \in ED$.

Let $r = ucv \wedge v \upharpoonright \{c\} = \varepsilon$. Then, $uv \ltimes st'$ and so $uv \in ED$ by induction hypothesis.

There are two cases to consider:

(i) $u = u_0u_1 \wedge u_0 \in T[\![P]\!] \setminus T[\![Q]\!]$.

This implies that $ucv = u_0u_1cv \in ED$.

(ii) $v = v_0v_1 \wedge uv_0 \in T[\![P]\!] \setminus T[\![Q]\!] \wedge v_1 \in (\mathcal{A} \cup \mathcal{B})^* \wedge u \in T[\![P]\!] \cap T[\![Q]\!]$

If $c \in \mathcal{A}$,

$$
\begin{array}{ll}
 & ucv_0 \ltimes_{\mathcal{A},\mathcal{B}} uv_0c \wedge uv_0 \in T[\![P]\!] \\
\Rightarrow & \quad \{ \ P \text{ is receptive to input } \} \\
 & ucv_0 \ltimes_{\mathcal{A},\mathcal{B}} uv_0c \wedge uv_0c \in T[\![P]\!] \\
\Rightarrow & \quad \{ \ T[\![P]\!] \text{ is closed under } \ltimes_{\mathcal{A},\mathcal{B}} \ \} \\
 & ucv_0 \in T[\![P]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & ucv_0 \ltimes_{\mathcal{A},\mathcal{B}} uv_0c \wedge ucv_0 \in T[\![Q]\!] \\
\Rightarrow & \quad \{ \text{ Property of } \ltimes \ \} \\
 & uv_0c \ltimes_{\mathcal{B},\mathcal{A}} ucv_0 \wedge ucv_0 \in T[\![Q]\!] \\
\Rightarrow & \quad \{ \ T[\![Q]\!] \text{ is closed under } \ltimes_{\mathcal{B},\mathcal{A}} \ \} \\
 & uv_0c \in T[\![Q]\!] \\
\Rightarrow & \quad \{ \ T[\![Q]\!] \text{ is prefix closed } \} \\
 & uv_0 \in T[\![Q]\!] \\
\Rightarrow & \quad \{ \ uv_0 \notin T[\![Q]\!] \ \} \\
 & false
\end{array}
$$

Therefore $ucv_0 \in T[\![P]\!] \setminus T[\![Q]\!]$ and so $r \in ED$.

If $c \in \mathcal{B}$,

$$
\begin{array}{ll}
& v \upharpoonright \{c\} = \varepsilon \wedge ucv \ltimes_{\mathcal{A},\mathcal{B}} st'c \wedge c \in \mathcal{B} \\
\Rightarrow & \quad \{ \text{ Property of reordering } \} \\
& v \in \mathcal{B}^* \\
\\
& uv_0 \notin T[\![Q]\!] \\
\Rightarrow & \quad \{ \ v \in \mathcal{B}^* \text{ and } Q \text{ is receptive to input } \} \\
& u \notin T[\![Q]\!] \\
\Rightarrow & \quad \{ \ u \in T[\![Q]\!] \ \} \\
& false
\end{array}
$$

Therefore such a situation cannot arise. ∎

3.5.3 Properties of the restriction operator

The following are some interesting properties satisfied by the restriction operator.

(P1) Restriction by a divergent environment is no restriction.

$P \upharpoonright \perp = P$

Proof:

We need to show that $F[\![P \upharpoonright \perp]\!] = F[\![P]\!]$.

$$
\begin{array}{ll}
& F[\![P \upharpoonright \perp]\!] \\
= & \quad \{ \text{ R1 } \} \\
& F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![\perp]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= & \quad \{ \ T[\![\perp]\!] = (\mathcal{A} \cup \mathcal{B})^* \ \} \\
& F[\![P]\!]
\end{array}
$$

∎

(P2) Restriction distributes through non-deterministic choice.

$(R \sqcap S) \upharpoonright Q = (R \upharpoonright Q) \sqcap (S \upharpoonright Q)$

Proof:

$F[\![R \sqcap S]\!] = F[\![R]\!] \cup F[\![S]\!]$ (i)

$T[\![R \sqcap S]\!] = T[\![R]\!] \cup T[\![S]\!]$ (ii)

$$
\begin{aligned}
& F[\![(R \sqcap S) \upharpoonright Q]\!] \\
= \quad & \{ \text{R1} \} \\
& F[\![(R \sqcap S)]\!] \cup \{s,t : s \in T[\![(R \sqcap S)]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= \quad & \{ i \text{ and } ii \} \\
& F[\![R]\!] \cup F[\![S]\!] \cup \{s,t : s \in (T[\![R]\!] \cup T[\![S]\!]) \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= \quad & \{ \text{set theory} \} \\
& F[\![R]\!] \cup F[\![S]\!] \cup \\
& \{s,t : s \in T[\![R]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \cup \\
& \{s,t : s \in T[\![S]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= \quad & \{ \text{R1} \} \\
& F[\![R \upharpoonright Q]\!] \cup F[\![S \upharpoonright Q]\!]
\end{aligned}
$$

∎

Note that $P \upharpoonright (R \sqcap S) = (P \upharpoonright R) \sqcap (P \upharpoonright S)$ does not hold in general. To see this, consider the following processes expressed in DI-Algebra:

$P = [\ a? \rightarrow c!; P \ \Box\ b? \rightarrow c!; P\ ]$

$R = a!; c?; R$

$S = b!; c?; S$

In $P \upharpoonright (R \sqcap S)$, the trace a is not a divergence since it is not a divergence of P and is a trace of $R \sqcap S$. But in process $(P \upharpoonright S)$, the trace a is a divergence and so is also one of $(P \upharpoonright R) \sqcap (P \upharpoonright S)$.

(P3) Restriction weakens a process.

$P \restriction Q \sqsubseteq P$

Proof: We need to show that $F[\![P \restriction Q]\!] \supseteq F[\![P]\!]$.

$$
\begin{array}{ll}
 & F[\![P \restriction Q]\!] \\
= & \quad \{ \text{R1} \} \\
 & F[\![P]\!] \cup ED \\
\supseteq & \\
 & F[\![P]\!]
\end{array}
$$

∎

(P4) Restriction is monotonic in its first argument.

If $P \sqsubseteq P'$ then $P \restriction Q \sqsubseteq P' \restriction Q$

Proof:

Given that $P \sqsubseteq P'$, we have

$(F[\![P]\!] \supseteq F[\![P']\!]) \wedge (T[\![P]\!] \supseteq T[\![P']\!])$ (i)

We need to show that $F[\![P \restriction Q]\!] \supseteq F[\![P' \restriction Q]\!]$.

$$
\begin{array}{ll}
 & F[\![P \restriction Q]\!] \\
= & \quad \{ \text{R1} \} \\
 & F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
\supseteq & \quad \{ i \} \\
 & F[\![P']\!] \cup \{s,t : s \in T[\![P']\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= & \quad \{ \text{R1} \} \\
 & F[\![P' \restriction Q]\!]
\end{array}
$$

∎

(P5) Restriction is anti-monotonic in its second argument.

If $Q \sqsubseteq Q'$ then $P \upharpoonright Q' \sqsubseteq P \upharpoonright Q$

Proof:

Given that $Q \sqsubseteq Q'$, we have $T[\![Q]\!] \supseteq T[\![Q']\!]$ (i)

We need to show that $F[\![P \upharpoonright Q]\!] \subseteq F[\![P \upharpoonright Q']\!]$.

$$
\begin{array}{ll}
& F[\![P \upharpoonright Q]\!] \\
= & \quad \{ \text{R1} \} \\
& F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
\subseteq & \quad \{ i \} \\
& F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q']\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= & \quad \{ \text{R1} \} \\
& F[\![P \upharpoonright Q']\!]
\end{array}
$$

∎

(P6) Only the traces of the environment are important as far as restriction is concerned.

If $T[\![Q]\!] = T[\![Q']\!]$, then $P \upharpoonright Q = P \upharpoonright Q'$

Proof:

$$
\begin{array}{ll}
& F[\![P \upharpoonright Q]\!] \\
= & \quad \{ \text{R1} \} \\
& F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
= & \quad \{ T[\![Q]\!] = T[\![Q']\!] \} \\
& F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q']\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\}
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{R1} \} \\
&\quad F[\![P \upharpoonright Q']\!]
\end{aligned}
$$

∎

(P7) Restriction is idempotent.
$(P \upharpoonright Q) \upharpoonright Q = P \upharpoonright Q$

Proof:

$$
\begin{aligned}
&\quad F[\![(P \upharpoonright Q) \upharpoonright Q]\!] \\
&= \quad \{ \text{R1} \} \\
&\quad F[\![(P \upharpoonright Q)]\!] \cup \{s,t : s \in T[\![(P \upharpoonright Q)]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
&= \quad \{ \text{R1, R2} \} \\
&\quad F[\![P]\!] \cup \\
&\quad \{s,t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \cup \\
&\quad \{s,t : s \in (T[\![P]\!] \setminus T[\![Q]\!]) \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
&= \quad \{ \text{set theory} \} \\
&\quad F[\![P]\!] \cup \{s,t : s \in T[\![P]\!] \setminus T[\![Q]\!] \wedge t \in (\mathcal{A} \cup \mathcal{B})^* : st\} \\
&= \quad \{ \text{R1} \} \\
&\quad F[\![P \upharpoonright Q]\!]
\end{aligned}
$$

∎

(P8) If a process refines another process restricted to some environment, then this process restricted to the same environment still refines the other process restricted to that environment. This condition is necessary and sufficient.
$P \upharpoonright Q \sqsubseteq P' \upharpoonright Q$ iff $P \upharpoonright Q \sqsubseteq P'$

Proof:
(onlyif) We assume that $P \upharpoonright Q \sqsubseteq P' \upharpoonright Q$.

$$
\begin{aligned}
& P \upharpoonright Q \\
\sqsubseteq \quad & \{ \text{ Assumption } \} \\
& P' \upharpoonright Q \\
\sqsubseteq \quad & \{ \text{ P3 } \} \\
& P'
\end{aligned}
$$

(if) We assume that $P \upharpoonright Q \sqsubseteq P'$.

$$
\begin{aligned}
& P \upharpoonright Q \\
= \quad & \{ \text{ P7 } \} \\
& (P \upharpoonright Q) \upharpoonright Q \\
\sqsubseteq \quad & \{ \text{ Assumption and P4 } \} \\
& P' \upharpoonright Q
\end{aligned}
$$

∎

3.5.4 Safety and progress requirement

As seen above, the restriction operator helps to obtain the effective behaviour of a process in a given environment. If the environment is divergent then the process can behave as per the specification, since a divergent environment makes no restrictions. The definition is correct to achieve the goal in theory. But if one considers the synthesis of such a process in a divergent environment, the synthesis tools are unable to get a circuit due to the divergent environment. The Petri net generated by di2pn for a divergent process is unbounded and hence Petrify is unable to synthesise a circuit. Therefore, for a synthesisable specification, the closed system (comprising of the process P and its environment Q) must be "safe", i.e. P and Q must not be initially divergent; and in the course of execution P must not make Q divergent and vice versa. Formally, this can be stated by the following

safety property of the closed system:

$$\varepsilon \notin D[\![P]\!] \cup D[\![Q]\!] \wedge D[\![P]\!] \cap (T[\![Q]\!] \setminus D[\![Q]\!]) = \emptyset \wedge D[\![Q]\!] \cap (T[\![P]\!] \setminus D[\![P]\!]) = \emptyset$$

Apart from safety, it is desirable that the system satisfies progress requirement, i.e. it is free from deadlock. For achieving this, any quiescent trace of P must not be a quiescent trace of Q and vice versa. Formally,

$$(F[\![P]\!] \setminus D[\![P]\!]) \cap (F[\![Q]\!] \setminus D[\![Q]\!]) = \emptyset$$

3.6 Representing Alternation Using the Restriction Operator

In this section it is shown how to represent an alternation $\langle S, T \rangle P$ using the restriction operator. This is possible provided that P cannot initially output on a signal in T. (First the definition of *out*(P), those outputs initially possible from P, and *div*(P), whether or not P is divergent, are given below.)

3.6.1 Auxiliary definitions

The initial possibility of divergence

1. *div*$(\bot) = true$
2. *div*$(P \sqcap Q) =$ *div*$(P) \vee$ *div*(Q)
3. A guarded choice, $P = [\ (c : c \in Y : c! \rightarrow Q_c) \ \Box \ (a : a \in X : a? \rightarrow Q_a)\ ]$, diverges if there is a guarded process that is not awaiting input and the output guard can interfere with an output of the following process or that process itself diverges.
   div$(P) = (\exists c : c \in Y : (c \cap$ *out*$(Q_c) \neq \emptyset) \vee$ *div*$(Q_c))$

The initial set of possible outputs

An output transition might initially be observed by the environment of process P for any signal in $out(P)$.

1. For a divergent process, all observations are considered possible.
   $out(\bot) = \mathcal{B}$
2. $out(P \sqcap Q) = out(P) \cup out(Q)$
3. For a guarded choice, $P = [\ (c : c \in Y : c! \to Q_c) \ \Box\ (a : a \in X : a? \to Q_a)\ ]$, outputs can only arise from a guarded process that is not awaiting input.
   $$out(P) = \begin{cases} \mathcal{B} & \text{, if } (\exists c : c \in Y : c \in out(Q_c)) \\ Y \cup (\bigcup c : c \in Y : out(Q_c)) & \text{, otherwise} \end{cases}$$

3.6.2 Representing alternation

For any process P, with input alphabet $\mathcal{A}$ and output alphabet $\mathcal{B}$, such that $out(P) \cap T = \emptyset$, the environment $\overline{\langle S, T\rangle}_{ys} P$ is defined as follows, where $ys \subseteq \mathcal{A}$ is a set of unsafe inputs from the environment.
Let $ys' = \{b : b \in \mathcal{A} \setminus (S \cup T) \wedge out(P/b?) \cap T \neq \emptyset : b\} \cup ys$. Then,

$$
\begin{aligned}
&\overline{\langle S, T\rangle}_{ys} P = \\
&[\quad (a : a \in out(P) \wedge a \in S : a? \to \overline{\langle T, S\rangle}_{\emptyset}(P/a!)) && (1)\\
&\Box\ (a : a \in out(P) \wedge a \notin S : a? \to \overline{\langle S, T\rangle}_{\emptyset}(P/a!)) && (2)\\
&\Box\ (a : a \in \mathcal{A} \wedge a \in S \wedge out(P/a?) \cap S = \emptyset \wedge \\
&\qquad \neg div(P/a?) \wedge a \notin ys : a! \to \overline{\langle T, S\rangle}_{ys'}(P/a?)) && (3)\\
&\Box\ (a : a \in \mathcal{A} \wedge a \notin S \wedge out(P/a?) \cap T = \emptyset \wedge \\
&\qquad \neg div(P/a?) \wedge a \notin ys : a! \to \overline{\langle S, T\rangle}_{ys}(P/a?)) && (4)\\
&]
\end{aligned}
$$

The claim is: $\langle S, T\rangle P = P \upharpoonright (\overline{\langle S, T\rangle}_{\emptyset} P)$.

Clause (1) and (2) state that the environment absorbs any output produced by the process. To keep track of alternation sequence, if this output is from S

then sequence of $\langle S, T\rangle$ is swapped to $\langle T, S\rangle$ (as in (1)). According to (3) the environment provides an input to the process provided this input is safe (i.e. not in ys), does not make P divergent or violate the condition *out*$(P) \cap T = \emptyset$. If this input belongs to S then the alternation sequence is swapped and all possible inputs considered unsafe are added to the set ys for future reference. All other possible inputs are generated by the environment as in (4), provided they satisfy the same safety conditions of clause (3).

Example:

Below we consider the process used by Mallon to illustrate alternation. Here the environment that can be used for restricting this process to implement alternation is illustrated.

Let $\mathcal{A} = \{a, c\}$, and $\mathcal{B} = \{b\}$ and $P = c?\ ;\ b!\ ;\ stop$.

To analyse the application of alternation $\langle\{a?\}, \{b!\}\rangle$, first we must compute the appropriate environment R for process P using the above definition. Note that *out*$(P) = \emptyset$ and

$$P/a? = [\ c? \rightarrow b!\ ;\ a?\ ;\ \bot \ \Box\ a? \rightarrow \bot\ ]$$

The environment (R) is obtained as follows:

R

$=$ { Initially none of the inputs are unsafe. Therefore $ys = \emptyset$. }

$\overline{\langle\{a\}, \{b\}\rangle}_{\emptyset} P$

$=$ { *out*$(P/a?) = \emptyset$ and *out*$(P/c?) = \{b\}$. Here only clause (3) can be applied. }

$[\ a! \rightarrow\ \overline{\langle\{b\}, \{a\}\rangle}_{\{c\}}(P/a?)\ ]$

$=$ { *div*$(P/a?/a?)$ and $c \in \{c\}$. None of the clauses apply. }

$[\ a! \rightarrow\ stop\ ]$

Now

$D[\![P]\!] = (cb + \varepsilon)(aa + cc + aca + cac + acc + caa)(a + b + c)^* + (acb + cab)(a + cc +$

$ca)(a+b+c)^*$

$T[\![P]\!] = (\varepsilon+a+c+ac+ca+cb+acb+cab+cbc+acbc+cabc+cba+cbac+cbca) \cup D[\![P]\!]$

$F[\![P]\!] = (\varepsilon + a + cb + cab + acb + cba + cbc + cbac + cbca + acbc + cabc) \cup D[\![P]\!]$

$D[\![R]\!] = bab(a+b+c)^* + (\varepsilon + a)bb(a+b+c)^*$

$T[\![R]\!] = (\varepsilon + b + a + ba + ab) \cup D[\![R]\!]$

We can calculate that

$F[\![P \upharpoonright R]\!] = (a+\varepsilon)\ c\ (a+b+c)^* \cup F[\![P]\!]$

This can be expressed in DI-Algebra by the following process.

$P' = [\ c? \rightarrow \perp\ ]$ ♦

3.7 Elimination of Mallon's alternation operator

An alternation restricts a process in so far as what is considered to be a safe sequence of events, thus weakening it. This can be viewed as an extra component in the system which monitors the protocol, i.e., another process runs in parallel with the original one. Applying algebraic laws of parallel composition to this combination, one can eliminate alternation to obtain a description of the weakened process.

Consider $\langle S, T\rangle P$ where P has input alphabet $\mathcal{A}$ and output alphabet $\mathcal{B}$. Define $Q^{S,T}$ with input alphabet $(\mathcal{A} \cap (S \cup T)) \cup \{a : a \in \mathcal{B} \cap (S \cup T) : a'\}$ and output alphabet $(\mathcal{B} \cap (S \cup T)) \cup \{a : a \in \mathcal{A} \cap (S \cup T) : a'\}$, as follows:

$$
\begin{aligned}
Q^{S,T} = \ & [\ (a : a \in \mathcal{A} \cap S : a? \rightarrow a'!\ ;\ Q^{T,S}) \\
& \Box\ (b : b \in \mathcal{B} \cap S : b'? \rightarrow b!\ ;\ Q^{T,S}) \\
& \Box\ (a : a \in \mathcal{A} \cap T : a? \rightarrow \perp) \\
& \Box\ (b : b \in \mathcal{B} \cap T : b'? \rightarrow \perp) \\
&]
\end{aligned}
$$

The signals in S and T of P are renamed by their primed versions before composing with $Q^{S,T}$. Thus the process $Q^{S,T}$ keeps track of the alternating events, relaying them between P and the environment.

Example:

Consider the example used by Mallon to illustrate alternation. The process P with input alphabet $\{a, c\}$ and output alphabet $\{b\}$, is described by

$P = c?\ ;\ b!\ ;\ stop$

and we must apply alternation $\langle\{a\},\{b\}\rangle$ to it. First the signals in the alternation sets are renamed to obtain

$P' = c?\ ;\ b'!\ ;\ stop$

Using the earlier definition, one obtains

$$\begin{aligned} Q^{\{a\},\{b\}} = &\ [\ a? \rightarrow a'!\ ;\ Q^{\{b\},\{a\}} \\ &\ \Box\ b'? \rightarrow \bot\ ] \\ Q^{\{b\},\{a\}} = &\ [\ b'? \rightarrow b!\ ;\ Q^{\{a\},\{b\}} \\ &\ \Box\ a? \rightarrow \bot\ ] \end{aligned}$$

The effect of alternation is then given by the parallel composition of P' with $Q^{\{a\},\{b\}}$ shown in Figure 3.7. Eliminating parallel composition as follows, gives a result that is consistent with that of Mallon.

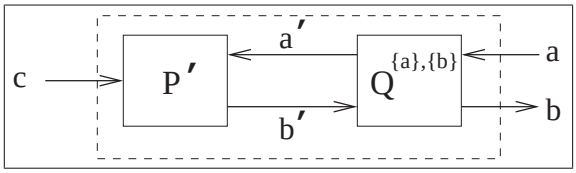


Figure 3.7: $\langle\{a\},\{b\}\rangle P$ re-expressed as a parallel composition

$$\begin{aligned} &\langle\{a\},\{b\}\rangle P \\ &= P' \ ||\ Q^{\{a\},\{b\}} \\ &= [\ c? \rightarrow (stop\ ||\ (Q^{\{a\},\{b\}}/b'?)) \\ &\quad \Box\ a? \rightarrow ((P'/a'?)\ ||Q^{\{b\},\{a\}})\ ] \\ &= [\ c? \rightarrow (stop\ ||\ \bot) \\ &\quad \Box\ a? \rightarrow ([\ c? \rightarrow b'!; a'?; \bot \Box\ a'? \rightarrow \bot\ ]\ ||Q^{\{b\},\{a\}})\ ] \\ &= [\ c? \rightarrow \bot \\ &\quad \Box\ a? \rightarrow [\ a? \rightarrow \bot \Box\ c? \rightarrow \ldots\ ]\ ] \\ &= c?\ ;\ \bot \end{aligned}$$

♦

3.8 Conclusion

DI-Algebra has been extended with a restriction operator and its semantics have been explored. This operator allows one to obtain the effective behaviour of a process in its environment. As the environment might act in a restrictive way, this resultant behaviour is useful to obtain the exact specification and also cheaper implementations.

A way of constructing a suitable environment for a process in order to mimic alternation was proposed. This result shows that the restriction operator subsumes alternation in that it is more general than the latter. Also, a way of eliminating alternation was given.

The concept of restricting a process to its environment has been implemented in two tools, di2pn and diana. Thus one can automatically synthesise cheaper circuits in restrictive environments using di2pn and Petrify, and also verify refinement steps using diana.

Chapter 4

Verification of Terminating DI Processes

4.1 Introduction

The earlier chapters considered the verification of processes expressed in DI-Algebra. As noted earlier, DI-Algebra has no notion of successful termination. However, sequential composition is a convenient operator in constructing specifications and so the concept of termination must be addressed. This is done by the language of DISP. Verification of finite processes expressed in this language is the topic of this chapter.

One of the ways to perform this task is by converting expressions into a canonical form and using syntactic comparison to establish equivalence. In the context of Communicating Sequential Processes (CSP) [Hoa85], algebraic reduction has been investigated as a means of performing this conversion [Ros98]. Moreover, in stepwise refinement, it is required to substitute one process for another. The two processes need not be equivalent. Rather, it has been pointed out [Hoa85] that refinement can be formulated in terms of equivalence. That is, P is refined by Q if and only if P `or` $Q = P$, where "`or`" means non-deterministic choice between P

and Q.

Where a process is to engage in buffered communication, only certain events (inputs from the environment to a buffer and outputs from a buffer to the environment) are observable, whereas others (inputs from a buffer to the process and outputs from the process to a buffer) are not. It is therefore more appropriate to check the equivalence of such processes in composition with their buffers, rather than in isolation [HJH90]. In the special cases of dataflow [Kah74] and delay-insensitivity [Udd86], it is possible to stipulate that the observable behaviour of a process is invariant under buffering, since infinite buffers in series and wires in series form an infinite buffer and a wire, respectively. In these cases, conversion to canonical form can be viewed as taking a description of a process in terms of internal events and replacing it with a description in terms of observable events, Figure 4.1.

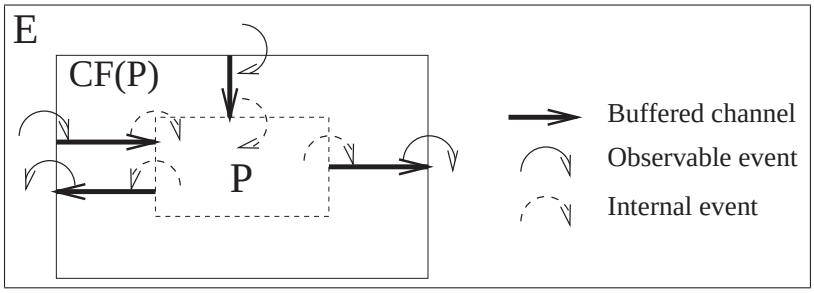


Figure 4.1: The canonical form $CF(P)$ describes a process that synchronises with its environment E. This is an abstraction of process P that communicates through buffered channels

This chapter first defines the terms normal form and canonical form in section 4.2. It then defines the abstract syntax of the subset of the DISP language required for this work in section 4.3, followed by the operator elimination laws in section 4.4. Section 4.5 introduces certain semantic functions and these are key to the conversion procedure in section 4.6. Section 4.7 discusses the implementation of the algebraic reduction and conversion procedures within a term-rewriting system. It is possible to translate from DI-Algebra descriptions to DISP. The method to perform this translation is given in section 4.8.

4.2 Algebra

One of the methods to understand a process algebraic language is to develop a set of algebraic laws which the operators satisfy. An algebraic law is the statement that two expressions, involving some operators and identities representing arbitrary processes are equal [Ros98]. By 'equal', we mean that the two sides are essentially the same: for DISP this means that their communicating behaviours are indistinguishable by the environment. Such laws provide a useful way of gaining understanding and intuition about the intended meaning (semantics) of the language constructs, and are also useful in checking process equivalence.

The aim of this work is to show equivalence of processes. This is based on the following definitions.

- **Normal Form:** An expression is said to be reduced to its normal form by applying a set of rules/laws when no more reductions are possible.
- **Canonical Form:** A canonical form is an unique representation of a process expression, i.e., only one representation exists in canonical form for each semantically equivalent process.

Equivalence can thus be established by a syntactic comparison of the canonical form of the processes.

This chapter is restricted to finite DISP processes and first postulates a set of algebraic laws. These allow certain operators, including the after-operator, sequential composition and parallel composition, to be eliminated by algebraic reduction. This procedure leaves a process in normal form. The laws in fact provide inductive definitions for these operators on processes in normal form, and so termination and convergence of this reduction system is guaranteed. Since the normal form is not sufficient to identify two equivalent terms, a conversion procedure from normal form to canonical form is given. The transformations have been automated using the term-rewriting system *Maude* (http://maude.cs.uiuc.edu/).

Processes in DISP being delay-insensitive, their canonical form must take into account wire delays and the possibility of transmission interference [Udd86, Sne85]. For example, the input/output burst $a, b/c$ is a process that waits for input transitions on a and b before generating an output transition on c and terminating. It transpires that the canonical form $CF(a, b/c)$ of the process is

$$
\begin{array}{lll}
[\ a/\emptyset \rightarrow & [\ a/\emptyset \rightarrow & \texttt{error} \\
 & \Box\ b/\emptyset \rightarrow & [\ \emptyset/c \rightarrow \texttt{pushback}\ \emptyset \\
 & & \Box\ a/\emptyset \rightarrow \texttt{error} \\
 & & \Box\ b/\emptyset \rightarrow \texttt{error}\ ]\ ] \\
\Box\ b/\emptyset \rightarrow & [\ b/\emptyset \rightarrow & \texttt{error} \\
 & \Box\ a/\emptyset \rightarrow & [\ \emptyset/c \rightarrow \texttt{pushback}\ \emptyset \\
 & & \Box\ a/\emptyset \rightarrow \texttt{error} \\
 & & \Box\ b/\emptyset \rightarrow \texttt{error}\ ]\ ]\ ]
\end{array}
$$

where all possible orderings of input and output events have been made explicit.

A canonical form has previously been given to DI-Algebra for finite processes [GJLU93] and for recursively-defined processes [LPU97].

4.3 DISP

4.3.1 Syntax

The abstract syntax used here to describe a finite process in DISP is as follows:

$$
\begin{array}{lll}
proc & ::= & \texttt{stop} \mid \texttt{skip} \mid \texttt{error} \\
 & & \mid \texttt{pushback}\ siglist \mid ioburst \\
 & & \mid [\ \textit{[}choice\textit{]}\ ] \mid proc\ \texttt{after}\ siglist \\
 & & \mid proc\ ;\ proc \mid proc\ \texttt{or}\ proc \\
 & & \mid proc \parallel proc \\
choice & ::= & ioburst \rightarrow proc\ \textit{[}\Box\ choice\textit{]} \\
ioburst & ::= & siglist/siglist
\end{array}
$$

Note 1: A process is associated with an input alphabet $\mathcal{A}$ and an output alphabet $\mathcal{B}$, which are disjoint sets of signals. A *siglist* can include signals from $\mathcal{A}$ or from $\mathcal{B}$, but not from both. The signals in a *siglist* must be distinct and their order is unimportant. Thus, a *siglist* represents a set of signals, though it is convenient to omit braces.

Note 2: Square brackets are used to delimit a process formed from a finite choice between processes guarded by *iobursts*. The guarded processes are separated by $\Box$ and their order is unimportant. Choice can also be expressed as $(i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i)$, where xs, ys and P are indexed families of n input bursts, output bursts and processes, respectively. (These bursts could be empty.)

4.3.2 Delay-insensitivity

For an input/output burst xs/ys to be executed, transitions on all of the input signals in xs must have been previously supplied by the environment. Transitions on all of the output signals in ys are then generated and will eventually reach the environment.

Not only is there a delay in the propagation of signal transitions along a wire connecting a circuit to its environment, but one must avoid transmission interference. Transmission interference is modelled by the process `error` in DISP. Not only does `error` ; $P =$ `error`, but also `error` $||$ $P =$ `error`.

4.3.3 Successful termination

What distinguishes DISP from DI-Algebra is the notion of successful termination. Finite DISP processes interact by input and output signal transitions with their environment and then terminate, successfully or otherwise. On the other hand, finite processes in DI-Algebra cannot terminate successfully, and so sequential composition and iteration are meaningless.

Upon (successful) termination, a DISP process may have signals in transit. In particular:

- `pushback` xs leaves transitions on the inputs xs unabsorbed, available for use by a possible continuation. This often provides a convenient way to describe the initialisation of a process [JF02].
- $\emptyset/ys$ leaves transitions on the outputs ys on their way to the environment.

Figure 4.2 shows a successfully terminating process.

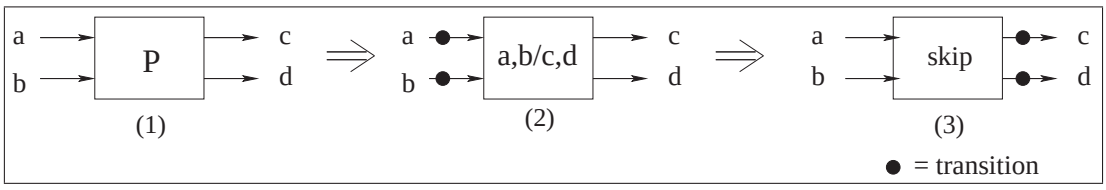


Figure 4.2: Process $P =$ `pushback` a, b ; $a, b/c, d$ as it evolves by engaging in internal events

The process `stop` neither outputs nor successfully terminates. In particular, `stop` ; $P =$ `stop`.

4.3.4 Normal form

Certain operators, such as sequential composition (;) and parallel composition (||) are inessential in DISP. They can be eliminated to leave a process in *normal form*.

Definition A process is in *head normal* form if it is in one of the following forms:

- `error`
- `pushback` xs
- $[\ (i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i)\ ]$

A process is in *normal* form if it is in head normal form and every sub-process is in normal form as well.

4.4 Reduction to Normal Form

Having defined the normal form for processes, algebraic laws need to be defined to reduce each type of process expression to that form. The interesting operators to eliminate are P `after` xs, P ; Q and P || Q. Note that in doing so, P and Q may be assumed to be already in normal form, since one can reduce processes by working outwards from the innermost sub-processes. Before considering these operators, elimination laws for operators that are special cases of more general ones are given. (Some important properties satisfied by the operators are stated and proved in Section 4.6.1.)

4.4.1 Elimination laws

`skip`

(L1) `skip` is a process that does not interact before terminating successfully.

`skip` $=$ `pushback` $\emptyset$

I/O bursts

(L2) An input/output burst can be expressed as a guarded choice consisting of one alternative.

$xs/ys = [\ xs/ys \rightarrow \texttt{pushback}\ \emptyset\ ]$

stop

(L3) `stop` can be expressed as a guarded choice consisting of no alternatives.

$\texttt{stop} = [\ ]$

Non-deterministic choice

(L4) Non-deterministic choice can be expressed as guarded choice using empty guards.

$P\ \texttt{or}\ Q = [\ \emptyset/\emptyset \rightarrow P \ \Box\ \emptyset/\emptyset \rightarrow Q\ ]$

After-input operator

P `after` us is defined (by structural induction on process P in normal form), where us is a set of input signals on which transitions have already been sent to P. (The after-output operator will be considered in section 4.5.3.)

(L5) There is no escape from `error`.

$\texttt{error after}\ us = \texttt{error}$

(L6) The behaviour of a `pushback` process after an input burst depends upon whether or not the signal transitions interfere.

$$(\texttt{pushback}\ xs)\ \texttt{after}\ us = \begin{cases} \texttt{pushback}\ (xs \cup us) & \text{, if } xs \cap us = \emptyset \\ \texttt{error} & \text{, otherwise} \end{cases}$$

(L7) For a guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$, some transitions of signals in us may cancel corresponding input guards, with any remaining

transitions forwarded to the guarded processes. Some extra input guards may have to be added to model the possibility of interference.

$$P \texttt{ after } us = [\ \ (i : 0 \leq i < n : xs'_i/ys_i \rightarrow (P'_i \texttt{ after } xs''_i)) \\ \Box\ (j : 0 \leq j < k \wedge (\forall i : 0 \leq i < n : xs_i \neq \emptyset) : u_j/\emptyset \rightarrow \texttt{error})\ ]$$

where $u_0 \ldots u_{k-1} = us$ and,

for all $i, 0 \leq i < n, xs'_i = xs_i \setminus us$ and $xs''_i = us \setminus xs_i$.

Example:

$$[\ a, b/e \rightarrow [\ a/f \rightarrow \texttt{pushback } c \\ \Box\ c/g \rightarrow \texttt{pushback } a\ ]\ ]\ \texttt{after } a, b, c$$

$$= \{L7\}$$

$$[\ \emptyset/e \rightarrow [\ a/f \rightarrow \texttt{pushback } c \\ \Box\ c/g \rightarrow \texttt{pushback } a\ ]\ \texttt{after } c \\ \Box\ a/\emptyset \rightarrow \texttt{error} \\ \Box\ b/\emptyset \rightarrow \texttt{error} \\ \Box\ c/\emptyset \rightarrow \texttt{error}\ ]$$

$$= \{L7, L6\}$$

$$[\ \emptyset/e \rightarrow [\ a/f \rightarrow \texttt{error} \\ \Box\ \emptyset/g \rightarrow \texttt{pushback } a \\ \Box\ c/\emptyset \rightarrow \texttt{error}\ ] \\ \Box\ a/\emptyset \rightarrow \texttt{error} \\ \Box\ b/\emptyset \rightarrow \texttt{error} \\ \Box\ c/\emptyset \rightarrow \texttt{error}\ ]$$

♦

Sequential composition

$P\ ;\ Q$ is defined (by structural induction on process P in normal form).

(L8) `error` followed by any other process is still `error`, as stated in section 4.3.2. (The reverse is not true in general though.)

$\texttt{error}\ ;\ Q = \texttt{error}$

(L9) Pushing back a set of signal transitions followed by any process is the same as that process after that set.

$\texttt{pushback}\ xs\ ;\ Q = Q\ \texttt{after}\ xs$

(L10) A process that follows a guarded choice distributes into each alternative.

$[(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]\ ;\ Q = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow (P'_i\ ;\ Q))]$

It follows from this law and from (L3) that $\texttt{stop}\ ;\ P = \texttt{stop}$, as stated in section 4.3.3.

Example:

$$
\begin{aligned}
& [\ a/e \rightarrow \texttt{pushback}\ c \\
& \Box\ b/f \rightarrow \texttt{pushback}\ \emptyset\ ]\ ;\ [\ b, c/g \rightarrow \texttt{pushback}\ \emptyset\ ] \\
= \quad & \{L10\} \\
& [\ a/e \rightarrow (\texttt{pushback}\ c\ ;\ [\ b, c/g \rightarrow \texttt{pushback}\ \emptyset\ ]) \\
& \Box\, b/f \rightarrow (\texttt{pushback}\,\emptyset\,;\,[\, b, c/g \rightarrow \texttt{pushback}\,\emptyset\,])\ ] \\
= \quad & \{L9\} \\
& [\ a/e \rightarrow ([\ b, c/g \rightarrow \texttt{pushback}\ \emptyset\ ]\ \texttt{after}\ c) \\
& \Box\ b/f \rightarrow ([\ b, c/g \rightarrow \texttt{pushback}\ \emptyset\ ]\ \texttt{after}\ \emptyset)\ ] \\
= \quad & \{L7, L6\} \\
& [\ a/e \rightarrow [\ b/g \rightarrow \texttt{pushback}\ \emptyset\ \Box\ c/\emptyset \rightarrow \texttt{error}\ ] \\
& \Box\ b/f \rightarrow [\ b, c/g \rightarrow \texttt{pushback}\ \emptyset\ ]\ ]
\end{aligned}
$$

♦

Parallel composition

When two processes P and Q are allowed to execute concurrently, they may communicate with each other, as well as with the environment. To describe the observable behaviour of such a system any inter-process communication is concealed. See, for example, Figure 4.3.

Formally, the input (output) alphabet $\mathcal{A}_2$ ($\mathcal{B}_2$) of the parallel composition of processes P and Q is defined as follows:

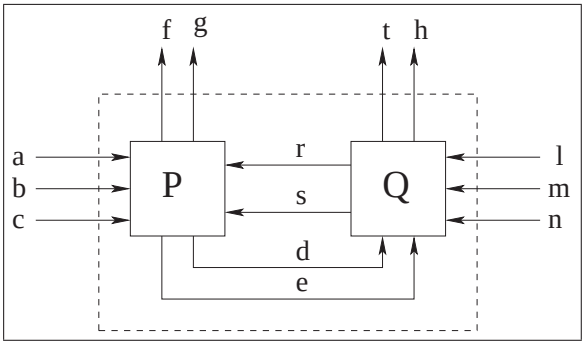


Figure 4.3: $P \mid\mid Q$

Let $\mathcal{A}_0(\mathcal{B}_0)$ be the input (output) alphabet of process P and let $\mathcal{A}_1(\mathcal{B}_1)$ be the input (output) alphabet of process Q. Then $\mathcal{A}_2 = (\mathcal{A}_0 \setminus \mathcal{B}_1) \cup (\mathcal{A}_1 \setminus \mathcal{B}_0)$ and $\mathcal{B}_2 = (\mathcal{B}_1 \setminus \mathcal{A}_0) \cup (\mathcal{B}_0 \setminus \mathcal{A}_1)$. It is required that $\mathcal{A}_0 \cap \mathcal{A}_1 = \emptyset$ and $\mathcal{B}_0 \cap \mathcal{B}_1 = \emptyset$.
$P \mid\mid Q$ is itself defined by the following laws (by structural induction on P and Q in normal form).

(L11) Parallel composition is commutative.
$P \mid\mid Q = Q \mid\mid P$

(L12) Parallel composition is a strict operator, i.e., if one component is `error` the complete system behaves like `error`, as stated in section 4.3.2.
`error` $\mid\mid Q =$ `error`

(L13) `pushbacks` in parallel can be combined. Note that transitions on $(xs_0 \cap B_1) \cup (xs_1 \cap B_0)$ will be invisible to any continuation.
`pushback` $xs_0 \mid\mid$ `pushback` $xs_1 =$ `pushback` $((xs_0 \setminus B_1) \cup (xs_1 \setminus B_0))$

(L14) Let $P =$ `pushback` us and $Q = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow Q'_i)]$. A choice can be made only from those alternatives in Q waiting upon input solely from the environment. For such alternatives we must separate the output signals (ys') to the environment from those (ys'') destined for P. Transitions by the environment on inputs in $(us \setminus B_1)$ will lead to interference.
$P \mid\mid Q =$

$[\ (i, ys', ys'' : 0 \leq i < n \wedge xs_i \cap \mathcal{B}_0 = \emptyset \wedge ys_i = ys' \cup ys'' \wedge ys' \cap \mathcal{A}_0 = \emptyset \wedge ys'' \subseteq \mathcal{A}_0$
$\quad : xs_i/ys' \rightarrow ((P \texttt{ after } ys'') \mid\mid Q_i'))$
$\Box\ (j : 0 \leq j < k \wedge u_j \notin B_1 : u_j/\emptyset \rightarrow \texttt{error})\ ]$
where $u_0 \ldots u_{k-1} = us$.

(L15) Let $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i')]$ and
$Q = [(j : 0 \leq j < m : us_j/vs_j \rightarrow Q_j')]$.
A choice can be made only from those alternatives in P and in Q waiting upon input solely from the environment. For such alternatives, we must separate the output signals to the environment from those destined for the other process.
$P \mid\mid Q =$
$[\ (i, ys', ys'' : 0 \leq i < n \wedge xs_i \cap \mathcal{B}_1 = \emptyset \wedge ys_i = ys' \cup ys'' \wedge ys' \cap \mathcal{A}_1 = \emptyset \wedge ys'' \subseteq \mathcal{A}_1$
$\quad : xs_i/ys' \rightarrow (P_i' \mid\mid (Q \texttt{ after } ys'')))$
$\Box\ (j, vs', vs'' : 0 \leq j < m \wedge us_j \cap \mathcal{B}_0 = \emptyset \wedge vs_j = vs' \cup vs'' \wedge vs' \cap \mathcal{A}_0 = \emptyset \wedge vs'' \subseteq \mathcal{A}_0$
$\quad : us_j/vs' \rightarrow ((P \texttt{ after } vs'') \mid\mid Q_j'))\ ]$

Examples:

Consider once more Figure 4.3, where
$\mathcal{A}_0 = \{a, b, c, r, s\},\ \mathcal{B}_0 = \{d, e, f, g\},$
$\mathcal{A}_1 = \{l, m, n, d, e\},$ and $\mathcal{B}_1 = \{r, s, t, h\}$.
Thus process P communicates with the environment on a, b, c, f, g; process Q communicates with the environment on l, m, n, t, h; P and Q communicate with each other on d, e, r, s. The following are two examples of reduction with respect to these alphabets.

1. $(\texttt{pushback}\ a, b, r) \mid\mid (\texttt{pushback}\ l, d, e)$
$= \{L13\}$
$\texttt{pushback}\ a, b, l$

2. Let $P = [\ c, r/e \rightarrow [\ s/f \rightarrow \texttt{pushback}\ \emptyset\ ]\ ]$ and
$Q = [\ l/r \rightarrow [\ e/s \rightarrow \texttt{pushback}\ \emptyset\ ]\ ]$. Then
$P \parallel Q$
$=$ { L15 }
$[\ l/\emptyset \rightarrow ((P\ \texttt{after}\ r) \parallel [\ e/s \rightarrow \texttt{pushback}\ \emptyset\ ])\ ]$
$=$ { L7, L15 }
$[\ l/\emptyset \rightarrow [\ c/\emptyset \rightarrow ([\ s/f \rightarrow \texttt{pushback}\ \emptyset\ ] \parallel$
$([\ e/s \rightarrow \texttt{pushback}\ \emptyset\ ]\ \texttt{after}\ e))\ ]\ ]$
$=$ { L7, L15 }
$[\ l/\emptyset \rightarrow [\ c/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow$
$((([\ s/f \rightarrow \texttt{pushback}\ \emptyset\ ]\ \texttt{after}\ s) \parallel$
$\texttt{pushback}\ \emptyset)\ ]\ ]\ ]$
$=$ {L7, L14 }
$[\ l/\emptyset \rightarrow [\ c/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow [\ \emptyset/f \rightarrow$
$(\texttt{pushback}\ \emptyset \parallel \texttt{pushback}\ \emptyset)\ ]\ ]\ ]\ ]$
$=$ { L13 }
$[\ l/\emptyset \rightarrow [\ c/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow [\ \emptyset/f \rightarrow \texttt{pushback}\ \emptyset\ ]\ ]\ ]\ ]$
(This is in fact semantically equivalent to $l, c/f$ as they convert into the same canonical form, as shown in section 4.7.2.)

♦

4.5 Semantic Functions

Syntactically distinct processes in normal form may still be semantically equivalent, i.e., indistinguishable to an "observer". Therefore to convert a process in normal form to its canonical representation, certain semantic functions are necessary as described by this section. What is meant by the initial behaviour of a process in terms of the semantic functions *div* and *out* (defined simultaneously), *terminates* and *refuses* is formalised below. (As with the operator elimination laws,

these functions are defined by structural induction on processes in normal form.) Elimination laws for the after-output operator are given using *div* and *out*. Elimination laws for the after-input operator having been given in section 4.4.1, one can therefore determine the behaviour of a process as it evolves by engaging in a trace of observable events.

4.5.1 The initial possibility of divergence

(F1) An `error` process immediately diverges.
$\mathit{div}(\texttt{error}) = true$

(F2) A `pushback` process does not.
$\mathit{div}(\texttt{pushback}\ xs) = false$

(F3) A guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$, diverges if there is a guarded process that is not awaiting input and an output in its burst can interfere with an output of the following process or that process itself diverges.
$\mathit{div}(P) = \big(\exists i : 0 \leq i < n \wedge xs_i = \emptyset : ys_i \cap \mathit{out}(P'_i) \neq \emptyset \vee \mathit{div}(P'_i)\big)$

4.5.2 The initial set of possible outputs

An output transition might initially be observed by the environment of process P for any signal in $\mathit{out}(P)$.

(F4) For a divergent process, all observations are considered possible:
$\mathit{out}(P) = \mathcal{B}$, if $\mathit{div}(P)$
Note that this clause covers both the case of an `error` process and the case of a divergent guarded choice.

(F5) A `pushback` process generates no outputs.
$\mathit{out}(\texttt{pushback}\ xs) = \emptyset$

(F6) For a guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i')]$, outputs can only arise from a guarded process that is not awaiting input.

$out(P) = \left(\bigcup i : 0 \leq i < n \wedge xs_i = \emptyset : ys_i \cup out(P_i')\right)$, if $\neg div(P)$

Examples:

1. According to (F3), (F5) and (F6),
$$div\left(\begin{array}{l}[\ a/e \rightarrow \texttt{pushback}\ \emptyset \\ \Box\ \emptyset/e \rightarrow [\ \emptyset/e, f \rightarrow \texttt{pushback}\ \emptyset\ ] \\ \Box\ \emptyset/g \rightarrow [\ b/f \rightarrow \texttt{pushback}\ \emptyset\ ]\ ]\end{array}\right) = true$$
because of interference on signal e, i.e. $\{e\} \cap \{e, f\} \neq \emptyset$.

2. According to (F2), (F3), (F5) and (F6),
$$out\left(\begin{array}{lll}[\ a/e \rightarrow & \texttt{pushback}\ \emptyset & \\ \Box\ \emptyset/e \rightarrow & [\ \emptyset/f \rightarrow \texttt{pushback}\ \emptyset\ ] & \\ \Box\ \emptyset/g \rightarrow & [\ \emptyset/e \rightarrow \texttt{pushback}\ \emptyset & \\ & \Box\ a/h \rightarrow \texttt{pushback}\ \emptyset\ ]\ ] & \end{array}\right) = \{e, f\} \cup \{g, e\} = \{e, f, g\}$$

♦

4.5.3 Elimination laws for the after-output operator

P `after` v is defined (by structural induction on process P in normal form), where $v \in out(P)$ is an output transition initially observable by the environment.

(L16) A divergent process remains divergent after any $v \in \mathcal{B}$.

P `after` $v =$ `error` , if $div(P)$

(L17) For a guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i')]$, we restrict ourselves to those guarded processes that are not waiting for input and are capable of outputting v.

P `after` $v =$
$[\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge v \in ys_i : \emptyset/(ys_i \setminus \{v\}) \rightarrow P'_i)$
$\Box\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge v \notin ys_i \wedge v \in \mathit{out}(P'_i) : \emptyset/ys_i \rightarrow (P'_i$ `after` $v))\ ]$
, if $\neg \mathit{div}(P)$

Note that output cannot be observed from a `pushback` process.

Example:

$[\ \ a, b/e \rightarrow [\ a/f \rightarrow$ `pushback` $\emptyset\ ]$
$\Box\ \emptyset/e, g \rightarrow [\ \emptyset/f, h \rightarrow$ `pushback` $a\ ]$
$\Box\ \emptyset/e \rightarrow [\ d/f \rightarrow$ `pushback` $\emptyset\ ]\ ]$ `after` f
$=$ { L17 }
$[\ \emptyset/e, g \rightarrow [\ \emptyset/h \rightarrow$ `pushback` $a\ ]\ ]$ ♦

4.5.4 The initial set of possible termination states

Recall from section 4.3.3 that termination leaves a process with a set xs of unabsorbed inputs and a set ys of pending outputs. As a process P may initially terminate in several ways, we need to consider the set $\mathit{terminates}(P)$ of all such pairs (xs, ys). This is defined as follows.

(F7) If a process diverges, then it can leave all possible combinations of input and output signals as termination states. This can be represented by the Cartesian product of the power sets of the input and output alphabets.
$\mathit{terminates}(P) = 2^{\mathcal{A}} \times 2^{\mathcal{B}}$, if $\mathit{div}(P)$

(F8) The process `pushback` xs leaves transitions on the inputs xs unabsorbed and no output transitions pending.
$\mathit{terminates}($`pushback` $xs) = \{(xs, \emptyset)\}$

(F9) A guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ terminates where there is an empty input burst that guards a terminating process.
$\mathit{terminates}(P) =$

$$\{i, us, vs : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us, vs) \in \textit{terminates}(P'_i) : (us, ys_i \cup vs)\}$$
, if $\neg \textit{div}(P)$

Example:

According to (F2), (F3), (F5), (F6) and (F9),

$$\textit{terminates} \left(\begin{array}{l} [\ a/e \rightarrow \texttt{error} \\ \Box\ \emptyset/e \rightarrow [\ \emptyset/f \rightarrow \texttt{pushback}\ \emptyset \\ \qquad\qquad \Box\ \emptyset/h \rightarrow \texttt{pushback}\ b\ ] \\ \Box\ \emptyset/f \rightarrow \texttt{pushback}\ a\ ] \end{array} \right)$$
$$= \{\ (\emptyset, \{e, f\}), (\{b\}, \{e, h\}), (\{a\}, \{f\})\ \}$$ ♦

4.5.5 The initial possibility of refusal

The *refuses* predicate indicates that a process may refuse both to output and to terminate.

(F10) A divergent process may refuse to do anything.

$\textit{refuses}(P) = true$, if $\textit{div}(P)$

(F11) A `pushback` process eventually terminates if left alone.

$\textit{refuses}(\texttt{pushback}\ xs) = false$

(F12) A guarded choice, $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$, refuses to output if all the input bursts are non-empty and may also do so if there is an empty I/O burst guarding a process capable of refusal.

$\textit{refuses}(P) =$

$(\forall i : 0 \leq i < n : xs_i \neq \emptyset) \vee (\exists i : 0 \leq i < n \wedge xs_i = \emptyset \wedge ys_i = \emptyset : \textit{refuses}(P'_i))$
, if $\neg \textit{div}(P)$

Example:

According to (F3) and (F12),

$$\mathit{refuses}\left(\begin{array}{l}[\ a/e \rightarrow \texttt{pushback}\ \emptyset \\ \Box\ b/f \rightarrow \texttt{pushback}\ \emptyset\ ]\end{array}\right) = true$$

♦

Properties of the semantic functions introduced in this section are explored in Section 4.6.1.

4.6 Conversion to Canonical Form

The processes are finite in the sense that each process P must diverge after engaging in any trace of length at least $depth(P)$, defined by

$$depth(P) = \begin{cases} 0 & \text{, if } div(P) \\ 1 + max(z : z \in \mathcal{A} \cup out(P) : depth(P\ \texttt{after}\ z)) & \text{, otherwise} \end{cases} \qquad (DP)$$

Note that $depth(P) > 0$ implies $\neg \mathit{div}(P)$ and
$depth(P\ \texttt{after}\ z) < depth(P)$, $z \in \mathcal{A} \cup \mathit{out}(P)$.

Such processes can now be converted from normal form to canonical form by a recursively-defined procedure. In canonical form, the behaviour of a process is described in a unique way. As it evolves through a trace of events observed by its environment, it is easy to see whether or not it is divergent and, if not, the set of possible termination states, the set of possible outputs, and whether or not the process can refuse both to output and to terminate. A theorem is proved in Section 4.6.1 that gives us confidence in the following definition of canonical form:

$$CF(P) = \begin{cases} \texttt{error} & \text{, if } \mathit{div}(P) \\ [\ (us, vs : (us, vs) \in \mathit{terminates}(P) : \emptyset / vs \rightarrow \texttt{pushback}\ us) & \\ \Box\ (y : y \in \mathit{out}(P) : \emptyset / y \rightarrow CF(P\ \texttt{after}\ y)) & \\ \Box\ (x : x \in \mathcal{A}\ \wedge (\mathit{refuses}(P) \Rightarrow \mathit{terminates}(P) = \emptyset \wedge \mathit{out}(P) = \emptyset) & \\ \qquad : x / \emptyset \rightarrow CF(P\ \texttt{after}\ x)) & \\ \Box\ (: \mathit{refuses}(P) \wedge (\mathit{terminates}(P) \neq \emptyset \vee \mathit{out}(P) \neq \emptyset) & \\ \qquad : \emptyset / \emptyset \rightarrow [\ (x : x \in \mathcal{A} : x / \emptyset \rightarrow CF(P\ \texttt{after}\ x))\ ]) & \\ ] & \text{, otherwise} \end{cases}$$

In particular,

$$\begin{aligned} & CF(\texttt{pushback}\ xs') \\ & = [\ \emptyset/\emptyset \rightarrow \texttt{pushback}\ xs' \\ & \quad \Box\ (x : x \in \mathcal{A} : x/\emptyset \rightarrow CF((\texttt{pushback}\ xs')\ \texttt{after}\ x))\ ] \\ & = [\ \emptyset/\emptyset \rightarrow \texttt{pushback}\ xs' \\ & \quad \Box\ (x : x \in \mathcal{A} \wedge x \in xs' : x/\emptyset \rightarrow \texttt{error}) \\ & \quad \Box\ (x : x \in \mathcal{A} \setminus xs' : x/\emptyset \rightarrow CF(\texttt{pushback}\ x, xs'))\ ] \end{aligned}$$

Example:

The canonical form of process P given by

$$\begin{aligned} P = [\ & \emptyset/\emptyset \rightarrow [\ a/e \rightarrow \texttt{pushback}\ \emptyset\ ] \\ & \Box\ b/f \rightarrow \texttt{pushback}\ \emptyset\ ] \end{aligned}$$

assuming $\mathcal{A} = \{a, b\}$ and $\mathcal{B} = \{e, f\}$ can be computed as follows:

$$\begin{aligned} & CF(P) \\ = [\ & a/\emptyset \rightarrow CF(P\ \texttt{after}\ a) \\ & \Box\ b/\emptyset \rightarrow CF(P\ \texttt{after}\ b)\ ] \end{aligned}$$

$$\begin{aligned} & CF(P\ \texttt{after}\ a) \\ = [\ & \emptyset/e \rightarrow \texttt{pushback}\ \emptyset \\ & \Box\ a/\emptyset \rightarrow \texttt{error} \\ & \Box\ b/\emptyset \rightarrow CF((P\ \texttt{after}\ a)\ \texttt{after}\ b)\ ] \end{aligned}$$

$CF((P \texttt{ after } a) \texttt{ after } b)$
$= [\ \emptyset/e \rightarrow \texttt{pushback } b$
$\quad \Box\ \emptyset/f \rightarrow \texttt{pushback } a$
$\quad \Box\ a/\emptyset \rightarrow \texttt{error}$
$\quad \Box\ b/\emptyset \rightarrow \texttt{error}\ ]$

Performing similar conversions and putting it all together,

$CF(P) =$
$[\ a/\emptyset \rightarrow [\ \ \emptyset/e \rightarrow \texttt{pushback } \emptyset$
$\qquad \Box\ a/\emptyset \rightarrow \texttt{error}$
$\qquad \Box\ b/\emptyset \rightarrow [\ \emptyset/e \rightarrow \texttt{pushback } b$
$\qquad\qquad \Box\ \emptyset/f \rightarrow \texttt{pushback } a$
$\qquad\qquad \Box\ a/\emptyset \rightarrow \texttt{error}$
$\qquad\qquad \Box\ b/\emptyset \rightarrow \texttt{error}\ ]\ ]$
$\Box\ b/\emptyset \rightarrow [\ \emptyset/f \rightarrow \texttt{pushback } \emptyset$
$\qquad \Box\ \emptyset/\emptyset \rightarrow [\ a/\emptyset \rightarrow [\ \ \emptyset/e \rightarrow \texttt{pushback } b$
$\qquad\qquad\qquad \Box\ \emptyset/f \rightarrow \texttt{pushback } a$
$\qquad\qquad\qquad \Box\ a/\emptyset \rightarrow \texttt{error}$
$\qquad\qquad\qquad \Box\ b/\emptyset \rightarrow \texttt{error}\ ]$
$\qquad\qquad \Box\ b/\emptyset \rightarrow \texttt{error}\ ]\ ]\ ]$

♦

The size of a process P in canonical form is $O(k^n)$, where $n = depth(P)$ and $k = |\mathcal{A}| + |\mathcal{B}|$. Informally, this is because P need only represent traces of length at most n and there is a choice of at most k events at each point in a trace. The proof of this result is given in Section 4.6.2.

4.6.1 Properties of semantic functions

The following six lemmas relate the semantic functions of section 4.5 and enable us to prove a theorem about the canonical form of section 4.6.

Lemma 1: $y \in \mathit{out}(P) \Rightarrow \{us, vs : (us, vs) \in \mathit{terminates}(P \texttt{ after } y) : (us, vs \cup \{y\})\} \subseteq \mathit{terminates}(P)$

Proof: Assume $y \in \mathit{out}(P)$ and use structural induction on P in normal form.

Case $\mathit{div}(P)$.

$$
\begin{array}{ll}
& \mathit{terminates}(P \texttt{ after } y) \\
= & \quad \{ \text{L16} \} \\
& \mathit{terminates}(\texttt{error}) \\
= & \quad \{ \text{F1 and F7} \} \\
& 2^{\mathcal{A}} \times 2^{\mathcal{B}} \\
= & \quad \{ \text{F7} \} \\
& \mathit{terminates}(P).
\end{array}
$$

Case $P = \texttt{pushback}\ xs$. This case cannot arise on account of F5.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ and $\neg \mathit{div}(P)$. We calculate

$$
\begin{array}{ll}
& \mathit{terminates}(P \texttt{ after } y) \\
= & \quad \{ \text{L17} \} \\
& \mathit{terminates} \left(\begin{array}{l} [\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \in ys_i : \emptyset/(ys_i \setminus \{y\}) \rightarrow P'_i) \\ \Box\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : \\ \qquad \emptyset/ys_i \rightarrow (P'_i \texttt{ after } y))\ ] \end{array} \right) \\
= & \quad \{ \text{F9} \} \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs') \in \mathit{terminates}(P'_i) \wedge y \in ys_i : (us', (ys_i \setminus \{y\}) \cup vs')\} \cup \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs') \in \mathit{terminates}(P'_i \texttt{ after } y) \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : (us', ys_i \cup vs')\}.
\end{array}
$$

Then

$$
\begin{array}{ll}
& \{us, vs : (us, vs) \in \mathit{terminates}(P \texttt{ after } y) : (us, vs \cup \{y\})\} \\
= & \quad \{ \text{Value of } \mathit{terminates}(P \texttt{ after } y) \text{ given above } \} \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs') \in \mathit{terminates}(P_i') \wedge y \in ys_i : \\
& (us', ys_i \cup vs')\} \cup \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs') \in \mathit{terminates}(P_i' \texttt{ after } y) \wedge y \notin \\
& ys_i \wedge y \in \mathit{out}(P_i') : (us', \{y\} \cup ys_i \cup vs')\} \\
\subseteq & \quad \{ \text{Induction hypothesis on } P_i' \} \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs') \in \mathit{terminates}(P_i') \wedge y \in ys_i : \\
& (us', ys_i \cup vs')\} \cup \\
& \{i, us', vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us', vs' \cup \{y\}) \in \mathit{terminates}(P_i') \wedge y \notin \\
& ys_i \wedge y \in \mathit{out}(P_i') : (us', \{y\} \cup ys_i \cup vs')\} \\
\subseteq & \quad \{ \text{F9} \} \\
& \mathit{terminates}(P).
\end{array}
$$

∎

Lemma 2: $y \in \mathit{out}(P) \Rightarrow \mathit{out}(P \texttt{ after } y) \subseteq \mathit{out}(P)$

Proof: Assume $y \in \mathit{out}(P)$ and use structural induction on P in normal form.

Case $\mathit{div}(P)$.

$$
\begin{array}{ll}
& \mathit{out}(P \texttt{ after } y) \\
= & \quad \{ \text{L16} \} \\
& \mathit{out}(\texttt{error}) \\
= & \quad \{ \text{F1 and F4} \} \\
& \mathcal{B} \\
= & \quad \{ \text{F4} \} \\
& \mathit{out}(P).
\end{array}
$$

Case $P = \texttt{pushback } xs$. This case cannot arise on account of F5.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P_i')]$ and $\neg \mathit{div}(P)$.

$$
\begin{array}{ll}
& \mathit{out}(P \texttt{ after } y) \\
= & \quad \{ \text{L17} \} \\
& \mathit{out}\left(\begin{array}{l} [\ (i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \in ys_i : \emptyset/(ys_i \setminus \{y\}) \rightarrow P'_i) \\ \Box\ (i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : \\ \qquad \emptyset/ys_i \rightarrow (P'_i \texttt{ after } y))\ ] \end{array} \right) \\
= & \quad \{ \text{F6} \} \\
& (\bigcup i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \in ys_i : (ys_i \setminus \{y\}) \cup \mathit{out}(P'_i)) \\
& \cup\ (\bigcup i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : ys_i \cup \mathit{out}(P'_i \texttt{ after } y)) \\
\subseteq & \quad \{ \text{Induction hypothesis on } P'_i \} \\
& (\bigcup i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \in ys_i : (ys_i \setminus \{y\}) \cup \mathit{out}(P'_i)) \\
& \cup\ (\bigcup i : 0 \le i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : ys_i \cup \mathit{out}(P'_i)) \\
\subseteq & \quad \{ \text{F6} \} \\
& \mathit{out}(P).
\end{array}
$$

∎

Lemma 3: $(us, vs) \in \mathit{terminates}(P) \Rightarrow vs \subseteq \mathit{out}(P)$

Proof: By structural induction on P in normal form.

Case $\mathit{div}(P)$.

$$
\begin{array}{ll}
& (us, vs) \in \mathit{terminates}(P) \\
\equiv & \quad \{ \text{F7} \} \\
& us \in 2^{\mathcal{A}} \wedge vs \in 2^{\mathcal{B}} \\
\Rightarrow & \quad \{ \text{F4} \} \\
& vs \subseteq \mathit{out}(P).
\end{array}
$$

Case $P = \texttt{pushback}\ xs$.

$$
\begin{array}{ll}
& (us, vs) \in \mathit{terminates}(P) \\
\equiv & \quad \{ \text{F8} \} \\
& us = xs \wedge vs = \emptyset
\end{array}
$$

$\Rightarrow$ { F5 }

$vs = \mathit{out}(P)$.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ and $\neg \mathit{div}(P)$.

$(us, vs) \in \mathit{terminates}(P)$

$\equiv$ { F9 }

$(\exists i, vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge (us, vs') \in \mathit{terminates}(P'_i) : vs = ys_i \cup vs')$

$\Rightarrow$ { Induction hypothesis on P'_i }

$(\exists i, vs' : 0 \leq i < n \wedge xs_i = \emptyset \wedge vs' \subseteq \mathit{out}(P'_i) : vs = ys_i \cup vs')$

$\Rightarrow$ { Set theory }

$(\exists i : 0 \leq i < n \wedge xs_i = \emptyset : vs \subseteq ys_i \cup \mathit{out}(P'_i))$

$\Rightarrow$ { F6 }

$vs \subseteq \mathit{out}(P)$.

∎

Lemma 4: $\mathit{out}(P) = \emptyset \wedge \mathit{terminates}(P) = \emptyset \Rightarrow \mathit{refuses}(P)$

Proof: By structural induction on P in normal form.

Case $\mathit{div}(P)$.

$\mathit{terminates}(P) = \emptyset$

$\Rightarrow$ { $(\emptyset, \emptyset) \in \mathit{terminates}(P)$ by F7 }

$false$

$\Rightarrow$ { Propositional calculus }

$\mathit{refuses}(P)$.

Case $P =$ `pushback` xs.

$\mathit{terminates}(P) = \emptyset$

$\Rightarrow$ { $(xs, \emptyset) \in \mathit{terminates}(P)$ by F8 }

$false$

$\Rightarrow$ { Propositional calculus }

$\mathit{refuses}(P)$.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ and $\neg \mathit{div}(P)$.

$\mathit{out}(P) = \emptyset \wedge \mathit{terminates}(P) = \emptyset$

$\Rightarrow$ { F6 and F9 }

$(\forall i : 0 \leq i < n \wedge xs_i = \emptyset : ys_i = \emptyset \wedge \mathit{out}(P'_i) = \emptyset \wedge \mathit{terminates}(P'_i) = \emptyset)$

$\Rightarrow$ { Induction hypothesis }

$(\forall i : 0 \leq i < n \wedge xs_i = \emptyset : ys_i = \emptyset \wedge \mathit{refuses}(P'_i))$

$\Rightarrow$ { F12 }

$\mathit{refuses}(P)$.

∎

Lemma 5: $y \in \mathit{out}(P) \cap \mathit{out}(P \texttt{ after } y) \Rightarrow \mathit{div}(P)$

Proof: By structural induction on P in normal form.

Case $\mathit{div}(P)$. There is nothing to prove.

Case $P = \texttt{pushback } xs$. This case cannot arise on account of F5.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ and $\neg \mathit{div}(P)$. We establish a contradiction, as follows:

$y \in \mathit{out}(P) \cap \mathit{out}(P \texttt{ after } y)$

$\Rightarrow$ { F6 and L17 }

$y \in \big((\bigcup i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \in ys_i : (ys_i \setminus \{y\}) \cup \mathit{out}(P'_i)) \cup$

$(\bigcup i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : ys_i \cup \mathit{out}(P'_i \texttt{ after } y))\big)$

$\equiv$ { Set theory }

$(\exists i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \in ys_i : y \in \mathit{out}(P'_i)) \vee$

$(\exists i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : y \in \mathit{out}(P'_i \texttt{ after } y))$

$\Rightarrow$ { F3 and induction hypothesis on P'_i }

$\mathit{div}(P)$.

∎

Lemma 6: $y \in \mathit{out}(P) \wedge \mathit{div}(P \texttt{ after } y) \Rightarrow \mathit{div}(P)$

Proof: By structural induction on P in normal form.

Case $\mathit{div}(P)$. There is nothing to prove.

Case $P = \texttt{pushback}\ xs$. This case cannot arise on account of F5.

Case $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$ and $\neg\mathit{div}(P)$. We establish a contradiction, as follows:

$$
\begin{array}{ll}
& y \in \mathit{out}(P) \wedge \mathit{div}(P \texttt{ after } y) \\
\equiv & \quad \{ \text{L17} \} \\
& \mathit{div}\left(\begin{array}{l} [\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \in ys_i : \emptyset/(ys_i \setminus \{y\}) \rightarrow P'_i) \\ \Box\ (i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : \\ \qquad \emptyset/ys_i \rightarrow (P'_i \texttt{ after } y))\ ] \end{array} \right) \\
\equiv & \quad \{ \text{F3} \} \\
& (\exists i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \in ys_i : (ys_i \setminus \{y\}) \cap \mathit{out}(P'_i) \neq \emptyset \vee \mathit{div}(P'_i)) \\
& \vee \\
& (\exists i : 0 \leq i < n \wedge xs_i = \emptyset \wedge y \notin ys_i \wedge y \in \mathit{out}(P'_i) : \\
& ys_i \cap \mathit{out}(P'_i \texttt{ after } y) \neq \emptyset \vee \mathit{div}((P'_i \texttt{ after } y))) \\
\Rightarrow & \quad \{ \text{F3 and induction hypothesis on } P'_i \} \\
& \mathit{div}(P).
\end{array}
$$

∎

Theorem 1:

(i) $\mathit{div}(CF(P)) \equiv \mathit{div}(P)$

(ii) $\mathit{terminates}(CF(P)) = \mathit{terminates}(P)$

(iii) $\mathit{out}(CF(P)) = \mathit{out}(P)$

(iv) $\mathit{refuses}(CF(P)) \equiv \mathit{refuses}(P)$

Proof: By induction on $\mathit{depth}(P)$.

Base case: $\mathit{depth}(P) = 0$.

$$
\begin{array}{ll}
& \mathit{depth}(P) = 0
\end{array}
$$

$\Rightarrow$ { DP }

$\mathit{div}(P)$

$\Rightarrow$ { Definition of CF }

$\mathit{div}(P) \wedge CF(P) = \texttt{error}$

$\Rightarrow$ { F1 }

$\mathit{div}(P) \wedge \mathit{div}(CF(P))$

$\Rightarrow$ { F4, F7 and F10 }

$\mathit{div}(P) \equiv \mathit{div}(CF(P)) \wedge \mathit{out}(CF(P)) = \mathit{out}(P) \wedge$
$\mathit{terminates}(CF(P)) = \mathit{terminates}(P) \wedge \mathit{refuses}(CF(P)) \equiv \mathit{refuses}(P)$

Inductive step: $depth(P) > 0$.

$\mathit{div}(CF(P))$

$\equiv$ { Definition of CF and F3 }

$(\exists y : y \in \mathit{out}(P) : y \cap \mathit{out}(CF(P \texttt{ after } y)) \neq \emptyset \vee \mathit{div}(CF(P \texttt{ after } y)))$

$\equiv$ { Induction hypothesis }

$(\exists y : y \in \mathit{out}(P) : y \cap \mathit{out}(P \texttt{ after } y) \neq \emptyset \vee \mathit{div}(P \texttt{ after } y))$

$\Rightarrow$ { Lemma 5 and 6 }

$\mathit{div}(P)$

$\mathit{terminates}(CF(P))$

$=$ { Definition of CF, F8 and F9 }

$\{us, vs : (us, vs) \in \mathit{terminates}(P) : (us, vs)\} \cup$
$\{us', vs', y : y \in \mathit{out}(P) \wedge (us', vs') \in \mathit{terminates}(CF(P \texttt{ after } y)) : (us', vs' \cup \{y\})\}$

$=$ { Induction hypothesis }

$\mathit{terminates}(P) \cup$
$\{us', vs', y : y \in \mathit{out}(P) \wedge (us', vs') \in \mathit{terminates}(P \texttt{ after } y) : (us', vs' \cup \{y\})\}$

$=$ { Using Lemma 1 the second term is subset of *terminates*(P) }

terminates(P)

out$(CF(P))$

$=$ { Definition of CF, F5 and F6 }

$(\bigcup us, vs : (us, vs) \in$ *terminates*$(P) : vs) \cup$

$(\bigcup y : y \in$ *out*$(P) : \{y\} \cup$ *out*$(CF(P \texttt{ after } y)))$

$=$ { Induction hypothesis }

$(\bigcup us, vs : (us, vs) \in$ *terminates*$(P) : vs) \cup$

$(\bigcup y : y \in$ *out*$(P) : \{y\} \cup$ *out*$(P \texttt{ after } y))$

$=$ { Lemma 2 and 3 }

out(P)

refuses$(CF(P))$

$\equiv$ { Definition of CF, F11 and F12 }

$($*out*$(P) = \emptyset \wedge$ *terminates*$(P) = \emptyset) \vee$

$($*refuses*$(P) \wedge ($*out*$(P) \neq \emptyset \vee$ *terminates*$(P) \neq \emptyset))$

$\equiv$ { Propositional calculus }

$($*out*$(P) = \emptyset \wedge$ *terminates*$(P) = \emptyset) \vee$ *refuses*(P)

$\equiv$ { Lemma 4 }

refuses(P)

∎

4.6.2 The size of processes in canonical form

The size of a process in normal form (by structural induction) is defined as follows:

1. $size(\texttt{error}) = 1$
2. $size(\texttt{pushback } xs) = 1 + |xs|$

3. For guarded choice,

   $P = [(i : 0 \leq i < n : xs_i/ys_i \rightarrow P'_i)]$,

   $size(P)$

   $= 1 + (\Sigma\, i : 0 \leq i < n : 1 + |xs_i| + |ys_i| + size(P'_i))$

Note that the size of a process is greater than zero. For example,
$[\ \emptyset/\emptyset \rightarrow [\ ]$
$\Box\ \emptyset/\emptyset \rightarrow \texttt{pushback}\ \emptyset\ ]$
has size 5. This is an upper bound on the size of the canonical form of a process with empty input and output alphabet.

The upper bound on the size of the canonical form for process having the size of input and output alphabet 1 is 11 as illustrated for the process
$P = \texttt{stop or skip or pushback}\ a$.

$CF(P) = [\ \emptyset/\emptyset \rightarrow \texttt{pushback}\ \emptyset$
$\Box\ \emptyset/\emptyset \rightarrow \texttt{pushback}\ a$
$\Box\ \emptyset/\emptyset \rightarrow [\ a/\emptyset \rightarrow \texttt{error}\ ]\ ]$

The size of the canonical form of P for $k = |\mathcal{A}| + |\mathcal{B}| > 1$ is bounded from above as follows:

$$size(CF(P)) \leq 5k^n 2^k - f(k)$$

where $n \geq depth(P)$ and $f(k) = (2 + 2k + 2^k(2 + k/2))/(k - 1)$.

Note that the term 2^k denotes the maximum number of ways a process can terminate after any trace. In practice, however, a process might be capable of terminating in just a few ways, or perhaps not at all.

Proof: By induction on n.
Base Case: $n = 0$

$size(CF(P))$

$=$ { Definition of canonical form }

$size(\texttt{error})$

$=$ { Definition of size }

1

$\leq$ { $k > 1$ }

$5 \times k^0 \times 2^k - f(k)$

Inductive step: $n > 0$.

$size(CF(P))$

$=$ { Definition of canonical form }

$size\big($

$[\ (us, vs : (us, vs) \in \mathit{terminates}(P) : \emptyset/vs \rightarrow \texttt{pushback}\ us)$

$\Box\ (y : y \in \mathit{out}(P) : \emptyset/y \rightarrow CF(P\ \texttt{after}\ y))$

$\Box\ (x : x \in \mathcal{A}\ \wedge (\mathit{refuses}(P) \Rightarrow \mathit{terminates}(P) = \emptyset \wedge \mathit{out}(P) = \emptyset)$

$\quad : x/\emptyset \rightarrow CF(P\ \texttt{after}\ x))$

$\Box\ (: \mathit{refuses}(P) \wedge (\mathit{terminates}(P) \neq \emptyset \vee \mathit{out}(P) \neq \emptyset)$

$\quad : \emptyset/\emptyset \rightarrow [\ (x : x \in \mathcal{A} : x/\emptyset \rightarrow CF(P\ \texttt{after}\ x))\ ])$

$])$

$\leq$ { Applying definition of size of process to this expression }

$1 + (\Sigma\ us, vs : (us, vs) \in \mathit{terminates}(P) : 2 + |vs| + |us|)$

$+ (\Sigma\ y : y \in \mathit{out}(P) : 2 + size(CF(P\ \texttt{after}\ y)))$

$+ 1 + (\Sigma\ x : x \in \mathcal{A} : 2 + size(CF(P\ \texttt{after}\ x)))$

$\leq$ { $|terminates(P)| \leq 2^k$; $|us| + |vs| \leq k/2$; and $|\mathit{out}(P)| \leq |\mathcal{B}|$ }

$2 + 2^k(2 + k/2) + 2|\mathcal{B}| +$

$(\Sigma\ y : y \in \mathit{out}(P) : size(CF(P\ \texttt{after}\ y))) +$

$2|\mathcal{A}| + (\Sigma\ x : x \in \mathcal{A} : size(CF(P\ \texttt{after}\ x)))$

$\leq$ { Induction hypothesis }

$2 + 2k + 2^k(2 + k/2) + |\mathcal{B}|(5k^{n-1}2^k - f(k)) + |\mathcal{A}|(5k^{n-1}2^k - f(k))$

$=$ { Definition of k }

$2 + 2k + 2^k(2 + k/2) + k(5k^{n-1}2^k - f(k))$

$=$ { Definition of $f(k)$ }

$(k-1)f(k) + 5k^n 2^k - kf(k)$

$=$ { Arithmetic }

$5k^n 2^k - f(k).$

∎

Example:

Consider the most nondeterministic, nondivergent process P with $\mathcal{A} = \{a, b\}$ and $\mathcal{B} = \emptyset$, namely,

$P =$ `stop or skip or pushback` a `or pushback` b `or pushback` a, b

Here $k = 2$ and $n = 1$.

An upper bound on the size of its canonical form is

$5k^n 2^k - f(k)$

$= 5 \times 2^1 \times 2^2 - f(2)$

$= 22.$

The normal form of P is given by the following expression of size 21.

$[\ \emptyset/\emptyset \rightarrow [\,]$

$\Box\ \emptyset/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow$ `pushback` $\emptyset$

$\Box\ \emptyset/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow$ `pushback` a

$\Box\ \emptyset/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow$ `pushback` b

$\Box\ \emptyset/\emptyset \rightarrow$ `pushback` $a, b\ ]\ ]\ ]\ ]$

Its canonical form is expressed as follows:

$$
\begin{aligned}
CF(P) = [\ & \emptyset/\emptyset \rightarrow \texttt{pushback}\ \emptyset \\
\Box\ & \emptyset/\emptyset \rightarrow \texttt{pushback}\ a \\
\Box\ & \emptyset/\emptyset \rightarrow \texttt{pushback}\ b \\
\Box\ & \emptyset/\emptyset \rightarrow \texttt{pushback}\ a, b \\
\Box\ & \emptyset/\emptyset \rightarrow [\ a/\emptyset \rightarrow \texttt{error} \\
& \qquad \Box\ b/\emptyset \rightarrow \texttt{error}\ ]\ ]
\end{aligned}
$$

This is also of size 21. ♦

4.7 Automatic Transformation

Equational reasoning is an important component for automated deduction, high-level programming languages, program verification, and artificial intelligence. Reasoning with equations involves deriving consequences and finding values for variables satisfying a given equation. Rewriting is a powerful method to deal with equations. A rewriting systems consists of set of "rewrite-rules" which are used to replace one term by another in the indicated direction. The theory of rewriting centers around the concept of "normal-form", an expression that cannot be rewritten any further [Der93, DJ90].

As the process of normalising a DISP specification consists of a set of laws these can be viewed as a set of rewrite rules and successive application of them will lead to the normal-form. Therefore, a term-rewriting system can be a helpful tool to automate the process of normalisation. The conversion to a canonical form can also be implemented in a rewriting system using additional operators.

To automate the reduction of process expressions to normal form and canonical form, rather than using more general verification tools such as HOL [GM93] or PVS [ORS92], the term-rewriting system *Maude* is chosen, as it suffices to implement the procedures. (An attempt [GHP+95] has previously been made to construct a theory of DI-Algebra in HOL.) Maude nevertheless provides higher-order functions and many-sorted algebra, which are found useful. Rewriting logic has

previously been proposed as a logical and semantic framework [MOM00], following the successful use of the Maude tool to capture the operational semantics for CCS [VMO00]. Other models of concurrency have also been expressed in Maude so as to understand the commonalities and differences between them [Mes96].

4.7.1 Implementation of DISP in the Maude system

Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Algebraic specifications are given in Maude to describe different types (sorts) of data and the operations to be performed on them. The operations can be defined as conditional equations. They are mainly divided into two categories: *constructors* that are used to generate data, and other remaining operators that modify the data.

The DISP elimination laws, semantic functions and conversion procedure are defined as many-sorted equational specifications in Maude. The operational semantics of Maude for such specifications is *equational simplification*, that is, rewriting of terms until no more rules are applicable. The basic language constructs are declared as constructors and the algebraic elimination laws and semantic functions are declared as additional operators in Maude. The complete details of these can be found in Appendix B.

Table 4.1 shows the time taken to generate the canonical form for the loop bodies of descriptions of some standard circuits [Ver03]. The experiments were carried out on a 1.4 GHz, Pentium 4 machine with 256 MB RAM. The size of the canonical form is in all cases orders of magnitude smaller than the claimed theoretical upper bound for the given values of k and n.

The implementation in Maude was also used to prove properties of operators as demonstrated in Section 4.7.3. The laws implemented as rewrite rules are used to reduce the left-hand side and right-hand side of the properties and the results

compared to prove equivalence.

Circuit Name	$\|\mathcal{A}\| + \|\mathcal{B}\|$ (k)	Depth (n)	size of NF	size of CF	Time (ms)	Number of Rewrites
Modulo-3 Counter	3	8	13	31	0	1796
Test-and-Set Element	4	6	13	58	0	2480
2-to-4 Phase Converter	4	9	25	55	0	3639
2-CALL	6	8	34	169	10	11946
Mutual-Exclusion Element	4	7	18	234	10	12002
4-to-2 Phase Converter	4	9	43	189	20	34485
RGD Arbiter	6	6	32	992	50	64016

Note that very few rewrites and negligible time were taken in each case over reduction to normal form.

Table 4.1: Transformation of loop bodies of descriptions of standard circuits

4.7.2 Verification using Maude

Equivalence checking of two processes P and Q is performed by the command

`Maude> red CF(`P`) == CF(`Q`)`

Similarly to check for refinement of P by Q one can use the command

`Maude> red CF(`P `or` Q`) == CF(`P`)`

For example, equivalence of the process

$[\ l/\emptyset \rightarrow [\ c/\emptyset \rightarrow [\ \emptyset/\emptyset \rightarrow [\ \emptyset/f \rightarrow$ `pushback` $\emptyset\ ]\ ]\ ]\ ]$ and $l, c/f$ as claimed in the example of section 4.4.1 can be proved using Maude as follows:

```
Maude>
red CF( select ({'l} / mt) then select ({'c} / mt) then
            select (mt / mt) then select (mt / {'f}) then
                pushback(mt) end end end end
      [{'a,'b,'c,'l,'m,'n},{'f,'g,'t,'h}])
==
    CF( select ({'l,'c} / {'f}) then pushback(mt) end
```

```
      [{'a,'b,'c,'l,'m,'n},{'f,'g,'t,'h}]) .
rewrites: 6373854 in 5470ms cpu (5720ms real) (1165238 rewrites/second)
result Bool: true
```

Note that in the implementation, guarded choice is represented by `[[ ... ]]` and an empty set by the keyword `mt`. Process descriptions are parameterised by input and output alphabets.

Case study

Consider the design of a controller for a micro-pipeline stage, Figure 4.4. The controller is described in three different ways and the refinement relationship amongst these descriptions is investigated.

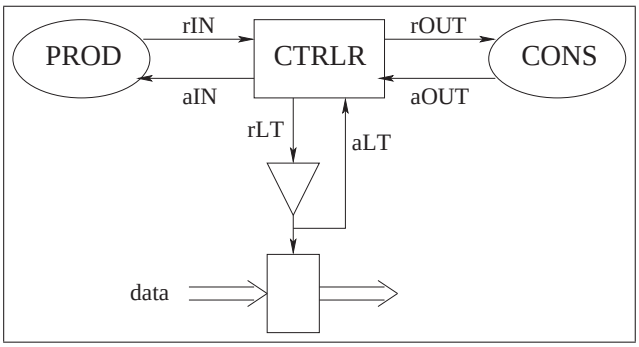


Figure 4.4: Micro-pipeline stage controller

Cyclic behaviour is described in DISP using the `forever` loop construct. The semantics of `forever do` P `end` is defined to be the limit of the finite processes "`error`", "P ; `error`", "P ; P ; `error`", etc. A sufficient condition for `forever do` P `end` to be refined by `forever do` Q `end` is that P is refined by Q.

The following processes, P, Q and R, are three possible descriptions of the controller [JF02]. *Maude* is used to compare their loop bodies, P', Q' and R', respectively, Table 4.2.

$P =$ `pushback` $aOUT$;
`forever do`
$[\ rIN/\emptyset \rightarrow (aOUT/rLT, rOUT\ ;\ aLT/aIN)$
$\Box\ rIN/rLT \rightarrow (aLT/aIN \parallel aOUT/rOUT)\ ]$
`end`

$Q =$ `pushback` $aOUT$;
`forever do`
$[\ rIN, aOUT/rLT, rOUT \rightarrow aLT/aIN\ ]$
`end`

$R =$ `pushback` $aOUT$;
`forever do`
$[\ rIN/rLT \rightarrow (aLT/aIN \parallel aOUT/rOUT)\ ]$
`end`

Property	Maude input	Result	Time Taken (ms)
Q' refines P'	CF(P' or Q') == CF(P')	true	1560
R' refines P'	CF(P' or R') == CF(P')	true	1450
P' behaves like Q' `or` R'	CF(Q' or R') == CF(P')	true	1400
Q' and R' are distinct	CF(Q') == CF(R')	false	610

Table 4.2: Comparison of controller descriptions

A further attempt to synthesise them using *di2pn* and *Petrify* gave the following results.

Process P : Petrify could not solve state coding conflicts

Process Q : Synthesised with area 12 after the addition of 1 state variable

Process R : Synthesised with area 60 after the addition of 3 state variables

Thus, a designer might select the implementation of Q and have a formal proof that Q refines the original description (P).

4.7.3 Properties proved using Maude

The properties given below were proved using the Maude implementation as shown by the sample run of the proof for each property.

Property 1: $(P \texttt{ or } Q) \texttt{ after } us = (P \texttt{ after } us) \texttt{ or } (Q \texttt{ after } us)$, where $us \in \mathcal{A}$ Note that `af` denotes `after` input operator in the implementation.

```
Maude> red (P or Q) af us == (P af us) or (Q af us) .
rewrites: 23 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 2: $[\ \emptyset/\emptyset \rightarrow P\ ] \texttt{ after } us = [\ \emptyset/\emptyset \rightarrow (P \texttt{ after } us)\ ]$

Note that the implementation uses "select ... end" to denote a guarded choice with $\rightarrow$ replaced by `then` and `mt` to denote an empty set.

```
Maude> red
select (mt / mt) then P end af us == select (mt / mt) then (P af us) end .
rewrites: 13 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 3: $(P \texttt{ or } Q)\ ;\ R = (P\ ;\ R) \texttt{ or } (Q\ ;\ R)$

```
Maude> red (P or Q) ; R == (P ; R) or (Q ; R) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 4: *div*$(P \texttt{ or } Q) =$ *div*$(P) \vee$ *div*(Q) Note that the implementation is supplied with the input (a1) and output (b1) alphabet of the processes to compute process divergence.

```
Maude> red (div(P[a1,b1]) or div(Q[a1,b1])) == div((P or Q)[a1,b1]) .
rewrites: 24 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 5: $div([\ \emptyset/\emptyset \rightarrow P\ ]) = div(P)$

```
Maude> red div(select (mt / mt) then P end [a1,b1]) == div(P[a1,b1]) .
rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 6: $out(P \texttt{ or } Q) = out(P) \cup out(Q)$ Note that to compute the intial outputs, the input (a1) and output (b1) alphabets are supplied to the process.

```
Maude> red out((P or Q)[a1,b1]) == out(P[a1,b1]) U out(Q[a1,b1]) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 7: $out([\ \emptyset/\emptyset \rightarrow P\ ]) = out(P)$

```
Maude> red out(select (mt / mt) then P end [a1,b1]) == out(P[a1,b1]) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 8: Property for $(P \texttt{ or } Q) \texttt{ after } v$, where $v \in \mathcal{B}$

$$(P \texttt{ or } Q) \texttt{ after } v = \begin{cases} (P \texttt{ after } v) & \text{,if } v \in out(P) \setminus out(Q) \\ (Q \texttt{ after } v) & \text{,if } v \in out(Q) \setminus out(P) \\ (P \texttt{ after} v) \texttt{ or } (Q \texttt{ after } v) & \text{,otherwise} \end{cases}$$

Note that `afOut` denotes `after` output operator in the implementation and `miracle` denotes a dummy choice which gets eventually removed from the set of choices. This needs to be added as an alternative during implementation. Manual observation and use of Property 7 proves the result.

```
Maude> red (P or Q)[a1,b1] afOut v .
rewrites: 19 in 0ms cpu (0ms real) (~ rewrites/second)
result Process:
select  if v in out (P[a1,b1]) then mt / mt then (P[a1,b1] afOut v)
    else  miracle fi
```

```
  alt
    if v in out (Q[a1,b1]) then mt / mt then (Q[a1,b1] afOut v)
    else  miracle fi
end
```

Property 9: $[\ \emptyset/\emptyset \rightarrow P\ ]$ `after` $v = [\ \emptyset/\emptyset \rightarrow (P$ `after` $v)\ ]$, where $v \in \mathcal{B}$

```
Maude> red select (mt / mt) then P end[a1,b1] afOut v .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
result Process:
select if v subset out (P[a1,b1]) then
     (mt / mt) then ((P[a1,b1]) afOut v)
   else miracle fi
end
```

Property 10: *terminates*$(P$ `or` $Q) =$ *terminates*$(P) \cup$ *terminates*(Q) Note that the implementation uses `term` to denote *terminates*.

```
Maude> red term((P or Q)[a1,b1]) == term(P[a1,b1]) U term(Q[a1,b1]) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 11: *terminates*$([\ \emptyset/\emptyset \rightarrow P\ ]) =$ *terminates*(P)

```
Maude> red term(select (mt / mt) then P end[a1,b1]) == term(P[a1,b1])  .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 12: *refuses*$(P$ `or` $Q) =$ *refuses*$(P) \vee$ *refuses*(Q)

```
Maude> red (refuses(P or Q)) == (refuses(P) or refuses(Q)) .
rewrites: 27 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Property 13: *refuses*($[\ \emptyset/\emptyset \rightarrow P\ ]$) = *refuses*($P$)

```
Maude> red refuses(select (mt / mt) then P end) == refuses(P) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

4.8 Translation from DI-Algebra to DISP

Consider the following abstract syntax for DI-Algebra adapted from Chapter 2 with some syntactic sugar to match with the syntax of DISP.

$$
\begin{array}{lll}
declaration & ::= & id = proc \\
process & ::= & highproc \mid proc \\
highproc & ::= & id\ [\ ||\ id]* \\
proc & ::= & \texttt{stop} \mid \texttt{error} \mid id \mid proc\ \texttt{after}\ sig? \\
 & & \mid [\,[choice]\ ] \mid sig?\ ;\ proc \mid sig!\ ;\ proc \\
 & & \mid proc\ \texttt{or}\ proc \\
choice & ::= & guard \rightarrow proc\ [\Box\ choice] \\
guard & ::= & sig? \mid sig! \mid \texttt{skip}
\end{array}
$$

Note that DI-Algebra does not contain the sequential composition of two processes as there is no concept of successful termination. Input prefixing and output prefixing is represented by $sig?\ ;\ proc$ and $sig!\ ;\ proc$, respectively. Process description can also contain process identifier denoted by id. The after-input operator of DI-Algebra is denoted by $proc$ `after` $sig?$. Guarded choice can have an input guard, an output guard or a skip guard.

Transformation rules

The input and output signal transition of DI-Algebra can be easily translated to DISP ioburst, and the after-input operator as a `pushback` statement:

- $x? \Rightarrow x/\emptyset$
- $x! \Rightarrow \emptyset/x$
- $proc$ `after` $sig? \Rightarrow$ `pushback` sig ; $proc$

DI processes are expressed as finite nonempty sets of equations, each of the shape $X_i = E_i$, where E_i is a process expression ($proc$). To translate such a set of equations the concept of a state machine is used. Whenever an equation refers to another, a state change occurs. The translated process is a single non-terminating process consisting of a set of guarded choices which is a combination of all the equations in the set, with the addition of an extra guard for the state of each process. The initial state is set to the first equation by use of `pushback` statement. Each time the state is changed by a `pushback` on the state variable, i.e. X_i is replaced by `pushback` s_i whenever X_i appears in the process expression. Thus, each equation is associated with a state variable and this is added to each initial guard of the equation to get the guarded choice for the DISP expression.

The set of equations $X_i = E_i$ is translated to the following DISP process. Let X_0 be the initial equation.

$$X'_0 = \texttt{pushback}\ s_0\ ;\ \texttt{forever do}\ [\ (i : 0 \leq i < n : s_i/\emptyset \rightarrow E'_i)\ ]\ \texttt{end}$$

where, E'_i is the translated version of E_i, and there are n such equations.

Example:

This example shows the DI-Algebra specification of a Nacking Arbiter and its translated DISP representation. Nacking Arbiter (N) expressed in DI-Algebra:

$N = \quad [\ r_0? \rightarrow a_0!\ ;\ B_0 \ \Box\ r_1? \rightarrow a_1!\ ;\ B_1\ ]$

$B_0 = \quad [\ r_0? \rightarrow a_0!\ ;\ N\ \Box\ r_1? \rightarrow n_1!\ ;\ B_0\ ]$

$B_1 = \quad [\ r_1? \rightarrow a_1!\ ;\ N\ \Box\ r_0? \rightarrow n_0!\ ;\ B_1\ ]$

Let s_0 represent state variable for process identifier N, s_1 for B_0 and s_2 for B_1. The translated specification (N') in DISP is as follows:

$N' =$ `pushback` s_0 ;

`forever do`

$[\ s_0/\emptyset \rightarrow [\ r_0/\emptyset \rightarrow \emptyset/a_0$; `pushback` s_1

$\Box\ r_1/\emptyset \rightarrow \emptyset/a_1$; `pushback` $s_2\ ]$

$\Box\ s_1/\emptyset \rightarrow [\ r_0/\emptyset \rightarrow \emptyset/a_0$; `pushback` s_0

$\Box\ r_1/\emptyset \rightarrow \emptyset/n_1$; `pushback` $s_1\ ]$

$\Box\ s_2/\emptyset \rightarrow [\ r_1/\emptyset \rightarrow \emptyset/a_1$; `pushback` s_0

$\Box\ r_0/\emptyset \rightarrow \emptyset/n_0$; `pushback` $s_2\ ]\ ]$

`end`

♦

4.9 Conclusion

Sequential processes that communicate delay-insensitively with each other were considered. A communication event can be interpreted as a signal transition.

17 algebraic laws were stated, using which one can systematically eliminate the `skip`, I/O burst, `stop`, non-deterministic choice, after, sequential composition and parallel composition constructs to reduce finite processes to normal form.

The semantic functions, *div*, *out*, *terminates* and *refuses* were defined in 12 clauses. These can be used to calculate the canonical form of finite processes. Certain important properties of the semantic functions and canonical form were also proved. Reduction to normal form and conversion to canonical form have been automated by incorporating the definitions and laws into the Maude term-rewriting tool, which was also used to prove certain properties automatically.

Non-terminating DISP processes can be translated automatically into Petri nets using the tool di2pn [JF02]. Asynchronous logic can then be synthesised using the tool Petrify [CKK+97a]. Given that Petrify employs heuristics, prior algebraic manipulation of loop bodies (checking for equivalence or refinement) can affect the quality of the circuit synthesised.

Chapter 5

Decomposition of DI Processes

5.1 Introduction

This chapter is concerned with digital logic synthesis from processes described in DISP. Petri nets, interpreted as Signal Transition Graphs (STGs) [Chu87], are widely used to specify asynchronous control circuits. The tool Petrify [CKK+97a] inputs such a description and converts it into a state graph (SG) prior to logic synthesis. Construction of Petri nets manually is cumbersome and error prone. More conveniently, the front-end tool di2pn takes a program in the language of DISP and automatically generates a Petri net [JF02].

In order to synthesise a circuit, Petrify requires an SG to have a complete state coding (CSC). That is, no two reachable markings of the net may be encoded by the same signal values (i.e. correspond to the same state) unless the same output signals are excited for each of them [Chu87, Mye01, SF01].

When an SG does not have CSC, the specification needs to be modified. One possibility is to change the dependencies between external signal transitions, e.g. [Man01], which may or may not be acceptable. Another approach is to introduce internal signals. Petrify takes the latter approach to solve CSC. It employs heuristics to insert internal signals (extra state variables) in order that different markings

might correspond to different states. These heuristics are based upon an analysis of the quiescent and excitation regions of the SG [CKK+97b].

In the situation of specifications that have "concurrent outputs" or "self-contained blocks", it is sometimes difficult for Petrify to solve CSC and hence synthesise a circuit. This chapter provides the designer with heuristics that can be applied to decompose such specifications into a form in which Petrify can solve CSC. The language of DISP in which this decomposition is carried out is high level compared to SGs and Petri nets.

The chapter first describes the concrete syntax of the DISP language used for this work in section 5.2 and then shows how concurrent outputs and self-contained blocks lead to CSC conflicts in section 5.3. A set of decomposition heuristics are described in section 5.4, followed by some experimental results obtained by applying the heuristics to benchmark examples in section 5.5. Appendix C provides details of these benchmarks, their decomposition and synthesis.

5.1.1 Related work

Decomposition is considered in [VW02], where the input STG is decomposed into a set of components. This is achieved by partitioning the set of output signals and generating components that produce these outputs. According to [VW02], "their reachability graphs taken together can be much smaller than the original one. Even if this is not achieved, several smaller components might be easier to handle". A similar approach was taken previously in [Chu87].

In [SKC+02] it is shown how to decompose a closed system consisting of a module and its environment such that both of them have delay-insensitive interfaces. As here, the modules themselves are to be implemented as speed independent (SI) circuits. Delay-insensitive interfaces are obtained by removing the dependencies from pairs of directly related input signals. The transformations were performed on STG representations of the modules and on average resulted in an area penalty

of about 36% and performance degradation of about 20%.

Petrify itself automatically applies a set of heuristics to solve CSC conflicts, but sometimes the results are sub-optimal. Alternatively, interactive insertion of state signals can be tried out. A tool has been developed [MBKY03, Mad03] which helps the user to visualise sets of transitions causing conflicts. The tool can also be applied in a fully automated or semi-automated way. The visualisation method is aimed at facilitating a manual refinement of an STG with CSC conflicts, and shows unfolded prefixes [KKY02, KKY03] of the graph. In order to avoid explicit enumeration of encoding conflicts, they are visualised as cores, i.e., sets of transitions causing one or more conflicts. Conflicting pairs of configurations are identified on the complete unfolded prefix for the STG, and the cores are generated from such conflicting pairs. All such cores must eventually be eliminated by adding new signals that resolve the encoding conflicts to yield an STG satisfying the CSC property.

Finally, CSC conflicts can be avoided by performing structural transformations on Petri nets [CCP02]. The encoding is done in such a way that a circuit implementation is guaranteed and the complexity of the method is polynomial on the size of the specification. The transformation is based on the insertion of a signal for each place of the STG that mimics the token flow on that place, and creation of transitions connecting the new signals to the original places. As the insertion of a new signal for each place may be too costly, in terms of size and performance of the resulting circuit, a set of structural transformations is performed on the inserted transitions. These reduce the size of the STG. The results obtained by the method were in many cases identical to those obtained by Petrify, and in other cases similar, with more internal signals than the ones inserted by Petrify.

5.2 DISP Language Syntax

As we have seen in Chapter 4, Delay-Insensitive Sequential Processes (DISP) is a variant of CSP [Hoa85] and DI-Algebra [JU93]. It is a description language which can be used in the synthesis of asynchronous control circuits with the help of the tools di2pn and Petrify. It is more convenient for the designer to describe circuits in DISP (a high-level language), rather than in the graphical notation of Petri nets. The tool di2pn [JF02] automatically translates from the former to the latter. (The algorithm to translate from DISP to Petri nets as described in [JF02] is given for reference in Appendix D.)

The following concrete syntax will be used in this chapter and Appendix C to describe processes:

$$
\begin{array}{lcl}
\mathit{proc} & ::= & \mathit{ioburst} \mid \texttt{select}\ \mathit{choice}\ \texttt{end} \mid \texttt{pushback}\ \mathit{xs} \mid \texttt{stop} \\
 & & \mid \mathit{proc}\ ;\ \mathit{proc} \mid \mathit{proc}\ \texttt{par}\ \mathit{proc} \\
 & & \mid \texttt{forever do}\ \mathit{proc}\ \texttt{end} \\
\mathit{choice} & ::= & \mathit{ioburst}\ [\texttt{then}\ \mathit{proc}]\ [\texttt{alt}\ \mathit{choice}] \\
\mathit{ioburst} & ::= & \mathit{siglist}/\mathit{siglist}
\end{array}
$$

siglist is a list of signal names; these must be distinct and their order is unimportant. The simplest process is an input/output burst ($ioburst$) where transitions of all the signals in the input burst must be absorbed before transitions of all the signals in the output burst are produced.

`select` and `end` delimit a process from a choice between *ioburst*-guarded processes; these are separated by `alt` and their order is unimportant. Choice is restricted to those guarded processes for which all required input transitions are available. For example, the process $P =$ `select` a/b `alt` $-/c$ `end` eventually outputs c and terminates, unless input a arrives, in which case it may non-deterministically decide to output on b or c before terminating.

Processes can be composed in sequence and in parallel, and infinite repetition

is provided by the `forever` construct.

Asynchronous control circuits must be modelled by *non-terminating* processes. This is the smallest class of processes satisfying the following rules: an infinite repetition is non-terminating; the sequential or parallel composition of two processes is non-terminating if either process is non-terminating; a choice is non-terminating if all of its guarded processes are non-terminating.

5.3 CSC Conflicts

To synthesise circuits, Petrify requires an SG to have CSC. If there are conflicting states in the graph, additional (internal) signals are required and their corresponding up-going and down-going transitions are inserted in the SG. This modification is based on the theory of regions [CKK+97b] and involves computing the excitation region for the new event. The new SG is then checked for CSC and the process is repeated until CSC is resolved.

CSC conflicts arise when distinct outputs are required of the circuit from states with the same coding. Two common causes considered in this chapter are when the specification includes

- concurrent outputs and these outputs are absorbed by different components in the environment. In such a situation the environment may respond to these outputs with corresponding input events in an arbitrary order.
- a self-contained block, i.e., one in which every signal is transitioned an even number of times, and the start state needs to be distinguished from the finish state for that block.

5.3.1 Concurrent outputs

Consider a circuit specified by the following DISP process (P) and its corresponding environment (E1).

E1 = forever do -/a,b ; c,d/- end

#environment E1

P = forever do a,b/c,d end

The process P synchronises on a and b (before outputting c and d), whereas E1 synchronises on c and d. The Petri net generated by di2pn for this specification is shown in Figure 5.1. It corresponds to an SG that has no CSC conflicts and Petrify synthesises a speed-independent (SI) circuit with an estimated area of 18 units (the number of literals in the Boolean equations generated by Petrify).

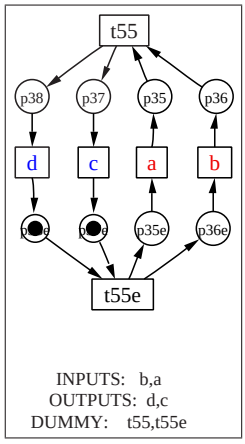


Figure 5.1: Petri net for P in environment $E1$

Consider once more process P, but this time assume its environment has more parallelism, i.e., instead of one process (E1) now two parallel processes (E2 par E3) represent the environment.

E2 = forever do -/a ; c/- end

E3 = forever do -/b ; d/- end

#environment E2 par E3

Figure 5.2 shows the Petri net resulting from this specification. Here the two environment components respond to c and d independently. It is now the case that Petrify needs to add one extra signal to resolve CSC conflicts before it can synthesise a speed-independent (SI) circuit with an estimated area of 11 units.

Thus failure of the environment to synchronise on the concurrent outputs led

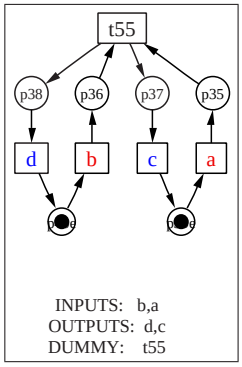


Figure 5.2: Petri net for P in environment $E2$ par $E3$

to CSC conflicts. These are shown as shaded states in Figure 5.3. When in state 1001 (0110) the circuit is confused as to whether to produce output event c+(-) or d-(+). The situation has arisen because the environment handles the concurrent outputs independently: it is free to supply an input as soon as it receives the corresponding output. From state 1100, after d+ has been output, E3 can produce the transition b- before the transition on c+ has been observed.

5.3.2 Self-contained blocks

Consider the specification of a modulo-3 counter described in DISP as follows:

E = forever do -/a; select b/- alt c/- end end

#environment E

M3 = forever do a/b ; a/b ; a/b ; a/b ; a/c ; a/c end

The process M3 performs two return-to-zero (RZ) handshakes involving signals a and b before performing a handshake involving a and c. The environment provides the input on a and is prepared to accept the output either on b or on c.

If we consider the sequence starting with the first output of signal b, we get four self-contained basic blocks (as shown in Figure 5.4):

- Three blocks each formed by the sequence “b a b a”.

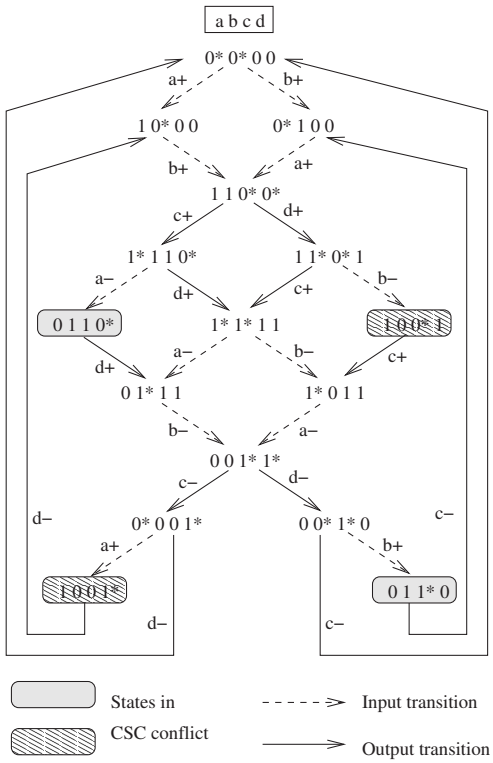


Figure 5.3: State graph for P in environment $E2$ par $E3$

- One block formed by the sequence "c a c a".

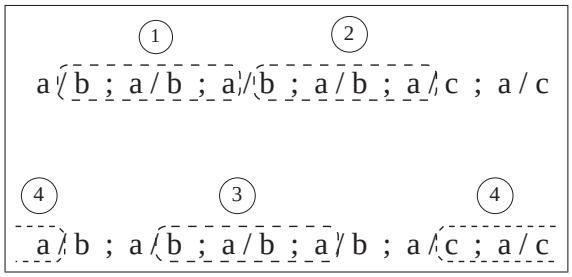
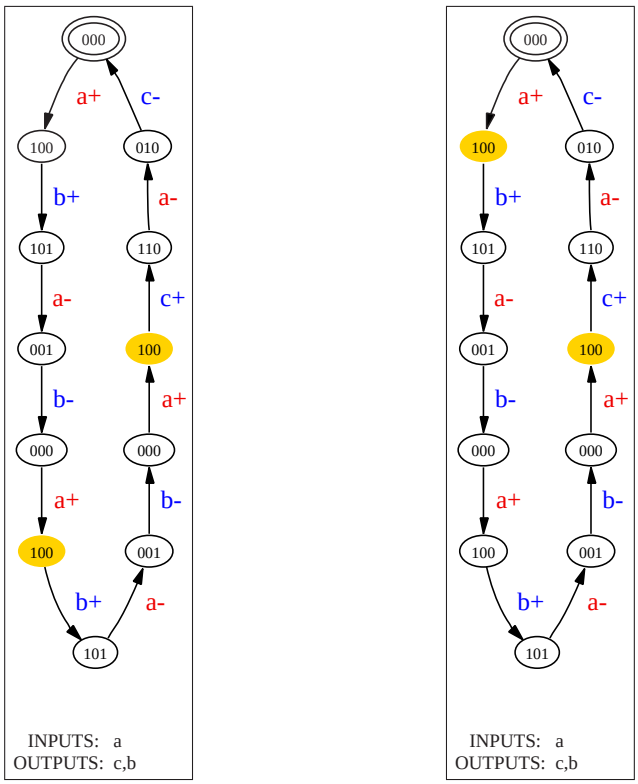


Figure 5.4: Self-contained blocks in mod-3 counter

The start/finish states in each case must be distinguished, even though there is only a CSC conflict in (2) and (4), Figure 5.5. When in state 100 the circuit is confused as to whether to produce output event b+ or c+. The situation has arisen because after the execution of each block the SG returns to the state 100 and there is no way for it to distinguish which self-contained block it is supposed to execute next.



(a) (b)

Figure 5.5: CSC conflicts between start/finish states of (a) block 2 and (b) block 4

To resolve the ambiguity Petrify inserts three state variables thus obtaining a SG that distinguishes the states that output a b+ from those that output a c+ to obtain a circuit with area of 42 units.

5.4 Decomposition Heuristics

This section gives a set of heuristics that either separate out Forks or Wires from the specification to reduce/resolve state coding conflicts. These apply to the output bursts of DISP processes.

5.4.1 Concurrent outputs

(H1) Consider a non-terminating process P and a list $y = y_1, \ldots, y_n$, where $1 < n$, of distinct signal names that always occur together in its output bursts, i.e., if y_i occurs in an output burst of P then so does y_j, $0 < i, j \leq n$. Let x be a fresh signal name, let P' be the process formed by substituting x for each occurrence of y in P, and let *FORK* = `forever do` x/y `end`. Then one can decompose P into P' `par` *FORK*.

(H2) Consider a non-terminating process P and a list $y = y_1, \ldots, y_n$, where $0 < n$, of distinct signal names that always occur together in its output bursts. Let x and z be fresh signal names, let P' be the process formed by substituting x followed by an input/output burst (z/t) for each output burst (y, t) that contains y in P, and let *FORK* = `forever do` $x/y, z$ `end`. Then one can decompose P into P' `par` *FORK*.

Forks provide the cheapest way of generating concurrent outputs. H1 reduces concurrency in the specification by replacing multiple outputs (y) by a single one (x). H2 reduces concurrency by removing one or more outputs (y) from each burst in which y occurs. Note that H2 is often effective for $n = 1$. One can also choose to apply H2 more than once to different signals from the output burst and then select the best possible solution.

Example 1 : Application of H1

This example shows the application of H1 to the process P (of section 5.3.1 with environment E2 par E3) that resolves the CSC conflicts. To decompose this specification we can replace the output-burst c,d by a fresh signal s1 and introduce a FORK component as shown below (E2 and E3 are unchanged):

FORK = forever do s1/c,d end

P1 = forever do a,b/s1 end

Figure 5.6 shows the Petri net for this process. The state graphs for P1 and FORK component are shown in Figure 5.7. Both of the components have no CSC conflicts and synthesis using Petrify gives SI circuits with area 7 and 2, respectively. (There is no need to run Petrify to synthesise a Fork, of course!).

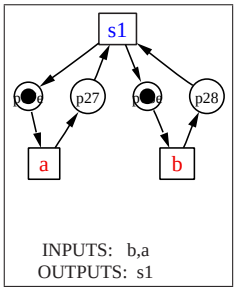


Figure 5.6: Petri net for $P1$

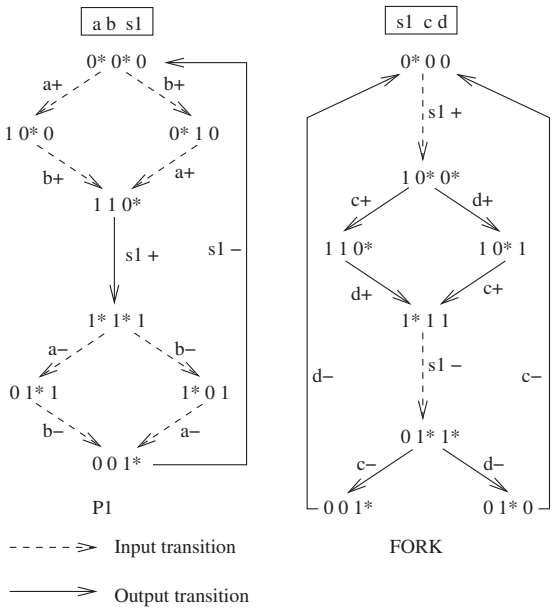


Figure 5.7: State graph for $P1$ and $FORK$

Example 2 : Application of H2

Consider a 2x1 Decision-Wait element with forked outputs. This can be described in DISP as follows:

L = forever do select -/a0 then d0/- alt -/a1 then d1/- end end

R = forever do -/b ; c/- end

#environment L par R

Q = forever do select a0,b/c,d0 alt a1,b/c,d1 end end

In each cycle the process L chooses non-deterministically between a0 and a1. If the process Q receives input on a0 and b, it produces output on d0 and c. Similarly, if it receives input on a1 and b, it produces output on d1 and c. (Figure 5.8 shows the Petri net that is generated by di2pn.) Petrify can synthesise a circuit with area 88 units after adding 2 state variables.

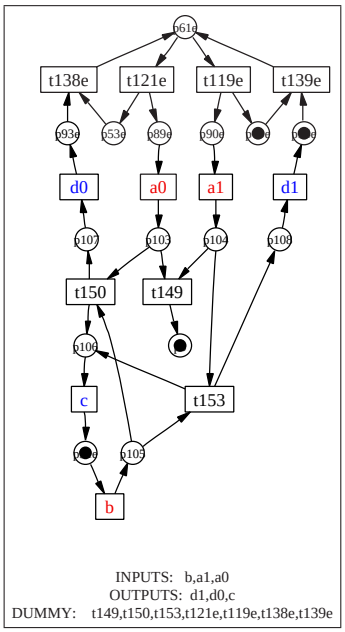


Figure 5.8: Petri net of Q in environment L par R

As can be observed the two output bursts c,d0 and c,d1 do not have the same signals and also the signal c is shared by both of them. Therefore, H1 is not applicable. To apply H2, one might select the signal c to be forked out. Using fresh signal names x and z one can perform the decomposition as shown below:

Q' = forever do

select a0,b/x then z/d0 alt a1,b/x then z/d1 end

end

FORK = forever do x/c,z end

Q' has no CSC conflicts and synthesis using Petrify gives an SI circuit with area 45.

5.4.2 Self-contained blocks

(H3) Consider a non-terminating process P containing m mutually exclusive self-contained basic blocks, B_j, $1 \leq j \leq m$. To introduce n Wires, let x_i and z_i be fresh signal names, and let $1 \leq i \leq n$. Let P' be the process formed by substituting x_{ρ_j} $(1 \leq \rho_j \leq n)$ followed by an input/output burst (z_{ρ_j}/t) for an output burst (t) in the self-contained block B_j, and let $Wire_i =$ `forever do` x_i/z_i `end`. Then one can decompose P into P' `par` $Wire_1$ `par` ... `par` $Wire_n$[1].

In applying H3 one possibility is to use one new wire per block, i.e. $m = n$. Alternatively, Gray codes can be used so that $n = log(m)$ wires are required for decomposition.

Example : Decomposition of modulo-3 counter

Consider once again the specification of a modulo-3 counter given in section 5.3.2.

E = forever do -/a; select b/- alt c/- end end

#environment E

M3 = forever do a/b ; a/b ; a/b ; a/b ; a/c ; a/c end

As seen earlier this specification has 4 self-contained basic blocks (cf. Figure 5.4). Using Gray codes we need $log(4) = 2$ wires to perform the decomposition

[1]Each Wire should interact with at least one block, so far all i, there exists j such that $\rho_j = i$.

as shown below. (Note that Petrify required 3 state variables to solve CSC.)

M3' =

forever do

a/b ; a/x1 ; y1/b ;

a/x2 ; y2/b ; a/x1 ; y1/b ;

a/c ; a/x2 ; y2/c

end

Wire1 = forever do x1/y1 end

Wire2 = forever do x2/y2 end

Synthesis of this using di2pn and Petrify reveals that no state variables need to be added and the circuit synthesised has area 41 units with 2 wires in the environment (i.e. total area of 43 units). In the case of modulo-3 counter, the circuit obtained from the decomposed version has almost the same area as that obtained from the original specification. As the size of the counter grows a significant decrease in area and the number of wires required to be added compared to the number of state variables inserted by Petrify is obtained (cf. Table 5.2).

5.5 Experimental Results

To evaluate the heuristics, they were applied to a number of benchmark examples. The results obtained are discussed below. These experiments were performed using Petrify 4.2 on a 1.4 GHz Pentium 4 machine with 256 MB RAM. The tables show the number of state signals added by Petrify to synthesise the circuit, the estimated area of the circuit for a generalised C element implementation and the time required to generate the solution. They are given for the specifications before and after decomposition.

5.5.1 Concurrent outputs

Heuristic H2 could be successfully applied to a number of benchmark examples [NYD92, YD92, YD95] and obtained results shown in Table 5.1. Synthesis using Petrify succeeded on the decomposed versions, having failed on the original specification. For the original specifications, the -csc[n] option (to solve CSC with blocks of at most n intersecting regions) of Petrify also could not solve CSC conflicts.

Circuit	Original				Decomposed				CSC Ratio	Area Ratio	SR Ratio	Time Ratio
	N i	A I	SR p1	T t1	N ii	A II	SR p2	T t2	ii/i	II/I	p2/p1	t2/t1
scsi-isend	4	103	7	135	2	77	4	70	0.5	0.75	0.57	0.52
scsi-tsend	2	84	3	30	1	65	3	13	0.5	0.77	1	0.43
scsi-trcv	2	90	3	47	1	72	2	17	0.5	0.80	0.67	0.36
pscsi-isend	5	83	5	50	2	59	2	12	0.4	0.71	0.4	0.24
pscsi-tsend	4	92	3	25	3	71	3	11	0.75	0.77	1	0.44
pscsi-ircv	2	37	3	10	1	24	1	2	0.5	0.65	0.33	0.20
pscsi-trcv	×	-	-	-	1	33	2	3	-	-	-	-
isend-fast	×	-	-	-	4	83	2	90	-	-	-	-
sbuf-send	3	72	5	12	1	57	3	6	0.33	0.79	0.60	0.50

× = Petrify could not solve CSC conflicts; N = number of state signals inserted by Petrify;
A = area in literals; SR = total number of set-reset pins; T = time taken in seconds for synthesis

Table 5.1: Experimental results for concurrent outputs

In the case of pscsi-trcv [YD92] and SCSI-fast-initiator-send [YD95], Petrify could not solve CSC on the original specification, but could successfully synthesise the decomposed specification.

The other cases can be summarised by calculating the geometric mean: synthesis for “Decomposed” was on average 2.7-times faster than direct synthesis, saving 25% in area, 52% in state variables inserted by Petrify, and 39% in set-reset pins required.

In all cases the original specifications required 2-5 state signals to be added by Petrify and 3-7 set-reset pins. Their decomposed versions required fewer state signals and at worst the same number of set-reset pins, in all cases decreasing the

area of the circuit.

5.5.2 Self-contained blocks

The table 5.2 shows the decomposition results for specifications with self-contained blocks. Here heuristic H3 was applied in all cases. As Gray codes were applied in each case, fewer wires were required to represent state variables and no new state variables were required to be added by Petrify.

Circuit	Original				Decomposed				CSC Ratio	Area Ratio	SR Ratio	Time Ratio
	N i	A I	SR p1	T t1	N ii	A II	SR p2	T t2	ii/i	II/I	p2/p1	t2/t1
lcounter	×	-	-	-	0	80	2	9	-	-	-	-
dme-fast	2	46	2	3	0	33	2	1.8	0	0.72	1	0.60
mod2	1	23	-	0.15	0	24	2	0.13	0	1.04	-	0.87
mod3	3	42	2	0.6	0	43	3	0.35	0	1.02	1.5	0.58
mod4	5	70	6	2.1	0	55	3	0.8	0	0.79	0.5	0.38
mod5	7	98	8	7.9	0	75	4	0.98	0	0.77	0.5	0.12
mod9	15	202	15	280	0	127	5	3.4	0	0.63	0.33	0.01
seq3	2	22	1	0.66	0	24	1	0.43	0	1.09	1	0.65
seq5	3	61	5	4.38	0	58	4	1.32	0	0.95	0.8	0.30
seq9	5	107	9	63	0	103	7	8.5	0	0.96	0.78	0.13

× = Petrify could not solve CSC conflicts; N = number of state signals inserted by Petrify;
A = area in literals; SR = total number of set-reset pins; T = time taken in seconds for synthesis

Table 5.2: Experimental results for self-contained blocks

An N-bit loadable counter [Jos02] inputs a number m, in response to which it performs m handshakes. The specification of this circuit was written in DISP and a proper environment was modelled for the same. Petrify was unable to solve the CSC conflicts and hence could not synthesise a circuit. After application of the heuristic H3 a synthesisable specification was obtained.

In the case of the DME controller, reduction in area was obtained by inserting one state variable. For modulo counters a significant reduction in the synthesised area and time was observed as the size of the counter grows. Also, the number

of wires used as state variables in the decomposed versions was much lesser than the state variables inserted by Petrify. For the sequencer [JB97] specifications a marginal reduction was obtained in area tough no extra signals were required to be added by Petrify.

In general the decomposed versions can be summarised by calculating the geometric mean: synthesis for "Decomposed" was on average 3.4-times faster than direct synthesis, saving 11% in area, 100% in state variables inserted by Petrify, and 23% in set-reset pins required.

5.6 Conclusion

During the synthesis of asynchronous control circuits Petrify applies a set of heuristics to resolve state coding conflicts. In cases where the specification exhibits concurrent outputs or has self-contained blocks, it is sometimes difficult for Petrify to resolve them. A set of decomposition heuristics are given which can be readily applied to help Petrify to rapidly synthesise area-efficient circuits. Note that performance evaluation of the original and decomposed specifications was not carried out in this work.

The decomposition heuristics introduce Wires and Fork elements that preserve the delay-insensitive behaviour typically required of asynchronous controllers. They offer a practical approach to delay-insensitive decomposition of specifications, where each component can be implemented as a speed-independent circuit.

Chapter 6

Conclusion

6.1 Summary

This thesis has demonstrated that the concept of a delay-insensitive (DI) process is useful in the design and verification of asynchronous control circuits. Such processes can be formally described in general-purpose (CCS) and special-purpose (DI-Algebra and DISP) languages and can be analysed with various existing verification tools (CWB and Maude). Specifications that consist of a single process, as well as ones consisting of pairs of processes (describing a module and its environment) were considered. It was shown that the process abstraction facilitates transformation and decomposition of design descriptions to improve upon direct synthesis on the initial specification. Figure 6.1 shows the various tools and techniques used in this thesis.

The thesis started with modelling DI processes within CCS in order to apply an existing verification tool, the Edinburgh Concurrency Workbench. The modelling of processes in CCS by surrounding them with carefully defined delay-insensitive wires (acting as delay elements) and the applicability of MUST-testing for equivalence and refinement checking was demonstrated. This method was successfully applied to small circuits, but encountered state-explosion problems on larger ex-

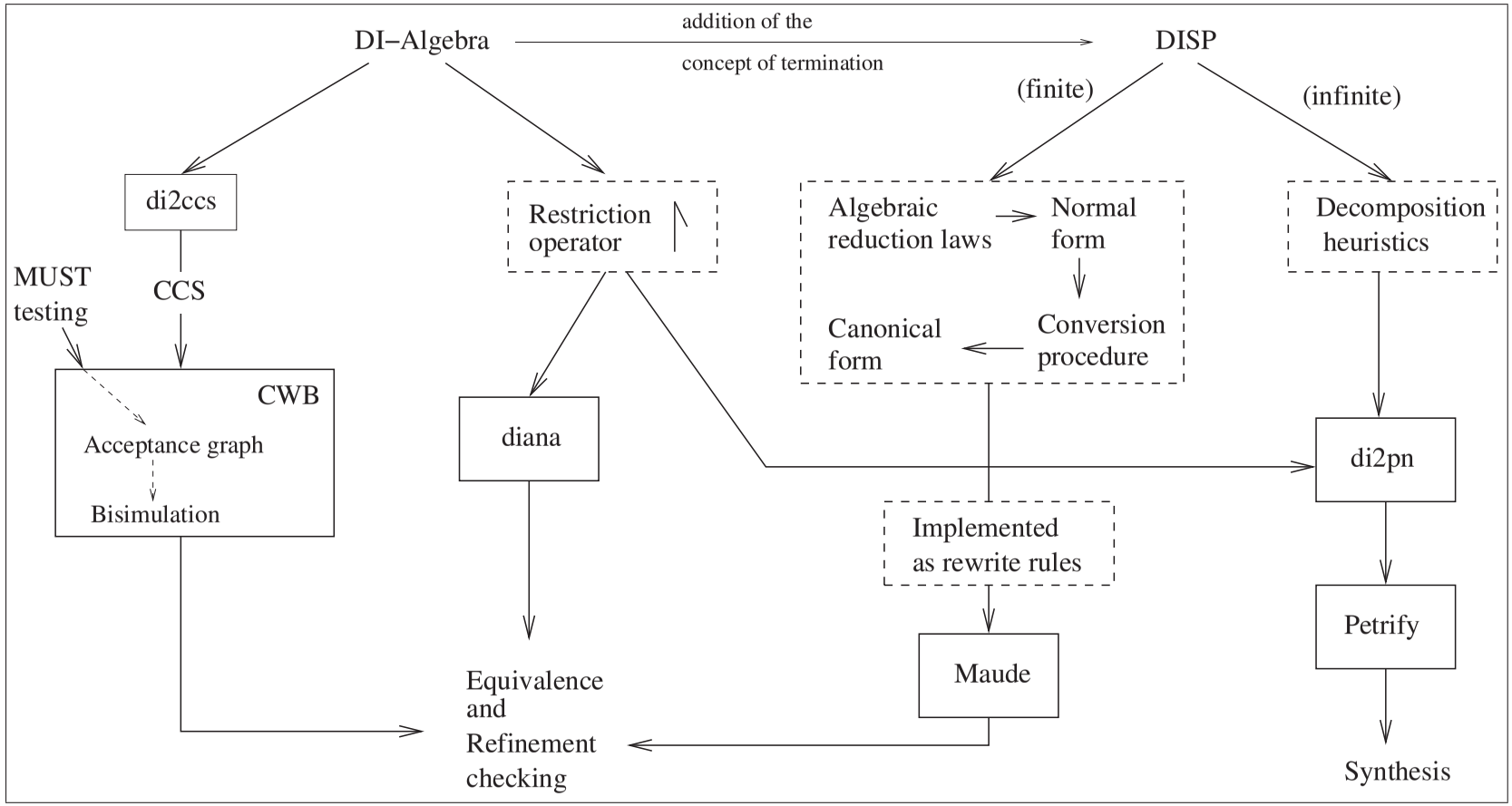


Figure 6.1: Flow graph of main contributions showing various tools and techniques used

amples. The potential of hierarchical verification was also highlighted when such an example was successfully verified in multiple steps.

The next contribution was the proposal of a restriction operator for DI-Algebra, that takes a process and its environment, and gives the effective behaviour of the process running in that environment. This helps to avoid exploring and implementing redundant features in a specification. It opens up new possibilities for refining one process into another, that would not otherwise be allowed, by taking the operating environment into account. Trace-theoretic semantics were given to the restriction operator. The application of the existing tools di2pn with Petrify in synthesising cheaper circuits in restrictive environments, and di2pn with diana for checking refinement-steps was demonstrated. A simple example showed that costly arbitration in circuits can be avoided if it is known that the environment serialises the inputs.

Processes expressed in DI-Algebra cannot terminate successfully. Of course, circuits cannot do so either. Nevertheless sequential composition is a convenient operator in constructing specifications and so the concept of termination must be addressed. To do so, the language DISP was introduced in [JF00]. The contribution of this thesis has been to investigate the semantics of finite processes in this language. Equivalence can be verified by syntactic comparison of their canonical representations. This form gives all possible combinations of inputs and outputs at each step and thus helps the designer to understand the behaviour of a process in the presence of wires. Algebraic laws and a conversion procedure were postulated to achieve this task. These were also implemented using an existing term-rewriting system, Maude. The limitation of finite processes meant that only the bodies of loops could be considered rather than the overall cyclic behaviour of circuits.

Asynchronous control circuits can be synthesised by specifying them using DISP and then using di2pn to obtain Petri nets for them. These nets can then be input into the tool Petrify to obtain speed-independent implementations [JF02]. The

thesis has shown a way to improve on this direct approach. Specifications having concurrent outputs which are absorbed by different environment components, or self-contained blocks that have the same encoding for start and finish states, pose a challenge to synthesis tools like Petrify to resolve state coding conflicts for successful circuit implementations. Simple Wire- and Fork-based decomposition heuristics were provided to help resolve state coding conflicts. The heuristics rely on the delay-insensitivity of a specification (as is the case when it is expressed in DISP) for their correctness. They were evaluated on a total number of 19 benchmark examples. In the cases of pscsi-trcv, scsi-fast-initiator-send and loadable counter, Petrify could not solve CSC on the original specifications, but could successfully synthesise the decomposed specification. In the other cases synthesis was significantly faster and the resulting circuits were smaller. Application of these heuristics is not automated, but they facilitate exploration of the design space.

6.2 Future Research Directions

Experience and results obtained from this thesis have opened up a number of research issues that could be pursued in the future.

6.2.1 Verification by state-exploration

State-space exploration is one of the most successful strategies for checking the correctness of finite state concurrent systems [God96]. Processes are interpreted as labelled transition systems (representing the combined behaviour of all the concurrent components) and proofs are established by automatically searching the resulting spaces. The limitation of this approach is the size of this state space, which often grows exponentially with respect to the size of the descriptions, leading to the *state-space explosion* problem. Among other reasons, this usually happens as all possible interleavings of the concurrent components are considered while con-

structing the state space. For example, the execution of n concurrent events is investigated by exploring $n!$ possible interleavings of these events.

Using partial-order methods

Recently, a collection of verification techniques, referred to as "partial-order methods", have demonstrated that exploring *all* interleavings of concurrent events is not a priori necessary for verification. Indeed, interleavings corresponding to the same concurrent execution contain related information. Some popular references to these methods are [God90, Val88a, Val88b, Val90]. Partial-order methods are now used in several existing verification tools and have been tested on numerous examples of real-protocols [GHP92, GPS96, HP94]. A detailed comparison of the results published in these papers is available in [God96].

As we have seen, the CWB suffers from the state-explosion problem; such partial-order methods can be developed and integrated with it to be able to verify very large size specifications. Furey's diana tool [Fur02] can be regarded as a first prototype verifier for DI processes that takes into account the independence of events that result from delay-insensitivity.

Using acceptance graphs for receptive processes

Another approach could be to modify the definition of MUST-testing in order to handle receptive processes [Jos92] and then building acceptance graphs [CH93] for this.

Using the CWB-NC tool

Verification of iterative DISP processes can be performed by using testing equivalence on the labelled transition systems (LTS). The CWB-NC [CS96] tool supports various process calculi like the CCS, CSP and LOTOS. To integrate a new process calculi in this tool one needs to give an operational semantics to the calculus and

the Process Algebra Compiler (PAC), which is a front-end of the CWB-NC, can then be used to integrate the new process calculus into the CWB-NC. Once this is accomplished, existing verification methods of the tool can then be applied.

Counter-example support

To differentiate two agents described in CCS, the CWB gives a trace of events that distinguishes them using the *dftrace* command. However, this support is provided only under bisimulation equivalence. As the verification of DI processes is based upon the MUST-testing preorder, a trace distinguishing two such agents cannot be obtained using the CWB. Developing the theory and implementation of such a utility could be helpful in finding bugs in the descriptions.

6.2.2 Verification by term-rewriting

A well defined technique for general theorem proving in equational logic is term-rewriting. But there are some difficulties involved in applying existing term-rewriting systems to process algebras. Such systems work only on *finite terms* (i.e., terms with fixed depths). Using these tools, one can only reason about finite processes [Kir93]. However, almost all useful processes are defined recursively. To deal with this, some forms of induction, such as *unique fixpoint induction* and *Scott Induction*, are needed. These induction principles are not supported by the existing term rewriting systems.

The Process Algebra Manipulator (PAM) is a general proof tool for process algebras [Lin94]. It allows users to define their own calculi and perform algebraic style proofs in these calculi by directly manipulating process terms. The logic implemented by PAM is equational logic together with recursion. Equational reasoning is implemented by rewriting, while recursion is dealt with by induction. Proofs are constructed interactively, giving users the freedom to control the proof process.

The contribution of this thesis in comparing canonical forms of DISP processes was limited to finite processes only. One can apply the PAM tool for verification of iterative DISP processes by defining the calculus in PAM and using its proofs techniques. To put it to use, the PAM tool needs support with respect to getting it running under current versions of the necessary compilers.

6.2.3 Deadlock detection

A process is considered to run in an environment that can veto the performance of certain events, as was demonstrated by the restriction operator introduced in this thesis. Moreover, the environment can decide to do so during the execution of processes. If at some moment in the execution, no action in which the process is prepared to engage is allowed by the environment, then deadlock occurs, which is considered to be observable. An additional operator based on the restriction operator can be defined which will help in detecting deadlock of a process with respect to a given environment.

6.2.4 Translation into STGs

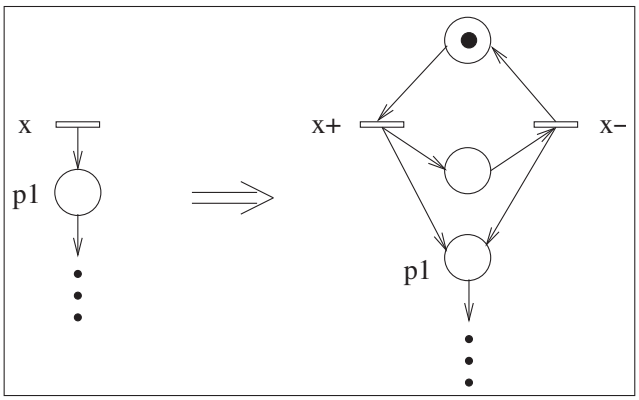


Figure 6.2: Transformation of a Petri net fragment to an STG fragment for an input transition

Petrify converts a Petri net into an Signal Transition Graph (STG) prior to synthesis, so that individual “toggle” transitions are replaced by alternating rising

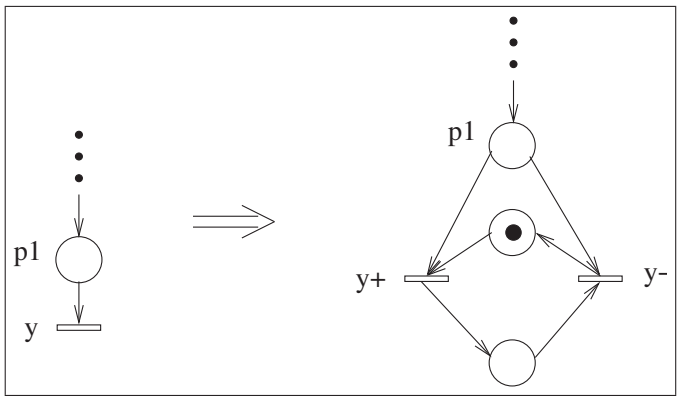


Figure 6.3: Transformation of a Petri net fragment to an STG fragment for an output transition

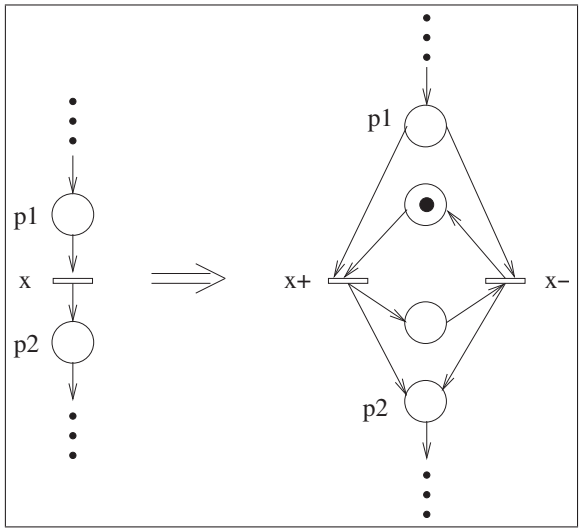


Figure 6.4: Transformation of a Petri net fragment to an STG fragment for an internal transition

and falling transitions. It might be more convenient and efficient if there were a method and tool support to convert DISP program descriptions into such STGs instead of Petri nets. To develop such a tool, one could take the Petri net generated by di2pn and then convert it into an STG. An algorithm to translate from DISP to Petri nets [JF02] is given for reference in Appendix D.

Figures 6.2, 6.3 and 6.4 respectively show the transformations required for converting an input transition, an output transition and an internal transition into an equivalent fragment of the corresponding STG.

6.3 Reflections

6.3.1 Delay-insensitive modules in system design

Asynchronous technology has existed since the beginning of digital electronics, but the straightforward synchronous design style using a global clock has always dominated. Recently, with the advances in CMOS technology and increase in the complexity of designs, the synchronous design approach is becoming problematic due to higher power dissipation and electromagnetic interference. Moreover, electromagnetic emissions are also a threat to the security of devices using them, e.g. smart cards [MAC+02]. Asynchronous design provides an alternative that helps to overcome the above mentioned limitations. However, designers will hesitate to adopt asynchronous approach until the necessary training, tools and testing methodologies are readily available.

Amongst the potential advantages of asynchronous design, the modularity property could be highly attractive to designers. Indeed, as a compromise approach one might use globally asynchronous interconnections of locally synchronous modules (GALS) [Cha84, MTMR02]. The local components being small in size, the clock skew can be kept under control and the various components can use their own clocks tuned to the requirements of their elements. Each component will need to be wrapped up with asynchronous-synchronous interface circuits, resulting in a delay-insensitive module. This style is independent of wire delays between modules which can become dominant in deep submicron CMOS. Verification of DI modules is potentially useful to gain confidence in such systems, whether or not modules implemented are asynchronous circuits.

6.3.2 Specialisation of formal methods to support DI processes

Theory and dedicated tool support [Luc94, MU97, Mal00b, Udd84] has been provided in the past for DI processes. This thesis demonstrated the embedding of

DI processes into mature, widely-used formal methods tools and has shown their applicability using various examples. As DI processes are concurrent in nature, research in verification techniques related to concurrency was found to be applicable. Likewise, research into term-rewriting is relevant to the implementation of the languages DI-Algebra and DISP on account of their algebraic semantics.

Finally, the concept of closed systems is used in compositional model checking and supervisory control of discrete event systems. The restriction operator defined in this thesis can be considered to be related to this concept.

The thesis has thus shown that generic research in formal methods can be tuned to special purpose applications instead of devising new methods.

Appendix A

Translation tool : di2ccs

A.1 Makefile

Compilation of the parser written for DI-Algebra requires JAVACC (Java Compiler Compiler). This can be obtained from the website https://javacc.dev.java.net.

```
# DiParser makefile

JAVAC        = jikes
JAVACC       = /cygdrive/d/javacc-3.0/bin/javacc.bat
PARSER_SRC   = parserSrc
DI_CLASSES   = classes
SAMPLES      = samples
MAIN_CLASS   = di2ccs.DiModule

SRC_FILES    = $(PARSER_SRC)/*.java \
               *.java

JAVACC_SRC   = DiParser.jj
PARSER_FILE  = DiParser.java

all : $(PARSER_SRC)/$(PARSER_FILE)
$(JAVAC) -d $(DI_CLASSES) $(SRC_FILES)

$(PARSER_SRC)/$(PARSER_FILE) : $(JAVACC_SRC)
$(JAVACC) -OUTPUT_DIRECTORY:$(PARSER_SRC) $(JAVACC_SRC)

clean :
rm -f parserSrc/*.java classes/di2ccs/*.class

jar : all
echo "Manifest-Version: 1.0" > classes/manifest
echo "Created-By: Hemangee Kapoor" >> classes/manifest
```

```
echo "Created-By: London South Bank University" >> classes/manifest
echo "Main-Class: di2ccs.DiModule" >> classes/manifest
cd classes; jar cfm di2ccs.jar manifest di2ccs
```

A.2 Class Files

DI Parser source : DiParser.jj

```
options {
  LOOKAHEAD = 1;
  FORCE_LA_CHECK = true;
}

PARSER_BEGIN(DiParser)

package di2ccs;

import java.util.Vector;

public class DiParser
{
  static DiModule module;  // module being parsed,
                           // initialized by parseModule
  static void initParser()
  {
    new DiParser(new java.io.StringReader(""));
  }
}

PARSER_END(DiParser)

SKIP :
{
    " "
  | "\r"
  | "\t"
  | "\n"
  | <"*" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
}

TOKEN : /* OPERATORS */
{
    < SEQ:   ";" >
  | < PAR:   "||" >
```

```
  | < OR:    "ND" >
}

TOKEN :  /* KEY WORDS */
{
    < ERROR:    "CHAOS" >
  | < D_SKIP:   "skip" >
  | < STOP:     "stop" >
  | < AFTER:    "/" >
  | < SELECT:   "[" >
  | < ARROW:    "->" >
  | < HASH:     "#" >
  | < END:      "]" >
}

TOKEN :
{
    < IDEN: <LETTER> (<LETTER> | <DIGIT> )* >
  | < #LETTER: ["A"-"Z","a"-"z","_"] >
  | < #DIGIT:  ["0"-"9"] >
}

void parseModule(DiModule newModule) :  // A module is one file
{
  DiProcess diProc;
  String    procName;
  module = newModule;
}
{
  (
    <IDEN> { procName = token.image.toUpperCase(); }
    "="
    (
      // process can be lowlevel or highlevel
      // when highlevel then it is only parallel composition and
      // nondeterministic choice
      diProc = highLevelProcess()
    | diProc = lowLevelProcess()
    ) { diProc.setName(procName); module.addProcess(diProc); }
  )+ <EOF>
}

DiProcess highLevelProcess() :
{
  DiProcess retProc = new DiProcess();
  PNode s1,s2;
```

```
}
{
  <IDEN> { s1 = new CompositeNode(token.image.toUpperCase()); }
  // high-level process can be composed using parallel composition only
  // non-determnistic choice of highlevel processes is taken care of
  // by lowlevel processes ... therefore here only <PAR> is considered
  ( <PAR> <IDEN>
    {
      s2 = new CompositeNode(token.image.toUpperCase());
      s1 = new ParNode(s1, s2, retProc);
    }
  )+ // user cannot write high level processes like P = Q
 { retProc.setParseTree(s1); return retProc; }
}

DiProcess lowLevelProcess() :
{
  DiProcess retProc = new DiProcess();
  PNode s;
}
{
  readProcessAlphabet(retProc)
  s = statements(retProc)
    { retProc.setParseTree(s); return retProc; }
}

void readProcessAlphabet(DiProcess retProc) :
{
}
{
  "{"
    [
      <IDEN> {retProc.addInput(token.image); }
      (
        "," <IDEN>
        { retProc.addInput(token.image); }
      )*
    ]
  "}"
  ","
  "{"
    [
      <IDEN> {retProc.addOutput(token.image); }
      (
        "," <IDEN>
        { retProc.addOutput(token.image); }
```

```
      )*
    ]
  "}"
}

PNode statements(DiProcess retProc) :
{
  PNode s1, s2;
}
{
  s1 = statement(retProc)
  (
    // parallel composition cannot occur in low-level process
    // due to the restriction of alphabets
    //(oper = <OR> | oper = <PAR>)
    <OR>
    s2 = statement(retProc)
    {
      if (s1 == null)
      {
        s1 = s2;
      }
      else if (s2 != null)
      {
        s1 = new OrNode(s1, s2);
      }
    }
  )* { return s1; }
}

PNode statement(DiProcess diProc) :
{
  PNode s1, s2;
}
{
  s1 = afterStatement(diProc)
  (
    <SEQ> s2 = afterStatement(diProc)
    {
      if (s1 == null) s1 = s2;
      else if (s2 != null) s1 = new SeqNode(s1, s2);
    }
  )* { return s1; }
}

PNode afterStatement(DiProcess diProc) :
```

```
{
  PNode  s1,s2,ret;
}
{
  s1 = basicStatement(diProc) { ret = s1; }
  [ <AFTER> s2 = inBurst(diProc) { ret = new AfterNode(s1,s2); }
  ]
  { return ret; }
}

PNode basicStatement(DiProcess diProc) :
{
  PNode s;
}
{
  (
    s = selectStatement(diProc)
  | LOOKAHEAD(2) s = inBurst(diProc)
  | LOOKAHEAD(2) s = outBurst(diProc)
  | <IDEN> { s = new CompositeNode(token.image.toUpperCase()); }
  | "(" s = statements(diProc) ")"
  | <ERROR> { s = PNode.ErrorNode; }
  | <D_SKIP> { s = PNode.SkipNode; }
  | <STOP> { s = PNode.StopNode; }
  ) { return s; }
}

PNode selectStatement(DiProcess diProc) :
{
  PNode s, choice;
}
{
  <SELECT>
    choice = getChoice(diProc)
    {
      s = new SelectNode();
      ((SelectNode) s).addChoice(choice);
    }
    (
      <HASH> choice = getChoice(diProc)
      {
        ((SelectNode) s).addChoice(choice);
      }
    )*
  <END> { return s; }
}
```

```
PNode inBurst(DiProcess diProc) :
{
  String s;
}
{
  <IDEN> { s = new String(token.image); } "?"
  {
    return new InBurstNode(s);
  }
}

PNode outBurst(DiProcess diProc) :
{
  String s;
}
{
  <IDEN> { s = new String(token.image); } "!"
  {
    return new OutBurstNode(s);
  }
}

PNode getChoice(DiProcess diProc) :
{
  PNode s1, s2, ret;
}
{
  (
    LOOKAHEAD(2) s1 = inBurst(diProc)   { ret = s1; }
  | LOOKAHEAD(2) s1 = outBurst(diProc)  { ret = s1; }
  | <D_SKIP>    { s1 = PNode.SkipNode; ret = s1; }
  )
  [
    <ARROW> s2 = statements(diProc)
    {
      ret = new ChoiceNode(s1, s2);
    }
  ] { return ret; }
}
```

Main module : DiModule.java

```
package di2ccs;
```

```
import java.io.FileReader;
import java.io.FileWriter;
import java.util.Iterator;

class DiModule
{
  public static DiModule currentModule;  // used by ParNode

  public static final  String LOOP_PROC = "LOOP_PROC_";
  java.util.Vector     procs;

  DiModule()
  {
    procs   = new java.util.Vector();   // processes in the module
    currentModule = this;
  }

  void addProcess(DiProcess diProcess)
  {
    procs.addElement(diProcess);
  }

  String getNewName()
  {
    return new String(LOOP_PROC + procs.size());
  }

  DiProcess getProcess(String searchName)
  {
    for(Iterator iter = procs.iterator(); iter.hasNext(); )
    {
     DiProcess p = (DiProcess) iter.next();
      if (p.name.equals(searchName))
      {
        return p;
      }
    }
    throw new java.lang.Error("Undefined process " + searchName);
  }

  public String toString()
  {
    java.lang.StringBuffer s = new java.lang.StringBuffer();

    // First print the definition of a input and output DI-wire
    s.append("agent Di_Win = x.W' + px.W' ; \n");
```

```
    s.append("agent W' = 'y.Di_Win + x.@ + px.@ ; \n");
    s.append("agent Di_W = x.('y.Di_W + x.@) ; \n");

    for(Iterator iter = procs.iterator(); iter.hasNext(); )
    {
      DiProcess p = (DiProcess) iter.next();
      s.append(p.toString());
      s.append("\n\n");
    }

    return s.toString();
  }

  public static void main(String args[]) {
    DiParser.initParser();
    try {
      for(int i = 0; i < args.length; i++) {
        System.out.println("Parsing file ... " + args[i]);

        FileReader fr = new FileReader(args[i]);
        DiParser.ReInit(fr);
        DiModule module = new DiModule();
        DiParser.parseModule(module);
        fr.close();

        FileWriter outFw = new FileWriter(args[i] + "2ccs");
        outFw.write(module.toString());
        outFw.close();
        // System.out.println(module);
      }
    }
    catch(java.lang.Error e) {
      System.out.println("error: " + e);
      System.exit(1);
    }
    catch(ParseException e) {
      System.out.println("error: " + e);
      System.out.println("Exiting.");
      System.exit(1);
    }
    catch(java.io.FileNotFoundException e) {
      System.out.println("error:(File not found) " + e);
      System.out.println("Exiting.");
      System.exit(1);
    }
    catch(java.io.IOException e) {
```

```
      System.out.println("error: " + e);
      System.out.println("Ignored");
    }
  }
}
```

DI process : DiProcess.java

```
package di2ccs;

import java.util.TreeSet;
import java.util.Iterator;

class DiProcess {

  String     name;
  PNode      parseTree;
  TreeSet    inputs;
  TreeSet    outputs;

  DiProcess()
  {
    parseTree = null;
    inputs  = new TreeSet();
    outputs = new TreeSet();
  }

  void setName(String procName)
  {
    name = procName;
  }

  void setParseTree(PNode node)
  {
    parseTree = node;
  }

  void addInput(String instr)
  {
    inputs.add(new String(instr));
  }

  void addOutput(String outstr)
  {
    outputs.add(new String(outstr));
```

```
}

public String toString()
{

  StringBuffer s = new StringBuffer();

  // Add the new name of process which would represent the DI process
  s.append("agent " + name + " = " + parseTree.toString() + ";");
  s.append("\n");
  s.append("agent Di_" + name + " = \n ( \n ");
  addDiWires(s);
  addProcessRenaming(s);
  s.append(" \n )\\{ ");
  printDiProcessRestriction(s);
  s.append(" } ; ");
  return s.toString();
}

void addDiWires(StringBuffer s)
{
  // Create wire strings for inputs and outputs

  for(Iterator iter = inputs.iterator(); iter.hasNext(); )
  {
    String signal = (String) iter.next();
    s.append("Di_Win[" + signal + "/x , p" + signal +"/px,"
              + signal + "i/y] | \n ");
  }

  for(Iterator iter = outputs.iterator(); iter.hasNext(); )
  {
    String signal = (String) iter.next();
    s.append("Di_W[" + signal + "o/x , " + signal + "/y] | \n ");
  }
}

void addProcessRenaming(StringBuffer s)
{
  s.append(name + "[");
  for(Iterator iter = inputs.iterator(); iter.hasNext(); )
  {
    String signal = (String) iter.next();
    if(iter.hasNext()) s.append(signal + "i/" + signal +", ");
    else if(!outputs.isEmpty()) s.append(signal + "i/" + signal + ", ");
    else s.append(signal + "i/" + signal);
```

```
    }

    for(Iterator iter = outputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "o/" + signal + ", ");
      else s.append(signal + "o/" + signal);
    }

    s.append("]");
  }

  void printDiProcessRestriction(StringBuffer s)
  {
    for(Iterator iter = inputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "i, p" + signal + ", ");
      else if(!outputs.isEmpty()) s.append(signal + "i, p" + signal + ", ");
      else s.append(signal + "i, p" + signal );
    }

    for(Iterator iter = outputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "o, ");
      else s.append(signal + "o");
    }
  }
}
```

Type of process : PNode.java

```
package di2ccs;

abstract class PNode
{
  abstract public String toString();
  abstract PNode  getRightmostNode();

  public static final PNode StopNode = new PNode() {
    boolean isStopNode() { return true; }
    PNode getRightmostNode() { return this; }
    public String toString() { return " 0 "; }
  };
```

```
  public static final PNode SkipNode = new PNode() {
    boolean isSkipNode() { return true; }
    PNode   getRightmostNode() { return this; }
    public  String toString() { return ""; }
  };

  public static final PNode ErrorNode = new PNode() {
    boolean isErrorNode() { return true;  }
    PNode   getRightmostNode() { return this; }
    public  String toString() { return " @ "; }
  };

  String getPushbackString() { return "" ;}

  boolean isChoiceNode() { return false; }
  boolean isCompositeNode() { return false; }
  boolean isInBurstNode() { return false; }
  boolean isOutBurstNode() { return false; }
  boolean isLoopNode() { return false; }
  boolean isOrNode() { return false; }
  boolean isParNode() { return false; }
  boolean isAfterNode() { return false; }
  boolean isSelectNode() { return false; }
  boolean isSeqNode() { return false; }
  boolean isSkipNode() { return false; }
  boolean isStopNode() { return false; }
  boolean isErrorNode() { return false; }
}
```

Input Signal : InBurstNode.java

```
package di2ccs;

import java.lang.String;
import java.lang.StringBuffer;

class InBurstNode extends PNode
{
  String inputSignal;

  InBurstNode(String inp)
  {
    inputSignal = inp;
  }
```

```
  boolean isInBurstNode() { return true; }
  PNode   getRightmostNode() { return this; }

  public String toString()
  {
    return inputSignal;
  }

  public String getPushbackString()
  {
    return "'p" + inputSignal ;
  }
}
```

Output Signal : OutBurstNode.java

```
package di2ccs;

import java.lang.String;
import java.lang.StringBuffer;

class OutBurstNode extends PNode
{
  String outputSignal;

  OutBurstNode(String inp)
  {
    outputSignal = inp;
  }

  boolean isOutBurstNode() { return true; }
  PNode   getRightmostNode() { return this; }

  public String toString()
  {
    return "'" + outputSignal;
  }

}
```

Guarded choice : SelectNode.java

```
package di2ccs;
```

```
import java.lang.StringBuffer;
import java.util.Vector;

class SelectNode extends PNode
{
  Vector choices;

  SelectNode()
  {
    choices = new Vector();
  }

  void addChoice(PNode n)
  {
    choices.addElement(n);
  }

  boolean isSelectNode() { return true; }

  PNode getRightmostNode()
  {
    final  PNode lastNode = (PNode) choices.lastElement();
    return lastNode.getRightmostNode();
  }

  public String toString()
  {
    StringBuffer s = new StringBuffer();
    for(java.util.Iterator iter = choices.iterator(); iter.hasNext(); )
    {
      PNode n = (PNode) iter.next();
      // s.append("(");
      String cn = n.toString();

      if (cn.length() == 0) s.append("tau");
      else s.append(n.toString());

      // s.append(")");
      if (iter.hasNext()) s.append(" + ");
    }
    return s.toString();
  }
}
```

Single choice : ChoiceNode.java

```
package di2ccs;

class ChoiceNode extends PNode
{
  PNode lNode;
  PNode rNode;

  ChoiceNode(PNode n1, PNode n2)
  {
    lNode = n1;
    rNode = n2;
  }

  boolean isChoiceNode() { return true; }
  PNode getRightmostNode() { return rNode.getRightmostNode(); }

  public String toString()
  {
    String lStr = lNode.toString();
    String rStr = rNode.toString();

    if (lStr.length() == 0)
      return rStr;
    else if (rStr.length() != 0)
       return lNode.toString() + " . (" + rNode.toString() + ")";

    return lStr;
  }
}
```

After input : AfterNode.java

```
package di2ccs;

class AfterNode extends PNode
{
  PNode lNode;
  PNode rNode;

  AfterNode(PNode n1, PNode n2)
  {
    lNode = n1;
    rNode = n2;
```

```
  }

  boolean isAfterNode() { return true; }
  PNode getRightmostNode() { return rNode.getRightmostNode(); }

  public String toString()
  {
    String lStr = lNode.toString();
    String rStr = rNode.getPushbackString();

    if (lStr.length() == 0 && rStr.length() == 0)
      return "";  // Syntactic transformation => (tau)

    if (lStr.length() == 0)
      return rStr; // return "(" + rStr + ")";

    if (rStr.length() == 0)
      return lStr; // return "(" + lStr + ")";

    return rStr + " . " + lStr;
    // return "(" + rStr + " . " + lStr + ")";
  }
}
```

Process identifier : CompositeNode.java

```
package di2ccs;

class CompositeNode extends PNode   // Name of the process
{
  String id;

  CompositeNode(String n)
  {
    id = new String(n);
  }

  boolean isCompositeNode() { return true; }
  PNode getRightmostNode() { return this; }

  public String toString()
  {
    return id;
  }
}
```

Sequential Composition : SeqNode.java

```
package di2ccs;

class SeqNode extends PNode
{
  PNode lNode;
  PNode rNode;

  SeqNode(PNode n1, PNode n2)
  {
    lNode = n1;
    rNode = n2;
  }

  boolean isSeqNode() { return true; }
  PNode getRightmostNode() { return rNode.getRightmostNode(); }

  public String toString()
  {
    String lStr = lNode.toString();
    String rStr = rNode.toString();

    if (lStr.length() == 0 && rStr.length() == 0)
      return "";  // Syntactic transformation => (tau)

    if (lStr.length() == 0)
      return rStr; // return "(" + rStr + ")";

    if (rStr.length() == 0)
      return lStr; // return "(" + lStr + ")";

    if (rNode.isErrorNode() || rNode.isStopNode() || rNode.isSkipNode()
        || rNode.isInBurstNode() || rNode.isOutBurstNode()
        || rNode.isCompositeNode() || rNode.isLoopNode())
      return lStr + " . " + rStr;
    else
      return lStr + " . (" + rStr + ")";
  }
}
```

Non-deterministic Choice : OrNode.java

```
package di2ccs;
```

```
class OrNode extends PNode
{
  PNode lNode;
  PNode rNode;

  OrNode(PNode n1, PNode n2)
  {
    lNode = n1;
    rNode = n2;
  }

  boolean isOrNode() { return true; }
  PNode   getRightmostNode() { return rNode.getRightmostNode(); }

  public String toString()
  {
    String lStr = lNode.toString();
    String rStr = rNode.toString();

    if (lStr.length() == 0 && rStr.length() == 0)
      return " tau + tau ";
      // return "(tau + tau)";

    if (lStr.length() == 0)
      return " tau + tau . " + rStr;
      // return "(tau + tau . " + rStr + ")";

    if (rStr.length() == 0)
      return " tau . " + lStr + " + tau ";
      // return "(tau . " + lStr + " + tau)";

    return " tau . " + lStr + " + tau . " + rStr;
    // return "(tau . " + lStr + " + tau . " + rStr + ")";
  }
}
```

Parallel composition : ParNode.java

```
package di2ccs;

import java.util.TreeSet;
import java.util.Iterator;

class ParNode extends PNode
{
```

```
PNode      lNode;
PNode      rNode;
DiProcess process;

TreeSet inputs;  // restricted set obtained from lNode and rNode
TreeSet outputs;
TreeSet internals;

ParNode(PNode n1, PNode n2, DiProcess proc)
{
  lNode   = n1;
  rNode   = n2;
  process = proc;

  inputs  = null;
  outputs = null;
  internals = null;
}

boolean isParNode() { return true; }
PNode   getRightmostNode() { return rNode.getRightmostNode(); }

void getIOForPair(PNode n1, PNode n2, TreeSet in, TreeSet out)
{
  getIOForNode(n1, in, out);
  getIOForNode(n2, in, out);
}

void getIOForNode(PNode n, TreeSet in, TreeSet out)
{
  if (n.isParNode())
  {
   ((ParNode) n).getInputOutput();
   in.addAll(((ParNode) n).inputs);
   out.addAll(((ParNode) n).outputs);
  }
  else if (n.isChoiceNode())
    getIOForPair(((ChoiceNode) n).lNode, ((ChoiceNode) n).rNode, in, out);
  else if (n.isSeqNode())
    getIOForPair(((SeqNode) n).lNode, ((SeqNode) n).rNode, in, out);
  else if (n.isOrNode())
    getIOForPair(((OrNode) n).lNode, ((OrNode) n).rNode, in, out);
  else if (n.isSelectNode())
  {
    for(Iterator i = ((SelectNode) n).choices.iterator(); i.hasNext(); )
    {
```

```
      PNode p = (PNode) i.next();
      getIOForNode(p, in, out);
    }
  }
  else if (n.isCompositeNode() || n.isLoopNode())
  {
    DiProcess proc = DiModule.currentModule.getProcess(n.toString());
    in.addAll(proc.inputs);
    out.addAll(proc.outputs);
  }
  else if (n.isInBurstNode() || n.isStopNode()
           || n.isErrorNode() || n.isSkipNode()
           //|| n.isPushNode()
           )
  {
    in.addAll(process.inputs);
    out.addAll(process.outputs);
  }
  else throw new java.lang.Error("ParNode: getInputOutput: unknown node");
}

void getInputOutput()
{
  if (internals != null) return;

  TreeSet in1 = new TreeSet();
  TreeSet out1 = new TreeSet();

  getIOForNode(lNode, in1, out1);

  TreeSet in2 = new TreeSet();
  TreeSet out2 = new TreeSet();

  getIOForNode(rNode, in2, out2);

  inputs = new TreeSet();
  outputs = new TreeSet();

  inputs.addAll(in1);
  inputs.removeAll(out2);
  inputs.addAll(in2);
  inputs.removeAll(out1);
  // set these inputs to the inputs of the process
  process.inputs.addAll(inputs);

  outputs.addAll(out1);
```

```
    outputs.removeAll(in2);
    outputs.addAll(out2);
    outputs.removeAll(in1);
    // set these outputs to the outputs of the process
    process.outputs.addAll(outputs);

    internals = new TreeSet();
    internals.addAll(in1);
    internals.retainAll(out2);

    TreeSet tmp = new TreeSet();
    tmp.addAll(out1);
    tmp.retainAll(in2);

    internals.addAll(tmp);

    /* Added by hemangee -- when we have par composition then the
     * process this belongs to must also have the internal signals
     * removed from its inputs and outputs */

    process.inputs.removeAll(internals);
    process.outputs.removeAll(internals);
  }

  public String toString()
  {
    String lStr = lNode.toString();
    String rStr = rNode.toString();

    // Parallel composition can never have empty left or right nodes
    // it will always be a high-level process name

    getInputOutput();

    StringBuffer s = new StringBuffer();
    s.append("( \n ");
    addInternalDiWires(s);

    if(lNode.isCompositeNode())
    {
      DiProcess lProc = DiModule.currentModule.getProcess(lStr);
      s.append(lStr + "[");
      for(Iterator iter = lProc.inputs.iterator(); iter.hasNext(); )
      {
        String signal = (String) iter.next();
        if(iter.hasNext()) s.append(signal + "i/" + signal +", ");
```

```
      else if(!outputs.isEmpty()) s.append(signal + "i/" + signal + ", ");
      else s.append(signal + "i/" + signal);
    }

    for(Iterator iter = lProc.outputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "o/" + signal + ", ");
      else s.append(signal + "o/" + signal);
    }
    s.append("]");

  }
  else
  {
    s.append(lStr + "\n");
  }

  s.append(" | \n ");

 if(rNode.isCompositeNode())
  {
    DiProcess rProc = DiModule.currentModule.getProcess(rStr);
    s.append(rStr + "[");
    for(Iterator iter = rProc.inputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "i/" + signal +", ");
      else if(!outputs.isEmpty()) s.append(signal + "i/" + signal + ", ");
      else s.append(signal + "i/" + signal);
    }

    for(Iterator iter = rProc.outputs.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      if(iter.hasNext()) s.append(signal + "o/" + signal + ", ");
      else s.append(signal + "o/" + signal);
    }
    s.append("]");

  }
  else
  {
    s.append(rStr);
  }
```

```
    s.append("\n ) ");

    s.append("\\{ ");

    for(java.util.Iterator i = internals.iterator(); i.hasNext(); )
    {
      String sig = (String) i.next();
      s.append(sig + "i, p" + sig + ", " + sig +"o");
      if (i.hasNext()) s.append(", ");
    }
    s.append("}");

    return s.toString();
  }

  void addInternalDiWires(StringBuffer s)
  {
    for(Iterator iter = internals.iterator(); iter.hasNext(); )
    {
      String signal = (String) iter.next();
      s.append("Di_Win[" + signal + "o/x , p" + signal +"/px,"
                  + signal + "i/y] | \n ");
    }
  }
}
```

Iterative process : LoopNode.java

```
package di2ccs;

class LoopNode extends PNode
{
  String id;

  LoopNode(String n)
  {
    id = new String(n);
  }

  boolean isLoopNode() { return true; }
  PNode   getRightmostNode() { return this; }

  public String toString()
  {
    return id;
```

```
  }
}
```

Appendix B

Implementation of DISP Laws in Maude

B.1 Module SET

It provides a set of variables and it gives operators used to perform set operations. This set is used to store the sequence of input and output signals.

Module declaration

```
fmod SET is
  protecting QID .
  protecting NAT .
```

Where `QID` (a module provided by maude to define quoted identifiers) and the `NAT` module defined above are used by the current module. Set consists of quoted identifiers. To denote empty set the symbol `mt` is used.

Two new sorts are declared in this module, viz., `Set` and `List`. `List` is used to store the elements of the set and hence is a subsort of `Set`.

```
  sorts Set List .
  subsorts Qid < List < Set .
```

The operators declared are:

```
  op mt : -> Set .                              *** empty set
  op _,_ : List List -> List [assoc comm] .  *** concatenation
  op {_} : List -> Set .                        *** set constructor
  op _U_ : Set Set -> Set [assoc comm] .     *** set union
  op _I_ : Set Set -> Set [assoc comm] .     *** set intersection
  op _\_ : Set Set -> Set .                     *** set difference
  op _in_ : Qid Set -> Bool .                   *** set membership
```

```
op _subset_ : Set Set -> Bool .          *** subset
op sz_ : Set -> Nat .                    *** size of set
```

Variable declarations include the name of variable and the sort it represents.

```
vars L1 L2 : List .
vars S1 S2 : Set .
vars X1 X2 : Qid .
```

Operations are defined as follows:

- Duplicates get removed during List construction.

```
eq { L1 , L1 , L2 } = { L1, L2 } .
eq { L1 , L1 } = { L1 } .
```

- Union of a set with an empty set remains unchanged. Union of two non-empty sets uses the List constructor.

```
eq S1 U mt = S1 .
eq { L1 } U { L2 } = { L1 , L2 } .
```

- Set membership returns true if the given element is in the set. If the set has a single element we compare the two elements, otherwise the comparison is done recursively on the set elements beginning from the head of the set.

```
eq X1 in mt = false .
eq X1 in { X2 } = (X1 == X2) .
eq X1 in { X2 , L1 } =
    if X1 == X2 then true else X1 in { L1 } fi .
```

- Set intersection with an empty set `mt` is empty. In the other case we use the set membership to generate the intersection set. We iterate over one set and if its element is a member of the other set, we add it to the intersection set.

```
eq mt I S1 = mt .
eq { X1 } I S1 = if X1 in S1 then { X1 } else mt fi .
eq { X1 , L1 } I S1 = ({ X1 } I S1) U ({ L1 } I S1) .
```

- To compute set difference we iterate over the first set and if the element of this set is not in the second set, we add it to the resultant set.

```
eq mt \ S1 = mt .
eq S1 \ mt = mt .
eq { X1 } \ S1 = if X1 in S1 then mt else { X1 } fi .
eq { X1, L1 } \ S1 = ({ X1 } \ S1) U ({ L1 } \ S1) .
```

- To check for subset relationship, all elements of the first set are checked for membership inside the second set.

```
eq mt subset S1 = true .
eq { X1 } subset S1 =  (X1 in S1) .
eq { X1, L1 } subset S1 =
    ({ X1 } subset S1) and ({ L1 } subset S1) .
```

- To compute size of set the NAT module is used. The size is calculated incrementally.

```
eq sz(mt) = 0 .
eq sz({ X1 }) = s(0) .
eq sz({ X1 , L1 }) = s(sz({ L1 })) .
```

After all these definitions the module can be closed using the `endfm` command.

```
endfm
```

B.2 Module NATSET

Similar to the module `SET` this module stores natural numbers instead of identifiers. Below is the complete listing of the module. Along with other set operations it also computes the maximum element in the set. It also defines an operator `gt` which return the maximum of the two arguments it takes.

```
fmod NATSET is
    protecting NAT .

    sorts NatSet NatList .
    subsorts Nat < NatList < NatSet .

    op mtNat : -> NatSet .                          *** empty set
    op _,_ : NatList NatList -> NatList [assoc comm] .
                                                    *** concatenation
    op {_} : NatList -> NatSet .                    *** set constructor
    op _U_ : NatSet NatSet -> NatSet [assoc comm] .
                                                    *** set union
    op _I_ : NatSet NatSet -> NatSet [assoc comm] .
                                                    *** set intersection
    op _\_ : NatSet NatSet -> NatSet .              *** set difference
    op _in_ : Nat NatSet -> Bool .                  *** set membership
    op max_ : NatSet -> Nat .                   *** max element in set
```

```
    op _gt_ : Nat Nat -> Nat [comm] .      *** greater element of two

    vars L1 L2 : NatList .
    vars S1 S2 : NatSet .
    vars N1 N2 : Nat .

  *** eliminate duplicates in the constructor
    eq { L1 , L1 , L2 } = { L1, L2 } .
    eq { L1 , L1 } = { L1 }.

  *** set union
    eq S1 U mtNat = S1 .
    eq { L1 } U { L2 } = { L1 , L2 } .

  *** set membership
    eq N1 in mtNat = false .
    eq N1 in { N2 } = (N1 == N2) .
    eq N1 in { N2 , L1 }
       = if N1 == N2 then true else N1 in { L1 } fi .

  *** set intersection
    eq mtNat I S1 = mtNat .
    eq { N1 } I S1 = if N1 in S1 then { N1 } else mtNat fi .
    eq { N1 , L1 } I S1 = ({ N1 } I S1) U ({ L1 } I S1) .

  *** set difference
    eq mtNat \ S1 = mtNat .
    eq { N1 } \ S1 = if N1 in S1 then mtNat else { N1 } fi .
    eq { N1, L1 } \ S1 = ({ N1 } \ S1) U ({ L1 } \ S1) .

  *** greater element of given two numbers
    eq 0 gt 0         = 0 .
    eq 0 gt N1        = N1 .
    eq s(N1) gt s(N2) = s( N1 gt N2 ).

  *** max element in the set
    eq max(mtNat) = mtNat .
    eq max({ N1 }) = N1 .
    eq max({N1 , L1 }) = N1 gt max({ L1 }) .

endfm
```

B.3 Module TERMSTATES

This provides a set of pairs and each pair itself has two sets. This is used to store the termination states of a process. The first element of the pair is the set of unabsorbed input signals and the second element is the set of pending outputs. These are generated for all the processes that terminate.

`pmt` is used as an empty set of terminating states. The `join` operator is used in the generation of termination states of a process. During this generation, we get some output signals and these output signals need to be combined with the termination states of the following process in the guarded choice. Therefore we need to add this output burst to all the outputs of the termination states of the following process. The join opeartor performs this function. For example, in case of $(xs, ys' \cup ys)$ where ys = output burst and (xs, ys') in termstates of following process, the join operator helps in combining ys and ys'.

Module declaration

```
fmod TERMSTATES is
  protecting QID .
  protecting SET .
```

It defines a sort called `Pair` that stores two sets. It defines a sort `ListOfPairs`, which stores a list of such `Pairs`. It defines the `SetOfPairs` which stores these pairs in the form of set, i.e. no duplicates.

```
  sorts Pair ListOfPairs TermStates .
  subsorts Set < Pair < ListOfPairs < TermStates .
```

The operators declared are:

```
    op pmt   : -> TermStates .
    op <_,_> : Set Set -> Pair [ctor] .
    op _,_   : ListOfPairs ListOfPairs -> ListOfPairs [assoc comm] .
                                                  *** concatenation
    op {_}   :  ListOfPairs -> TermStates .       *** constructor

    op _U_   : TermStates TermStates -> TermStates [assoc comm] .
                                                  *** union
    op _I_   : TermStates TermStates -> TermStates [assoc comm] .
                                                  *** intersection
    op _in_  : Pair TermStates -> Bool .          *** membership

    op _join_ : Set TermStates -> TermStates .
```

Variable declarations include the name of variable and the sort it represents.

```
vars p1 p2    : Pair .
vars LP1 LP2  : ListOfPairs .
vars T1 T2    : TermStates .
vars S1 S2 S3 : Set .
```

Operations are defined as follows:

- Duplicates get removed during List of pairs construction.

```
eq { LP1 , LP1 , LP2 } = { LP1, LP2 } .
eq { LP1 , LP1 } = { LP1 }.
```

- Union of a set with an empty set remains unchanged. Union of two non-empty sets uses the List of pairs constructor.

```
eq T1 U pmt = T1 .
eq { LP1 } U { LP2 } = { LP1 , LP2 } .
```

- Termination state membership returns true if the given pair of termination state is in the set of pairs. If the set has a single element we compare the two elements, otherwise the comparison is done recursively on the set elements beginning from the head of the set.

```
eq p1 in pmt = false .
eq p1 in { p2 } = (p1 == p2) .
eq p1 in { p2 , LP1 } =
    if p1 == p2 then true else p1 in { LP1 } fi .
```

- Intersection with an empty set `pmt` is empty. In the other case we use the set membership to generate the intersection set. We iterate over one set and if its element is a member of the other set, we add it to the intersection set.

```
eq pmt I T1 = pmt .
eq { p1 } I T1 = if p1 in T1 then { p1 } else pmt fi .
eq { p1 , LP1 } I T1 = ({ p1 } I T1) U ({ LP1 } I T1) .
```

- The `join` operator adds the elements of the given set to the second element of each pair in the current set of pairs. This functionality is used by the canonical form operators of module PROCESS. `join` with either an empty set of an empty set of pairs makes no changes. In the other case we iterate over the set of pairs and add the elements to the second component of each pair.

```
eq mt join T1 = T1 .
eq S1 join pmt = pmt .
eq S1 join { < S2 , S3 > } = { < S2 , (S3 U S1) > } .
eq S1 join { p1 , LP1 } = (S1 join { p1 }) U (S1 join { LP1 }) .
```

The module ends with `endfm`.

B.4 Module BURST

This module helps us declare an input/output burst each as a set of variables. The burst is give precedence value of 38 which is high precedence then others so that it gets bound earlier.

Module and sort declaration

```
fmod BURST is
    protecting SET .

    sort Burst .
    subsort Burst < Set .

    op _/_ : Set Set -> Burst [ctor prec 38] .  *** constructor

endfm
```

B.5 Module PROCESS

This module defines all the operators of DISP and gives the algebraic laws for operator elimination and conversion to canonical form.

Module and sort declaration:
The sorts are declared as follows. If a sort is contained in another then it is declared as a sub-sort of the latter.

```
fmod PROCESS is
    protecting BURST .
    protecting TERMSTATES .
    protecting NATSET .

    sorts Choice Process AlphaProcess .
    subsorts Burst < Choice < Process .
```

The constructors are defined as. The single choice is given precdence value of 39 which is lower in precedence than the input-output burst, but higher than others so that the whole guarded process gets bound before composing with other processes. The `formal` attribute is used to get a formatted output after reduction.

```
    op _‘[_‘,_‘]  : Process Set Set -> AlphaProcess [ctor] .

    op error      : -> Process [ctor] .          *** error
```

```
op skip       : -> Process [ctor] .          *** error

op stop       : -> Process [ctor] .          *** error

op pushback_  : Set -> Process [ctor] .      *** pushback

op mtchoice   : -> Choice [ctor] .

op miracle    : -> Choice [ctor] .           *** dummy choice

op _then_     : Burst Process -> Choice [ctor prec 39 gather (E e)] .
                                             *** single-choice

op select_end : Choice -> Process
                [ctor format ( n++i d ni-- d ) ] .
                                             *** guarded process

op _alt_      : Choice Choice -> Choice
                [ctor assoc comm format ( d ni ssss d ) ] .
                                             *** set of choices
```

Variables used are

```
vars X1 X2       : Qid .
vars N1 N2       : Nat .
vars L1 L2       : List .
vars B1 B2       : Burst .
vars C1 C2 C3 C4 : Choice .
vars P1 P2 Q1 Q2 : Process .
vars S1 S2 S3 S4 : Set .
vars S5 S6 S7 S8 : Set .
vars p1 p2       : Pair .
vars l1 l2       : ListOfPairs .
vars T1 T2       : TermStates .
vars a0 a1 a2 b0 b1 b2 : Set .
vars NL1 NL2     : NatList .
vars NS1 NS2     : NatSet .
```

Auxiliary specifications

A set of guarded choices does not have duplicates.

```
eq C1 alt C1 = C1 .
```

An empty choice combined with a non-empty choice gets removed.

```
eq C1 alt mtchoice = C1 .
```

During elimination we cannot just drop a choice, so we replace it by a dummy choice called `miracle`. Such choices need to be removed from the resultant set of choices.

```
eq C1 alt miracle = C1 .
eq select miracle end = skip .
```

An operator `removeMtMt` is defined which removes empty input/output guards from a choice if that is the only choice in the set.

```
op removeMtMt_ : Process -> Process .

eq removeMtMt( miracle ) = miracle .
eq removeMtMt(error) = error .
eq removeMtMt(pushback(S1)) = pushback(S1) .
eq removeMtMt( (S1 / S2) then P1 )
    = if S1 == mt and S2 == mt
      then
         removeMtMt(P1)
      else
        (S1 / S2) then P1
      fi .
```

A single choice as a process also does the same as above.

```
eq removeMtMt( select ((S1 / S2) then P1) end )
    = if S1 == mt and S2 == mt
      then
         removeMtMt(P1)
      else
        select (S1 / S2) then P1 end
      fi .
```

No removals are possible from `mtchoice` and a set of choices.

```
eq removeMtMt( mtchoice ) =  mtchoice .
eq removeMtMt( C1 alt C2 ) = C1 alt C2 .

eq removeMtMt( select mtchoice end ) = select mtchoice end .
eq removeMtMt( select C1 alt C2 end ) = select C1 alt C2 end .
```

An operator `cherr` is defined which helps in modelling of interference. This operator takes a set of signals and generates choices which lead to error on inputting any of these signals. If the set is empty, it generates the dummy choice, which has empty input/output burst, and an empty pushback. In other case it recursively generates the choices from the set.

```
op cherr_     : Set -> Choice .

eq cherr( mt ) = miracle .
eq cherr({ X1 }) = ({ X1 } / mt ) then error .
eq cherr({ X1 , L1 }) =
      (  (({ X1 } / mt) then error ) alt (cherr( { L1 })) ) .
```

stop, skip and non-deterministic choice

```
*** law for skip
   eq skip = pushback(mt) .

*** law for stop
  eq stop = select mtchoice end .

  op _or_     : Process Process -> Process [assoc comm] .

  eq P1 or P2 =
      select (((mt / mt) then P1) alt ((mt / mt) then P2) ) end .
```

Sequential Composition

This operator has higher precedence than `or` and `par`. Operator declaration:

```
op _;_    : Process Process -> Process [assoc prec 38] .
```

Defining the elimination laws as equational specifications

- `error` process

  ```
  eq error ; P1 = error .
  ```

  `pushback` process: `skip` terminates immediately.

  ```
  eq P1 ; pushback(mt)  = P1 .
  eq pushback(mt) ; P1  = P1 .
  ```

  `pushback` followed by `error` is `error`.

  ```
  eq pushback(S1) ; error = error .
  ```

  If intersection of the two sets pushed back is empty then we combine them else it is `error`.

```
eq pushback(S1) ; pushback(S2) =
    if (S1 I S2) == mt then pushback(S1 U S2) else error fi .
```

`pushback` followed by a guarded choice is the guarded choice after the set of signals in pushback set.

```
eq pushback(S1) ; select C1 end  = select C1 end af S1 .
```

- Process following a guarded choice distributes through each choice. This is done for set of choices and also for a process made up of a set of choices. `mtchoice` models `stop` process which does not terminate, and therefore any process following it is never enabled.

```
eq (B1 then P1) ; P2 = B1 then (P1 ; P2) .
eq (C1 alt C2) ; P1  = (C1 ; P1) alt (C2 ; P1) .
eq (select mtchoice end) ; P1  = select mtchoice end .
eq (select C1 end) ; P1  = select C1 ; P1 end .
```

Combining two consecutive input/output bursts

```
ceq (S1 / S2) ; (S3 / S4) = ((S1 U S3) / S4)
    if S2 == mt and (S1 I S3) == mt .

ceq (S1 / S2) ; (S3 / S4) = error
    if S2 == mt and (S1 I S3) =/= mt .

ceq (S1 / S2) ; (S3 / S4) = (S1 / (S2 U S4))
    if S3 == mt and (S2 I S4) == mt .

ceq (S1 / S2) ; (S3 / S4) = ((S1 / mt) then error)
    if S3 == mt and (S2 I S4) =/= mt .

eq (S1 / S2) ; (S3 / S4) =
    select (S1 / S2) then skip end ;
    select (S3 / S4) then skip end .

eq (S1 / S2) ; select C1 end =
    select (S1 / S2) then skip end ; select C1 end .
```

After-input operator

Operator declaration

```
op _af_  : Process Set -> Process .
```

Elimination laws:

- `error` after any set of input signals remains unchanged.

```
eq error af S1 = error .
```

- Any process after empty set of inputs remains unchanged.

```
eq P1 af mt    = P1 .
```

- `pushback` after an input set depends on whether the two sets cause interference.

```
eq pushback(S1) af S2 =
    if (S1 I S2) == mt then pushback(S1 U S2) else error fi .
```

- For guarded choices, we perform the necessary set cancellations due to available input signals and also generate choices with the available signals leading to `error` using the `cherr` operator. This is done recursively for each choice.

```
eq ((S1 / S2) then P1) af S3 =
    (((S1 \ S3) / S2) then (P1 af (S3 \ S1))) .

eq mtchoice af S1  = cherr(S1) .
eq (C1 alt C2) af S1 = (C1 af S1) alt (C2 af S1) .

eq select C1 end af S1 =
  if allInputguardsNonempty(C1) or (C1 == mtchoice) then
      select (C1 af S1) alt cherr(S1) end
  else
      select (C1 af S1) end
  fi .
```

After-output operator

The after-output operator requires the alphabets of the process and therefore we use `AlphaProcess` as its parameter.
Operator declaration:

```
op _afOut_  : AlphaProcess Qid -> Process .
```

Elimination laws:

- `error` after any set of input signals remains unchanged.

```
eq (error[a1,b1]) afOut X1 = error .
```

- pushback make no outputs

```
eq (pushback(S1)[a1,b1]) afOut X1 = pushback(S1) .
```

- For guarded choices we perform the necessary cancellation of output signals. This is done only if the given set of output signals is a subset of the output generated by the choice. To compute the outputs generated by the choice, we use the out operator.

```
eq (((S1 / S2) then P1)[a1,b1]) afOut X1
   = if (S1 == mt) and (X1 in out(((S1 / S2) then P1)[a1,b1]) )
     then
       if X1 in S2
       then ((mt / (S2 \ { X1 })) then P1)
       else ((mt / S2) then (P1[a1,b1] afOut X1))
       fi
     else
       miracle
     fi .

eq ((mtchoice)[a1,b1]) afOut X1 = mtchoice .

eq ((C1 alt C2)[a1,b1]) afOut X1
    =  ((C1[a1,b1]) afOut X1) alt ((C2[a1,b1]) afOut X1) .

eq ((select C1 end)[a1,b1]) afOut X1
    = select (C1[a1,b1]) afOut X1 end .
```

Parallel Composition

Operator declaration

```
op _par_    : AlphaProcess AlphaProcess -> Process [comm] .
op _parNC_  : AlphaProcess AlphaProcess -> Process .
```

Elimination laws:

- error parallel with any process leads to error.

```
eq (error[a1,b1]) par (P2[a2,b2]) = error .
```

- pushback parallel with pushback

```
eq (pushback(S1)[a1,b1]) par (pushback(S2)[a2,b2])
    = pushback((S1 \ b2) U (S2 \ b1)) .
```

- pushback in parallel with a guarded choice

```
ceq (pushback(S1)[a1,b1]) par (((S2 / S3) then P2)[a2,b2])
    = ( ((S2 / mt) then error) alt (cherr(S1 \ b2)) )
        if (S2 I b1) == mt and (S3 I S1) =/= mt .

ceq (pushback(S1)[a1,b1]) par (((S2 / S3) then P2)[a2,b2])
    = (
       ((S2 / (S3 I (b2 \ a1)) ) then
          removeMtMt(
              ((pushback(S1 U (S3 I (a1 \ S1)))[a1,b1])
               par (P2[a2,b2]))
          )
       )
       alt (cherr(S1 \ b2))
      )
      if (S2 I b1) == mt and (S3 I S1) == mt .
```

If none of above condition satisfy we need to provide other alternative equations. There are not given as algebraic laws in the main test of the thesis, as one need not construct them. But here as this is automated we need to have all possible alternatives available. The following take care of them and reduce them to the dummy choice that get deleted eventually.

```
ceq (pushback(S1)[a1,b1]) par (((S2 / S3) then P2)[a2,b2])
    = miracle if (S2 I b1) =/= mt .
```

For a set of choices and guarded process we perform the same operations recursively. Though parallel composition is commutative, in case of solving guarded choices we need to check both the processes composed and iteratively select the enabled choices at a given time. To do this we need to define a non-commutative parallel composition operator. This operator does parallel composition of each process with the other, with a process kept on LHS and RHS. stop in parallel with stop results in stop, but in parallel with a set of choices can be further reduced.

```
eq (pushback(S1)[a1,b1]) par ((C1 alt C2)[a2,b2])
    = ( ((pushback(S1)[a1,b1]) par (C1[a2,b2])) alt
              ((pushback(S1)[a1,b1]) par (C2[a2,b2])) ) .
```

```
eq (pushback(S1)[a1,b1]) par ((select mtchoice end)[a2,b2])
    = select mtchoice end .
eq (pushback(S1)[a1,b1]) par ((select C1 end)[a2,b2])
    = select (pushback(S1)[a1,b1]) par ((C1)[a2,b2]) end .

eq (C1[a1,b1]) par (C2[a2,b2])
    = ((C1[a1,b1]) parNC (C2[a2,b2]))
      alt  ((C2[a2,b2]) parNC (C1[a1,b1])) .

eq ((select mtchoice end)[a1,b1]) par
     ((select mtchoice end)[a2,b2])
        = select mtchoice end .

eq ((select mtchoice end)[a1,b1]) par ((select C2 end)[a2,b2])
    = select ((C2[a2,b2]) parNC ((mtchoice)[a1,b1])) end .

eq ((select C1 end)[a1,b1]) par ((select C2 end)[a2,b2])
    = select (C1[a1,b1]) par (C2[a2,b2]) end .
```

- Laws for the non-commutative parallel composition.
  If a single choice is in parallel with an empty choice, we select the choice if its input guards are all from the environment otherwise it results into empty choice.

```
eq (((S1 / S2) then P1)[a1,b1]) parNC ((mtchoice)[a2,b2])
    = if (S1 I b2) == mt then
      ((S1 / (S2 \ a2)) then
         (
         removeMtMt( (P1[a1,b1]) par
           (select((mtchoice) af (S2 I a2))end[a2,b2]))
         )
      )
      else
      ( mtchoice )
      fi .
```

  We perform the similar operation recursively on a set of choices in parallel with an empty choice.

```
eq (((((S1 / S2) then P1) alt C1)[a1,b1])
   parNC ((mtchoice)[a2,b2]) )
    = if (S1 I b2) == mt then
      (
```

```
        ((S1 / (S2 \ a2)) then
           (
            removeMtMt( (P1[a1,b1]) par
               ( select((mtchoice) af (S2 I a2))end[a2,b2]) )
            )
         )
         alt ( (C1[a1,b1]) parNC ((mtchoice)[a2,b2]) )
        )
        else
            ( (C1[a1,b1]) parNC ((mtchoice)[a2,b2]) )
        fi .
```

Two single non-empty choices in parallel result in a dummy choice if they do not wait explicitly for inputs from the environment.

```
ceq (((S1 / S2) then P1)[a1,b1])
     parNC (((S3 / S4) then P2)[a2,b2])
    = miracle if (S1 I b2) =/= mt and (S3 I b1) =/= mt .
```

If both of them wait for the environment then we select both the choices.

```
*** single choice on side-1 and side-2 : both enabled
ceq (((S1 / S2) then P1)[a1,b1])
     parNC (((S3 / S4) then P2)[a2,b2])
    =
    ( ( (S1 / (S2 \ a2)) then
       (
       removeMtMt( (P1[a1,b1]) par
        (select (((S3 / S4) then P2) af (S2 I a2))end[a2,b2]))
       )
      )
      alt
      ( (S3 / (S4 \ a1)) then
        (
        removeMtMt( (P2[a2,b2]) par
         (select(((S1 / S2) then P1) af (S4 I a1))end[a1,b1]))
        )
      )
    ) if (S1 I b2) == mt and (S3 I b1) == mt .
```

Alternatively, if either of them waits for inputs explicitly from environment then we select this choice.

```
*** single choice on side-1 and  side-2 : left side enabled
ceq (((S1 / S2) then P1)[a1,b1])
    \ parNC (((S3 / S4) then P2)[a2,b2])
    =
    ( (S1 / (S2 \ a2)) then
       (
       removeMtMt( (P1[a1,b1]) par
         (select(((S3 / S4) then P2) af (S2 I a2))end[a2,b2])
       ))
    ) if (S1 I b2) == mt and (S3 I b1) =/= mt .

*** single choice one side-1 and side-2 : right side enabled
ceq (((S1 / S2) then P1)[a1,b1])
    parNC (((S3 / S4) then P2)[a2,b2])
    =
    ( (S3 / (S4 \ a1)) then
       (
       removeMtMt( (P2[a2,b2]) par
         (select(((S1 / S2) then P1) af (S4 I a1))end[a1,b1])
       ))
   ) if (S1 I b2) =/= mt and (S3 I b1) == mt .
```

There is a single non-empty choice in parallel with a set of non-empty choices, we select this choice if it waits for inputs from the environment and then recursively continue the parallel composition, otherwise it results into a dummy choice.

```
*** single choice on side-1 and set of choices on side-2
eq ((((S1 / S2) then P1)[a1,b1]) parNC (C2[a2,b2]) )
    = if (S1 I b2) == mt
      then
       ( (S1 / (S2 \ a2)) then
            (
            removeMtMt( (P1[a1,b1]) par
                      ( select( C2 af (S2 I a2))end[a2,b2]))
            )
       )
      else
        miracle
      fi .
```

If there are set of non-empty choices in parallel with each other, if a choice on the left side is enabled, we select the choice and recursively continue

the parallel composition. Otherwise the composition is continued for other choices on the left side.

```
*** set of choices on both sides
*** 1 from side-1 and 0 from side-2 enabled
eq (((((S1 / S2) then P1) alt C1)[a1,b1]) parNC (C2[a2,b2]) )
    = if (S1 I b2) == mt
      then
         ( ( (S1 / (S2 \ a2)) then
                (
                removeMtMt( (P1[a1,b1]) par
                    ( select(C2 af (S2 I a2))end[a2,b2]) )
                )
           )
           alt ( (C1[a1,b1]) parNC (C2[a2,b2]) )
         )
      else
         ( (C1[a1,b1]) parNC (C2[a2,b2]) )
      fi .
```

Initial outputs of a process

To compute the initial set of possible outputs generated by a process we need a process with its alphabet set and return the set of output signals.

```
op out_  : AlphaProcess -> Set .
```

Elimination laws:

- Output of an error process is the complete output alphabet.

  ```
  eq out(error[a1,b1]) = b1 .
  ```

- `pushback` makes no outputs.

  ```
  eq out(pushback(S1)[a1,b1]) = mt .
  ```

- Output of a guarded choice is the output burst if its input burst is empty and the outputs of the following process.

  ```
  *** out of a single choice
  ceq out( ((S1 / S2) then P1)[a1,b1] )
      = S2 U out(P1[a1,b1])  if S1 == mt .
  ```

```
    *** out of a single choice
    ceq out( ((S1 / S2) then P1)[a1,b1] ) = mt  if S1 =/= mt .

    eq out( (mtchoice)[a1,b1] ) = mt .

    eq out( (miracle)[a1,b1] ) = mt .

    *** out of set of choices
    eq out( (C1 alt C2)[a1,b1] )
        = out( C1[a1,b1] ) U out( C2[a1,b1] ) .

    eq out( (select C1 end)[a1,b1] ) = out( C1[a1,b1] ) .
```

Process divergence

This takes a process with its alphabets and returns a boolean stating if the process diverges.

```
op div_    : AlphaProcess -> Bool .
```

Elimiation laws:

```
eq div(error[a1,b1]) = true .

eq div(pushback(S1)[a1,b1]) = false .

*** divergence of a single choice
  eq div( ((S1 / S2) then P1)[a1,b1] )
      = if S1 == mt then
           ((S2 I out(P1[a1,b1])) =/= mt ) or div(P1[a1,b1])
        else false
        fi  .

  eq div( (mtchoice)[a1,b1] ) = false .

*** divergence of set of choices
  eq div( (C1 alt C2)[a1,b1] )
      = div( C1[a1,b1] ) or div( C2[a1,b1] ) .

  eq div( (select C1 end)[a1,b1] ) = div( C1[a1,b1] ) .
```

Termination states

This gives the possible states of termination of a process. Here we define the laws only for non-divergent processes. Because we need this only for the canonical

form definition which does not use terminates definition for divergent processes.

```
op term_   : AlphaProcess -> TermStates .
```

Clauses:

```
ceq term(pushback(S1)[a1,b1])
    = { < S1 , mt > } if not div(pushback(S1)[a1,b1]) .

eq term( (miracle)[a1,b1] ) = pmt .
```

For a single non-empty choice, we use the join operator of the TERMSTATES module to combine the output burst of the current choice with the terminations of the following process (say P1), i.e. to all the termination states of P1, the output burst is added to all the pending outputs of all pairs in the termination states of P1.

```
eq term( ((S1 / S2) then P1)[a1,b1] )
    = if S1 == mt then S2 join term(P1[a1,b1]) else pmt fi .

eq term( (mtchoice)[a1,b1] ) = pmt .

*** termination states of set of choices
eq term( (C1 alt C2)[a1,b1] ) i
    = term( C1[a1,b1] ) U term( C2[a1,b1] ) .

eq term( (select C1 end)[a1,b1] ) = term( C1[a1,b1] ) .
```

Refuses to output

Returns true if the given process refuses to output. For a single non-empty choice, the choice refuses to output if it has a non-empty input burst. For a set of choices, this set refuses to output if all its input guards are non-empty or there is an empty input/output burst and its guarded process refuses to output.

```
*** operator declaration
op refuses_  : Process -> Bool .

*** equations
eq refuses(error) = true .
eq refuses(pushback(S1)) = false .
eq refuses( (S1 / S2) then P1 ) =
    (S1 =/= mt) or ((S1 == mt) and (S2 == mt) and (refuses(P1))) .
eq refuses( mtchoice ) = true .
eq refuses( C1 )
    = allInputguardsNonempty(C1) or oneEmptyBurst(C1) .
eq refuses( select C1 end) = refuses(C1) .
```

Auxiliary specs for canonical form

terminates
Takes a process and return true if the process terminates immediately.

```
*** operator declaration
  op terminates_  : Process -> Bool .

*** pushback terminates immediately
  eq terminates(pushback(S1)) = true .

*** a guarded choice terminates if it has an empty input burst
*** and the guarded process terminates as well
  eq terminates( (S1 / S2) then P1 )
      = if S1 =/= mt then
           false
        else
           terminates(P1) fi .

  eq terminates(mtchoice) = false .

*** A set of choices terminates if any one
*** of its choices terminates
  eq terminates(C1 alt C2) = terminates(C1) or terminates(C2) .

  eq terminates(select C1 end) = terminates(C1) .
```

allTerminations
Gives a set of choices to be added to the canonical form. It generates the choices using the set of possible termination states.

```
op allTerminations_ : TermStates -> Choice .
```

If the set of termination states is empty then we return a dummy choice. Otherwise we generate an input/output burst with an input burst followed by the output burst containing the pending outputs from the set of termination states and a `pushback` with the unabsorbed inputs. This is done recursively for all the set elements.

```
eq allTerminations(pmt) = miracle .
eq allTerminations({ < S1 , S2 > })
    = ( mt / S2 ) then pushback(S1) .
eq allTerminations({ p1 , l1 })
    = (allTerminations({ p1 })) alt (allTerminations({ l1 })) .
```

allOutputs
Generates set of choices for all possible outputs of a given process i.e. [-/y then CF(P after y)]

```
*** operator declaration
  op allOutputs(_,_) : Set AlphaProcess -> Choice .

*** equations
  eq allOutputs(mt , (P1[a1,b1])) = miracle .

  eq allOutputs({ X1 }, (P1[a1,b1]))
      = ( mt / { X1 }) then CF( ((P1[a1,b1]) afOut X1 )[a1,b1] ) .

  eq allOutputs({ X1 , L1 }, (P1[a1,b1]))
      = allOutputs({ X1 }, (P1[a1,b1]))
        alt
        allOutputs({ L1 }, (P1[a1,b1])) .
```

allInputs
For the given set generate cases like [a/- then CF(P1 after a) with xsUa as unsafe. Here the set of inputs is the first set and the unsafe signals is the second set.

```
*** operator declaration
  op allInputs(_,_,_)  : Set AlphaProcess Set -> Choice .

*** equations
  eq allInputs(mt , (P1[a1,b1]) ) = miracle .

  eq allInputs({ X1 } , (P1[a1,b1]) )
      = ({ X1 } / mt) then (CF ((P1 af { X1 })[a1,b1])) .

  eq allInputs({ X1 , L1 }, (P1[a1,b1]))
      = allInputs({ X1 }, (P1[a1,b1]))
        alt
        allInputs({ L1 }, (P1[a1,b1])) .
```

allInputguardsNonempty
Returns true if all the input guards in a choice are non-empty. We use logical-and to combine the results of recursion.

```
*** operator declaration
  op allInputguardsNonempty_ : Choice -> Bool .

*** equations
```

```
  eq allInputguardsNonempty( (S1 / S2) then P1 ) = (S1 =/= mt) .
  eq allInputguardsNonempty( C1 alt C2 )
      = allInputguardsNonempty(C1) and allInputguardsNonempty(C2) .
```

oneEmptyBurst
Returns true if any one input/output burst is empty and the following process also refuses. To find if there exists such a burst, we use logical or to combine the results of recursion.

```
*** operator declaration
  op oneEmptyBurst_ : Choice -> Bool .

*** equations
  eq oneEmptyBurst( (S1 / S2) then P1)
      = (S1 == mt) and (S2 == mt) and (refuses(P1)) .
  eq oneEmptyBurst( C1 alt C2)
      = oneEmptyBurst(C1) or oneEmptyBurst(C2) .
```

refusesTermOut
The clause in the canonical form definition: *refuses*$(P) \wedge$ (*terminates*$(P) \neq \emptyset \vee$ *out*$(P) \neq \emptyset$) is implemented by this routine.

```
*** operator declaration
  op refusesTermOut_ : AlphaProcess -> Bool .

*** equations
  eq refusesTermOut(P1[a1,b1])
    = refuses(P1) and
      ( (term(P1[a1,b1]) =/= pmt) or
        (out(getNonTerminatingChoices(P1)[a1,b1]) =/= mt) ) .
```

getTerminatingChoices
Takes a process and returns a set of choices which do not terminate immediately.

```
*** operator declaration
  op getTerminatingChoices_ : Process -> Choice .

*** equations
  eq getTerminatingChoices(pushback(S1)) = pushback(S1) .

  eq getTerminatingChoices((S1 / S2) then P1)
      = if terminates( (S1 / S2) then P1) then
            ( (S1 / S2) then P1 )
        else
            miracle fi .
```

```
  eq getTerminatingChoices(mtchoice)
      = miracle .

  eq getTerminatingChoices(C1 alt C2)
      = getTerminatingChoices(C1) alt getTerminatingChoices(C2) .

  eq getTerminatingChoices(select C1 end) = getTerminatingChoices(C1) .
```

getNonTerminatingChoices
Takes a process and return a set of choices from that process which do not terminate immediately.

```
*** operator declaration
  op getNonTerminatingChoices_ : Process -> Choice .

*** equations
  eq getNonTerminatingChoices(pushback(S1))
      = miracle .

  eq getNonTerminatingChoices((S1 / S2) then P1)
      = if terminates( (S1 / S2) then P1) then
            miracle
        else
            ( (S1 / S2) then P1 ) fi .

  eq getNonTerminatingChoices(mtchoice) = mtchoice .

  eq getNonTerminatingChoices(C1 alt C2)
      = getNonTerminatingChoices(C1)
        alt
        getNonTerminatingChoices(C2) .

  eq getNonTerminatingChoices(select C1 end)
      = getNonTerminatingChoices(C1) .
```

terminatingChoicesCF
Generates the set of choices to be added to the canonical form of a process. These choices are those that terminate immediately. The `allTerminations` operator generates the required set of choices, using the set of terminating choices as input.

```
*** operator declaration
  op terminatingChoicesCF(_) : AlphaProcess -> Choice .
```

```
*** equation
  eq terminatingChoicesCF( (P1[a1,b1]) )
      = allTerminations( term(P1[a1,b1]) ) .
```

nonTerminatingChoicesCF
Generates the set of choices to be added to the canonical form of a process. These choices are those that do not terminate immediately. The set of non-terminating choices is sent as parameter. It calls the `allOutputs` operator which generates the required choices.

```
*** opearator declaration
  op nonTerminatingChoicesCF(_) : AlphaProcess  -> Choice .

*** equation
  eq nonTerminatingChoicesCF( (P1[a1,b1]) )
     = allOutputs( out(P1[a1,b1]) , (P1[a1,b1]) ) .
```

refusesChoicesCF
Generates the set of choices to be added to the canonical form of a process. These choices are those that can refuse to output or terminate. If the given process has some initial outputs or can immediately terminate then we maintain the non-determinism by inserting an empty input/output burst before the canonisised choice.

```
*** operator declaration
  op refusesChoicesCF(_) : AlphaProcess  -> Choice .

*** equation
  eq refusesChoicesCF(P1[a1,b1])
      = if (refusesTermOut(P1[a1,b1]))
        then
        (
           (mt / mt) then select allInputs( a1 , (P1[a1,b1]) ) end
        )
        else
           allInputs( a1 , (P1[a1,b1]) )
        fi .
```

Canonical form specifications

The canonical form operator takes the process with its alphabets and generates the canonical form for the given process.

```
*** operator declaration
  op CF(_)     : AlphaProcess -> Process .
```

```
*** equation
  eq CF(P1[a1,b1])
      = if div(P1[a1,b1])
        then error
        else
        select
            terminatingChoicesCF(
                (getTerminatingChoices(P1))[a1,b1])
            alt
            nonTerminatingChoicesCF(
                (getNonTerminatingChoices(P1))[a1,b1])
            alt
            refusesChoicesCF(P1[a1,b1])
       end
       fi .
```

Size of process

Computes the size of a given process. Uses the `sz` operator of `SET` module to compute size of a set. The values are computed using `NAT` module for natural numbers.

```
*** operator declaration
  op size_ : Process ->  Nat .

*** equations
  eq size(error) = s(0) .
  eq size(pushback(S1)) = s(sz(S1)) .
  eq size( (S1 / S2) then P1 )
      = s(0) + ((sz(S1) + sz(S2)) + size(P1)) .
  eq size( mtchoice ) = s(0) .
  eq size( select C1 end) = s(0) + size(C1) .
  eq size( C1 alt C2 ) = s(0) + (size( C1 ) + size( C2 )) .
```

Depth of process

To compute the depth of a process, all internal depths are calculated and then the maximum of these values is chosen. The `getInternaldepths` operator computes recursively depths of a given process expression.The `getAlldepths` operator collects all the internal depths in a set of natural numbers.

```
*** operator declarations
  op depth_ : AlphaProcess ->  Nat .
```

```
 op getAlldepths_ : AlphaProcess -> NatSet .

 op getInternaldepths(_,_) : Set AlphaProcess -> NatSet .

*** equations
 eq getInternaldepths(mt , (P1[a1,b1])) = mtNat .

 eq getInternaldepths({ X1 }, (P1[a1,b1]))
     = if (X1 in a1) then
          { depth( (P1 af { X1 })[a1,b1] ) }
       else
          { depth( ((P1[a1,b1]) afOut X1 )[a1,b1] ) }
       fi .

 eq getInternaldepths({ X1, L1 }, (P1[a1,b1]))
    = getInternaldepths({ X1 }, (P1[a1,b1]))
      U
      getInternaldepths({ L1 }, (P1[a1,b1])) .

*** returns a list of Nats
 eq getAlldepths(P1[a1,b1]) =
    getInternaldepths(out(P1[a1,b1]) , P1[a1,b1])
    U
    getInternaldepths(a1, P1[a1,b1]) .

 eq depth(P1[a1,b1])
      = if div(P1[a1,b1])
           then 0
        else
           s(0) + max(getAlldepths(P1[a1,b1]))
        fi .
```

Appendix C

Decomposition of Benchmark Circuits

C.1 Circuits Related to Concurrent Outputs

C.1.1 scsi isend

The Burst-Mode description of SCSI initiator send controller is shown in Figure C.1. The DISP description of the controller and its environment is given below. To mantain delay-insensitive behaviour an additional signal `Next` is added to the specification. This signal avoids transmission interference on the signal `DAckNormN` in transitions from state 4 to 5 and from state 5 to 6.

```
R = forever do -/ReqInN ; AckOutN/- end

T =
forever do -/StartDMASend ; EndDMAInt/- end

L = pushback Next ;
forever do
  select Next,DReqN/DAckNormN then
         ReadyN/DTCN ;
         Next,ReadyN/DAckNormN,DTCN ;
         pushback DReqN
     alt Next,DReqN/DAckLastN then
         ReadyN,DReqN/DTCN ;
         Next,ReadyN/DAckLastN,DTCN
  end
end
```

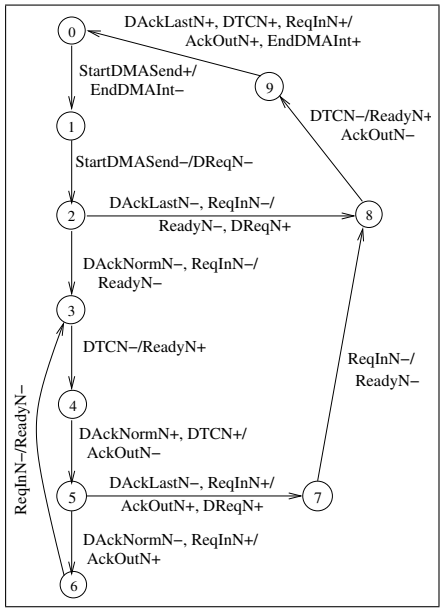


Figure C.1: scsi-isend

```
#environment R par T par L

ISEND =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DReqN ;
       select DAckLastN,ReqInN/ReadyN,DReqN then
                DTCN/ReadyN,AckOutN,Next ;
                DAckLastN,DTCN,ReqInN/AckOutN,EndDMAInt,Next
          alt DAckNormN,ReqInN/ReadyN
       end
     alt DTCN/ReadyN,Next then DAckNormN,DTCN/AckOutN,Next ;
           select DAckLastN,ReqInN/AckOutN,DReqN then ReqInN/ReadyN ;
                    DTCN/ReadyN,AckOutN,Next ;
                    DAckLastN,DTCN,ReqInN/AckOutN,EndDMAInt,Next
              alt DAckNormN,ReqInN/AckOutN then ReqInN/ReadyN
           end
  end
end
```

Petrify output:

```
# EQN file for model ISEND
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 103.00

INORDER = StartDMASend ReqInN DTCN DAckNormN DAckLastN ReadyN Next
EndDMAInt DReqN AckOutN csc0 csc1 csc2 csc3;
OUTORDER = [ReadyN] [Next] [EndDMAInt] [DReqN] [AckOutN]
[csc0] [csc1] [csc2] [csc3];
[0] = ReqInN' DTCN csc2 csc0 csc3;
[1] = DTCN' (csc0' + csc1');
[2] = DTCN DAckNormN (csc1 + DReqN');
[Next] = [2]' ([1] + Next) + Next [1];     # mappable onto gC
[4] = csc2' csc3';
[5] = csc2 StartDMASend;
[EndDMAInt] = [5]' ([4] + EndDMAInt) + EndDMAInt [4];  # mappable onto gC
[7] = DAckLastN' csc3 (csc0 + ReqInN);
[8] = DAckLastN csc2 EndDMAInt' StartDMASend';
[DReqN] = [8]' ([7] + DReqN) + DReqN [7];     # mappable onto gC
[10] = DTCN csc1 csc0;
[11] = DTCN csc0' (DAckNormN + Next') + DTCN' DAckNormN csc0 + csc1';
[AckOutN] = [11]' ([10] + AckOutN) + AckOutN [10];     # mappable onto gC
[13] = DReqN' DTCN';
[14] = DAckLastN' csc2 csc0 + csc3';
```

```
[15] = csc2' csc3 + ReqInN csc0';
[csc1] = [15]' ([14] + csc1) + csc1 [14];     # mappable onto gC
[17] = csc1 EndDMAInt;
[18] = DTCN' DAckLastN';
[csc2] = [18]' ([17] + csc2) + csc2 [17];     # mappable onto gC
[20] = ReqInN' (DAckNormN' + DAckLastN');
[21] = ReqInN (DTCN DAckLastN csc2' + DAckNormN' csc0);
[csc3] = [21]' ([20] + csc3) + csc3 [20];     # mappable onto gC
[ReadyN] = DTCN' ([0]' + ReadyN) + ReadyN [0]';     # mappable onto gC
[csc0] = csc1' ([13]' + csc0) + csc0 [13]';     # mappable onto gC

# Set/reset pins: set(EndDMAInt) set(DReqN) set(csc1) set([17])
reset(csc3) set(ReadyN) set(csc0)
```

Decompositions applied were as follows:

```
#environment R par T par L par Fork1 par Fork2

ISENDD =
forever do
  select StartDMASend/y then t/- ; StartDMASend/DReqN ;
       select DAckLastN,ReqInN/ReadyN,DReqN then
              DTCN/x ; z/ReadyN,AckOutN ;
              DAckLastN,DTCN,ReqInN/x ; z/y ; t/AckOutN
          alt DAckNormN,ReqInN/ReadyN
       end
     alt DTCN/x then z/ReadyN ; DAckNormN,DTCN/x ; z/AckOutN ;
       select DAckLastN,ReqInN/AckOutN,DReqN then ReqInN/ReadyN ;
              DTCN/x ; z/ReadyN,AckOutN ;
              DAckLastN,DTCN,ReqInN/x ; z/y ; t/AckOutN
          alt DAckNormN,ReqInN/AckOutN then ReqInN/ReadyN
      end
  end
end

Fork1 = forever do x/Next,z end
Fork2 = forever do y/EndDMAInt,t end
```

Petrify output:

```
# EQN file for model ISENDD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 73.00
```

```
INORDER = z t StartDMASend ReqInN DTCN DAckNormN DAckLastN y x
ReadyN DReqN AckOutN csc0 csc1;
OUTORDER = [y] [x] [ReadyN] [DReqN] [AckOutN] [csc0] [csc1];
[0] = StartDMASend csc0;
[1] = DReqN' DAckLastN z' csc0';
[y] = [1]' ([0] + y) + y [0];     # mappable onto gC
[3] = ReqInN' z' csc0 csc1;
[ReadyN] = [3]' (z + ReadyN) + z ReadyN;     # mappable onto gC
[5] = StartDMASend' t csc1';
[6] = DAckLastN' csc1 (csc0 + ReqInN);
[DReqN] = [6]' ([5] + DReqN) + DReqN [5];     # mappable onto gC
[8] = z' csc0;
[9] = z' csc0' + DReqN' z;
[AckOutN] = [9]' ([8] + AckOutN) + AckOutN [8];     # mappable onto gC
[11] = DAckLastN' ReqInN + csc1';
[12] = z DTCN (ReqInN DAckLastN + DAckNormN DReqN);
[csc0] = [12]' ([11] + csc0) + csc0 [11];     # mappable onto gC
[14] = ReqInN' (DAckLastN' + DAckNormN');
[15] = DAckNormN' ReqInN + t';
[csc1] = [15]' ([14] + csc1) + csc1 [14];     # mappable onto gC
[x] = csc0 (DTCN' + x) + DTCN' x;     # mappable onto gC

# Set/reset pins: reset(y) set(ReadyN) reset(DReqN) reset(x)
```

C.1.2 scsi tsend

The Burst-Mode specification of SCSI target send controller is shown in Figure C.2. The DISP description of the controller and its environment is given below. To mantain delay-insensitive behaviour an additional signal `Next` is added to the specification. This signal avoids transmission interference on the signal `DAckNormN` in transitions from state 4 to 5 and from state 5 to 6.

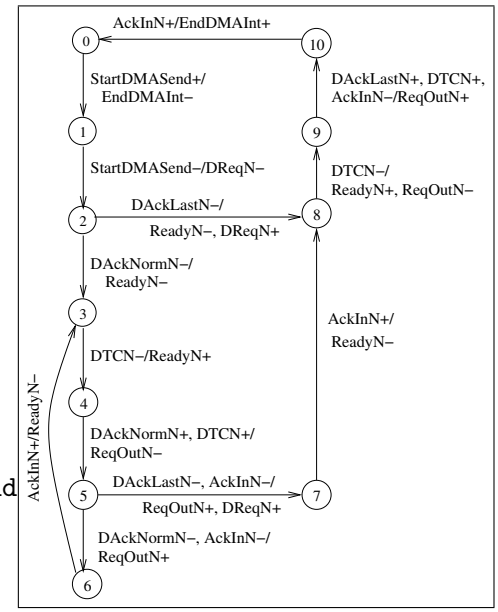


Figure C.2: scsi-tsend

```
R = forever do ReqOutN/AckInN end

T =
forever do -/StartDMASend ; EndDMAInt/- end

L = pushback Next ;
forever do
  select Next,DReqN/DAckNormN then
         ReadyN/DTCN ;
         Next,ReadyN/DAckNormN,DTCN ;
         pushback DReqN
     alt Next,DReqN/DAckLastN then
         ReadyN,DReqN/DTCN ;
         Next,ReadyN/DAckLastN,DTCN
  end
end

#environment R par T par L

TSEND =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DReqN ;
       select DAckLastN/ReadyN,DReqN then DTCN/ReadyN,ReqOutN,Next ;
                DAckLastN,DTCN,AckInN/ReqOutN,Next ; AckInN/EndDMAInt
          alt DAckNormN/ReadyN
       end
     alt DTCN/ReadyN,Next then DAckNormN,DTCN/ReqOutN,Next ;
           select DAckLastN,AckInN/ReqOutN,DReqN then AckInN/ReadyN ;
                    DTCN/ReadyN,ReqOutN,Next ;
                    DAckLastN,DTCN,AckInN/ReqOutN,Next ;
                    AckInN/EndDMAInt
              alt DAckNormN,AckInN/ReqOutN then AckInN/ReadyN
           end
  end
```

```
end
```

Petrify output:

```
# EQN file for model TSEND
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 84.00

INORDER = StartDMASend DTCN DAckNormN DAckLastN AckInN ReqOutN
ReadyN Next EndDMAInt DReqN csc0 csc1;
OUTORDER = [ReqOutN] [ReadyN] [Next] [EndDMAInt] [DReqN] [csc0] [csc1];
[0] = DTCN (DReqN DAckLastN csc1 + csc0);
[1] = DTCN DReqN' csc0' (DAckNormN + Next') + DTCN' DAckLastN' + csc1';
[ReqOutN] = [1]' ([0] + ReqOutN) + ReqOutN [0];     # mappable onto gC
[3] = DAckLastN DTCN' + csc0';
[4] = AckInN csc0 (DTCN DAckNormN' + DAckLastN');
[ReadyN] = [4]' ([3] + ReadyN) + ReadyN [3];     # mappable onto gC
[6] = DTCN' (csc0' + DAckLastN');
[7] = DTCN DAckNormN csc1;
[Next] = [7]' ([6] + Next) + Next [6];     # mappable onto gC
[9] = AckInN DReqN csc0' csc1;
[EndDMAInt] = csc0' ([9] + EndDMAInt) + EndDMAInt [9]; # mappable onto gC
[11] = DAckLastN' (csc0 + AckInN');
[12] = DAckLastN StartDMASend' csc0;
[DReqN] = [12]' ([11] + DReqN) + DReqN [11];     # mappable onto gC
[14] = AckInN' (DAckLastN' csc1 + DAckNormN') + StartDMASend;
[15] = DAckLastN DTCN' + csc1';
[csc0] = [15]' ([14] + csc0) + csc0 [14];     # mappable onto gC
[17] = AckInN' DAckLastN;
[18] = DAckLastN' DTCN';
[csc1] = [18]' ([17] + csc1) + csc1 [17];     # mappable onto gC

# Set/reset pins: set(DReqN) reset(csc0) set(csc1)
```

Decompositions applied were as follows:

```
#environment R par T par L par Fork1

TSENDD =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DReqN ;
       select DAckLastN/ReadyN,DReqN then DTCN/x ; z/ReadyN,Next ;
              DAckLastN,DTCN,AckInN/x ; z/Next ; AckInN/EndDMAInt
          alt DAckNormN/ReadyN
```

```
       end
     alt DTCN/ReadyN,Next then DAckNormN,DTCN/x ; z/Next ;
           select DAckLastN,AckInN/x then z/DReqN ; AckInN/ReadyN ;
                  DTCN/x ; z/ReadyN,Next ;
                  DAckLastN,DTCN,AckInN/x ; z/Next ;
                  AckInN/EndDMAInt
              alt DAckNormN,AckInN/x then z/- ; AckInN/ReadyN
           end
  end
end

Fork1 = forever do x/ReqOutN,z end
```

Petrify output:

```
# EQN file for model TSENDD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 63.00

INORDER = z StartDMASend DTCN DAckNormN DAckLastN AckInN x
ReadyN Next EndDMAInt DReqN csc0;
OUTORDER = [x] [ReadyN] [Next] [EndDMAInt] [DReqN] [csc0];
[0] = csc0' (DTCN DReqN' DAckNormN + DAckLastN');
[1] = AckInN' DTCN DReqN DAckLastN + csc0;
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = DAckLastN csc0' + z;
[4] = z' AckInN (DAckNormN' csc0 + DAckLastN');
[ReadyN] = [4]' ([3] + ReadyN) + ReadyN [3];     # mappable onto gC
[Next] = z' DReqN' csc0' + z DReqN;
[7] = AckInN z' DReqN DAckLastN csc0';
[EndDMAInt] = csc0' ([7] + EndDMAInt) + EndDMAInt [7];  # mappable onto gC
[9] = z' DAckLastN';
[10] = StartDMASend' DAckLastN csc0;
[DReqN] = [10]' ([9] + DReqN) + DReqN [9];     # mappable onto gC
[12] = AckInN' (DReqN' DAckLastN' + DAckNormN') + StartDMASend;
[csc0] = DTCN ([12] + csc0) + csc0 [12];     # mappable onto gC

# Set/reset pins: reset(x) set(DReqN) reset(csc0)
```

C.1.3 scsi trcv

The Burst-Mode specification of target send protocol for SCSI controller is shown in Figure C.3. The DISP description of the controller and its environment is given below. To mantain delay-insensitive behaviour an additional signal `Next` is added to the specification. This signal avoids transmission interference on the signal `DAckNormN` in transitions from state 4 to 5 and from state 5 to 6.

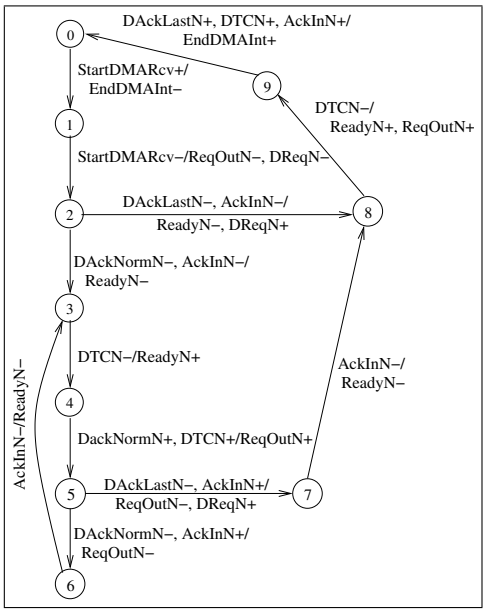


Figure C.3: scsi-trcv

```
R = forever do ReqOutN/AckInN end

T =
forever do -/StartDMARcV ; EndDMAInt/- end

L = pushback Next ;
forever do
  select Next,DReqN/DAckNormN then
         ReadyN/DTCN ;
         Next,ReadyN/DAckNormN,DTCN ;
         pushback DReqN
     alt Next,DReqN/DAckLastN then
         ReadyN,DReqN/DTCN ;
         Next,ReadyN/DAckLastN,DTCN
  end
end

#environment R par T par L

TRCV =
forever do
  select StartDMARcV/EndDMAInt then StartDMARcV/ReqOutN,DReqN ;
       select DAckLastN,AckInN/ReadyN,DReqN then DTCN/ReadyN,ReqOutN,Next ;
                DAckLastN,DTCN,AckInN/Next,EndDMAInt
          alt DAckNormN,AckInN/ReadyN
       end
     alt DTCN/ReadyN,Next then DAckNormN,DTCN/ReqOutN,Next ;
           select DAckLastN,AckInN/ReqOutN,DReqN then AckInN/ReadyN ;
                  DTCN/ReadyN,ReqOutN,Next ;
                  DAckLastN,DTCN,AckInN/Next,EndDMAInt
             alt DAckNormN,AckInN/ReqOutN then AckInN/ReadyN
           end
  end
end
```

Petrify output:

```
# EQN file for model TRCV
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 90.00

INORDER = StartDMARcV DTCN DAckNormN DAckLastN AckInN ReqOutN
ReadyN Next EndDMAInt DReqN csc0 csc1;
OUTORDER = [ReqOutN] [ReadyN] [Next] [EndDMAInt] [DReqN] [csc0] [csc1];
[0] = DTCN' DAckNormN csc0 + csc0' (DTCN (DAckNormN + Next) + DReqN);
[1] = DTCN csc0 StartDMARcV';
[ReqOutN] = [1]' ([0] + ReqOutN) + ReqOutN [0];     # mappable onto gC
[3] = csc1 (DAckNormN' + csc0');
[4] = DTCN AckInN' csc0 (DAckNormN' + csc1);
[ReadyN] = [4]' ([3] + ReadyN) + ReadyN [3];     # mappable onto gC
[6] = DTCN (DReqN' DAckNormN + csc1');
[7] = DTCN' (DAckLastN' + csc0');
[Next] = [7]' ([6] + Next) + Next [6];     # mappable onto gC
[9] = AckInN DTCN csc0' DReqN DAckLastN;
[10] = csc0 csc1';
[EndDMAInt] = [10]' ([9] + EndDMAInt) + EndDMAInt [9];  # mappable onto gC
[12] = DAckLastN' csc1 (csc0 + AckInN);
[13] = csc0 StartDMARcV' DAckLastN;
[DReqN] = [13]' ([12] + DReqN) + DReqN [12];     # mappable onto gC
[15] = AckInN (DTCN Next DAckLastN' + csc1' DAckNormN') + StartDMARcV;
[16] = AckInN' DAckLastN' + DTCN';
[17] = AckInN (DTCN DReqN DAckLastN + DAckNormN');
[csc1] = [17]' ([16] + csc1) + csc1 [16];     # mappable onto gC
[csc0] = DTCN ([15] + csc0) + csc0 [15];     # mappable onto gC

# Set/reset pins: set(ReadyN) set(DReqN) reset(csc0)
```

Decompositions applied were as follows:

```
#environment R par T par L par Fork1

TRCVD =
forever do
  select StartDMARcV/EndDMAInt then StartDMARcV/x ; z/ReqOutN ;
       select DAckLastN,AckInN/x then z/ReadyN ; DTCN/ReadyN,ReqOutN,Next ;
                DAckLastN,DTCN,AckInN/Next,EndDMAInt
          alt DAckNormN,AckInN/ReadyN
       end
     alt DTCN/ReadyN,Next then DAckNormN,DTCN/ReqOutN,Next ;
```

```
            select DAckLastN,AckInN/x then z/ReqOutN ; AckInN/ReadyN ;
                   DTCN/ReadyN,ReqOutN,Next ;
                   DAckLastN,DTCN,AckInN/Next,EndDMAInt
               alt DAckNormN,AckInN/ReqOutN then AckInN/ReadyN
            end
  end
end

Fork1 = forever do x/DReqN,z end
```

Petrify output:

```
# EQN file for model TRCVD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 70.00

INORDER = z StartDMARcV DTCN DAckNormN DAckLastN AckInN x
ReqOutN ReadyN Next EndDMAInt csc0;
OUTORDER = [x] [ReqOutN] [ReadyN] [Next] [EndDMAInt] [csc0];
[0] = DAckLastN StartDMARcV' csc0;
[1] = DAckLastN' (AckInN csc0' + AckInN' csc0);
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = DTCN csc0' (Next + DAckNormN) + z' DTCN';
[4] = csc0 (DAckLastN' DTCN x' + z);
[ReqOutN] = [4]' ([3] + ReqOutN) + ReqOutN [3];     # mappable onto gC
[6] = AckInN' csc0 (z' + DAckNormN');
[7] = DTCN (DAckNormN x + DAckLastN AckInN);
[8] = DAckLastN AckInN DTCN x' csc0';
[EndDMAInt] = csc0' ([8] + EndDMAInt) + EndDMAInt [8];  # mappable onto gC
[10] = DAckLastN' Next DTCN z' + DAckNormN' AckInN + StartDMARcV;
[ReadyN] = csc0' ([6]' + ReadyN) + ReadyN [6]';     # mappable onto gC
[Next] = DTCN ([7] + Next) + Next [7];     # mappable onto gC
[csc0] = DTCN ([10] + csc0) + csc0 [10];     # mappable onto gC

# Set/reset pins: reset(x) reset(csc0)
```

C.1.4 pipelined ircv

The Burst-Mode specification of non-block mode initiator receive protocol for pipelined SCSI controller is shown in Figure C.4. The DISP description of the controller and its environment is as follows:

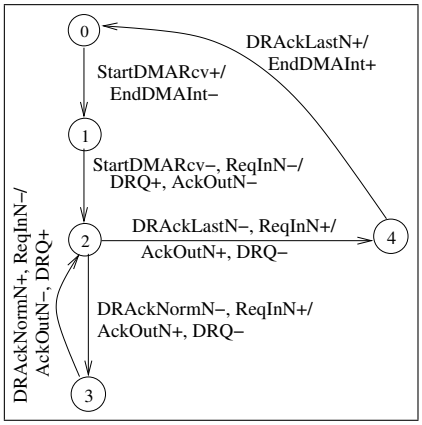


Figure C.4: pscsi-ircv

```
T =
forever do -/StartDMARcv ; EndDMAInt/- end

R = forever do -/ReqInN ; AckOutN/- end

L =
forever do
  select DRQ/DRAckNormN then DRQ/DRAckNormN
     alt DRQ/DRAckLastN then DRQ/DRAckLastN
  end
end

#environment T par R par L

PIRCV =
forever do
  select StartDMARcv/EndDMAInt then StartDMARcv,ReqInN/DRQ,AckOutN
     alt DRAckNormN,ReqInN/AckOutN,DRQ then DRAckNormN,ReqInN/AckOutN,DRQ
     alt DRAckLastN,ReqInN/AckOutN,DRQ then DRAckLastN/EndDMAInt
  end
end
```

Petrify output:

```
# EQN file for model PIRCV
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 37.00

INORDER = StartDMARcV ReqInN DRAckNormN DRAckLastN EndDMAInt
DRQ AckOutN csc0 csc1;
OUTORDER = [EndDMAInt] [DRQ] [AckOutN] [csc0] [csc1];
[0] = DRAckLastN csc0';
[1] = StartDMARcV csc0;
[EndDMAInt] = [1]' ([0] + EndDMAInt) + EndDMAInt [0];
                                          # mappable onto gC
[3] = csc1' (csc0' + DRAckNormN');
[DRQ] = [3]' (csc1 + DRQ) + DRQ csc1;     # mappable onto gC
```

```
[AckOutN] = csc1';
[6] = StartDMARcV' EndDMAInt' DRAckNormN DRAckLastN ReqInN' csc0;
[7] = ReqInN (DRAckLastN' + DRAckNormN');
[csc1] = [7]' ([6] + csc1) + csc1 [6];     # mappable onto gC
[csc0] = DRAckLastN (EndDMAInt + csc0) + EndDMAInt csc0;
                                          # mappable onto gC

# Set/reset pins: set(EndDMAInt) reset(DRQ) reset(csc1)
```

Decompositions applied were as follows:

```
#environment T par R par L par Fork1

PIRCVD =
forever do
  select StartDMARcV/EndDMAInt then StartDMARcV,ReqInN/x ; z/DRQ
     alt DRAckNormN,ReqInN/x then z/DRQ ;
         DRAckNormN,ReqInN/x ; z/DRQ
     alt DRAckLastN,ReqInN/x then z/DRQ ; DRAckLastN/EndDMAInt
  end
end

Fork1 = forever do x/AckOutN,z end
```

Petrify output:

```
# EQN file for model PIRCVD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 22.00

INORDER = z StartDMARcV ReqInN DRAckNormN DRAckLastN
x EndDMAInt DRQ csc0;
OUTORDER = [x] [EndDMAInt] [DRQ] [csc0];
[0] = DRAckNormN ReqInN' StartDMARcV' csc0;
[1] = DRAckNormN' ReqInN + csc0';
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[EndDMAInt] = DRAckLastN csc0';
[DRQ] = z;
[5] = ReqInN DRAckLastN';
[csc0] = [5]' (StartDMARcV + csc0) + StartDMARcV csc0;
                                       # mappable onto gC

# Set/reset pins: reset(csc0)
```

C.1.5 pipelined trcv

The Burst-Mode specification of non-block mode target receive protocol for pipelined SCSI controller is shown in Figure C.5. The DISP description of the controller and its environment is as follows:

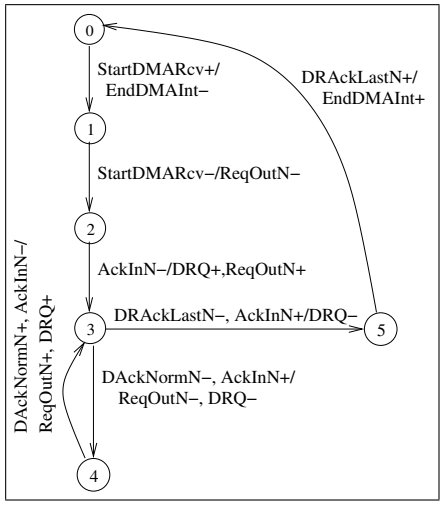


Figure C.5: pscsi-trcv

```
T =
forever do -/StartDMARcv ; EndDMAInt/- end

R = forever do ReqOutN/AckInN end

L =
forever do
  select DRQ/DRAckNormN then DRQ/DRAckNormN
     alt DRQ/DRAckLastN then DRQ/DRAckLastN
  end
end

#environment T par R par L

PTRCV =
forever do
  select StartDMARcv/EndDMAInt then
           StartDMARcv/ReqOutN ; AckInN/DRQ,ReqOutN
     alt DRAckNormN,AckInN/ReqOutN,DRQ then
           DRAckNormN,AckInN/ReqOutN,DRQ
     alt DRAckLastN,AckInN/DRQ then DRAckLastN/EndDMAInt
  end
end
```

Petrify output:

```
Error: CSC cannot be solved.
No irreducible CSC conflicts found.
```

Decompositions applied were as follows:

```
#environment T par R par L par Fork1

PTRCVD =
forever do
  select StartDMARcV/EndDMAInt then
           StartDMARcV/ReqOutN ; AckInN/x ; z/ReqOutN
     alt DRAckNormN,AckInN/x then z/ReqOutN ;
```

```
          DRAckNormN,AckInN/x ; z/ReqOutN
     alt DRAckLastN,AckInN/x then z/- ; DRAckLastN/EndDMAInt
  end
end

Fork1 = forever do x/DRQ,z end
```

Petrify output:

```
# EQN file for model PTRCVD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 31.00

INORDER = z StartDMARcV DRAckNormN DRAckLastN AckInN x
ReqOutN EndDMAInt csc0;
OUTORDER = [x] [ReqOutN] [EndDMAInt] [csc0];
[0] = DRAckNormN AckInN';
[1] = AckInN (csc0' + DRAckNormN');
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = z' StartDMARcV' csc0;
[ReqOutN] = [3]' (z + ReqOutN) + z ReqOutN;   # mappable onto gC
[5] = DRAckLastN z' csc0';
[EndDMAInt] = csc0' ([5] + EndDMAInt) + EndDMAInt [5];
                                              # mappable onto gC
[csc0] = DRAckLastN (StartDMARcV + csc0) + StartDMARcV csc0;
                                              # mappable onto gC

# Set/reset pins: set(ReqOutN) reset(csc0)
```

C.1.6 pipelined isend

The Burst-Mode specification of non-block mode initiator send protocol for pipelined SCSI controller is shown in Figure C.6. The DISP description of the controller and its environment is as follows:

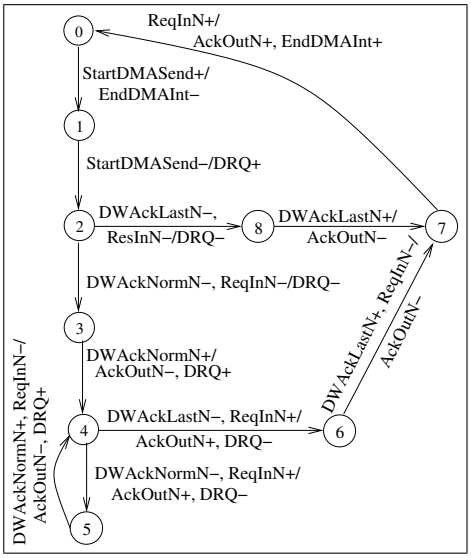


Figure C.6: pscsi-isend

```
T =
forever do -/StartDMASend ; EndDMAInt/- end

R = forever do -/ReqInN ; AckOutN/- end

L =
forever do
  select DRQ/DWAckNormN then DRQ/DWAckNormN
     alt DRQ/DWAckLastN then DRQ/DWAckLastN
  end
end

#environment T par R par L

PISEND =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DRQ ;
         select DWAckNormN,ReqInN/DRQ then DWAckNormN/AckOutN,DRQ
            alt DWAckLastN,ReqInN/DRQ then DWAckLastN/AckOutN ;
                ReqInN/AckOutN,EndDMAInt
         end
     alt DWAckNormN,ReqInN/AckOutN,DRQ then
             DWAckNormN,ReqInN/AckOutN,DRQ
     alt DWAckLastN,ReqInN/AckOutN,DRQ then DWAckLastN,ReqInN/AckOutN ;
             ReqInN/AckOutN,EndDMAInt
  end
end
```

Petrify output:

```
# EQN file for model PISEND
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 83.00

INORDER = StartDMASend ReqInN DWAckNormN DWAckLastN EndDMAInt
DRQ AckOutN csc0 csc1 csc2 csc3 csc4;
OUTORDER = [EndDMAInt] [DRQ] [AckOutN] [csc0] [csc1] [csc2] [csc3] [csc4];
```

```
[0] = csc2' (AckOutN + ReqInN);
[EndDMAInt] = csc2' ([0] + EndDMAInt) + EndDMAInt [0];  # mappable onto gC
[2] = csc1 StartDMASend' csc2 csc3';
[3] = csc4 (ReqInN' csc1' + csc0);
[DRQ] = [3]' ([2] + DRQ) + DRQ [2];     # mappable onto gC
[5] = csc4 (ReqInN csc1 + DWAckLastN' + csc0);
[6] = csc0' csc1' DWAckLastN + csc3' csc4';
[AckOutN] = [6]' ([5] + AckOutN) + AckOutN [5];     # mappable onto gC
[8] = ReqInN' csc1' DWAckLastN;
[csc0] = [8]' (csc3 + csc0) + csc0 csc3;     # mappable onto gC
[10] = csc1 StartDMASend;
[11] = AckOutN' csc0' DRQ' csc1';
[csc2] = [11]' ([10] + csc2) + csc2 [10];     # mappable onto gC
[13] = DWAckNormN' (ReqInN csc4' + ReqInN' csc4);
[14] = DWAckNormN csc4';
[csc3] = [14]' ([13] + csc3) + csc3 [13];     # mappable onto gC
[16] = ReqInN (csc3 + csc1');
[17] = ReqInN' DWAckNormN csc3;
[csc4] = [17]' ([16] + csc4) + csc4 [16];     # mappable onto gC
[csc1] = DWAckLastN (csc2' + csc1) + csc1 csc2';     # mappable onto gC

# Set/reset pins: reset(DRQ) reset(csc0) reset(csc2) reset(csc3) set(csc4)
```

Decompositions applied were as follows:

```
#environment T par R par L par Fork1 par Fork2

PISENDD =
forever do
  select StartDMASend/y ; t/- ; StartDMASend/DRQ ;
         select DWAckNormN,ReqInN/DRQ ; DWAckNormN/x ; z/DRQ
            alt DWAckLastN,ReqInN/DRQ ; DWAckLastN/x ; z/- ;
                ReqInN/x ; z/y ; t/-
         end
     alt DWAckNormN,ReqInN/x ; z/DRQ ;
             DWAckNormN,ReqInN/x ; z/DRQ
     alt DWAckLastN,ReqInN/x ; z/DRQ ; DWAckLastN,ReqInN/x ;z/- ;
             ReqInN/x ; z/y ; t/-
  end
end

Fork1 = forever do x/AckOutN,z end
Fork2 = forever do y/EndDMAInt,t end
```

Petrify output:

```
# EQN file for model PISENDD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 55.00

INORDER = z t StartDMASend ReqInN DWAckNormN DWAckLastN y x DRQ csc0 csc1;
OUTORDER = [y] [x] [DRQ] [csc0] [csc1];
[0] = z' csc0' csc1;
[1] = ReqInN' DWAckLastN DWAckNormN csc0;
[2] = ReqInN (csc0' + DWAckNormN' + DWAckLastN');
[x] = [2]' ([1] + x) + x [1];     # mappable onto gC
[4] = csc1' (t StartDMASend' csc0' + z);
[5] = DWAckNormN' z' csc0 + csc1;
[DRQ] = [5]' ([4] + DRQ) + DRQ [4];     # mappable onto gC
[7] = ReqInN' (DWAckNormN' + DWAckLastN');
[8] = z csc1;
[csc0] = [8]' ([7] + csc0) + csc0 [7];     # mappable onto gC
[10] = DWAckLastN' z' csc0;
[11] = t' StartDMASend;
[csc1] = [11]' ([10] + csc1) + csc1 [10];     # mappable onto gC
[y] = csc1' ([0]' + y) + y [0]';     # mappable onto gC

# Set/reset pins: reset(csc0) set(csc1)
```

C.1.7 pipelined tsend

The Burst-Mode specification of non-block mode target send protocol for pipelined SCSI controller is shown in Figure C.7. The DISP description of the controller and its environment is as follows:

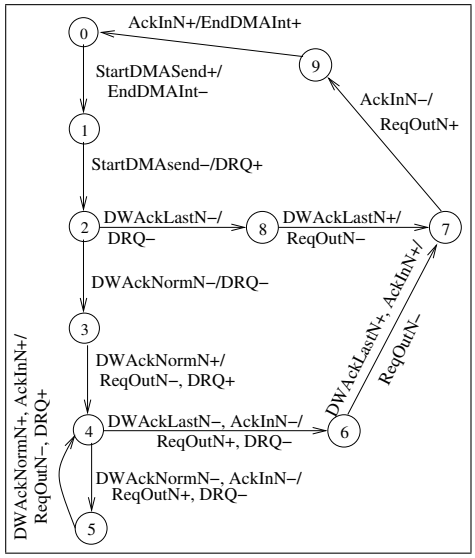


Figure C.7: pscsi-tsend

```
T =
forever do -/StartDMASend ; EndDMAInt/- end

R = forever do ReqOutN/AckInN end

L =
forever do
  select DRQ/DWAckNormN then DRQ/DWAckNormN
     alt DRQ/DWAckLastN then DRQ/DWAckLastN
  end
end

#environment T par R par L

PTSEND =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DRQ ;
         select DWAckNormN/DRQ then DWAckNormN/ReqOutN,DRQ
            alt DWAckLastN/DRQ then DWAckLastN/ReqOutN ; AckInN/ReqOutN;
                AckInN/EndDMAInt
         end
     alt DWAckNormN,AckInN/ReqOutN,DRQ then
             DWAckNormN,AckInN/ReqOutN,DRQ
     alt DWAckLastN,AckInN/ReqOutN,DRQ then DWAckLastN,AckInN/ReqOutN ;
             AckInN/ReqOutN; AckInN/EndDMAInt
  end
end
```

Petrify output:

```
# EQN file for model PTSEND
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 92.00

INORDER = StartDMASend DWAckNormN DWAckLastN AckInN ReqOutN
EndDMAInt DRQ csc0 csc1 csc2 csc3;
OUTORDER = [ReqOutN] [EndDMAInt] [DRQ] [csc0] [csc1] [csc2] [csc3];
```

```
[0] = csc2 csc3' + csc2' csc3;
[1] = DWAckLastN csc2' csc3' csc0' + csc2 csc3;
[ReqOutN] = [1]' ([0] + ReqOutN) + ReqOutN [0];     # mappable onto gC
[3] = AckInN csc2 csc0';
[4] = csc0 (csc2' csc3' StartDMASend' + csc2 csc3);
[5] = DWAckNormN' csc2' csc3 + csc2 csc3' + csc1 csc0';
[DRQ] = [5]' ([4] + DRQ) + DRQ [4];     # mappable onto gC
[7] = DWAckLastN' (csc2' + AckInN');
[csc0] = [7]' (StartDMASend + csc0) + StartDMASend csc0;
                                          # mappable onto gC
[9] = ReqOutN' csc2;
[10] = DRQ' csc0 (EndDMAInt' csc2 csc3' + csc2' csc3);
[csc1] = [10]' ([9] + csc1) + csc1 [9];     # mappable onto gC
[12] = DWAckNormN AckInN csc1' + csc3' (AckInN' + DWAckNormN');
[13] = csc1 (AckInN' (DWAckNormN' + DWAckLastN') + csc0' csc3)
       + StartDMASend csc0;
[csc2] = [13]' ([12] + csc2) + csc2 [12];     # mappable onto gC
[15] = DWAckNormN csc1';
[16] = DWAckLastN AckInN csc0';
[csc3] = [16]' ([15] + csc3) + csc3 [15];     # mappable onto gC
[EndDMAInt] = csc2 ([3] + EndDMAInt) + EndDMAInt [3];
                                           # mappable onto gC

# Set/reset pins: reset(csc0) set(csc1) set(csc2)
```

Decompositions applied were as follows:

```
#environment T par R par L par Fork1

PTSENDD =
forever do
  select StartDMASend/EndDMAInt then StartDMASend/DRQ ;
         select DWAckNormN/DRQ then DWAckNormN/x ; z/DRQ
            alt DWAckLastN/DRQ then DWAckLastN/x ; z/- ; AckInN/x ; z/-;
                AckInN/EndDMAInt
         end
     alt DWAckNormN,AckInN/x then z/DRQ ;
             DWAckNormN,AckInN/x ; z/DRQ
     alt DWAckLastN,AckInN/x then z/DRQ ; DWAckLastN,AckInN/x ; z/- ;
             AckInN/x ; z/- ; AckInN/EndDMAInt
  end
end

Fork1 = forever do x/ReqOutN,z end
```

Petrify output:

```
# EQN file for model PTSENDD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 69.00

INORDER = z StartDMASend DWAckNormN DWAckLastN AckInN x
EndDMAInt DRQ csc0 csc1 csc2;
OUTORDER = [x] [EndDMAInt] [DRQ] [csc0] [csc1] [csc2];
[0] = DWAckLastN csc1' (DWAckNormN AckInN csc2 + csc0);
[1] = AckInN' (DWAckLastN' csc2' + DWAckNormN') + csc1;
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = DWAckLastN AckInN z' csc0 csc1;
[4] = csc0' (StartDMASend' csc1 + z);
[5] = z' csc1';
[DRQ] = [5]' ([4] + DRQ) + DRQ [4];     # mappable onto gC
[7] = DWAckLastN AckInN csc1' csc2' + DWAckLastN' csc1;
[csc0] = StartDMASend' ([7] + csc0) + csc0 [7];  # mappable onto gC
[9] = AckInN' z csc0;
[10] = DWAckLastN' csc0 + DWAckNormN' csc2;
[csc1] = [10]' ([9] + csc1) + csc1 [9];     # mappable onto gC
[12] = csc1 + csc0;
[13] = DWAckLastN' z;
[csc2] = [13]' ([12] + csc2) + csc2 [12];     # mappable onto gC
[EndDMAInt] = csc0 ([3] + EndDMAInt) + EndDMAInt [3]; # mappable onto gC

# Set/reset pins: reset(DRQ) set(csc0) set(csc1)
```

C.1.8 fast isend

The Burst-Mode specification of fast SCSI initiator send protocol for non-pipelined SCSI controller is shown in Figure C.8. The DISP description of the controller and its environment is as follows:

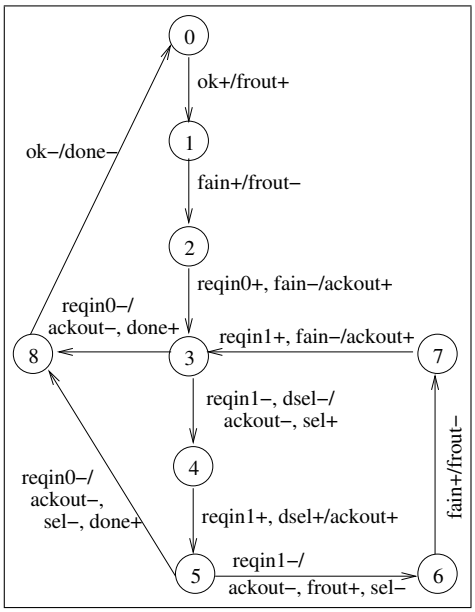


Figure C.8: fast-isend

```
L = forever do frout/fain end

M = forever do -/ok ; done/- end

R = -/reqin0 ;
forever do
  select ackout/reqin0 then ackout/reqin0
     alt ackout/reqin1 then ackout/reqin1
  end
end

B = forever do -/dsel ; sel/- end

Dum = stop ; -/s3

#environment L par M par R par B par Dum

FASTVAR =
forever do
 select ok/frout then fain/frout ; reqin0,fain/ackout ; pushback s3
    alt s3,reqin1,dsel/ackout,sel then reqin1,dsel/ackout ;
        select reqin1/ackout,frout,sel then fain/frout ;
               reqin1,fain/ackout ; pushback s3
           alt reqin0/done,ackout,sel then ok/done
        end
    alt s3,reqin0/done,ackout then ok/done
 end
end
```

Petrify output:

```
Error: CSC cannot be solved.
No irreducible CSC conflicts found.
Warning: non-commutative behavior between reqin1+ and fain+
Warning: non-commutative behavior between reqin1+ and dsel-
Warning: non-commutative behavior between reqin0+ and ok+
Warning: non-commutative behavior between reqin0+ and ok-
Warning: non-commutative behavior between reqin0+ and fain+
```

```
Warning: non-commutative behavior between reqin0+ and dsel-
Warning: non-commutative behavior between reqin0- and dsel-
Warning: non-commutative behavior between ok+ and dsel-
Warning: non-commutative behavior between ok- and dsel-
Warning: non-commutative behavior between fain+ and dsel-
Warning: non-commutative behavior between fain- and dsel-
Input non-commutativity produces irreducible CSC conflicts.
```

Decompositions applied were as follows:

```
#environment L par M par R par B par Dum par Fork1 par Fork2

FASTVARD =
forever do
 select ok/frout then fain/frout ; reqin0,fain/y ; t/- ; pushback s3
    alt s3,reqin1,dsel/x then z/y ; t/- ; reqin1,dsel/y ; t/- ;
        select reqin1/x then z/y ; t/frout ; fain/frout ;
               reqin1,fain/y ; t/- ; pushback s3
           alt reqin0/x then z/y ; t/done ; ok/done
        end
    alt s3,reqin0/y then t/done ; ok/done
 end
end

Fork1 = forever do x/sel,z end
Fork2 = forever do y/ackout,t end
```

Petrify output:

```
# EQN file for model FASTVARD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 79.00

INORDER = z t reqin1 reqin0 ok fain dsel y x frout done csc0 csc1 csc2 csc3;
OUTORDER = [y] [x] [frout] [done] [csc0] [csc1] [csc2] [csc3];
[0] = csc1' csc3 (t' dsel csc2 + x');
[1] = z csc2' + csc3';
[y] = [1]' ([0] + y) + y [0];     # mappable onto gC
[3] = t csc2';
[4] = reqin0' t + csc1;
[x] = [4]' ([3] + x) + x [3];     # mappable onto gC
[6] = t' ok csc0' csc1;
[frout] = csc0' ([6] + frout) + frout [6];     # mappable onto gC
[done] = csc0' csc1';
```

```
[9] = t' csc1' csc3' + y z' csc1;
[csc0] = [9]' (fain + csc0) + fain csc0;     # mappable onto gC
[11] = reqin1' t z csc2 + ok';
[12] = reqin1 csc3;
[csc1] = [12]' ([11] + csc1) + csc1 [11];     # mappable onto gC
[14] = reqin1' dsel' csc1';
[csc2] = [14]' (reqin1 + csc2) + reqin1 csc2;     # mappable onto gC
[16] = reqin0 fain' csc0 csc1;
[17] = reqin0' t z' + csc0';
[csc3] = [17]' ([16] + csc3) + csc3 [16];     # mappable onto gC

# Set/reset pins: reset(frout) reset(csc0)
```

C.1.9 sbuf send

The Burst-Mode specification of SBUF-SEND protocol is shown in Figure C.9. The DISP description of the controller and its environment is given below. To mantain delay-insensitive behaviour an additional signal NEXT is added to the specification. This signal avoids transmission interference on the signal REJPKT in transitions from state 4 to 6 and from state 6 to 7.
Note that for sbuf-read-ctl the specfication has no CSC conflicts and there no decomposition performed.

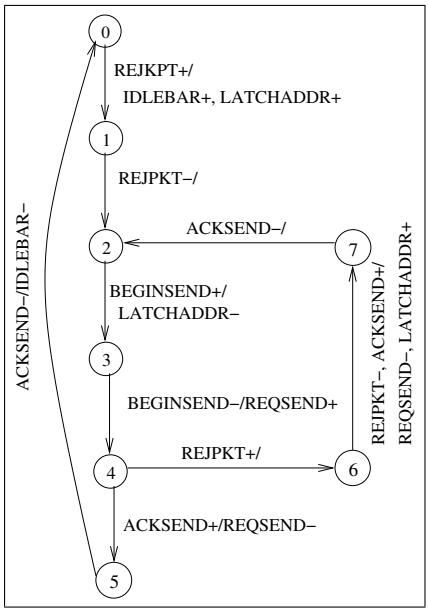


Figure C.9: sbuf send

```
R = forever do LATCHADDR/BEGINSEND end

L = pushback e0 ;
forever do
  select e0/REJPKT then NEXT,IDLEBAR/REJPKT ;
         pushback e2
     alt e2,NEXT,REQSEND/ACKSEND then
         REQSEND/ACKSEND ; IDLEBAR/- ;
         pushback e0
     alt e2,NEXT,REQSEND/REJPKT then
         NEXT/REJPKT,ACKSEND ;
         REQSEND/ACKSEND ; pushback e2
  end
end

Edum = stop ; -/e0,e2

#environment R par L par Edum

SBUFSD =
forever do
  select REJPKT/IDLEBAR,LATCHADDR,NEXT then REJPKT/NEXT
     alt BEGINSEND/LATCHADDR then BEGINSEND/REQSEND ;
         select ACKSEND/REQSEND then ACKSEND/IDLEBAR
            alt REJPKT/NEXT then REJPKT,ACKSEND/REQSEND,LATCHADDR,NEXT ;
                ACKSEND/-
         end
  end
end
```

Petrify output:

```
# EQN file for model SBUFSD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 72.00

INORDER = REJPKT BEGINSEND ACKSEND REQSEND NEXT LATCHADDR
IDLEBAR csc0 csc1 csc2;
OUTORDER = [REQSEND] [NEXT] [LATCHADDR] [IDLEBAR] [csc0] [csc1] [csc2];
[0] = REJPKT csc2;
[1] = ACKSEND (csc1' csc2 + REJPKT' NEXT) + csc0' (REJPKT + csc1' + NEXT);
[2] = REQSEND' csc0;
[LATCHADDR] = [2]' ([1] + LATCHADDR) + LATCHADDR [1];  # mappable onto gC
[4] = NEXT' csc1 REJPKT' ACKSEND' REQSEND';
[IDLEBAR] = [4]' (REJPKT + IDLEBAR) + REJPKT IDLEBAR;  # mappable onto gC
[6] = BEGINSEND REJPKT' ACKSEND';
[7] = REJPKT' ACKSEND csc2;
[csc0] = [7]' ([6] + csc0) + csc0 [6];     # mappable onto gC
[9] = NEXT' REQSEND csc2';
[10] = NEXT (REJPKT' (REQSEND' + ACKSEND) + csc2');
[csc1] = [10]' ([9] + csc1) + csc1 [9];     # mappable onto gC
[12] = NEXT' csc1;
[13] = BEGINSEND' REQSEND' csc0;
[csc2] = [13]' ([12] + csc2) + csc2 [12];     # mappable onto gC
[NEXT] = csc1 ([0] + NEXT) + NEXT [0];     # mappable onto gC
[REQSEND] = csc0 (csc2' + REQSEND) + REQSEND csc2';   # mappable onto gC

# Set/reset pins: reset(LATCHADDR) reset(csc0) set(csc1)
  set([12]) reset(NEXT)
```

Decompositions applied were as follows:

```
#environment R par L par Edum par Fork1

SBUFSDD =
forever do
  select REJPKT/x then z/IDLEBAR,NEXT ; REJPKT/NEXT
     alt BEGINSEND/x then z/- ; BEGINSEND/REQSEND ;
         select ACKSEND/REQSEND then ACKSEND/IDLEBAR
            alt REJPKT/NEXT ; REJPKT,ACKSEND/x ; z/REQSEND,NEXT ; ACKSEND/-
         end
  end
end

Fork1 = forever do x/LATCHADDR,z end
```

Petrify output:

```
# EQN file for model SBUFSDD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 55.00

INORDER = z REJPKT BEGINSEND ACKSEND x REQSEND NEXT IDLEBAR csc0;
OUTORDER = [x] [REQSEND] [NEXT] [IDLEBAR] [csc0];
[0] = REJPKT' NEXT ACKSEND + REJPKT IDLEBAR';
[1] = REJPKT' BEGINSEND ACKSEND';
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = BEGINSEND' z' csc0';
[4] = REJPKT' NEXT' csc0 + z;
[REQSEND] = [4]' ([3] + REQSEND) + REQSEND [3];     # mappable onto gC
[6] = REJPKT (REQSEND csc0 + z);
[7] = REJPKT' NEXT' ACKSEND' csc0;
[IDLEBAR] = [7]' (z + IDLEBAR) + z IDLEBAR;     # mappable onto gC
[9] = ACKSEND z' + REJPKT;
[10] = REJPKT' (NEXT REQSEND' + z);
[csc0] = [10]' ([9] + csc0) + csc0 [9];     # mappable onto gC
[NEXT] = csc0 ([6] + NEXT) + NEXT [6];     # mappable onto gC

# Set/reset pins: reset(x) set(csc0) reset(NEXT)
```

C.2 Circuits Related to Self-contained Blocks

C.2.1 loadable counter

An N-bit loadable counter can take input a number m, in response to which it does m handshakes. The DISP description is given as follows:

```
LEFT =
forever do
   select loadL/- then pushback reqL
      alt reqL/ackL_true
      alt reqL/ackL_false
   end
end

RIGHT =
pushback ackR_false;
forever do
   select ackR_false/load_true,loadL
      alt ackR_false/load_false,loadL
      alt ackR_true/reqR
   end
end

#environment LEFT par RIGHT

CELL =
forever do
   select load_true/ackR_true
      alt load_false/- then pushback reqR
      alt reqR,ackL_true/reqL,ackR_true then reqR/ackR_true
      alt reqR,ackL_false/ackR_false
   end
end
```

Petrify output:

```
Error: CSC cannot be solved.
No irreducible CSC conflicts found.
Warning: non-commutative behavior between reqR+ and ackL_true+
Warning: non-commutative behavior between reqR- and ackL_true+
Warning: non-commutative behavior between reqR+ and ackL_true-
Warning: non-commutative behavior between reqR- and ackL_true-
Warning: non-commutative behavior between reqR+ and ackL_false+
Warning: non-commutative behavior between reqR- and ackL_false+
Warning: non-commutative behavior between reqR+ and ackL_false-
```

```
Warning: non-commutative behavior between reqR- and ackL_false-
Warning: non-commutative behavior between load_true+ and ackL_true+
Warning: non-commutative behavior between load_true- and ackL_true+
Warning: non-commutative behavior between load_true+ and ackL_true-
Warning: non-commutative behavior between load_true- and ackL_true-
Warning: non-commutative behavior between load_true+ and ackL_false+
Warning: non-commutative behavior between load_true- and ackL_false+
Warning: non-commutative behavior between load_true+ and ackL_false-
Warning: non-commutative behavior between load_true- and ackL_false-
Warning: non-commutative behavior between load_false+ and ackL_true+
Warning: non-commutative behavior between load_false- and ackL_true+
Warning: non-commutative behavior between load_false+ and ackL_true-
Warning: non-commutative behavior between load_false- and ackL_true-
Warning: non-commutative behavior between load_false+ and ackL_false+
Warning: non-commutative behavior between load_false- and ackL_false+
Warning: non-commutative behavior between load_false+ and ackL_false-
Warning: non-commutative behavior between load_false- and ackL_false-
Input non-commutativity produces irreducible CSC conflicts.
```

Decompositions applied were as follows:

```
#environment LEFT par RIGHT par Wire1 par Wire2

CELLD =
forever do
  select load_true/ackR_true
       alt load_false/- then pushback reqR
       alt reqR,ackL_true/y then t/reqL,ackR_true ; reqR/x ; z/ackR_true
       alt reqR,ackL_false/ackR_false
  end
end

Wire1 = forever do y/t end
Wire2 = forever do x/z end
```

Petrify output:

```
# EQN file for model CELLD
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 78.00

INORDER = z t reqR load_true load_false ackL_true ackL_false
y x reqL ackR_true ackR_false;
OUTORDER = [y] [x] [reqL] [ackR_true] [ackR_false];
```

```
[0] = z' ackL_true (ackL_false (reqR' load_false' + reqR load_false)
     + ackL_false' (reqR' load_false + reqR load_false'));
[1] = z ackL_true' (ackL_false (reqR' load_false' + reqR load_false)
     + ackL_false' (reqR' load_false + reqR load_false'));
[y] = [1]' ([0] + y) + y [0];     # mappable onto gC
[3] = t (reqR load_true' + reqR' load_true);
[4] = t' (reqR load_true' + reqR' load_true);
[x] = [4]' ([3] + x) + x [3];     # mappable onto gC
[reqL] = t;
[ackR_true] = t (z' load_true' + z load_true)
            + t' (load_true z' + load_true' z);
[8] = ackL_false (x' reqL' + x reqL)(reqR load_false' + reqR' load_false);
[9] = ackL_false' (x' reqL' + x reqL)(reqR' load_false' + reqR load_false);
[ackR_false] = [9]' ([8] + ackR_false) + ackR_false [8];
                                  # mappable onto gC

# Set/reset pins: reset(y) reset(x)
```

C.2.2 dme-fast-e

The Burst-Mode specification of DME Fast controller is shown in Figure C.10. The DISP description of the controller and its environment is as follows:

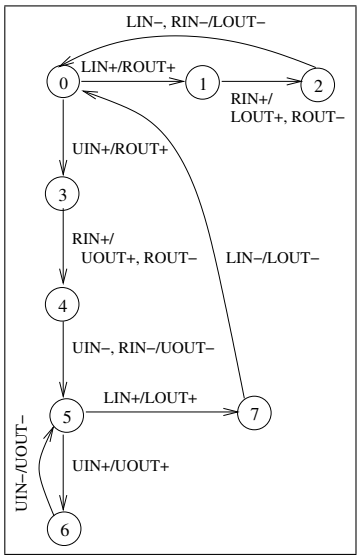


Figure C.10: dme-fast-e

```
L =
forever do
  select -/UIN then UOUT/UIN ; UOUT/-
     alt -/LIN then LOUT/LIN ; LOUT/-
  end
end

R = forever do ROUT/RIN end

Dum = stop ; -/s0,s5

#environment L par R par Dum

DME = pushback s0 ;
forever do
  select s0,LIN/ROUT then RIN/LOUT,ROUT ; LIN,RIN/LOUT ; pushback s0
     alt s0,UIN/ROUT then RIN/UOUT,ROUT ; UIN,RIN/UOUT ; pushback s5
     alt s5,UIN/UOUT then UIN/UOUT ; pushback s5
     alt s5,LIN/LOUT then LIN/LOUT ; pushback s0
  end
end
```

Petrify output:

```
# EQN file for model DME
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 46.00

INORDER = UIN RIN LIN UOUT ROUT LOUT csc0 csc1;
OUTORDER = [UOUT] [ROUT] [LOUT] [csc0] [csc1];
[0] = UIN csc0;
[1] = RIN' UIN';
[UOUT] = [1]' ([0] + UOUT) + UOUT [0];     # mappable onto gC
[3] = csc0' (LIN csc1 + UIN);
[4] = csc1' + csc0;
[ROUT] = [4]' ([3] + ROUT) + ROUT [3];     # mappable onto gC
[6] = csc0' csc1;
[7] = RIN UIN;
[8] = LOUT csc1;
[csc0] = [8]' ([7] + csc0) + csc0 [7];     # mappable onto gC
```

```
[10] = RIN' LIN';
[11] = LIN (csc0 + RIN);
[csc1] = [11]' ([10] + csc1) + csc1 [10];     # mappable onto gC
[LOUT] = csc1' ([6]' + LOUT) + LOUT [6]';     # mappable onto gC

# Set/reset pins: reset(ROUT) reset(csc0)
```

Decompositions applied were as follows:

```
#environment L par R par Dum par Wire1

DMED = pushback s0 ;
forever do
  select s0,LIN/ROUT ; RIN/x1 ; y1/LOUT,ROUT ;
         LIN,RIN/x1 ; y1/LOUT ; pushback s0
     alt s0,UIN/ROUT ; RIN/x1 ; y1/UOUT,ROUT ;
         UIN,RIN/UOUT ; pushback s5
     alt s5,UIN/UOUT ; UIN/UOUT ; pushback s5
     alt s5,LIN/LOUT ; LIN/x1; y1/LOUT ; pushback s0
  end
end

Wire1 = forever do x1/y1 end
```

Petrify output:

```
# EQN file for model DMED
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 32.00

INORDER = y1 UIN RIN LIN x1 UOUT ROUT LOUT;
OUTORDER = [x1] [UOUT] [ROUT] [LOUT];
[0] = LIN' RIN' LOUT;
[x1] = [0]' (RIN + x1) + RIN x1;     # mappable onto gC
[2] = UIN y1;
[3] = UIN' RIN';
[UOUT] = [3]' ([2] + UOUT) + UOUT [2];     # mappable onto gC
[5] = y1' (LIN + UIN);
[ROUT] = y1' ([5] + ROUT) + ROUT [5];     # mappable onto gC
[7] = LIN y1;
[LOUT] = y1 ([7] + LOUT) + LOUT [7];     # mappable onto gC

# Set/reset pins: reset(x1) reset(ROUT)
```

C.2.3 mod-2 counter

```
E = forever do  -/a ; select b/- alt c/- end end

#environment E
MOD2 =
forever do
  a/b ; a/b ; a/c ; a/c
end
```

Petrify output:

```
# EQN file for model MOD2
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 23.00

INORDER = a c b csc0;
OUTORDER = [c] [b] [csc0];
[0] = a' csc0;
[1] = a' csc0';
[b] = csc0' ([1] + b) + b [1];     # mappable onto gC
[3] = a b;
[4] = a c;
[csc0] = [4]' ([3] + csc0) + csc0 [3];     # mappable onto gC
[c] = csc0 ([0] + c) + c [0];     # mappable onto gC
# The initial state is unstable. No reset information generated.
# Signal b enabled in the initial state.
```

Decompositions applied were as follows:

```
#environment E par Wire1

MOD2D =
forever do
  a/b ; a/x ; y/b ;
  a/c ; a/x ; y/c
end

Wire1 = forever do x/y end
```

Petrify output:

```
# EQN file for model MOD2D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 23.00

INORDER = y a x c b;
OUTORDER = [x] [c] [b];
[0] = a' b;
[1] = a' c;
[x] = [1]' ([0] + x) + x [0];     # mappable onto gC
[3] = a y;
[4] = a y';
[b] = y' ([4] + b) + b [4];     # mappable onto gC
[c] = y ([3] + c) + c [3];     # mappable onto gC

# Set/reset pins: reset(x) reset(b)
```

C.2.4 mod-3 counter

```
E = forever do -/a ; select b/- alt c/- end end

#environment E
M3 =
forever do
  a/b ; a/b ;
  a/b ; a/b ;
  a/c ; a/c
end
```

Petrify output:

```
# EQN file for model M3
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 42.00

INORDER = a c b csc0 csc1 csc2;
OUTORDER = [c] [b] [csc0] [csc1] [csc2];
[0] = a csc0';
[c] = csc0' ([0] + c) + c [0];     # mappable onto gC
[2] = csc0 (a csc1 + csc2);
[3] = csc1' csc2' + csc0';
[b] = [3]' ([2] + b) + b [2];     # mappable onto gC
[5] = a' csc2';
[6] = a' csc2;
[csc0] = [6]' ([5] + csc0) + csc0 [5];     # mappable onto gC
[8] = a' b;
[csc1] = [8]' (c + csc1) + c csc1;     # mappable onto gC
[10] = a csc1';
[csc2] = csc1' ([10] + csc2) + csc2 [10];     # mappable onto gC

# Set/reset pins: reset(b) set(csc1)
```

Decompositions applied were as follows:

```
#environment E par Wire1 par Wire2

M3D =
forever do
  a/b ; a/x ; y/b ;
  a/z ; t/b ; a/x ; y/b ;
  a/c ; a/z ; t/c
```

```
end

Wire1 = forever do x/y end
Wire2 = forever do z/t end
```

Petrify output:

```
# EQN file for model M3D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 41.00

INORDER = y t a z x c b;
OUTORDER = [z] [x] [c] [b];
[0] = a y;
[1] = a' c;
[z] = [1]' ([0] + z) + z [0];     # mappable onto gC
[3] = a' b t';
[4] = a' t;
[x] = [4]' ([3] + x) + x [3];     # mappable onto gC
[6] = a z x';
[7] = a t' y' + t y;
[8] = t y' + t' y;
[b] = [8]' ([7] + b) + b [7];     # mappable onto gC
[c] = t ([6] + c) + c [6];     # mappable onto gC

# Set/reset pins: reset(z) reset(x) reset(b)
```

C.2.5 mod-4 counter

```
E = forever do -/a ; select b/- alt c/- end end

#environment E
M4 =
forever do
  a/b ; a/b ;
  a/b ; a/b ;
  a/b ; a/b ;
  a/c ; a/c
end
```

Petrify output:

```
# EQN file for model M4
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 70.00

INORDER = a c b csc0 csc1 csc2 csc3 csc4;
OUTORDER = [c] [b] [csc0] [csc1] [csc2] [csc3] [csc4];
[0] = a csc0';
[c] = csc0' ([0] + c) + c [0];     # mappable onto gC
[2] = csc3 csc2 csc0 csc4 + csc3' csc4' + csc1';
[3] = csc1 (csc2 csc0' + csc2' csc0) + csc3' csc4;
[b] = [3]' ([2] + b) + b [2];     # mappable onto gC
[5] = a' csc1';
[6] = a csc2';
[7] = csc0' + csc1';
[8] = a' csc3 csc1 csc0 csc4;
[csc2] = [8]' ([7] + csc2) + csc2 [7];     # mappable onto gC
[10] = a csc4;
[11] = a csc4';
[csc3] = [11]' ([10] + csc3) + csc3 [10];     # mappable onto gC
[13] = a' csc3';
[14] = a' c;
[csc4] = [14]' ([13] + csc4) + csc4 [13];     # mappable onto gC
[csc0] = csc4' ([5]' + csc0) + csc0 [5]';     # mappable onto gC
[csc1] = csc0' ([6]' + csc1) + csc1 [6]';     # mappable onto gC

# Set/reset pins: reset(b) set(csc2) set(csc3)
reset(csc4) set(csc0) set(csc1)
```

Decompositions applied were as follows:

```
#environment E par Wire1 par Wire2 par Wire3

M4D =
forever do
  a/b  ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x3 ; y3/b ;
  a/x1 ; y1/b ; a/x2 ; y2/b ;
  a/c ; a/x3 ; y3/c
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
Wire3 = forever do x3/y3 end
```

Petrify output:

```
# EQN file for model M4D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 52.00

INORDER = y3 y2 y1 a x3 x2 x1 c b;
OUTORDER = [x3] [x2] [x1] [c] [b];
[0] = a' y2;
[1] = a' c;
[x3] = [1]' ([0] + x3) + x3 [0];     # mappable onto gC
[3] = y1 a;
[4] = x1' a';
[x2] = [4]' ([3] + x2) + x2 [3];     # mappable onto gC
[6] = a' b x3';
[7] = a x3;
[x1] = [7]' ([6] + x1) + x1 [6];     # mappable onto gC
[9] = x2' a x3;
[10] = y3' (x1' a + y2) + y2 y1';
[11] = y3 (y2' + x1) + y2' y1;
[b] = [11]' ([10] + b) + b [10];     # mappable onto gC
[c] = y3 ([9] + c) + c [9];     # mappable onto gC

# Set/reset pins: reset(x3) reset(x1) reset(b)
```

C.2.6 mod-5 counter

```
E = forever do -/a ; select b/- alt c/- end end

#environment E
M5 =
forever do
  a/b ; a/b ;
  a/b ; a/b ;
  a/b ; a/b ;
  a/b ; a/b ;
  a/c ; a/c
end
```

Petrify output:

```
# EQN file for model M5
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 98.00

INORDER = a c b csc0 csc1 csc2 csc3 csc4 csc5 csc6;
OUTORDER = [c] [b] [csc0] [csc1] [csc2] [csc3] [csc4] [csc5] [csc6];
[0] = a csc0';
[c] = csc0' ([0] + c) + c [0];     # mappable onto gC
[2] = csc1 csc2 (csc0 csc4 csc5 csc6 + csc3') + csc5' csc6' + csc1' csc3;
[3] = csc2 csc3 csc4' + csc1 (csc4 csc2' csc0 + csc2 csc0') + csc5' csc6;
[b] = [3]' ([2] + b) + b [2];     # mappable onto gC
[5] = a' csc1';
[6] = a csc2';
[7] = csc0' + csc1';
[8] = a' csc3';
[csc2] = [8]' ([7] + csc2) + csc2 [7];     # mappable onto gC
[10] = csc1' + csc2';
[11] = a csc4';
[csc3] = [11]' ([10] + csc3) + csc3 [10];     # mappable onto gC
[13] = csc2' + csc3';
[14] = a' csc5 csc3 csc2 csc1 csc0 csc6;
[csc4] = [14]' ([13] + csc4) + csc4 [13];     # mappable onto gC
[16] = a csc6;
[17] = a csc6';
[csc5] = [17]' ([16] + csc5) + csc5 [16];     # mappable onto gC
[19] = a' csc5';
[20] = a' c;
[csc6] = [20]' ([19] + csc6) + csc6 [19];     # mappable onto gC
```

```
[csc0] = csc6' ([5]' + csc0) + csc0 [5]';     # mappable onto gC
[csc1] = csc0' ([6]' + csc1) + csc1 [6]';     # mappable onto gC

# Set/reset pins: reset(b) set(csc2) set(csc3) set(csc4)
set(csc5) reset(csc6) set(csc0) set(csc1)
```

Decompositions applied were as follows:

```
#environment E par Wire1 par Wire2 par Wire3

M5D =
forever do
  a/b  ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/x3 ; y3/b ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/c  ; a/x3 ; y3/c
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
Wire3 = forever do x3/y3 end
```

Petrify output:

```
# EQN file for model M5D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 72.00

INORDER = y3 y2 y1 a x3 x2 x1 c b;
OUTORDER = [x3] [x2] [x1] [c] [b];
[0] = y1' x2 a;
[1] = a' c;
[x3] = [1]' ([0] + x3) + x3 [0];     # mappable onto gC
[3] = y1 a x3';
[4] = x1 a x3;
[x2] = [4]' ([3] + x2) + x2 [3];     # mappable onto gC
[6] = a' (y2' x3' b + y2 x3);
[7] = a' (y2 x3' + y2' x3);
[x1] = [7]' ([6] + x1) + x1 [6];     # mappable onto gC
[9] = x1' x2' a x3;
[10] = y3 y1' x2 + y2' y1 x3 + x3' (y1' x2' a + y1 y2);
[11] = x3 (y2' y1' + y2 y1) + x3' (y1 y2' + y1' y2);
[b] = [11]' ([10] + b) + b [10];     # mappable onto gC
```

```
[c] = y3 ([9] + c) + c [9];     # mappable onto gC

# Set/reset pins: reset(x3) reset(x2) reset(x1) reset(b)
```

C.2.7 mod-9 counter

```
E = forever do -/a ; select b/- alt c/- end end

#environment E
M9 =
forever do
  a/b ; a/b ; a/b ; a/b ;
  a/b ; a/b ; a/b ; a/b ;
  a/b ; a/b ; a/b ; a/b ;
  a/b ; a/b ; a/b ; a/b ;
  a/c ; a/c
end
```

Petrify output:

```
# EQN file for model M9
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 202.00

INORDER = a c b csc0 csc1 csc2 csc3 csc4 csc5 csc6 csc7
csc8 csc9 csc10 csc11 csc12 csc13 csc14;
OUTORDER = [c] [b] [csc0] [csc1] [csc2] [csc3] [csc4] [csc5]
[csc6] [csc7] [csc8] [csc9] [csc10] [csc11] [csc12] [csc13] [csc14];
[0] = csc0' a;
[c] = csc0' ([0] + c) + c [0];     # mappable onto gC
[2] = csc1 csc2 csc3' csc5 + a c' csc13
      + csc7 (csc8 csc9' csc11 + csc3 csc4 csc5')
      + csc9 (csc5 csc6 csc7' + csc10 csc11') + csc1' csc3 + csc14';
[3] = csc0 csc1 csc2' csc4 + csc8 (csc9 csc10' csc12 + csc4 csc5 csc6')
  + csc6 (csc7 csc10 (csc1 csc2 csc3 csc5 csc9 csc11 csc13' csc14 + csc8')
  + csc4' csc5');
[b] = [3]' ([2] + b) + b [2];     # mappable onto gC
[5] = a' csc13;
[6] = csc1' a';
[csc0] = [6]' ([5] + csc0) + csc0 [5];     # mappable onto gC
[8] = csc2' a;
[9] = csc1' + csc0';
[10] = csc3' a';
[csc2] = [10]' ([9] + csc2) + csc2 [9];     # mappable onto gC
[12] = csc2' + csc1';
[13] = csc4' a;
[csc3] = [13]' ([12] + csc3) + csc3 [12];     # mappable onto gC
[15] = csc3' + csc2';
```

```
[16] = a' csc5';
[csc4] = [16]' ([15] + csc4) + csc4 [15];     # mappable onto gC
[18] = csc4' + csc3';
[19] = a csc6';
[csc5] = [19]' ([18] + csc5) + csc5 [18];     # mappable onto gC
[21] = csc5' + csc4';
[22] = a' csc7';
[csc6] = [22]' ([21] + csc6) + csc6 [21];     # mappable onto gC
[24] = csc6' + csc5';
[25] = a csc8';
[csc7] = [25]' ([24] + csc7) + csc7 [24];     # mappable onto gC
[27] = csc7' + csc6';
[28] = a' csc9';
[csc8] = [28]' ([27] + csc8) + csc8 [27];     # mappable onto gC
[30] = csc8' + csc7';
[31] = a csc10';
[csc9] = [31]' ([30] + csc9) + csc9 [30];     # mappable onto gC
[33] = csc9' + csc8';
[34] = a' csc11';
[csc10] = [34]' ([33] + csc10) + csc10 [33];     # mappable onto gC
[36] = csc9' + csc10';
[37] = a csc12';
[csc11] = [37]' ([36] + csc11) + csc11 [36];     # mappable onto gC
[39] = csc11' + csc10';
[40] = a' csc14';
[csc12] = [40]' ([39] + csc12) + csc12 [39];     # mappable onto gC
[42] = a' b;
[csc13] = [42]' (c + csc13) + c csc13;     # mappable onto gC
[44] = csc0 csc1 csc2 csc3 csc4 a csc5 csc6 csc7
       csc8 csc10 csc9 csc11 csc12 csc13';
[csc1] = csc0' ([8]' + csc1) + csc1 [8]';     # mappable onto gC
[csc14] = csc12' ([44]' + csc14) + csc14 [44]';     # mappable onto gC

# Set/reset pins: reset(b) set(csc2) set(csc3) set(csc4) set(csc5)
  set(csc6) set(csc7) set(csc8) set(csc9) set(csc10) set(csc11)
  set(csc12) set(csc13) set(csc1) set(csc14)
```

Decompositions applied were as follows:

```
#environment E par Wire1 par Wire2 par Wire3 par Wire4

M9D =
forever do
  a/b  ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/x3 ; y3/b ; a/x1 ; y1/b ;
```

```
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/x4 ; y4/b ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/x3 ; y3/b ; a/x1 ; y1/b ;
  a/x2 ; y2/b ; a/x1 ; y1/b ;
  a/c  ; a/x4 ; y4/c
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
Wire3 = forever do x3/y3 end
Wire4 = forever do x4/y4 end
```

Petrify output:

```
# EQN file for model M9D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 123.00

INORDER = y4 y3 y2 y1 a x4 x3 x2 x1 c b;
OUTORDER = [x4] [x3] [x2] [x1] [c] [b];
[0] = a x1' x3 x2';
[1] = a' c;
[x4] = [1]' ([0] + x4) + x4 [0];     # mappable onto gC
[3] = a x1' x4' y2;
[4] = a y1' x4 x2;
[x3] = [4]' ([3] + x3) + x3 [3];     # mappable onto gC
[6] = y1 a (x4' x3' + x4 x3);
[7] = y1 a (x4 x3' + x4' x3);
[x2] = [7]' ([6] + x2) + x2 [6];     # mappable onto gC
[9] = a' (x2 (x4 x3' + x4' x3) + x2' (b x4' x3' + x4 x3));
[10] = a' (x2 (x4' x3' + x4 x3) + x2' (x4 x3' + x4' x3));
[x1] = [10]' ([9] + x1) + x1 [9];     # mappable onto gC
[12] = a y1' x4 x3' x2';
[13] = y4 y3 x2' x1' + x4 (y3' x2 x1' + x1 (y3' y2' + y3 y2))
       + x4' (y3 (x2 x1' + y2' x1) + y3' (a x1' x2' + x1 y2));
[14] = (y3 x2' + y3' x2) (y1' x4' + y1 x4)
      + (y3' x2' + y3 x2) (y1 x4' + y1' x4);
[b] = [14]' ([13] + b) + b [13];     # mappable onto gC
[c] = y4 ([12] + c) + c [12];     # mappable onto gC

# Set/reset pins: reset(x4) reset(x3) reset(x2) reset(x1) reset(b)
```

C.2.8 seq3

```
E0 = forever do -/a0 ; c0/- end
E1 = forever do c1/a1 end
E2 = forever do c2/a2 end

#environment E0 par E1 par E2
S3 =
forever do
  a0/c1 ; a1/c1 ;
  a1/c2 ; a2/c2 ;
  a2/c0 ; a0/c0
end
```

Petrify output:

```
# EQN file for model S3
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 22.00

INORDER = a2 a1 a0 c2 c1 c0 csc0 csc1;
OUTORDER = [c2] [c1] [c0] [csc0] [csc1];
[0] = a1' csc0 csc1;
[c1] = a0 csc1';
[c0] = a2' csc0';
[c2] = csc0 ([0] + c2) + c2 [0];     # mappable onto gC
[csc0] = csc1' (a2' + csc0) + a2' csc0;     # mappable onto gC
[csc1] = a0 (a1 + csc1) + a1 csc1;     # mappable onto gC

# Set/reset pins: reset(c2)
```

Decompositions applied were as follows:

```
#environment E0 par E1 par E2 par Wire1 par Wire2

S3D =
forever do
  a0/c1 ; a1/x1 ; y1/c1 ;
  a1/c2 ; a2/x2 ; y2/c2 ;
  a2/c0 ; a0/x1 ; y1/x2 ; y2/c0
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
```

Petrify output:

```
# EQN file for model S3D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 22.00

INORDER = y2 y1 a2 a1 a0 x2 x1 c2 c1 c0;
OUTORDER = [x2] [x1] [c2] [c1] [c0];
[0] = a1' x1 y2';
[c2] = y2' ([0] + c2) + c2 [0];     # mappable onto gC
[c1] = y1' a0;
[c0] = y2 a2';
[x2] = y1 (a2 + x2) + a2 x2;     # mappable onto gC
[x1] = a0 (a1 + x1) + a1 x1;     # mappable onto gC

# Set/reset pins: reset(c2)
```

C.2.9 seq5

```
E0 = forever do -/a0 ; c0/- end
E1 = forever do c1/a1 end
E2 = forever do c2/a2 end
E3 = forever do c3/a3 end
E4 = forever do c4/a4 end

#environment E0 par E1 par E2 par E3 par E4
S5 =
forever do
  a0/c1 ; a1/c1 ;
  a1/c2 ; a2/c2 ;
  a2/c3 ; a3/c3 ;
  a3/c4 ; a4/c4 ;
  a4/c0 ; a0/c0
end
```

Petrify output:

```
# EQN file for model S5
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 61.00

INORDER = a4 a3 a2 a1 a0 c4 c3 c2 c1 c0 csc0 csc1 csc2;
OUTORDER = [c4] [c3] [c2] [c1] [c0] [csc0] [csc1] [csc2];
[0] = csc2' a3';
[1] = a4 csc2;
[c4] = [1]' ([0] + c4) + c4 [0];     # mappable onto gC
[3] = csc0 a2' csc2 csc1;
[4] = csc0 csc1' a1' c1';
[c2] = csc1' ([4] + c2) + c2 [4];     # mappable onto gC
[6] = a0 csc0' csc1';
[7] = csc0 a1;
[c1] = [7]' ([6] + c1) + c1 [6];     # mappable onto gC
[9] = a4' c4' csc0' csc1;
[csc0] = c4' (c1 + csc0) + c1 csc0;     # mappable onto gC
[c3] = csc2 ([3] + c3) + c3 [3];     # mappable onto gC
[c0] = csc1 ([9] + c0) + c0 [9];     # mappable onto gC
[csc1] = a0 (a2 + csc1) + a2 csc1;     # mappable onto gC
[csc2] = csc0' (a3' + csc2) + a3' csc2;     # mappable onto gC

# Set/reset pins: reset(c4) reset(c2) reset(c1) reset(csc0) reset(c3)
```

Decompositions applied were as follows:

```
#environment E0 par E1 par E2 par E3 par E4
             par Wire1 par Wire2 par Wire3

S5D =
forever do
  a0/c1 ; a1/x1 ; y1/c1 ;
  a1/c2 ; a2/x2 ; y2/c2 ;
  a2/c3 ; a3/x3 ; y3/c3 ;
  a3/c4 ; a4/x1 ; y1/c4 ;
  a4/c0 ; a0/x2 ; y2/x3 ; y3/c0
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
Wire3 = forever do x3/y3 end
```

Petrify output:

```
# EQN file for model S5D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 55.00

INORDER = y3 y2 y1 a4 a3 a2 a1 a0 x3 x2 x1 c4 c3 c2 c1 c0;
OUTORDER = [x3] [x2] [x1] [c4] [c3] [c2] [c1] [c0];
[x1] = a4' (a1 + x1) + a1 x1;     # mappable onto gC
[1] = a3' y3 x1;
[2] = y3' x2 a2';
[c3] = y3' ([2] + c3) + c3 [2];     # mappable onto gC
[4] = y2' x1 a1';
[c2] = y2' ([4] + c2) + c2 [4];     # mappable onto gC
[6] = x2' a0 y1';
[c1] = y1' ([6] + c1) + c1 [6];     # mappable onto gC
[8] = y3 a4' x1';
[x3] = y2 (a3 + x3) + a3 x3;     # mappable onto gC
[x2] = a0 (a2 + x2) + a2 x2;     # mappable onto gC
[c4] = y1 ([1] + c4) + c4 [1];     # mappable onto gC
[c0] = y3 ([8] + c0) + c0 [8];     # mappable onto gC

# Set/reset pins: reset(x1) reset(c3) reset(c2) reset(c1)
```

C.2.10 seq9

```
E0 = forever do -/a0 ; c0/- end
E1 = forever do c1/a1 end
E2 = forever do c2/a2 end
E3 = forever do c3/a3 end
E4 = forever do c4/a4 end
E5 = forever do c5/a5 end
E6 = forever do c6/a6 end
E7 = forever do c7/a7 end
E8 = forever do c8/a8 end

#environment E0 par E1 par E2 par E3 par E4
             par E5 par E6 par E7 par E8
S9 =
forever do
  a0/c1 ; a1/c1 ; a1/c2 ; a2/c2 ;
  a2/c3 ; a3/c3 ; a3/c4 ; a4/c4 ;
  a4/c5 ; a5/c5 ; a5/c6 ; a6/c6 ;
  a6/c7 ; a7/c7 ; a7/c8 ; a8/c8 ;
  a8/c0 ; a0/c0
end
```

Petrify output:

```
# EQN file for model S9
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 107.00

INORDER = a8 a7 a6 a5 a4 a3 a2 a1 a0 c8 c7 c6 c5 c4 c3 c2 c1 c0
csc0 csc1 csc2 csc3 csc4;
OUTORDER = [c8] [c7] [c6] [c5] [c4] [c3] [c2] [c1] [c0] [csc0]
[csc1] [csc2] [csc3] [csc4];
[0] = csc1' c7' a7' csc4;
[1] = csc1 a6' csc4;
[2] = csc1' a7;
[c7] = [2]' ([1] + c7) + c7 [1];     # mappable onto gC
[4] = a5' csc3 csc0 csc4';
[c6] = csc4' ([4] + c6) + c6 [4];     # mappable onto gC
[6] = a4' csc2 csc3';
[c5] = csc3' ([6] + c5) + c5 [6];     # mappable onto gC
[8] = csc2' a3' c3' csc0;
[c4] = csc2' ([8] + c4) + c4 [8];     # mappable onto gC
```

```
[10] = a2' csc3' csc0';
[11] = a3 csc0;
[c3] = [11]' ([10] + c3) + c3 [10];     # mappable onto gC
[13] = a1' csc2' csc3;
[14] = a0 csc2 csc0' csc1;
[15] = csc1' a8' csc4';
[c0] = csc1' ([15] + c0) + c0 [15];     # mappable onto gC
[csc0] = c8' (c3 + csc0) + c3 csc0;     # mappable onto gC
[csc2] = a1' (a4 + csc2) + a4 csc2;     # mappable onto gC
[csc3] = a2' (a5 + csc3) + a5 csc3;     # mappable onto gC
[20] = csc0' a8;
[csc4] = [20]' (a6 + csc4) + a6 csc4;     # mappable onto gC
[c8] = csc4 ([0] + c8) + c8 [0];     # mappable onto gC
[c2] = csc3 ([13] + c2) + c2 [13];     # mappable onto gC
[c1] = csc2 ([14] + c1) + c1 [14];     # mappable onto gC
[csc1] = a0' (c7' + csc1) + c7' csc1;     # mappable onto gC

# Set/reset pins: reset(c7) reset(c6) reset(c3) reset(csc0)
  set(csc2) set(csc3) reset(csc4) reset(c2) reset(c1)
```

Decompositions applied were as follows:

```
#environment E0 par E1 par E2 par E3 par E4 par E5 par E6
             par E7 par E8 par Wire1 par Wire2 par Wire3 par Wire4

S9D =
forever do
  a0/c1 ; a1/x1 ; y1/c1 ;
  a1/c2 ; a2/x2 ; y2/c2 ;
  a2/c3 ; a3/x1 ; y1/c3 ;
  a3/c4 ; a4/x3 ; y3/c4 ;
  a4/c5 ; a5/x1 ; y1/c5 ;
  a5/c6 ; a6/x2 ; y2/c6 ;
  a6/c7 ; a7/x1 ; y1/c7 ;
  a7/c8 ; a8/x4 ; y4/c8 ;
  a8/c0 ; a0/x3 ; y3/x4 ; y4/c0
end

Wire1 = forever do x1/y1 end
Wire2 = forever do x2/y2 end
Wire3 = forever do x3/y3 end
Wire4 = forever do x4/y4 end
```

Petrify output:

```
# EQN file for model S9D
# Generated by petrify 4.2 (compiled 15-Oct-03 at 3:06 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 99.00

INORDER = y4 y3 y2 y1 a8 a7 a6 a5 a4 a3 a2 a1 a0 x4 x3 x2
x1 c8 c7 c6 c5 c4 c3 c2 c1 c0;
OUTORDER = [x4] [x3] [x2] [x1] [c8] [c7] [c6] [c5] [c4]
[c3] [c2] [c1] [c0];
[x2] = a6' (a2 + x2) + a2 x2;     # mappable onto gC
[1] = a5 + a1;
[2] = a7 + a3;
[x1] = [2]' ([1] + x1) + x1 [1];     # mappable onto gC
[4] = x1' x3 y4' a7' x2';
[c8] = y4' ([4] + c8) + c8 [4];     # mappable onto gC
[6] = y1 x3 x2' a6';
[7] = y1 x3 a5' x2;
[8] = y1' x3 a4' x2;
[c5] = y1' ([8] + c5) + c5 [8];     # mappable onto gC
[10] = x1' a3' y3' x2;
[c4] = y3' ([10] + c4) + c4 [10];     # mappable onto gC
[12] = x1 y3' a2' x2;
[13] = a1' x1 x3' y2';
[c2] = y2' ([13] + c2) + c2 [13];     # mappable onto gC
[15] = y1' a0 x3' x2';
[c1] = y1' ([15] + c1) + c1 [15];     # mappable onto gC
[c0] = y4 a8';
[x4] = y3 (a8 + x4) + a8 x4;     # mappable onto gC
[x3] = a0 (a4 + x3) + a4 x3;     # mappable onto gC
[c7] = y1 ([6] + c7) + c7 [6];     # mappable onto gC
[c6] = y2 ([7] + c6) + c6 [7];     # mappable onto gC
[c3] = y1 ([12] + c3) + c3 [12];     # mappable onto gC

# Set/reset pins: reset(x2) reset(x1) reset(c8) reset(c5)
reset(c4) reset(c2) reset(c1)
```

Appendix D

Translation from DI processes into Petri nets

This appendix reproduces the algorithm to translate processes expressed in DISP into Petri nets as it appears in [JF02].

D.1 The Algorithm

The input/output behaviour of a logic block and of its environment are specified by a pair of programs. Each is translated into a Petri net fragment, as described below, and the two fragments are combined to form a closed Petri net.

The translation algorithm operates upon two data structures, a list L and a Petri net fragment N. The list consists of tuples of the form (α, ω, Φ), where α and ω are places in N, and Φ is either a process, or a set of alternatives, that has yet to be translated. When the list is empty, the algorithm terminates, returning N. A "1-safe" interpretation is to be put on the net, i.e., a computation in which a place becomes marked with more than one token is unsafe.

Given a program P, the data structures are initialised as follows: L contains a single tuple (0, 1, P), whilst N consists of

- a marked place 0 and an unmarked place 1,
- a transition (labelled x) with an empty pre-set and a post-set consisting of a single unmarked place (also labelled x) for each input signal x of P,
- a transition (labelled x) with an empty post-set and a pre-set consisting of a single unmarked place (also labelled x) for each output signal x of P,
- a single unmarked place (labelled x) for each local signal x of P.

While L is non-empty, any tuple is removed from the list and an expansion rule is applied to it. The algorithm terminates because each expansion rule strictly

reduces the sum over each tuple in L of the size of its third component. The rules for each of the most common language constructs are given below.

Expansion rules for tuples

$(\alpha, \omega, xs/ys)$: add dummy transition to N with pre-set α, xs and post-set ω, ys.

$(\alpha, \omega, \texttt{pushback}\ xs)$: add dummy transition to N with pre-set α and post-set ω, xs.

$(\alpha, \omega, \texttt{forever do}\ P\ \texttt{end})$: add tuple (α, α, P) to L.

$(\alpha, \omega, P\ ;\ Q)$: add place β to N and tuples (α, β, P) and (β, ω, Q) to L.

$(\alpha, \omega, P\ \texttt{par}\ Q)$: add one dummy transition to N with pre-set α and post-set α_0, α_1 (for new places α_0, α_1), another dummy transition to N with pre-set ω_0, ω_1 and post-set ω (for new places ω_o, ω_1) and the tuples (α_0, ω_0, P) and (α_1, ω_1, Q) to L.

$(\alpha, \omega, \texttt{select}\ \Phi\ \texttt{end})$: add tuple (α, ω, Φ) to L.

$(\alpha, \omega, xs/ys\ \texttt{then}\ P)$: add place β to N and tuples $(\alpha, \beta, xs/ys)$ and (β, ω, P) to L.

$(\alpha, \omega, \Phi\ \texttt{alt}\ \Psi)$: add place (α, ω, Φ) and (α, ω, Ψ) to L.

Example

Consider the One-Hot Join element described as follows:

$$\texttt{pushback}\ a\ ;\ \texttt{forever do}\ a, b/c\ \texttt{end}$$

In this case L is initialised to

$$(0, 1, \texttt{pushback}\ a\ ;\ \texttt{forever do}\ a, b/c\ \texttt{end})$$

and N consists of 5 places (labelled 0, 1, a, b and c) and 3 transitions (labelled a, b and c), with transitions a and b connected to places a and b, respectively, and place c connected to transition c. Place 0 is marked.

The algorithm then proceeds as follows:

1. The single tuple in L is replaced by two, namely, (0, 2, $\texttt{pushback}\ a$) and (2, 1, $\texttt{forever do}\ a, b/c\ \texttt{end}$) and a place labelled 2 is added to N.

2. The tuple (0, 2, $\texttt{pushback}\ a$) is removed and a transition t is added to N, with place 0 connected to t and transition t connected to places 2 and a.

3. The tuple (2, 1, `forever do` $a, b/c$ `end`) is replaced by (2, 2, $a, b/c$).

4. The tuple (2, 2, $a, b/c$) is removed and a transition u is added to N, with places 2, a and b connected to u and transition u connected to places 2 and c.

L is now empty and N can be returned. Figure D.1 shows the Petri net fragment returned by di2pn after it has simplified N by applying various peephole optimisations.

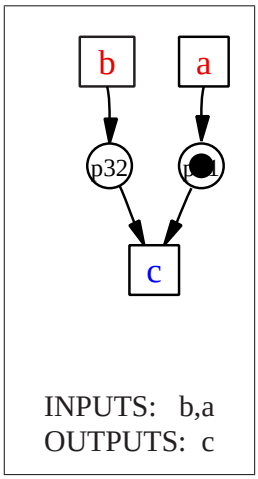


Figure D.1: Petri net fragment for One-Hot Join generated by di2pn

References

[BBC+96] N. S. Bjørner, A. Browne, E. Chang, M. Col'on, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems. *Proceedings of 8th International Conference on Computer Aided Verification, LNCS*, 1102:415–418, July 1996.

[BCDM86] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits using Temporal Logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[BE97] A. Bardsley and D. Edwards. Compiling the Language Balsa to Delay-insensitive Hardware. *Hardware Description Languages and their Applications*, pages 89–91, April 1997.

[Ber93] K. van Berkel. *Handshake Circuits - An Asynchronous architecture for VLSI programming*. Cambridge University Press, 1993.

[BHSV+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, 1102:428–432, 1996.

[BJN99] C. H. van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the Technology: Applications of Asynchronous Circuits. *Proceedings of IEEE*, 87(2):223–233, February 1999.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[Bre93] G. Brebner. A CCS based Investigation of Deadlock in a Multi-process Electronic Mail System. *Formal Aspects of Computing*, 5(5):467–479, 1993.

[BS89] E. Brunvand and R. F. Sproull. Translating Concurrent Programs into Delay-insensitive Circuits. *ICCAD*, pages 262–265, November 1989.

[BS94] J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1994.

[Bur92] G. Burns. A Case-study in Safety Critical Design. Technical Report ECS-LFCS-92-239, University of Edinburgh, 1992.

[Bur97] G. Burns. *Distributed Systems Analysis with CCS*. Prentice-Hall, London, 1997.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panagaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[CCP01] J. Carmona, J. Cortadella, and E. Pastor. A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. *Proceedings of Second International Conference on Application of Concurrency to System Design*, pages 157–166, 2001.

[CCP02] J. Carmona, J. Cortadella, and E. Pastor. A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. *Fundamenta Informaticae,* 34:1–23, 2002.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of Synchronisation Skeletons for Branching Time Temporal Logic. *Logic of Programs: Workshop. LNCS*, 131, May 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[CH93] R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. In *Formal Aspects of Computing*, volume 5, pages 1–20. BCS, 1993.

[Cha84] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[Chu87] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Department of Electrical Engineering and Computer Science, June 1987. MIT/LCS/TR-393.

[CKK+96] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete State Encoding Based on the Theory of Regions. *Proceedings of Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–47, March 1996.

[CKK+97a] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, 3(E80-D):315–325, 1997.

[CKK+97b] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A Region-based Theory for State Assignment in Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):793–812, August 1997.

[CKM+88] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Nielson, B. Pase, and M. Saaltink. m-EVES: A Tool for Verifying Software. In *Proceedings of the 11th International Conference on Software Engineering (ICSE'11)*, pages 324–333, April 1988.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, June 1989.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems. *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems, LNCS*, 407, 1989.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15:36–72, 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV), LNCS*, volume 1102, pages 394–397, July 1996.

[CT97] G. Clark and G. S. Taylor. The Verification of Asynchronous Circuits using CCS. Technical Report ECS-LFCS-97-369, Department of Computer Science, University of Edinburgh, October 1997.

[Der93] N. Dershowitz. A Taste of Rewrite Systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning (LNCS)*, pages 199–228, 1993.

[Dil89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[DJ90] N. Dershowitz and J. P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[DJ01] J. Desel and G. Juhás. What is Petri Net. *Unifying Petri Net, Lecture Notes in Computer Science*, 2128:1–25, 2001.

[Ebe87] J. C. Ebergen. *Translating programs into delay-insensitive circuits*. PhD thesis, Dept. of Mathematics and Computer Science, Eindhoven Univ. of Technology, 1987.

[Ebe91] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 3(5):447–450, 1991.

[Eve87] H. Eveking. Verification, Synthesis, and Correctness Preserving Transformations - Cooperative Approaches to Correct Hardware Design. *HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 229–239, 1987. Elsevier Science.

[FGR+99] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. *Proceedings of the IEEE*, 87(2):243–256, 1999.

[FNT+99] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines. Technical Report CUCS-020-99, Columbia University Computer Science Department, July 1999.

[Ful03] Asynchronous Circuit Technology: An Alternative for High-Speed Semiconductors, 2003. White Paper, Fulcrum Microsystems.

[Fur02] D. P. Furey. State based DI circuit Verification and Synthesis. *Proceedings of the 13th UK Asynchronous Forum, Cambridge University*, December 2002.

[GG88] S. J. Garland and J. V. Guttag. Inductive Methods for Reasoning about Abstract Data Types. In *Proceedings of 15th Symposium on Principles of Programming Languages*, pages 219–228, 1988.

[GH93] J. V. Guttag and J. J. Horning. *Larch : Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Proceedings 4th Workshop on Computer Aided Verification, LNCS*, volume 663, pages 178–191. Springer-Verlag, June 1992.

[GHP+95] R. Groenboom, C. Hendriks, I. Polak, J. Terlouw, and J. Tijmen Udding. Algebraic Proof Assistants in HOL. In B Muller, editor, *Mathematics of Program Construction, LNCS*, volume 947, pages 304–321, 1995.

[GJLU93] R. Groenboom, M. B. Josephs, P. G. Lucassen, and J. T. Udding. Normal Form in a Delay-Insensitive Algebra. *Proceedings of the IFIP Transactions on Asynchronous Design Methodologies*, pages 57–70, April 1993.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL - A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[God90] P. Godefroid. Using Partial Orders to improve Automatic Verification Methods. In *Proceedings 2nd Workshop on Computer Aided Verification, LNCS*, volume 531, pages 176–185. Springer-Verlag, June 1990.

[God96] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. In *LNCS*, volume 1032. Springer-Verlag, January 1996.

[Goy91] J. H. Goyer. Communications protocols for the B-HIVE multicomputer. Master's thesis, North Carolina State University, 1991.

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using Partial-Order Methods in the Formal Verification of Industrial Concurrent Programs. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 261–269, January 1996.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hau95] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1), January 1995.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[HJH90] J. He, M. B. Josephs, and C. A. R. Hoare. A Theory of Synchrony and Asynchrony. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 459–478. North-Holland, 1990.

[HK90] Z. Harel and R. P. Kurshan. Software for Analytical Development of Communications Protocols. *AT&T Bell Laboratories Technical Journal*, 69(1):45–59, January-February 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol03] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[HP94] G. J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings of FORTE'94*, pages 177–191, 1994.

[JB97] M. B. Josephs and A. M. Bailey. The Use of SI-Algebra in the Design of Sequencer Circuits. *Formal Aspects of Computing*, 9:395–408, 1997.

[JF00] M. B. Josephs and D. P. Furey. Delay-Insensitive Interface Specification and Synthesis. *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 169–173, March 2000.

[JF02] M. B. Josephs and D. P. Furey. A Programming Approach to the Design of Asynchronous Logic Blocks. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design Advances in Petri-Nets, LNCS*, volume 2549, pages 34–60. Springer Verlag, 2002.

[JHH89] M. B. Josephs, C. A. R. Hoare, and J. He. A Theory of Asynchronous Processes. Technical Report PRG-TR-6-89, Oxford University Computing Laboratory, Oxford, England, 1989.

[JLUV94] M. B. Josephs, P. G. Lucassen, J. T. Udding, and T. Verhoeff. Formal Design of an Asynchronous DSP Counterflow Pipeline: A Case Study in Handshake Algebra. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'94)*, pages 206–215, 1994.

[JMU+92] M. B. Josephs, R. H. Mak, J. T. Udding, T. Verhoeff, and J. T. Yantchev. High Level Design of an Asynchronous Packet Routing Chip. *Designing Correct Circuits. IFIP Transactions*, A-5:261–274, 1992.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[Jos92] M. B. Josephs. Receptive Process Theory. *Acta Informatica*, 29(1):17–31, 1992.

[Jos02] M. B. Josephs. Verification of a Loadable Counter. *Lecture Notes of Summer School on Asynchronous Circuit Design, TIMA Laboratory, Grenoble, France*, July 2002.

[JU90a] M. B. Josephs and J. T. Udding. Delay-Insensitive Circuits: An Algebraic Approach to their Design. *CONCUR'90, Lecture Notes in Computer Science*, 458:342–366, 1990.

[JU90b] M. B. Josephs and J. T. Udding. The Design of a Delay-insensitive Stack. In *Designing Correct Circuits, Workshop in Computing*, pages 132–152, 1990.

[JU93] M. B. Josephs and J. T. Udding. An Overview of D-I Algebra. *System Sciences, 1993, IEEE Proceeding of the Twenty-Sixth Hawaii International Conference*, 1:329–338, January 1993.

[JUY93] M. B. Josephs, J. T. Udding, and J. T. Yantchev. Handshake Algebra. Technical Report SBU-CISM-93-1, School of Computing, Information Systems and Mathematics, South Bank Univeristy, London, December 1993.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of IFIP Congress 74*. North Holland Publishing Co., 1974.

[KG99] C. Kern and M. R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[Kir93] C. Kirkwood. Automated (Specification $\equiv$ Implementation) using Equational Reasoning and LOTOS. In *Proceedings TOPSOFT'93 : Theory and Practice of Software Development, LNCS*, volume 668. Springer-Verlag, Berlin, 1993.

[KJ02] H. K. Kapoor and M. B. Josephs. Handshaking Expansion Versus Delay-Insensitive Sequential Processes. *Postgraduate Research in Electronics, Photonics, Communications and Software, PREP*, April 2002.

[KKY02] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting State Coding Conflicts in STGs using Integer Programming. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 338–345, March 2002.

[KKY03] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting State Coding Conflicts in STGs using SAT. In *Proceedings of International Conference on Application of Concurrency to System Design (ACSD)*, pages 51–60, June 2003.

[KMM00] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[KvBB+92] J. Kessels, K. van Berkel, R. Burgess, M. Roncken, and F. Schalij. An Error Decoder for the Compact Disc Player as an example of VLSI Programming. In *Proceedings of European Conference on Design Automation (EDAC)*, pages 69–75, 1992.

[KZ95] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[Lam84] L. Lamport. The Temporal Logic of Actions. *ACM TOPLAS*, pages 872–923, May 1984.

[Lin94] H. Lin. PAM : A Process Algebra Manipulator. *Formal Methods in System Design*, 7:243–259, 1994.

[Liu95] Y. Liu. *AMULET1 Specification and Verification in CCS*. PhD thesis, Department of Computer Science, Univeristy of Calgary, September 1995.

[LPU97] P. G. Lucassen, I. Polak, and J. T. Udding. Normal Form in DI-Algebra with Recursion. *Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 167–174, April 1997.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness Proofs for Distributed Algorithms. *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LT89] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[LU96] P. G. Lucassen and J. T. Udding. On the Correctness of the Sproull Counterflow Pipeline Processor. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'96)*, pages 112–120, 1996.

[Luc94] P. G. Lucassen. *A Denotational Model and Composition Theorems for a Calculus of Delay-Insensitive Specifications*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, May 1994.

[MAC+02] S. Moore, R. Anderson, P. Cunningham, R. Mullins, and G. Taylor. Improving Smart Card Security using Self-timed Circuits. In *Proceedings of Eighth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'02)*, pages 211– 218, 2002.

[Mad03] A. Madalinski. CONFRES: Interactive Coding Conflict Resolver Based on Core Visualisation. In *Proceedings of International Conference on Application of Concurrency to System Design (ACSD)*, pages 243–244, June 2003.

[Mal00a] W. C. Mallon. On Directed Transformations of Delay-Insensitive Specifications, Alternations and Dynamic Nondeterminism. *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.

[Mal00b] W. C. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. PhD thesis, Dept. of C.S., Univ. of Groningen, The Netherlands, January 2000.

[Man01] R. Manohar. An Analysis of Reshuffled Handshaking Expansions. *Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, March 2001.

[Mar87] A. J. Martin. Programming in VLSI: From Communicating Processes to Self-timed VLSI Circuits. In *Proceedings of UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, March 1987.

[MBKY03] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualisation and Resolution of Encoding Conflicts in Asynchronous Circuit Design. *IEE Proceedings on Computers and Digital Techniques*, 150(5):285–293, September 2003.

[MBL+89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The Design of an Asynchronous Microprocessor. In *Proceedings of Decennial Caltech Conference on VLSI*, pages 351–373, 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Press, 1993.

[Mes96] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report. *Proceedings CONCUR'96, LNCS*, 1119:331–372, August 1996.

[MFR85] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of Delay-Insensitive Modules. In Henry Fuchs, editor, *Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science, 1989.

[MLD92] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design*. Kluwer Academic Press, 1992.

[MLM99] R. Manohar, T-K. Lee, and A. J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 125–134, April 1999.

[MM98] R. Manohar and A. J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the fourth International Conference on the Mathematics of Program Contruction, Lecture Notes in Computer Science*, pages 272–285, July 1998.

[Mol91] F. Moller. *The Edinburgh Concurrency Workbench (Version 6.0)*. University of Edinburgh, August 1991.

[MOM00] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[MS] F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[MTMR02] S. Moore, G. Taylor, R. Mullins, and P. Robinson. Point to Point GALS Interconnect. In *Proceedings of Eighth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'02)*, pages 69–75, 2002.

[MU97] W. C. Mallon and J. T. Udding. Construction of an operational semantics for DI-algebra. Technical Report CSN 9710, Dept. of Computer Science, Univ. of Groningen, 1997.

[MU98] W. C. Mallon and J. T. Udding. Building Finite Automata from DI Specifications. *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, 1998.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Mye95] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Department of Electrical Engineering, Stanford University, October 1995.

[Mye01] C. J. Myers. *Asynchronous Circuit Design*. Wiley-Interscience, 2001.

[Neg98] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Waterloo, Ontario, Canada, August 1998.

[NYD92] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical Asynchronous Controller Design. *International Conference on Computer Design, ICCD*, pages 341–345, October 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. *11th International Conference on Automated Deduction (CADE), LNAI*, 607:748–752, June 1992.

[Par88] J. Parrow. Verifying a CSMA/CD-protocol with CCS. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*, pages 373–387. North-Holland, 1988.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Faculty of Mathematics and Physics, Technische Universitat Darmstadt, Germany, 1962.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International Series in Computer Science, 1998.

[RW89] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[Sei80] C. L. Seitz. System Timing. In Mead and Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[SF01] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design A Systems Perspective*. Kluwer Academic Publishers, 2001.

[SKC+02] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, A. Yakovlev, and T. Nanya. Design of Asynchronous Controllers with Delay Insensitive Interface. *IEICE transactions on Fundamentals*, E85-A(12):2577–2585, December 2002.

[Sne85] Jan L. A. van de Snepscheut. Trace Theory and VLSI Design. *LNCS*, 200, 1985.

[Spi88] J. M. Spivey. *Introducing Z : A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanno-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report Electronics Research Lab. Memo. No. UCB/ERL M92/41, Dept. of EECS, Univ. of California, Berkeley, U.S.A., 1992.

[Ste94] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, Department of Computer Science, University of Calgary, September 1994.

[TB98] G. S. Taylor and G. M. Blair. Reduced Complexity Two-Phase Micropipeline Latch Controller. *IEEE Journal of Solid-State Circuits*, 33(10):1590–1593, October 1998.

[Udd84] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Technical University of Eindhoven, 1984.

[Udd86] J. T. Udding. A Formal Model for Defining and Classifying Delay-Insensitive Circuits. *Distributed Computing*, 1(4):197–204, 1986.

[Val88a] A. Valmari. Error Detection by Reduced Reachability Graph Generation. In *Proceedings 9th International Conference on Application and Theory of Petri Nets*, pages 95–112, 1988.

[Val88b] A. Valmari. Heuristics for Lazy State Generation Speeds up Analysis of Concurrent Systems. In *Proceedings of the Finnish Artificial Intelligence Symposium STeP-88*, volume 2, pages 640–650, 1988.

[Val88c] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, Tampere University of Technology, 1988.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings 2nd Workshop on Computer Aided Verification, LNCS*, volume 531, pages 156–165. Springer-Verlag, June 1990.

[Ver94] T. Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[Ver01] IEEE Standard Verilog Hardware Description Language, March 2001. IEEE Std 1364-2001.

[Ver03] T. Verhoeff. Encyclopedia of Delay-Insensitive Systems (EDIS), 2003. http://edis.win.tue.nl/edis.html (Last accessed 28/10/2003).

[VHD00] IEEE Standard VHDL Language Reference Manual, January 2000. IEEE Std 1076, 2000 Edition.

[VMO00] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *FORTE/PSTV 2000*, pages 351–366. Kluwer Academic Publishers, October 2000.

[VW02] W. Vogler and R. Wollowski. Decomposition in Asynchronous Circuit Design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design Advances in Petri-Nets, LNCS*, volume 2549, pages 152–190. Springer Verlag, 2002.

[Wal89] D. J. Walker. Analysing Mutual Exclusion Algorithms using CCS. *Formal Aspects of Computing*, 1:273–292, 1989.

[WC96] J. M. Wing and E. M. Clarke. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, pages 626–643, December 1996.

[YD92] K. Y. Yun and D. L. Dill. Automatic Synthesis of 3D Asynchronous Controllers. *International Conference on Computer-Aided Design, ICCAD*, pages 576–580, November 1992.

[YD95] K. Y. Yun and D. L. Dill. A High-Performance Asynchronous SCSI Controller. *International Conference on Computer Design, ICCD*, pages 44–49, October 1995.

[YKSK96] A. V. Yakovlev, A. M. Koelmans, A. Semenov, and D. J. Kinniment. Modelling, Analysis and Synthesis of Asynchronous Control Circuits using Petri Nets. *Integration, the VLSI Journal*, 21(3):143–170, December 1996.