Simulation Techniques

Anders Gammelgaard

December 1991

Summary

In the papers surveyed in this thesis a number of simulation techniques are presented together with their applications to several examples. The papers improve upon existing techniques and introduce new techniques.

The improvement of existing techniques is motivated in programming methodology: It is demonstrated that existing techniques often introduce a double proof burden whereas the improved techniques alleviate such a burden. One application is to ensure delay insensitivity in a class of self-timed circuits.

A major part of the thesis is concerned with the deduction and use of two simulation techniques to prove the correctness of translations from subsets of occam-2 to transputer code.

The first technique is based on weak bisimulation. It is argued that Milner's notion of observation equivalence must be modified to cope with nondeterminism and silent divergence as found in occam-2. In the resulting technique a stronger, but asymmetric, simulation condition must be proved and an additional index to the simulation relation must be shown to decrease at certain computation steps. An application of the technique to a sequential subset of occam-2 is successful but results in a large proof.

The second technique drastically reduces the size of the proof in the first technique. It marks a major departure from traditional simulation ideas; instead of simulating single transitions, only conveniently long sequences of transitions are simulated. This idea turns out to remove the previous need for indexing the simulation relation and gives more natural proofs.

Parallelism presents a slight problem to the second technique. Three different proofs for the parallel construct are consequently carried out. The first two build on generalizations of the technique to parallel processes; of these generalizations the second employs a new notion of truly concurrent executions. The third uses a more abstract "external" semantics and the fact that this semantics is compositional in the parallel construct.

Dansk resumé

Nærværende licentiatafhandling er en sammenfatning af nedenstående publikationer, der alle omhandler udvikling og brug af simuleringsteknikker.

I A. Gammelgaard. Reuse of invariants in proofs of implementation. DAIMI PB-360, Aarhus Universitet, 1991.

II A. Gammelgaard. Implementation Conditions for Delay Insensitive Circuits. *Proc. PARLE 1989*, LNCS 365, Vol. 1, pp. 341–355.

III A. Gammelgaard og F. Nielson. Verification of a compiling specification. DAIMI IR-105, Aarhus Universitet, 1991.

IV A. Gammelgaard. Constructing simulations chunk by chunk. DAIMI IR-106, Aarhus Universitet, 1991.

V Kapitel tre og kapitel fire af en kommende monografi dokumenteret i [19]:
A. Gammelgaard, H. H. Løvengreen (ed.), C. Ø. Rump og J. F. Søgaard-Andersen. *Volume 4: Base System Verification*, ProCoS teknisk rapport, ID/DTH HHL 4/1, Danmarks Tekniske Højskole, 1991.

Simuleringsteknikker har været kendt længe inden for teoretisk datalogi. Teknikkerne kan bruges til at bevise, at et program opfylder en specifikation. Teknikkerne bygger på en operationel forståelse af både programmer *og* specifikationer: Den grundlæggende idé er, at hvert skridt i et program modsvares af nul, et, eller flere skridt i specifikationen. Såfremt dette krav, samt eventuelle ekstra krav, er opfyldt, vil programmet opfylde specifikationen.

I *I* benyttes en simuleringsteknik til at vise sikkerhedsegenskaber for implementerende programmer. Først vises en sikkerhedsegenskab for en specifikation af det konkrete program, og derefter vises kravene i simuleringsteknikken at være overholdt; de ønskede egenskaber fås derpå ved nedarvning fra specifikation til implementation.

Det påvises at eksisterende bevisteknikker, i den konkrete anvendelse, ofte fører til dobbelte bevisbyrder; den nedarvede egenskab må genbevises som en del af simuleringsbeviset. Derefter vises det, hvordan en eksisterende teknik kan modificeres, så den dobbelte bevisbyrde fjernes.

I II anvendes den modificerede teknik til at vise hastighedsuafhængighed i en speciel type elektroniske kredse. I stedet for at vise hastighedsuafhængighed direkte, i en model med ledninger, vises det, ved hjælp af den modificerede teknik, at en række implementations-betingelser, i en model uden ledninger, er tilstrækkelige. Disse betingelser udtrykker at kredsen er opbygget af firefase logik.

Publikationerne III, IV og V behandler alle anvendelsen af simuleringsteknikker på et specielt problem: Korrekthed af oversættelser. De betragtede oversættelser går fra forskellige delmængder af programmeringssproget occam-2 til abstrakte versioner af assemblerkode for transputere.

I III tages der udgangspunkt i Milners svage bisimulation. Der argumenteres for, at en serie af modifikationer er nødvendige for at opnå en teknik, der kan bruges på den valgte delmængde af occam-2. Vigtigst er svækkelsen af kravet om en symmetrisk relation begrundet i behandlingen af non-determinisme og tilføjelsen af et indeks til simuleringsrelationen, der gør det muligt at sikre, at intern divergens simuleres af intern divergens.

Korrekthedsbeviset i afhandling III viser sig at være uforholdsmæssigt stort. I IV føres størrelsen delvis tilbage til det faktum, at opbygningen af en simulerende occam-beregning sker ved at sammenstykke delberegninger i occam, der svarer til enkeltskridt på transputeren. Derfor foreslås en ny teknik, hvor kun længere bidder af transputerberegninger simuleres i occam. Den resulterende teknik giver væsentlig kortere korrekthedsbeviser for sekventielle programmer i occam.

Parallelisme kan ikke behandles direkte med den nye teknik; problemet er, at det er vanskeligt naturligt at opsplitte en flettet udførsel af et parallelt program i bidder, således at hver enkelt bid kun indeholder skridt fra en enkelt af de parallelle processer. I afhandling IV foreslås der to generaliseringer af teknikken.

I den første generalisering udnævnes et enkelt skridt i hver bid til at være et "principal" skridt . En occam-simulering opbygges atter ved at sammen-

stykke occam-delberegninger svarende til enkeltskridt på transputeren, men nu vil kun principale skridt blive simuleret af ikke-tomme sekvenser af skridt. I stedet for at relatere globale konfigurationer relateres projektioner af konfigurationer på enkeltprocesser i systemet. For at sikre at hvert skridt resulterer i en konfiguration for hvilken alle projektioner kan relateres, er det nødvendigt at indicere hver relation med en tæller, der måler afstanden til næste principale skridt.

Den anden generalisering af teknikken er mere radikal. I stedet for at ændre grundideen i teknikken ændres den underliggende model for beregning. Den sædvanlige semantik for parallelle processer baseret på fletningen af udførsler af enkeltprocesser erstattes således af begrebet en ikke-flettet udførsel. En sådan udførsel er basalt set en graf og ikke en linær sekvens. Med ikke-flettede udføersler kan bidder af udførsler af enkeltprocesser direkte bruges til at finde bidder (delgrafer) af udførsler af parallelle programmer, og den nye teknik er derfor relativt let at generalisere.

I *IV* tages en anden tilgangsvinkel til behandlingen af parallelisme. I stedet for at prøve at generalisere simuleringsteknikken vises det, at korrekthed af oversættelsen på sekventielle processer er tilstrækkelig: Hvis hver process i et program af parallelle processer implementeres korrekt, så vil den parallelle sammensætning af implementerende processer være en korrekt implementation af programmet. Rækkevidden af dette resultat illustreres også af de øvrige bidrag til [19], hvor det benyttes til at vise korrektheden af kerner, der samarbejder om at afvikle transputerkode.

Acknowledgements

My first thanks must go to Flemming Nielson without whose patient advice, encouragement, and administrative work this thesis would have been much poorer if it had been finished at all; his constructive criticism to drafts of my papers has helped me a lot. Also many thanks to Erik Meineche Schmidt for the many improvements he suggested and for the inspiring talks which I had with him in the early stages of the project.

I have participated in several incentive ProCoS meetings. Especially the cooperation with Hans Henrik Løvengreen, Camilla Rump, and Jørgen Søgaard-Andersen has been stimulating.

Uffe Engberg is thanked for his personal support and for the many interesting discussions we have had through the years that we have been roommates.

Lastly my thanks go to Jørgen Staunstrup who introduced me to delay insensitive circuits and who had the gift of cheering me up—even via e-mail—when a felt down.

The work reported here was supported by a scholarship from Aarhus University and by the ESPRIT BRA project no. 3104: Provably Correct Systems.

Contents

Chapter 1

Introduction

This thesis consists of the present survey and the five papers below.

I A. Gammelgaard. Reuse of invariants in proofs of implementation.
DAIMI PB-360, Aarhus University, 1991.

II A. Gammelgaard. Implementation Conditions for Delay Insensitive Circuits.
Proc. PARLE 1989, LNCS 365, Vol. 1, pp. 341–355.

III A. Gammelgaard and F. Nielson. Verification of a compiling specification.
DAIMI IR-105, Aarhus University, 1991.

IV A. Gammelgaard. Constructing simulations chunk by chunk.
DAIMI IR-106, Aarhus University, 1991.

V Chapters three and four of a forthcoming monograph documented in [19]:
A. Gammelgaard, H. H. Løvengreen (ed.), C. Ø. Rump, and J. F. Søgaard-Andersen.
Volume 4: Base System Verification, ProCoS technical report ID/DTH HHL 411, 1991.

The thesis is about simulation techniques. Simulation techniques are used to establish that one program correctly implements another program or a specification. Such techniques are an important tool in the construction of correct computing systems. The papers introduce various simulation techniques and demonstrate how they can be applied at different stages in the development of correct computing systems.

Computing systems are often described at different levels of abstraction. A high-level description may abstract away many architecture dependent details and may thus be easier to reason about than the corresponding low-level description where these details are present. This observation is often exploited to split up a correctness proof into a series of smaller subproofs: First an abstract description of the program is given as a program in a high level programming language; this program is either proven to meet some specification in logic or is taken to be a specification in itself. Then the program is refined through a series of steps ending with a description right down at the hardware level. For each successive pair of descriptions one has to carry out a proof that the more concrete description correctly implements the more abstract description. If all these proofs can be successfully carried out, then it has been proven that the most concrete description of the computing system satisfies the original specification.

There are different views at the meaning of the phrase "description D' implements description D". Here we shall follow a linear time view [32, 50]. In this view D' implements D if it is possible, for each execution of D', to find a simulating execution of D [1, 22, 28, 31, 38, 59]'[1]. A simulation technique is a proof technique which reduces the obligation of finding simulations in D for executions of D' to simpler conditions relating individual transitions in D and D'.

A number of simulation techniques have been proposed already [1, 22, 28, 31, 38, 48, 59]. This thesis modifies some of the known techniques, presents new techniques, and gives a number of examples of their application. Furthermore results are presented which ensure that double work in some of the techniques can be avoided.

The concept of an implementation arises naturally at several places in the

[1]Descriptions are constructed such that they define non-empty sets of executions. Consequently the trivial implementation having no executions is not feasible.

development of computing systems. Starting from a specification we can identify (at least) the following steps.

1. The initial design of the program is written in some high level programming language. Transformations to the program are then made within the language in order to cope with more and more implementation specific details.

2. The final program is subsequently translated into machine code by a compiler.

3. The machine code is interpreted by the hardware of the machine.

In all these steps it is necessary to prove that the lower level description implements the higher level description. In the latter two steps we are really interested in making *one* proof for a class of programs instead of making individual proofs for each program.

The paper I presents a technique along with a number of examples of its use for the first of the above steps. The paper II uses the same technique in the third step to deal with a specific problem in hardware built of self-timed four phase logic. Finally III, IV, and V present and apply two different techniques for dealing with the second of the above steps. The techniques used in IV and V are the same except for the treatment of parallelism.

Section 2 surveys the technique in I and describes its use on hardware verification in II. Section 3 describes the technique from III and its application to compiling verification. Section 4 describes how the technique of III can be modified into the technique of IV and furthermore demonstrates how the new technique can be applied to parallel languages. In Section 5 the algebraic approach to parallelism from V is described. Finally Section 6 surveys the relevant literature and points out directions for future research.

1.1 Transition systems

Simulation techniques to deal with concurrency is a main thread through the entire thesis. The first concern in all the papers consequently is to model con-

current programs. Numerous models have been suggested in the literature. In this thesis we will focus on models based on transition systems.

We consider two types of communication. In $[I]$ and $[II]$ communication is performed through shared variables; in $[III]$, $[IV]$, and $[V]$ communication is through channels. To deal with the latter form of communication we will, as usual, introduce transition labels, either of form $ch : v$ where ch is a channel and v is a communicated value, or of form τ where τ indicates an internal (silent) step in a system. No such labelling is needed in order to deal with communication through shared variables. Thus we end up with two different types of transition systems, labelled and unlabelled.

Besides a set of configurations (or states) and a transition relation, transition systems often include various other components [8, 49, 51]. Here we will only include a set of initial states. By requiring this set to be non-empty it is ensured that the behaviour of a transition system will be non-trivial.

An unlabelled transition system S is a triple $(C, \mathit{Init}, \rightarrow)$ where

- C is a set of configurations (or states);
- $\mathit{Init} \subseteq C$ is a non-empty set of initial configurations (states);
- $\rightarrow \subseteq C \times C$ is the transition relation.

A labelled transition system S is a 4-tuple $(C, \mathit{Init}, L, \rightarrow)$ where

- C is a set of configurations (or states);
- $\mathit{Init} \subseteq C$ is a non-empty set of initial configurations (states);
- L is a set of labels;
- $\rightarrow \subseteq C \times L \times C$ is the relation of labelled transitions.

An execution of an unlabelled transition system is a finite or infinite sequence of configurations such that the first configuration belongs to *Init* and such that adjacent configurations belong to $\rightarrow$. An execution of a labelled transition system is likewise defined except that the label of the mediating transition has to be inserted between each pair of adjecent configurations.

1.2 Open and closed systems

When modelling concurrent systems another distinction than the labelled-unlabelled one can be made—the distinction between open and closed systems.

An open system is a system with an interface to some external environment. Values are communicated forth and back over this interface. In an unlabelled transition system such communications are usually modelled by picking out some part of the configuration (the shared interface variables) and letting this part be under the control of both the system and its environment (e.g. [1]). In a labelled transition system communications with the environment are usually modelled by selecting some of the labels as being input/output labels (e.g. [28, 31, 38]). The environment then communicates with the system by executing transitions with input/output labels in cooperation with the system.

A closed system is a system which does not communicate with any external agent. Such systems explicitly describe both the program in question and its environment (see e.g. [8] and [47]). In transition systems describing such systems there is no partitioning of neither the individual configurations nor the transition labels to reflect the distinction between the program and its environment. When specifying and constructing such a system the designer instead has to be fully aware of the actions which the program can control and those which it cannot control.

In open systems one can compare two systems at their respective interfaces to the environment. Executions of each system give rise to observations at the interface[2]. If the two systems have the same interface—i.e. the same interface variables or the same input/output labels—, then observations at the two systems' interfaces become simple to compare; as indicated earlier we say that a concrete system implements an abstract system if all observations at the concrete system's interface are also possible observations at the abstract system's interface. The notion of one system implementing another system can thus be expressed by an external criterion relating the two systems at their interface.

[2]These observations need not merely be projected executions; in [IV] and [V] they also contain information about refused interactions.

In closed systems the situation is rather different because of the lack of interfaces at which two systems can be compared. Here it is up to the designer to decide what the important properties of a system are and how to reformulate them for the implementation. Since there is no interface to the environment, the designer has to suggest some interpretation or some relation which shows how to interpret properties of the high level as properties of the low level. On the other hand this enables the designer to refine interfaces of a program: Since there is no formal notion of an interface to the environment in closed systems, the designer is free to substitute certain interface variables or interface communications with others as long as he is able to reformulate and retain the import ant properties of thy system; he is not bound to show that all observations at a predefined interface of the lower level are also possible observationsat a predefined interface at the higher level.

In both $[I]$ and $[II]$ we consider closed systems with communicatio between parallel processes via shared variables. In $[III]$ $[IV]$, and $[V]$ we consider open systems with communication through channels. The lack of papers combining closed systems with channel communication or open systems with communication through shared variables is purely coincidental. We believe that there is nothing which prevents such combinations of concurrency models from being exploited.

Chapter 2

A simulation technique for inheritance of safety properties

The first simulation technique is developed in [*I*] and builds on ideas of Lamport [33, 34, 35]. In this overview we only describe a simple version of the technique.

The technique is used to reduce the effort needed to prove that each state in executions of some low level description of a program satisfy some property. Such properties belong to the category of safety properties [2]. The reduction is obtained by treating the low level description as an implementation of some high level description.

2.1 Basic technique

We need some notation. Given an unlabelled transition system $S = (C, \mathit{Init}, \rightarrow)$, let the set of reachable states *Reach* be the minimal set such that

1. $\mathit{Init} \subseteq \mathit{Reach}$;
2. if $c \in \mathit{Reach}$ and $(c, c') \in \rightarrow$, then $c' \in \mathit{Reach}$.

The importance of this definition is that all states occurring in executions of S will belong to *Reach*. So to prove properties of these states we only need

to prove properties of states in *Reach*.

To demonstrate that a system has a property we must express this property as a set $P \subseteq C$ and we must prove $Reach \subseteq P$. For instance the system could describe an algorithm for achieving mutual exclusion between some number of processes; the property P would then consist of all states in which at most one process is in its critical section. We say that P is *safe* in S whenever $Reach \subseteq P$.

Traditionally, to prove that a P is safe in S, one finds a property $Inv \subseteq C$ and proves

1. $Init \subseteq Inv$;
2. if $c \in Inv$ and $(c, c') \in \rightarrow$, then $c' \in Inv$;
3. $Inv \subseteq P$.

The first two conditions obviously ensure $Reach \subseteq Inv$ and the third condition then gives $Reach \subseteq P$. The property Inv is called an *invariant* whenever it satisfies 1 and 2.

For systems describing simple algorithms the invariant Inv can often be chosen to be equal to the property P. For systems describing more complicated algorithms the invariant has to be strengthened such that Inv becomes a proper subset of P and a proper superset of $Reach$.

In concrete systems containing a lot of architecture dependent details invariants can be difficult to find and to work with. In a realisation of the mutual exclusion algorithm, for example, it could be necessary to implement some shared data structure by a whole set of low level data-structures; the invariant then has to describe how manipulations change the lower level data-structures along with the high level properties which enforce mutual exclusion. This mixture of high and low level properties may make the invariant difficult to work with and consequently it may be difficult to establish that P is safe in S by these methods.

We want instead to prove P safe in S in two steps; first we prove it safe for an abstract description of the system; and next we prove that the concrete system can be treated as an implementation of the abstract system. This will enable us to inherit P from the abstract to the concrete level.

So assume that we are given two transition systems $S_1 = (C_1, Init_1, \rightarrow_1)$ and $S_2 = (C_2, Init_2, \rightarrow_2)$ which describe closed systems and for which S_1 is supposed to be an implementation of S_2. Since S_1 and S_2 describe closed systems we do not have a priori interfaces at which the systems can be compared. Instead we have to explicitly relate the states of the two systems. This must be done by defining a map $\alpha : C_1 \rightarrow C_2$. This map should be thought of as abstracting away the implementation specific details in S_1.

The next thing we must do is to prove that transitions in S_1 are simulated by transitions in S_2. This is not always quite possible, however, because the more refined data structures of S_1 may require more operations than the corresponding high level data structures of S_2. Consequently we also allow S_1 transitions to be simulated by transitions which simply repeat the configuration (called *stuttering* transitions in [1, 33, 35].)

Given two transition systems S_1 and S_2 and a map $\alpha : C_1 \rightarrow C_2$ the simulation conditions then become

1. $\alpha(Init_1) \subseteq Init_2$;
2. if $(c, c') \in \rightarrow_1$, then $\alpha(c) = \alpha(c')$ or $(\alpha(c), \alpha(c')) \in \rightarrow_2$.

If these conditions hold, then executions of S_1 map into (possibly stuttered) executions of S_2. Thus for any property P_2 which is safe in S_2 we also have that the inverse image $\alpha^{-1}(P_2)$ of P_2 is safe in S_1. So properties can be inherited.

2.2 Extensions

Thus far the development has not been materially different from Lamport's original ideas. The subsequent extensions will be a deviation and are the main contribution of the paper [I]. As will be discussed, the first extension has a parallel in [38] whereas the second part is new.

After constructing the mapping α it often turns out that some transitions in $\rightarrow_1$ do not map as desired (examples are given in both [I] and [II]). This does not always detrude the technique since actually only the transitions

$(c, c') \in \rightarrow_1$ with $c \in \mathit{Reach}_1$ need to map as described above. Consequently one may have to restrict somehow the set of S_1-transitions which are investigated. This means that one has to find a property P_1 which can be proven safe in S_1; and this may once again involve finding an invariant Inv_1 of S_1 such that $\mathit{Inv}_1 \subseteq P_1$.

The proof of safety of Inv_1 often turns out to be double work: Apparently, to inherit a property we first have to prove the safety of another property which is nothing but a sharpened version of the property that we want to inherit. For instance, to prove that all low level transitions map as desired we may have to exploit mutual exclusion between the processes. And if mutual exclusion is the property that we want to inherit, then the technique does not buy us anything; we anyway have to give a traditional proof of mutual exclusion at the low level.

First extension

There is a quite easy remedy in many of the above situations. It turns out that one can simply inherit the properties an use them as restrictions to the set of investigated transitions; if the subsequent simulation proof goes through, then, as proved in [*I*] page 13, it turns out that reachable transitions are mapped as desired.

A similar result has been reported by Lynch and Tuttle in [38]. Their setup is slightly different, however; formulated in our framework their result is that if all transitions $(c, c') \in \rightarrow_1$ such that $c \in \mathit{Reach}_1$ and $\alpha(c) \in \mathit{Reach}_2$ map as desired, then all properties of S_2 can be inherited. The condition $\alpha(c) \in \mathit{Reach}_2$ corresponds to our usage of $\alpha^{-1}(\mathit{Reach}_2)$ as an S_1-property in the "Characterisation Lemma 1" on page 13 in [*I*],

Second extension

Still there are examples for which the thus strengthened technique—and hence also Lynch and Tuttle's—will fail (again see [*I*] and [*II*]). It may be necessary to further shrink down the set of considered transitions. In mutual exclusion algorithms it may thus be necessary—in order to make the simulation proof go through—to know more about the low level system than

just the mutual exclusion property. The question then is whether inherited properties can be used in proofs of additional low level properties. It turns out that they can. The proof of such relative invariants can be incorporated into the simulation technique (pages 16-17 in [*I*]). The conditions for inheriting properties then become that there should exist some $Inv_1 \subseteq C_1$ such that the following conditions hold

1. $\alpha(Init_1) \subseteq Init_2$ and $Init_1 \subseteq Inv_1$;
2. if $(c, c') \in \rightarrow_1$ and $c \in Inv_1 \cap \alpha^{-1}(Reach_2)$, then $c' \in Inv_1$;
3. if $(c, c') \in \rightarrow_1$ and $c \in Inv_1 \cap \alpha^{-1}(Reach_2)$, then $\alpha(c) = \alpha(c')$ or $(\alpha(c), \alpha(c')) \in \rightarrow_2$.

In [*I*] a number of examples are given on how to use this simulation technique (it is called an inheritance technique in [*I*] page 17 because the above checks allow one to inherit properties). It is also described how more general safety properties than those expressible by a safe property can be inherited. E.g. the paper considers properties of the form "process i cannot overtake process j in queue q".

Finally it is discussed in [*I*] what completeness of the simulation technique means when dealing with closed systems. As mentioned previously it is not possible to compare two closed systems independently of an interpretation or relation between the two systems.

So a traditional completeness result saying that if S_1 implements S_2, then the simulation technique can be applied is not adequate here. Since part of the simulation technique in [*I*] is to construct the map α, the designer has full control over this interpretation. We exploit this fact to state a different kind of result which says that under very mild restrictions on S_2 it is possible to obtain any safety property of S_1 as an inherited property. This is not a traditional completeness result but it has, unfortunately, been called a completeness result in [*I*].

2.3 An application to delay insensitive circuits

In $[II]$ the technique from $[I]$ is used to facilitate the construction of delay insensitive hardware.

A circuit is delay insensitive if it is robust to variations in delays of signals through the wires of the components of the circuit [58, 63]. To model such circuits one has to describe explicitly the possibility of delays in wires. In a shared variable model this is done by introducing two bit-valued variables $w!$ and $w?$ for each wire w. These two variables model the two endpoints of the wire, and the delay is modelled by an assignment

$$w? := w!$$

For wires the condition of delay insensitivity then says that $w!$ may not change value as long as $w? \neq w!$, i. e. the circuitry may not attempt sending a new value via w when the old value is still in progress.

The explicit modelling of delays in wires tends to complicate the correctness arguments for delay insensitive circuits. The aim in $[II]$ is to reduce this complexity by allowing the designer to reason about a more abstract model. In the more abstract model each wire w is simply represented by a single variable. Instead of treating the delays explicitly the designer is requested to verify a number of so-called implementation conditions. These implementation conditions are nothing but safe properties of the abstract model; they express that the circuit should follow the rules for so-called self-timed four-phase logic (see [58]). An application of the simulation technique from $[I]$ then formally proves the claim that four-phase logic is insensitive to delays in wires.

Chapter 3

A simulation technique for compiling correctness

The second simulation technique is documented in [*III*].

The aim in [*III*] is to prove a compiling specification correct. The source language of the compilation is called PL_0 (see [15]) and is a subset of the occam-2 programming language [26]. A PL_0 program is a sequential process prefixed with declarations. Declarations introduce integer valued variables. The sequential process communicates with the environment via one input and one output channel. Sequential processes are constructed front primitive statements—input/output statements, the two processes `SKIP` and `STOP`—and these processes are composed into compound sequential, conditional and iterative processes.

Finiteness is an important aspect of PL_0: integers are assumed to lie in the interval $[-2^n, 2^n - 1]$ for some fixed n and the workspace needed by processes for allocation of both permanent and temporary variables is assumed to be bounded.

The target language of the compilation is called BAL_0 (see [17]) and is a block structured assembly language for the transputer [27]. The block structure is used to avoid difficulties in generating unique jump labels. Input and output is performed through the same two predefined channels as in PL_0.

In [16, 17] the languages PL_0 and BAL_0 are given Plotkin style structural op-

erational semantics [49]. Such a semantics defines for each of PL_0 and BAL_0 a single global transition system which resembles, but is slightly different from our labelled transition system. Instead of a set of initial configurations the Plotkin style semantics works with a set of terminal configurations. The set of terminal configurations is only used to define the transitions through an inference system and we will ignore it for the present. Our set of initial configurations is implicitly defined by any concrete program that we consider. In the subsequent discussion we will consequently assume that for each program in PL_0 or BAL_0 we have a labelled transition system as defined earlier.

The shape of PL_0-configurations depend on the syntactic category under consideration. For processes, e.g., they have shape $\langle p, \sigma \rangle$ where p is a process term and σ is a store. In BAL_0 the configurations have shape $\langle [u, v], ms \rangle$ here u and v are BAL_0 code sequences, where the comma indicates the position of the program counter, and where ms is a machine state indicating the contents of certain registers and the workspace.

The compiling specification is given as an ordinary, recursively defined, function t from PL_0 to BAL_0 (see [18]).

3.1 Definition of correctness

Let S_p, be the labelled transition system defined be the PL_0 program p and let $S_{t(p)}$ be the labelled transition system defined by the corresponding compiled program $t(p)$. We will take t to be correct if $S_{t(p)}$ correctly implements S_p for each p. Since p and $t(p)$ describe open systems with the same interface, we require each observation of $S_{t(p)}$ to be an observation of S_p.

This definition has not been formally stated in $[III]$. It has been used, however, to give an informal argument for the shape of what we will call the internal correctness predicate. Part of this argument will be repeated here.

The internal correctness predicate expresses correctness as a requirement on individual transitions and simulations of these. Thus it highly resembles a proof obligation in a simulation technique. It should be noted, however, that the only purpose of the internal correctness predicate is to ensure that $S_{t(p)}$ implements S_p, for each p. It is another matter whether the internal correctness predicate can serve as induction hypothesis in a proof by induction.

Quite in analogy with the need to consider an $inv \subset P$ in the previous technique it turns out to be necessary to considerably strengthen the internal correctness predicate in order to get a working induction hypothesis.

3.2 Towards an internal correctness predicate

The internal correctness predicate in [*III*] page 25 arose through a series of modifications to a predicate taken from the literature. Our initial attempt at expressing correctness of the compiling specification was to use the well-known notion of observation equivalence from [42] page 108. For the present we assume that both $Init_p$ and $Init_{t(p)}$ consist of just a single configuration. Then we say that S_p and $S_{t(p)}$ are observation equivalent if there exists a relation $\mathbf{R}$ in $C_p \times {C_{t(p)}}^3$ such that $Init_p \mathbf{R} Init_{t(p)}$ and such that

1. if $c_p \mathbf{R} c_{t(p)}$ and $c_p \xrightarrow{l} c'_p$, then there is a $c'_{t(p)}$ such that $c_{t(p)} \stackrel{\hat{l}}{\Longrightarrow} c'_{t(p)}$ and $c'_p \mathbf{R} c'_{t(p)}$;

2. if $c_p \mathbf{R} c_{t(p)}$ and $c_{t(p)} \xrightarrow{l} c'_{t(p)}$ then there is a c'_p such that $c_p \stackrel{\hat{l}}{\Longrightarrow} c'_p$ and $c'_p \mathbf{R} c'_{t(p)}$.

Here the expression $\hat{l}$ yields the empty sequence ε if $l = \tau$ (the label indicating an invisible step) and it yields the sequence consisting of just l otherwise. The expression $c \stackrel{s}{\Longrightarrow} c'$ means that there is a sequence of transitions from c to c' such that the non-τ labelled transitions are labelled, in sequence, with the labels in s.

It was soon realized that observation equivalence is not an adequate internal correctness notion in the ProCoS project: In successive stages of the project PL_0 is extended with non-deterministic constructs. Non-determinism is then treated as underspecification. This means that if p is non-deterministic, then $t(p)$ may be made deterministic by cutting off some of the internal choices made by p.

[3] The two sets of configurations are swapped in [*III*].

The treatment of non-determinism as underspecification makes it impossible to meet requirement 1 above: If there is a sequence of high level transitions

$$c_p \xrightarrow{l} c'_p \xrightarrow{l'} c''_p \ldots$$

then there is no guarantee that we have a low level simulation

$$c_{t(p)} \stackrel{\hat{l}}{\Longrightarrow} c'_{t(p)} \stackrel{\hat{l'}}{\Longrightarrow} c''_p \ldots$$

as would be the case if requirement 1 were met. We weaken requirement 1 by removing the last condition

$$c'_p \mathbf{R} c'_{t(p)}$$

So the simulation is not required to continue from the pair of configurations $(c'_p, c'_{t(p)})$.

On the other hand we do not want to completely remove requirement 1 for the remaining condition

$$c_{t(p)} \stackrel{\hat{l}}{\Longrightarrow} c'_{t(p)}$$

ensures that external communication capabilities at the high level are reflected at the low level. We will state this condition differently, however, since we want to use the mental picture of constructing high-level simulations for low level executions. So we formulate the requirement contrapositively: if the low level can refuse to participate in some external communication, then so can the high level.

To express the modified requirement we only have to analyse the circumstances under which a communication can be refused.

If the program reaches a configuration $c_{t(p)}$ where execution has stopped, then certainly no communication is possible. We will write that the execution has stopped as

$$\text{for all } l : c_{t(p)} \xrightarrow{l} \!\!\!\!/ \ .$$

In this case we require that if $c_p \mathbf{R} c_{t(p)}$ holds, then it is possible from c_p via some number of τ-steps to reach a configuration where also p has stopped execution.

At both levels we have that if a configuration is reached where the program is waiting for communication via channel ch (that is $c \xrightarrow{ch:v} c'$ for some v and c'), then it simultaneously refuses to communicate via any other channel. But requirement 2 already ensures that low-level transitions are simulated by high-level transitions with the same label so if the low level refuses some communication by being ready to communicate via ch, then so does the high level. We consequently do not need any extra condition to express that the high level simulates this kind of low-level refusal.

Finally the low level can refuse to make any communication by diverging internally. For the translation from PL_0 to BAL_0 it turns out that this only happens when the high level also diverges internally. We want our correctness predicate to express also this simulation of internal divergence by internal divergence. It is not possible to use requirement 2 directly for this because this requirement only states that each internal low level transition $c_{t(p)} \xrightarrow{\tau} c'_{t(p)}$ be simulated by some, maybe empty, sequence of high-level transitions $c_p \stackrel{\hat{\tau}}{\Longrightarrow} c'_p$. If all but finitely many of these sequences *were* empty, then an internal divergence could be simulated by a finite sequence of transitions in which the last configuration could be able to accept communication via some channel. Consequently some low level refusals would not be simulated at the high level.

To express that infinitely many of the simulations $c_p \stackrel{\hat{\tau}}{\Longrightarrow} c'_p$ must be non-empty we index the simulation relation $\mathbf{R}$ by elements w from a well-founded order $(W, <)$. Requirement 2 is then hanged to say that if $c_p \mathbf{R}_w c_{t(p)}$ holds and $c_{t(p)} \stackrel{l}{\Longrightarrow} c_{t(p)'}$, then there must be a configuration c'_p such that $c_p \stackrel{\hat{l}}{\Longrightarrow} c'_p$ and such that $c'_p \mathbf{R}_{w'} c'_{t(p)}$ holds with $w' < w$ whenever $c_p \xrightarrow{\hat{l}} c'_p$ is an empty sequence; the index, then, is only allowed to increase when $c_p \stackrel{\hat{l}}{\Longrightarrow} c'_p$ is non-empty. Since $(W, <)$ is well-founded, this means that when constructing a simulation of a divergent computation we occasionally have to choose non-empty simulations of transitions and this forces the constructed simulating computation to be infinite.

Two additional comments are necessary before we can present the resulting correctness predicate.

First, the sets $Init_p$ and $Init_{t(p)}$ of initial configurations do not consist of just a single state each. E.g. the program p really defines a bunch of initial configurations, one for each choice of initial value for the variables. We consequently cannot require $\mathbf{R}$ to relate all of $Init_p$ with all of $Init_{t(p)}$. Since we want to construct high level simulations of low level executions we instead require that for each $c_{t(p)} \in Init_{t(p)}$ there should exist a $c_p \in Init_p$ such that $c_p \mathbf{R}_w c_{t(p)}$ holds for some w.

Next, we have not yet taken into account the limited size of workspace and of transputer words. This is taken into account by recursively defining a set of functions which measure the words needed by each subpart of the program and the necessary word-size for jumps within the program. A program is then said to be compilable whenever the total workspace needed is less than the available workspace and whenever all jumps can be implemented with the available word-size.

The resulting correctness predicate can be summed up as follows. Let S_p be the labelled transition system defined by p and let $S_{t(p)}$ be the labelled transition system defined by $t(p)$. We say that t is correct if for each compilable program p there exists a family of relations $\mathbf{R}_w$ such that

1. if $c_p \mathbf{R}_w c_{t(p)}$ and $c_{t(p)} \xrightarrow{l} c'_{t(p)}$, then there is a c'_p such that $c_p \stackrel{\hat{l},m}{\Longrightarrow} c'_p$ and $c'_p \mathbf{R}_{w'} c'_{t(p)}$ where $m = 0$ implies $w' < w$;
2. for each $c_{t(p)} \in Init_{t(p)}$ there is a $c_p \in Init_p$ and a $w \in W$ such that $c_p \mathbf{R}_w c_{t(p)}$;
3. If $c_p \mathbf{R}_w c_{t(p)}$ and for all $l : c_{t(p)} \not\xrightarrow{l}$, then there is a c'_p such that $c_p \stackrel{\epsilon}{\Longrightarrow} c'_p$ and for all $l : c'_p \not\xrightarrow{l}$.[4]

3.3 The correctness proof

We next turn to how the proof of this correctness predicate is conducted. We are required, for each p, to define an appropriate family of relations $\mathbf{R}^p_w$ and

[4] In [*III*] page 25 the predicate is slightly stronger since we distinguish between proper and improper termination. This is due to the view that it is possible to communicate with the system after its termination to see how it terminated.

to prove that it satisfies the three conditions. Basically we want to reuse the corresponding families of $\mathbf{R}_w^q$-relations defined for each subconstruct q of p along with the properties enjoyed by them. This could be done by defining the relations $\mathbf{R}_w^p$ by recursion on the structure of p. It turns out, however, that for some p it is inconvenient to give a direct closed form definition of $\mathbf{R}_w^p$ For instance the process $\mathtt{WHILE}(b, p)$ gets packed into a sequence $\mathtt{SEQ}[\cdot]$ for each iteration so that the process becomes $\mathtt{SEQ}^n[\mathtt{WHILE}(b, p)]$ after n iterations whereas the BAL_0 program counter just cycles n times through the code; for this process we rather want from

$$\langle p, \sigma\rangle \mathbf{R}_w^{\mathtt{WHILE}(b,p)} \langle [u, v], ms\rangle$$

to infer

$$\langle \mathtt{SEQ}[p], \sigma\rangle \mathbf{R}_w^{\mathtt{WHILE}(b,p)} \langle [u, v], ms\rangle$$

This suggests to use inference systems for the definition of $\mathbf{R}_w^p$ for each p. Since definition by inference systems encompasses definition by recursion in p we have made the choice to give a simultaneous definition of all $\mathbf{R}_w^p$ by means of an inference system. The proof that the three above requirements are met is consequently conducted by induction on the inference trees of relation instances $c_p \mathbf{R}_u^p (c_{t(p)}$.

As already mentioned the three requirements have to be strengthened in order to serve as an induction hypothesis in the induction proof. Consider for instance the process $\mathtt{SEQ}[p_1, p_2]$ where p_1 translates into u and p_2 translates into v. When the process p_1 has been executed, then one needs to relate $\langle p_2, \sigma\rangle$ to $\langle [u, v], ms\rangle$. This is only possible if ms is a proper representation of σ. And consequently it has to be explicitly expressed that if some BAL_0 transition is simulated by some sequence of high level transitions, then the final ms actually *is* a proper representation of σ.

Even though PL_0 is just a small language, the proof in [III] is large; it runs over more than 40 pages. It seems that there are (at least) three different reasons for this.

First, the languages and the compiling specification have been defined independently of the proof effort. This means that all the "fine-tuning" of the definitions which usually precedes the publication of similar proofs has not been possible here.

Second, the bounded size of transputer words and of the available memory makes it necessary to make a lot of extra checks. It has to be demonstrated that references to words do not fall outside the workspace in use; and for each arithmetic operation one has to distinguish between situations with normal and abnormal outcomes.

Third, and most importantly, the proof technique itself seems to introduce complications since it requires us to find simulations for every single transition at the low level. This means that we are often forced to relate quite awkwardly. For instance a process like $ch!x$ is translated to a BAL_0 sequence of seven instructions. So execution of the construct requires seven steps at the low level but only a single step at the high level and the problem then is to find out with what high level configuration the six intermediate configurations of the low level should be related.

Chapter 4

Chunk by chunk simulation

The third simulation technique is documented in [IV]. The technique is an attempt to cut down on the size of the proof for sequential processes in [III]. Furthermore the paper gives two different ways to extend the third simulation technique to deal with parallelism. The sequential part of the technique and its use is also documented in [V] and the presentation there is a slight improvement over the presentation in [IV].

The programming language PL in [IV] page 7 is both an extension and a simplification of PL_0 from [15]. It contains parallelism at the outermost level, an arbitrary number of channels can be used for communication, and a `CHOICE`-construct introducing internal non-determinism has been included. On the other hand the declarations have been removed (they were only present in PL_0 in order to make the statical semantics of PL_0 non-trivial) and integers may take on arbitrarily large values.

The assembly language AL is adjusted accordingly. Furthermore configurations of a running AL program are made more low level as they now contain an instruction pointer pointng into the code instead of the comma notation in BAL_0.

4.1 In search for a better technique

The new simulation technique is an attempt to overcome two problems with the previous simulation technique. We want to get rid of the indices w on the simulation relation and we want to relax the strict demand of relating high and low level configurations after each transition at the low level. It will turn out that a single idea overcomes both problems.

The indices w were introduced to ensure that infinite sequences at the low level are simulated by infinite sequences at the high level. We now try to ensure this differently.

The first suggestion is to require that each transition be simulated by a non-empty sequence of high level transitions. This of course forces infinite executions to be simulated by infinite executions. But it also has the consequence that high level executions must contain more transitions than the low level executions which they simulate and this is clearly undesirable.

To avoid forcing the high level execution to contain more transitions we suggest to find simulations only for chunks of low level executions and not for individual transitions. So our requirement will be that if $c_p \mathbf{R} c_{t(p)}$ holds and $c_{t(p)} \to c'_{t(p)}$ is the first transition of a sufficiently long low level execution, then it is possible to find a prefix $c_{t(p)} \to c'_{t(p)} \Rightarrow c''_{t(p)}$ of this execution and a non-empty high level simulation $c_p \Rightarrow c''_p$ such that $c''_p \mathbf{R} c''_{t(p)}$ holds. This again ensures that infinite low level executions are simulated by infinite high level executions and now high level simulations are not forced to contain more transitions than the low level executions.

The sketched idea is also appropriate for relaxing the requirement of relating configurations after each low level step. Consider for instance the high level process $x := e$ and its translation $\mathit{eval}(e); \mathit{stl}(x)$[5]. They give rise to the following executions.

$$\begin{array}{ll} \text{high level} & \langle x := e, \sigma\rangle \xrightarrow{\tau} \langle \mathbf{terminated}, \sigma \oplus \{x \mapsto e(\sigma)\}\rangle \\ \text{low level} & \langle 0, \mathit{reg}, \sigma\rangle \xrightarrow{\tau} \langle 1, e(\sigma), \sigma\rangle \xrightarrow{\tau} \langle 2, e(\sigma), \sigma \oplus \{x \mapsto e(\sigma)\}\rangle \end{array}$$

In PL the value $e(\sigma)$ of the expression e evaluated in store σ is saved in

[5]For simplicity we assume the existence of AL-instructions which evaluate expressions in one step.

variable x whereby the process becomes terminated. In AL the instruction pointer first moves from 0 to 1 while $e(\sigma)$ is loaded into the general purpose register; next the instruction pointer moves from 1 to 2 while the value in the general purpose register is stored in variable x.

The problem in the previous simulation technique is that the intermediate configuration $\langle 1, e(\sigma), \sigma\rangle$ is required to be related to some of the two high level configurations and neither choice seems to be natural. The best would be simply to avoid relating $\langle 1, e(\sigma), \sigma\rangle$ to anything while relating only $\langle 0, \mathit{reg}, 0\rangle$ to $\langle x := e, \sigma\rangle$ and $\langle 2, e(\sigma), \sigma \oplus \{x \mapsto e(\sigma)\}\rangle$ to $\langle \mathbf{terminated}, \sigma \oplus \{x \mapsto e(\sigma)\}\rangle$. The seketched idea allows us to do just so by grouping the two low level transitions into a chunk such that only the initial and the final configurations need to be related to high level configurations.

Unfortunately the sketched idea does not directly work. There are both pragmatic and theoretical reasons for this.

The sketched idea requires us to match non-empty low level sequences of transitions with non-empty high level sequences. This is not always as natural as we may hope for. Consider e.g. the process

$$p_1 \texttt{ SEQ SKIP SEQ } p_2$$

The process `SKIP` just translates into the empty code sequence so the result of translating the above process is the code sequence

$$t(p_1); t(p_2)$$

At the high level we get the execution

$$\cdots \langle \texttt{SKIP SEQ } p_2, \sigma\rangle \xrightarrow{\tau} \langle p_2, \sigma\rangle \cdots$$

So it takes one transition to remove the process `SKIP`. At the low level there is no such transition

$$\cdots \langle i, \mathit{reg}, \sigma\rangle \cdots$$

The problem then is to what simulation the `SKIP` transition should belong; it could belong either to the simulation of a chunk ending in $\langle i, \mathit{reg}, \sigma\rangle$ or to the simulation of a chunk beginning in $\langle i, \mathit{reg}, \sigma\rangle$. Neither choice seems attractive. Rather it seems desirable to have the ability of inserting an *empty*

chunk between the chunks ending and beginning in $\langle i, reg, \sigma\rangle$ and to simulate this *empty* chunk by the transition

$$\langle \texttt{SKIP SEQ}\ p_2, \sigma\rangle \longrightarrow \langle p_2, \sigma\rangle$$

The reverse effect can also be observed—that some low level transitions most naturally are simulated by empty sequences of high level transitions. This effect occurs e.g. when executing a loop $\texttt{WHILE}(b, p)$. At the low level, after executing the body of p, there is an explicit jump transition which leaves control at the beginning of the loop. At the high level there is no such transition, rather the last transition in the execution of p leads directly to a configuration of form $\langle \texttt{WHILE}(b, p), \sigma\rangle$. Now it would be desirable to simulate the non-empty chunk consisting of the jump transition by an *empty* high level execution.

As we did in the previous simulation technique, we also want the chunk by chunk technique to ensure that low level refusals are simulated by high level refusals. This is the origin of the theoretical problem with the sketched idea.

To explain the problems consider first a finite low level execution for which a simulating high level execution can be found. We can identify two necessary requirements in the technique. To start the construction of the high level execution we must require that the initial low level configuration and the initial high level configuration are related.

Furthermore, to ensure that the two executions define the same set of refusals, it must be the case that the two final configurations enable the same set of communications. Since the two final configurations will be related, this is ensured by requiring that related configurations enable the same set of communications.

But now consider the process $ch!e$ and its translation $eval(e); out(ch)$ which give rise to the following executions

$$\begin{array}{ll} \text{high level} & \langle ch!e, \sigma\rangle \overset{ch:e(\sigma)}{\longrightarrow} \langle(\mathbf{terminated}, \sigma\rangle \\ \text{low level} & \langle 0, reg, \sigma\rangle \overset{\tau}{\longrightarrow} \langle 1, e(\sigma), \sigma\rangle \overset{ch:e(\sigma)}{\longrightarrow} \langle 2, e(\sigma), \sigma\rangle \end{array}$$

It is easily seen that the two above requirements collide in this case: First we are required to relate $\langle ch!e, \sigma\rangle$ with $\langle 0, reg, \sigma\rangle$ and next we are required to

show that the thus related configurations enable the same set of communications which is clearly not the case since $\langle ch!e, \sigma\rangle$ enables a communication along ch whereas $\langle 0, reg, \sigma\rangle$ does not. Again in this case it seems desirable to simulate the first transition from $\langle 0, reg, \sigma\rangle$ to $\langle 1, e(\sigma), \sigma\rangle$ by an *empty* high level execution such that $\langle ch!e, \sigma\rangle$ and $\langle 1, e(\sigma), \sigma\rangle$ which enable the same communications can subsequently be related. Furthermore the relation between the two initial configurations should be of another sort not requiring the same set of communications to be enabled.

As hinted at both the pragmatic and the theoretical problems can be solved by introducing two relations **S** and **R** with different requirements. If two configurations are **S**-related, then they enable the same set of communications and from any sufficiently long execution it is possible to cut out a non-empty chunk which has a non-empty high level simulation such that the respective final configurations are **R**-related. **R**-related configurations need not have the same communication capabilities as S**S**-related. We only require that if two configurations are **R**-related, then it is possible from any execution originating in the low level configuration to cut out a chunk which has a high level simulation such that the respective final configurations are **S**-related.

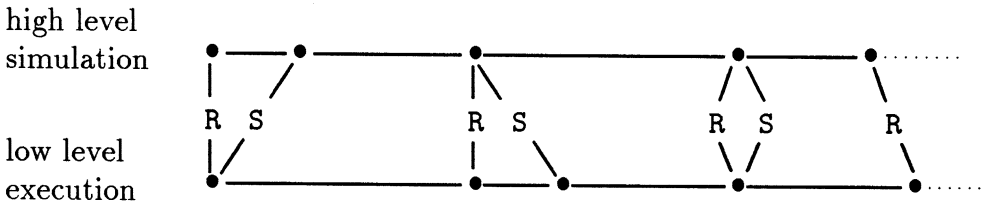

Figure 4.1: The chunk-by-chunk technique on infinite executions.

When using the **R**- and **S**-relations to construct simulations of low level executions we end up with a picture like in figure 1 for infinite executions. For finite executions a typical picture is given in figure 2. Notice that a finite execution and its simulation must end in a pair of configurations which are **S**-related.

To express the requirements on **R**- and **S**-related configurations formally we have to express that an execution γ is sufficiently long. In $[IV]$ the approach taken is to require γ to be complete which means that γ cannot be extended with internal transitions (page 15).

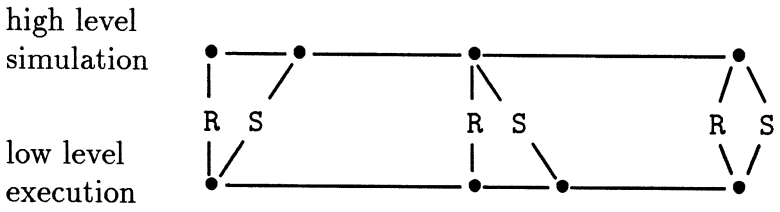

Figure 4.2: The chunk-by-chunk technique on finite executions.

In the following formalization we assume that γ is any complete execution. The notation $\beta \sqsubseteq \gamma$ means that β is an initial chunk of γ, i.e. β is a prefix of γ. The initial configuration of γ is denoted ${}^{\bullet}\gamma$. Finally the function *enable* yields the set of enabled channels when applied to a PL configuration pc and the (code-dependend) function $enable_{t(p)}$ yields the set of enabled channels (in code $t(p)$) when applied to an AL configuration mc. We say that t is correct if, for each p, it is possible to find $\mathbf{R}^p$ and $\mathbf{S}^p$ satisfying the below requirements

1. $\Downarrow$ $\quad pc\mathbf{R}mc \wedge b = {}^{\bullet}\gamma$
   $\exists \alpha, \beta : (\alpha = pc \stackrel{\varepsilon}{\Rightarrow} pc') \wedge (\beta = mc \stackrel{\varepsilon}{\Rightarrow} mc') \wedge (\beta \sqsubseteq \gamma) \wedge pc'\mathbf{S}mc'$

2. $\Downarrow$ $\quad pc\mathbf{S}mc \wedge \gamma = mc \stackrel{l}{\longrightarrow} \cdots$
   $\exists \alpha, \beta : (\alpha = pc \stackrel{l}{\Rightarrow} pc') \wedge (\beta = mc \stackrel{l}{\Rightarrow} mc') \wedge (\beta \sqsubseteq \gamma) \wedge pc'\mathbf{R}mc'$

3. $pc\mathbf{S}mc \Rightarrow (enable(pc) = enable_{t(p)}(mc))$

4.2 The correctness proof

As in the previous simulation technique we define the two relations $\mathbf{R}^p$ and $\mathbf{S}^p$ by an inference system and the proof that the above requirements hold is consequently performed by induction on the structure of inference trees.

Again it is necessary to strengthen the above predicate in order to get a working induction hypothesis. The strengthening is only very slight; condition 3 has to be strengthened into

$3'.\ pc\mathbf{S}mc \Rightarrow (enable(pc) = enable_{t(p)}(mc)) \wedge (pc \in T_{\mathrm{PL}} \Leftrightarrow mc \in T_{\mathrm{AL}}^{t(p)})$

The stronger version $3'$ is necessary to deal with the `SEQ`-construct, It ensures that an **S** to **R** chunk inside a $t(p_1); t(p_2)$-execution corresponds to an **S** to **R** chunk of a $t(p_1)$-execution whenever the instruction pointer of the $\mathbf{S}^{\mathtt{SEQ}[p_1,p_2]}$-related configurations points to a location inside $t(p_1)$. And it enables us to treat a special problem which arises, namely the problem that both the final chunk of $t(p_1)$ and the initial chunk of $t(p_2)$ will be of the **R** to **S** type. If no countermeasures were taken, this would lead to a violation of the principle that the two types of chunks should strictly alternate. We will not go deeper into the technical motivation for using $3'$ but refer to $[V]$ page 65.

4.3 External semantics

In the discussion leading up to the internal correctness predicates in both the previous and the present technique we used arguments which build on our expectations to the externally observable behaviour of transition systems. These expectations can be formalized. When this is done it becomes possible to give an external definition of the relationship that one transition system implements another. Furthermore it becomes possible to formally prove that the simulation techniques establish that the transition system given by $t(p)$ implements the transition system given by p for each p.

As mentioned previously the external semantics is only used as an informal guide in $[III]$. In $[IV]$ the external semantics is treated fully formally. There are *two* definitions of external semantics, however, due to different treatments of parallelism. Since these two definitions coincide for programs consisting of just a single sequential process, we will take the opportunity to sketch the external semantics of sequential processes. Furthermore we describe how correctness of sequential processes can be expressed by this external semantics and how the internal correctness predicate is correct with repsect to the external notion of correctness.

In correspondence with our general linear time view (see the Introduction) we take the external semantics of a system to be a set of observations where each observation is derived from an execution of the system. There are two

things which can be observed in an execution. The first is the sequence of communications with the environment—the trace [6]. The second is the set of channels along which the system refuses to communicate—the refusal set [6].

Although we follow the failure semantics of [6] in the structure of observations we do not follow it in the interpretation of refusal sets and in the treatment of diverging processes. In [6] a channel is only refused if the execution ends in a stable configuration (a configuration not allowing internal transitions) in which a communication labelled with the channel is not possible. In our approach we take a channel to be refused if it is refused—under an assumption of weak fairness—in the long run. More precisely, in addition to the refusals in [6] we also regard a channel as being refused in an infinite execution if communication via the channel only occurs a finite number of times and if communication along the channel is infinitely often disabled. The latter requirement says that we can only force the system to participate in a communication if the system is continuously prepared to do so from some point in the execution.

Notice that a system which diverges internally will refuse every channel. This is in contrast with failure semantics where diverging processes are treated to be equivalent with the chaotic process from the point where the divergence starts. The chaotic process can exhibit any behaviour, also a behaviour where no channel is refused.

It is now easy to express that transition system S' implements transition system S. With the view that non-determinism in S is just a means of underspecification we get that every observation made of S' should also be a possible observation of S—in other words the observations of S' should be included in the observations of S. If we let $[\![S]\!]$ and $[\![S']\!]$ denote the external semantics of S and S' respectively, then the formal statement of S' implements S becomes

$$[\![S']\!] \subseteq [\![S]\!]^6$$

[6]This inclusion property could be weakened slightly to correspond more closely with our informal explanation. If some refusal set is observed in an execution at the low level, then it is not necessary to require that the constructed high level simulation defines the *same* refusal set as the low level execution; it is sufficient to require that the simulation defines a larger refusal set since refusing a set also means refusing each of its subsets. This

With this definition we say that translation t is correct on sequential processes if, for each sequential p, transition system $S_{t(p)}$ implements transition system S_p. We call this kind of correctness predicate an external correctness predicate since it relates the external semantics of the systems.

A simulation technique is said to be sound with respect to the external semantics if its internal correctness predicate entails the external correctness predicate. It is not directly proven in $[IV]$ that the chunk by chunk technique is sound for sequential processes. Instead, as will be described below, a similar claim is proved with two different external semantics for parallel processes, and the claim for sequential processes follow as an easy specialization of both these results.

4.4 Treatment of parallelism

The major part of $[IV]$ is concerned with generalizations of the chunk by chunk technique to deal with parallelism also.

A PL program is a parallel composition $\texttt{PAR}[p_1, \ldots, p_N]$ of sequential processes $p_1, \ldots, p_N$. To describe the distribution of code on different processors in AL we introduce the operator `par` where $\texttt{par}(\pi_1, \ldots, \pi_N)$ is interpreted as the system consisting of N processors where code π_i is executed of processor i (in $[IV]$ we use $\|$ instead of `par`, but we want to reserve $\|$ for an operation introduced in $[V]$). Then the translation of parallel programs in PL simply is

$$t(\texttt{PAR}[p_1, \ldots, p_N]) = \| \ (t(p_1), \ldots, t(p_N))$$

If we try to follow the chunk by chunk technique for parallel constructs also, then we must break down each execution of $\texttt{par}(t(p_1), \ldots, t(p_N))$ into smaller chunks, simulate each chunk at the PL-level, and glue all these subsimulations together to one simulating execution of $\texttt{PAR}[p_1, \ldots, p_N]$. In this approach we of course want to reuse the chunks and simulations found for each of the processes p_i. But as the following example shows, this may be very difficult to do directly.

---

leads to a weaker notion of implementation which is used in the next section.

Assume that $N = 2$, that p_1 is the process $x_1 := e_1$ and p_2 the process $x_2 := e_2$. The translation of each of these processes is $eval(e_i); stl(x_i)$ for $i = 1, 2$. For each of $i = 1, 2$ this machine code executes in two transitions and these two transitions are chosen to constitute a chunk when when using the chunk by chunk technique. But when we execute the two pieces of machine code in parallel, then the chunks may interleave very inconveniently as demonstrated in figure 4.3. The chunks for the two processes get entangled in such a way that we cannot directly use a chunk for $t(p_i)$ as a chunk for the entire program; the chunk is mixed with transitions from other processes.

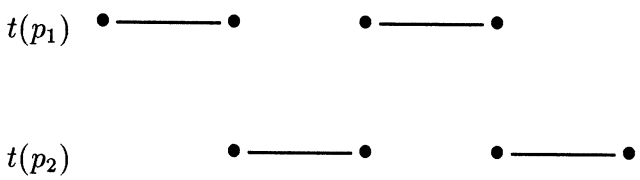

Figure 4.3: Inconvenient interleaving of two chunks.

The first idea to get around this problem is the idea of simulating *principal* transitions instead of chunks [*IV*] page 42. A principal transition is a transition in a chunk which is chosen to represent the entire chunk. In contrast to the chunks which may be heavily interleaved and thus may be only partially ordered, the principal transitions will be linearly ordered and it is thus possible to glue together a linear simulating execution from the subsimulations found for each principal transition. So whereas it is quite possible that the chunks are very entangled in the low level execution, in the constructed high level execution the subsimulation will be nicely separated.

When refraining from breaking down the global low level executions into chunks, we once again have to relate after each low level transition. The only thing we want to express, however, is that if some process reaches a principal transition, then this transition can be simulated from the current high level configuration. This means that we do not relate the global low level configuration to a global high level configuration, instead we relate the projection of the low level configuration onto each of the processes to a likewise projected high level configuration. For the sequential processes we have only defined relations at the endpoints of chunks so we need to create new relations for all the intermediate configurations. This is done by taking the relation which holds at one endpoint and attaching a counter to it which

simply counts the number of transitions in process $t(p_i)$ till the endpoint of the chunk. Since it turns out to be convenient to place principal transitions first in chunks, the counter also indicates the number of transitions to the next principal transition.

The use of counters also proves helpful when it comes to the demonstration that the constructed high level simulation is fair when the considered low level execution is. When parallel processes are introduced an execution is called fair if no continuously enabled internal transitions are postponed indefinitely. In parallel systems internal transitions encompass both τ-labelled transitions and transitions labelled with communications via internal channels in the system. All non-principal transitions at the low level are τ-labelled so the fairness condition at the low level ensures that these transitions are eventually executed when enabled. This means that the counter for each process is eventually decreased and since counters are natural numbers, each counter eventually gets the value 0; so the principal transition is enabled eventually. The fairness requirement then forces the principal transition to be taken if it is an internal transition and this ensures that a subsimulation can be found which extends the high level execution under construction. Consequently the high level fairness constraint gets satisfied.

In $[IV]$ the sketched simulation technique is proved sound relative to an external semantics (pages 55 to 58). This external semantics is a direct generalization of the already described semantics. The meaning of a parallel program is taken to be a set of observations where an observation consists of a trace and a refusal set defined as before. The executions which are considered must satisfy the above mentioned fairness constraint.

The basic structure in the proof of soundness is to construct a simulating execution as a limit of approximations. The next approximation is obtained by concatenating the current approximation with the subsimulation found for the next transition in the considered low level execution.

The paper $[IV]$ also presents another way around the problem that chunks may overlap very inconveniently when executions of individual processes are interleaved. The crucial observation is that it is really the interleaving of the sequential executions that causes the problem. So instead of sticking with the traditional view that executions of parallel processes are just interleavings of

sequential executions we introduce the notion of non-interleaved executions (page 66 in [IV]).

The basic observation which leads to non-interleaved executions is that an execution of a sequential program can be viewed as labelled graph consisting of a single path where vertices are labelled with configurations and edges are labelled with transitions. The generalization to N sequential processes then is a graph consisting of N paths. Because of communications it is not possible to directly use the traditional concept of a graph; a communication is conceptually just a single transition, but it has two initial and two final configurations. We consequently introduce the notion of a multi-graph where edges are allowed to have more than one origin and more than one destination.

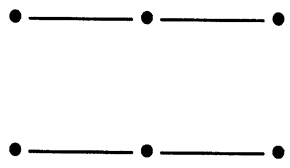

Figure 4.4: Non-interleaved execution without communications.

With the concept of a non-interleaved execution the execution of

$$t(\texttt{PAR}[x_1 := e_1, x_2 := e_2])$$

will now be depicted as in figure 4.4. As is seen the two chunks found for the respective two processes are kept completely separated. If the translated program were instead, say, $\texttt{PAR}[ch!e, ch?x]$, then the non-interleaved execution would give rise to a picture like in figure 4.5. Here the process $t(ch!e)$ first evaluates e; then the two processes communicate; finally process $t(ch?x)$ stores the communicated value in variable x.

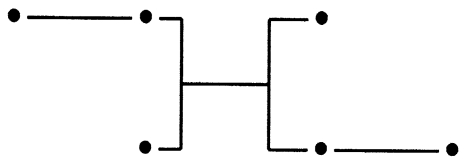

Figure 4.5: Non-interleaved execution with a communication.

Non-interleaved executions allow a simpler expression of our fairness condition. Now we can simply say that a non-interleaved execution is fair if it is inextensible by internal transitions.

With non-interleaved executions it is possible to directly reuse the internal correctness predicate for sequential processes: The non-interleaved low level execution can be broken down into the chunks found for sequential processes, their respective simulations can subsequently be glued together into a simulating non-interleaved high level execution.

Non-interleaved executions could be used as a basis of extracting the already described external semantics. They do, however, also open up for the definition of a more general external semantics based on pomsets [52]. In this kind of external semantics we extract a pomset instead of a trace from the execution. A pomset is a partially ordered multiset where the elements in the multiset are labels of transitions and the partial order is the causal dependencies between the labels. A non-interleaved execution identifies two labels as being ordered if there is a path in the execution from the smaller label to the larger label.

After the definition of the new kind of external semantics (page 71) the paper is concerned with the soundness proof for the simulation technique built on non-interleaved executions. As with interleaved executions the idea is to construct a simulating execution as the limit of a sequence of approximations. Now we have to face the problem, however, that the constructed simulation may have more than one infinite path. So when we glue subsimulations together we either have to do this in a carefully balanced way such that all paths approach infinity simultaneously or we have to take more than one turn, i.e. we have to make transfinite sequences (of length at most ωN where ωN is the N'th infinite limit ordinal, see [20] pages 76-77) of approximations where one path is completed for every limit ordinal. In the paper the second of these approaches is taken.

Chapter 5

An algebraic approach to parallelism

Parallelism is difficult to treat with the chunk-by-chunk technique. In $[IV]$ two third of the entire document is devoted to generalizations of the technique to deal with parallelism.

Also the more traditional simulation technique presented in $[III]$ may cause trouble when generalized to cope with parallelism. For instance, for an execution of a parallel system in which k processes diverge we must ensure that k processes also diverge in the constructed simulation. And this may call for the definition of a simulation relation indexed by a whole tuple of elements from well-founded orders, one element for each process in the system.

The difficulties with the simulation techniques have led to the consideration of using other techniques to deal with parallelism. The key observation in the search for an alternative technique is that the parallel composition of N processes simply compiles into the parallel composition of the corresponding N instruction sequences. So if a compositional definition of the external semantics of algebraic parallel processes can be found, then standard manipulations should enable us to establish the correctness of the translation.

The algebraic approach to parallelism has been fully developed in submission $[V]$ which is contained in the draft monograph [19]; in the monograph the approach is also used in proofs of the correctness of a kernel. The monograph shows that a combined use of simulation techniques and algebraic techniques

is feasible for a certain class of languages.

5.1 The idea

The definitions of PL and its semantics are slightly different in $[IV]$ and [19]. These differences will be treated in detail in the next subsection. For the moment—to present the central idea in the algebraic approach—it is sufficient to mention that the flat n-ary parallel composition of $[IV]$ has been replaced in [19] with a binary parallel operator which can be nested to arbitrary depth at the outermost level and that the fact that transition system S is implemented by transition system S' now is expressed in the external semantics by

$$[\![S]\!] \sqsubseteq [\![S']\!]$$

The crucial observation is that, in both PL and ML, the external semantics of the parallel composition of two programs only depends on the external semantics of each of the two programs.

This observation is formalized by defining a binary operator $\|$ within the external semantics and by proving that it reflects the parallel composition of programs

$$[\![S_{p_1 \texttt{ PAR } p_2}]\!] = [\![S_{p_1}]\!] \,\|\, [\![S_{p_2}]\!] \tag{5.1}$$

$$[\![S_{\pi_1 \texttt{ par } \pi_2}]\!] = [\![S_{\pi_1}]\!] \,\|\, [\![S_{\pi_2}]\!] \tag{5.2}$$

$$\tag{5.3}$$

It turns out that the defined operator is monotonic in each of its two arguments. Thus

$$[\![S_1]\!] \sqsubseteq [\![S_1']\!] \text{ and } [\![S_2]\!] \sqsubseteq [\![S_2']\!] \text{ implies } [\![S_1]\!] \,\|\, [\![S_2]\!] \sqsubseteq [\![S_1']\!] \,\|\, [\![S_2']\!] \tag{5.4}$$

These observations allow us to prove the following theorem by a simple calculation

Theorem: If $[\![S_{p_1}]\!] \sqsubseteq [\![S'_{t(p_1)}]\!]$ and $[\![S_{p_2}]\!] \sqsubseteq [\![S'_{t(p_2)}]\!]$ then $[\![S_{p_1 \texttt{ PAR } p_2}]\!] \sqsubseteq [\![S'_{t(p_1 \texttt{ PAR } p_2)}]\!]$.

Proof

$$\begin{array}{rcll} [\![S_{p_1 \texttt{ PAR } p_2}]\!] & = & [\![S_{p_1}]\!] \parallel [\![S_{p_2}]\!] & \text{(by 1)} \\ & \sqsubseteq & [\![S_{t(p_1)}]\!] \parallel [\![S_{t(p_2)}]\!] & \text{(by assumptions ad 3)} \\ & = & [\![S_{t(p_1) \texttt{ par } t(p_2)}]\!] & \text{(by 2)} \\ & = & [\![S_{t(p_1 \texttt{ PAR } p_2)}]\!] & \text{(by definition of } t\text{)} \end{array}$$

(End of proof)

The theorem says that in order to demonstrate that t correctly compiles parallel programs it is sufficient to prove that sequential processes are compiled correctly.

5.2 Treatment of alternations

The chunk-by-chunk technique is used in both $[IV]$ and $[V]$—in $[V]$ only to prove the correctness of t when applied to sequential processes. Nevertheless there are some differences between the formal details in the documents. The most important differences comes from the inclusion of the alternation construct in PL_0 in [19].

An alternation $\texttt{ALT}[gc_1, \ldots, gc_n]$ where each guarded command gc_i has form either $b_i \;\&\; ch_i?x_i \rightarrow p_i$ or $b_i \;\&\; \texttt{SKIP} \rightarrow p_i$, i. e. it consists of a guard and a process where the guard may be a SKIP-guard. The alternation can execute and become p_i if b_i holds true in the current configuration. If the guard in gc_i is $b_i \;\&\; ch_i?x_i$, then another process must be ready to output on channel ch_i; if the guard is $b_i \;\&\; \texttt{SKIP}$, then no synchronization with the environment is necessary.

The inclusion of the alternation construct forces us to give another definition of the set of channels refused in an execution. In both $[IV]$ and $[V]$ we intuitively take an external channel to be refused in an execution if the process that uses the channel either makes only finitely many steps and disables the channel in the final state or makes an infinite number of steps among which only finitely many are communications along the channel. The latter condition says that if the process can continuously choose other actions, then

communication along the channel need not occur.

The condition that infinite other activity leads to refusal of a channel has been formalized in [*IV*] by the condition that a channel is refused when communications along the channel only occur a finite number of times and are infinitely often disabled in the entire execution. This formalization is possible because communication is blocking in [*IV*]: If a process enables a communication (input or output) in some state, then the process cannot progress before the communication is married out. So if a process makes an infinite number of other steps, then the communication must be infinitely often disabled.

This situation changes when alternations are included in the language. An alternation with both a `SKIP`-guard and an ordinary guard containing a communication along a channel can enable the channel and nevertheless choose the `SKIP`-guard. This gives problems if an optimizing compiler is used. Consider e. g. the program

```
WHILE TRUE DO
    ALT
        TRUE & ch?x → p
        TRUE & SKIP → SKIP
```

An optimizing compiler could recognize that the entry conditions in the loop and the guarded commands are always true and thus it could skip evaluation of these conditions. Furthermore it could recognize that the body in the second guarded command is just the process `SKIP` so that a jump directly back to the beginning of the `ALT`-construct could be generated. But then we could have an infinite execution where channel *ch* is enabled in every configuration of the execution but where the cllannel nevertheless is refused because the second guarded command is continuously chosen Using the formalisation from [*IV*] page 39 and page 73 of the refusal set we would incorrectly get that *ch* is not refused.

To find a formalization which can deal with alternations we need a more direct way of expressing that a process executes one of its own actions. It should be possible to identify the action of the process even when its execution is interleaved with the execution of other cesses.

The idea is to use labels of transitions to identify processes. A slight change is necessary to use this idea. Previously we used τ as a common label on arbitrary internal actions in any process. To distinguish between processes we need to distinguish between τ-labels of different processes. This is done by colouring the τ-actions by the relevant process identity in the definition of the transition relation. Labels of form $ch : z$ need not be coloured because each ch is a point-to-point connection and thus directly identifies the processes which participate in the action.

With the colouring of action labels it becomes possible to express that a channel connected to some process is refused if the process makes a finite number of communications along the channel and an infinite number of other actions. We define a channel to be *ready* if it is enabled and the next action executed, if any, is an action of another process. A channel then is refused if it is taken a finite number of times and is not *ready* an infinite number of times.

In [19] the colouring of actions has also been used to express fairness among a set of executing processes. The fairness condition says that each enabled internal action must eventually be taken if the process(es) participating in the action perform(s) no other actions. This is formalized by extending the predicate *ready* to τ-actions in the obvious way and by requiring that no internal action be continuously *ready* from any point in any execution (choosing the action makes the predicate *ready* momentarily false).

5.3 Redundant information in the external semantics

In $[IV]$ a transition system S' is taken to refine transition system S whenever $[\![S']\!] \subseteq [\![S]\!]$ that is each observation of S' should also be an observation of S. In $[V]$ this condition is weakened slightly; it is sufficient to require that S refuses *at least* the channels refused by S' for if S refuses some set of channels, then it also refuses each subset of this set. In $[V]$ the symbol $\sqsubseteq$ is used for refinement, and to correspond with failure semantics it is used in the opposite direction of the above set inclusion symbol. So S' refines S is written $[\![S']\!] \sqsupseteq [\![S]\!]$.

The observation that refusal of some set of channels implies refusal of each of its subsets also has consequences for the definition of external semantics $[\![S]\!]$ of a transition system S. We want to make the external semantics of transition systems fully abstract which means that if two transition systems cannot be distinguished by any experiment, then they should have the same external semantics. The external semantics of a transition system is just the set of observations which can be made in executions of the systems. It is possible to make transition systems enabling two observations which are equal except that the refusal set of one of the observations is larger than the refusal set of the other. Then the other observation is redundant; the first already describes the traces and the refusals of the other. In the definition of the mapping $[\![\cdot]\!]$ from transition systems to external semantics in $[V]$ we explicitly remove such redundant information.

5.4 Limitations and difficulties

The development in $[V]$ demonstrates that it is possible to verify the correctness of a translation by using a simulation technique combined with an algebraic argument for parallelism. Such a combined use of techniques has limited applicability, however.

The introduction of parallelism nested into other constructs, e.g. the sequential, presents a problem. We have used simulation to prove that the translation of sequential processes is correct and we would like to do the same in a language extended with nested parallelism. But if a sequential process contains a nested parallel process, then our combined technique only establishes that the *external* semantics of the nested parallel process is refined by the *external* semantics of its translation; it gives no information about how high- and low-level configurations could be related and how (sequences of) low level transitions could be simulated by sequences of high level transitions. A switch back to operational arguments consequently seems hopeless.

Instead one could try to shift to algebraic arguments like the one we have given. But this is only feasible if the external semantics of the other constructs can also be defined compositionally. To give a compositional definition elements in the external semantics at least have to be extended with the input/output behaviour of constructs. Such a compositional definition for a

language like our PL has been given in a model based on failure semantics [23], but as discussed earlier there is no notion of fairness in failure semantics and furthermore chaos and divergence are not distinguished. It seems to be difficult to make our semantics compositional; some indication of the way to proceed has been given in [30], however.

One could also try to stick with operational arguments all the way through. Then parallelism could be dealt with in one of the two ways suggested in [*IV*]. However, also in these approaches the generalized simulation technique differs from the technique applied to sequential processes. So it has to be investigated whether the generalized technique can make the basis for constructing simulations for translations of programs with nested parallel processes.

In spite of the problems with nested parallelism the combined proof technique may turn out to be very useful as programs with parallelism only at the outermost level are very common in practice. They arise whenever a system is constructed as a network of cooperating machines, for instance as a network connecting stand-alone computers with each other or a bus connecting different peripherals of a computer with each other. Parallelism only at the outermost level is the rule rather than the exception.

Chapter 6

Discussion

The work documented in this thesis has been centered around simulation techniques. The aim has been to improve, invent, and apply simulation techniques.

6.1 Summary of results

In [I] a simulation technique suggested by Lamport [33, 34, 35] is improved. A couple of results shows that the size of the required simulation proof can be reduced drastically. That part of the proofs which can be saved is seen to amount to unnecessary repetition of invariance proofs. The modified technique is applied to a number of examples. A larger example is the new proof in [II] which shows that self-timed four phase logic is insensitive to delays in gates and wires.

In [III] a technique based on weak bisimulations [42] is used as stepping stone for proving that a translation is correct. The notion of weak bisimulation is changed as to cope with removal of non-determinism and simulation of divergence. In the resulting technique a stronger, but only one-way, simulation result must be proved; furthermore, an index on the simulation relation is added and must be shown to decrease for each step simulated by an empty sequence of steps. The technique is applied to a translation from a subset of occam containing various sequential programming constructs to a block

structured assembly language. The application is successful but the resulting proof is rather large.

In $[IV]$ the simulation technique from $[III]$ is substantially changed and this results in a much shorter proof. The central idea is to simulate only conveniently long sequences of transitions, chunks, instead of single transitions and to simulate non-empty chunks by non-empty sequences of transitions. The technique is applied to a translation from a language which extends the one from $[III]$ with parallelism. Parallel processes, however, cannot be directly treated with the simplest version of the chunk-by-chunk technique so two different generalizations are suggested and the corresponding proofs for the parallel construct are carried out. A new feature in $[IV]$ is that correctness is not expressed as the mere existence of a simulation relation with the required properties. It is expressed instead in an abstract "external" semantics.

In $[V]$ a more pragmatic approach to parallelism is taken; since parallelism only occurs at the outermost level in most applications, a hybrid proof technique can be used: For the parallel composition of sequential processes we use an algebraic argument where only the external semantics of processes is considered.

In addition to these improvements, inventions, and applications of simulation techniques two by-products deserve a comment.

One method to deal with parallelism in the chunk-by-chunk technique is to avoid inter-leaving of execution sequences. This led to the invention of non-interleaved executions. Non-interleaved executions seem to be a natural way of modelling executions of transition systems for true concurrency as those suggested in the chemical abstract machine [5] and in the grape semantics [10]. From non-interleaved executions one can also directly derive an external semantics based on pomsets. Furthermore non-interleaved executions may prove to be convenient in real-time computation models because the time between two configurations in such an execution is simply the sum of transition durations along any path between the configurations.

To give a simple and abstract definition of correct implementation we introduced the concept of external semantics in $[IV]$ and $[V]$. This resembles failure semantics, but also differs crucially. Our external semantics distinguishes between chaos and divergence and takes fairness between a pool of processes into account. This accords well with intuitive expectations to au-

tonomous computers working in cooperation. On the other hand external semantics is not compositionally defined so it cannot be used for giving an independent and self-contained definition of the semantics of each language construct; this must instead be done via operational semantics.

6.2 The landscape of verification

The simulation techniques presented in this thesis only add a small piece to the puzzle of proving implementations correct. A great variety of other techniques exist already. These techniques are not directly comparable, however. They use different formal frameworks and the aims in the techniques are not completely the same.

One source of variety is the use of different models of computation. E. g. communication between processes (if the computation model contains parallelism at all) can be modelled by asynchronous communication through shared variables [8, 12, *I*] and queues [28, 33] or by synchronous communication through channels [26] and label matching [42].

Another source of variety is the different styles of defining the semantics of systems. There are denotational semantics [6, 55, 61], operational semantics [33, 42, 49], axiomatic semantics [24], Petri net semantics [53], and so on. And there may even be variety between the semantics within a single style. Thus, the denotational semantics for input/output programs [55, 61] and for reactive systems [6] differ significantly.

Likewise the notions of refinement or correct implementation may depend on the intended usage of programs. Thus focus on branching aspects [50] gives another notion of refinement [41] than does the one used when linear time [50] is considered [*IV*, *V*] . And also considering internal divergence to be harmful [6] gives another refinement notion than ours [*IV*, *V*].

Finally the aims can be different in the different techniques. One can strive for complete proof techniques as in [31] or one can strive for techniques which seem easy to use as in [33, 38, 59, *I*, *V*]. One can also try to make the proofs compositional as in [28, *V*].

The systems studied in this thesis are assumed to be parallel, reactive sys-

tems. The semantics of the systems are given operationally by means of transition systems. Communication is through shared variables in [I, II] and through channels in [III, IV, V]. Only the linear-time aspects of the systems are taken to be important and divergence is considered harmless. Refinement is basically defined by a containment between sets of behaviours—in [I] with the additional requirement that low level be behaviours must be interpreted. The focus is on obtaining techniques which can be applied in practice.

6.3 Related work

Simulation techniques for proving safety

The simulation technique used in [I] was proposed by Lamport in [33, 34, 35]. Similar techniques have been used by many others [22, 28, 31, 38, 59].

These techniques mainly differ in the way high and low level states are related. Lamport's and our work use ordinary mappings from low level states to high level states. In [22, 28, 59] the more general notion of a *relation* between high and low level states is used; the equivalent notion of a multi-valued (possibilities) mapping is introduced in [38]. Klarlund [31] uses the even more general idea of a mapping into *sets of sets* of states (ND-measures).

The use of more complex ways of relating high and low level states make the techniques applicable to a wider range of examples. Stark [59] gives an example on which ordinary mappings do not work, but where relations do. Klarlund [31] gives another example to the same effect and also gives an example where even relations are not enough. On the other hand Klarlund shows that mappings into *sets of sets* of states *are* sufficient to prove that a transition system with bounded non-determinism is implemented by a so-called complete transition system.

In [1] Abadi and Lamport present a sort of anti-dote to the apparent deficiency of using their ordinary maps. Under similar restrictions as Klarlund's they prove that it is possible to transform the low level transition system in such a way that its semantics is preserved and such that an ordinary mapping can be used in the implementation proof. This way their method also becomes complete.

In contrast to all the mentioned work our method $[I]$ h as been developed for verification of closed systems. As discussed in section 2 this gives full control over the high level interpretation of low level states and thus the method can be used to prove any safety property of the low level transition system if only the high level system contains unreachable states.

Our method could easily be formulated for open systems also. Since the proof obligation in our method encompass the obligations in [1] as a special case, the result in [1] immediately yields a traditional completeness result for our technique when applied to verification of open systems.

The main focus in $[I]$ is on another aspect of the applicability issue than completeness. We analyse the total amount of work needed when using simulation techniques to verify properties of low level transition systems. In many cases it turns out that the existing techniques, despite their completeness, forces proofs of high level properties to be repeated in the simulation proofs The main contribution of $[I]$ is a couple of sharpened proof obligations which discard such double proof obligations. It should not be too hard to incorporate these sharpened obligations into the esisting simulation techniques.

Delay insensitive circuits

In $[II]$ the voltage changes around gates in a circuit are presented by rules of form $c \rightarrow \bar{x} := \bar{e}$ where c is a conditional expression and $\bar{x} := \bar{e}$ is a multiple assignment of the values in the sequence of expressions $\bar{e}$ to the sequence of variables $\bar{x}$. This notation is heavily inspired by the Synchronized Transitions notation suggested in [60] and it also resembles UNITY [8].

In the eighties the main part of the formal work on delay insensitive circuits has been carried out in Eindhoven University of Technology [14, 29, 56, 57, 63]. In this work a quite different formal model is used, the model of directed trace structures. Gates are modelled by sets of directed traces, where each symbol in a trace corresponds to a transition from low to high or from high to low voltage on a wire. The resulting formalization of delay insensitivity seems to be less direct than the one found in $[8, II, 60]$. The main focus in this work has been either on characterising delay insensitive circuits [57, 63] or on ensuring delay insensitivity by selecting proper composition principles

[14, 56]. In contrast our work is based on the idea of describing a system at different levels of abstraction. Finally it should be mentioned that Udding [63] has a classification of delay insensitive circuits in which the circuits considered in $[II]$ belong to the data communication class.

Recently a new formal approach to delay insensitivity has been undertaken at the Technical University of Denmark [21]. The so-called duration calculus is used to give a direct account of delays in wires. In this calculus a circuit C with m gates and wires may be described by a formula $[\![C(\delta_1, \ldots, \delta_m)]\!]$ where each δ_i is the delay of gate or wire i. The circuit is said to be delay insensitive with respect to some circuit specification—a formula free of delays—if $[\![C(\delta_1, \ldots, \delta_m)]\!]$ logically implies the specification for any choice of $\delta_1, \ldots, \delta_m$. This formalisation of delay insensitivity seems to be even more direct than ours. Furthermore the duration calculus allows one to express additional assumptions on the relative size of delays, and this has not been possible previously.

Correctness of translations

Quite a bit of work has been carried out on correctness of translations of sequential programs with focus on their input/output behaviour (e. g. [11, 13, 40, 43, 46, 62]). The sequential programming constructs treated in these techniques are often far more complex than those sequential constructs which have (yet) been treated in techniques for parallel languages; e.g. higher order functions and composite data-structures are considered. Having said this it seems that the success of most techniques for sequential languages hinges on the (relative) simplicity of the semantics of the considered programs—be it denotational or operational. Furthermore the determinism of the languages seems to make the correctness criterion self-evident for languages with onely simple constructs: The source and target programs should compute the same function (an exception to this is [11] where the possibility of non-determinism gives rise to two proof obligations establishing a set-containment either way between the results of executing the program and the results of executing its translation). It is doubtful whether the techniques generalize to concurrent, non-deterministic, and reactive systems as those considered in this thesis—at any rate such a generalization has not yet been claimed to be feasible.

It should be noted, however, that the structure of some of these proofs may also be useful for more powerful languages—in particular the algebraic approach in [13, 43, 44, 62]. Here the source language is treated as an initial (many-sorted) algebra with one operator for each construct in the language. Correctness is ensured by initiality: Two homomorphisms from the initial algebra to any other algebra over the same signature must be identical. The work then consists in making the meanings of the target language into an algebra (which is done by composing the translation with the map from the target language to the target semantical domain) and in proving that the map from the source semantical domain to the target semantical domain is a homomorphism (the source semantical domain is treated as the algebra induced by the map from the source language). This algebraic set-up is clearly independent of the particular constructs in the language. It could, thus, also serve as a basis for a proof of correctness for a language with reactive, parallel, and non-deterministic programs.

The investigation of translation correctness for parallel languages has been more sporadic [3, 4, 37, 39]. But several notions of refinement have been developed for parallel languages. In addition to those notions already mentioned when discussing inheritance of safety properties, we would like to mention the notions encountered in the treatments of CCS [42] and (T)CSP [25]. Some of these notions form the basis for defining correctness of translations in [3, 4, 39]. A minor difference from these applications should be noted, however, namely that the CCS and CSP notions all express refinement between processes in a single language whereas the definitions of translation correctness have to employ a notion of correctness between processes in two different languages.

CSP has been given various gradually more discriminating semantics [6, 7, 25, 54]. All these semantics assign to a CSP process a set of observations where an observation consists of a set of traces and various other sets according to the strength of the semantics. Refinement is defined by set inclusion(s). The intuition is that p refines q when p is more deterministic than q; removal of non-determinism makes the sets of possible observations smaller. This is much in correspondence with our definitions of correctness. The treatments of internal divergence and fairness are quite different, however. These differences will be discussed in the subsequent description of Barrett's thesis [4] as Barrett takes his notion of correctness from the infinite traces model [54].

Various preorders on the set of CCS terms have been suggested as a means of expressing refinement. The main suggestions lie within the testing framework [45] or the bisimulation framework [64]. In his thesis Millington [39] uses both testing and bisimulation ideas. But whereas he directly uses the testing preorders from [45] (see below), he only focuses on equivalence in his bisimulation approach to correctness translations.

To our knowledge none of the preorders from [64] have been used for verifying translations. Basically, the idea in all these preorders is that $p \sqsubseteq q$ holds whenever p *and* q are weakly bisimilar and q is less divergent than p. More specifically, q must be able to simulate each communication of p; and if q is convergent, then so is p and in this case p must be able to simulate each communication of q; as with ordinary bisimulations the resulting p and q derivatives must be related by the preorders.

It should be noted that the use of the term "divergence" in [64] is quite different from our use. In [64] a process is divergent if it can engage in an infinite sequence of τ transitions *or* if it is, or can silently evolve into, and underdefined process. The usage of underdefined processes plays a crucial role in the use of these preorders (see e.g. [64] and [9]). They give the designer the freedom of leaving parts of programs unimplemented when it can be seen from the context that these program parts are unreachable. But such processes are not included in traditional programming languages and this may indicate the reason for the apparent absence of approaches to compiling correctness based on the preorders from [64].

Another bisimulation concept needs to be mentioned. This is the $\frac{2}{3}$-bisimulation employed by Larsen and Skou in [36]. Larsen and Skou focus on the *symmetric* notion of $\frac{2}{3}$-bisimilarity, but to formalize that two processes p and q are $\frac{2}{3}$-bisimilar they require that both (p, q) and (q, p) belong to an *asymmetric* relation. For two processes p and q to be in this relation q must be able to simulate each p transition such that the resulting derivatives are related and p must be able to simulate each q transition but in this case the derivatives need not be related. This relation is very much like the relation that we have put forward in $[III]$ except that we use weak $\frac{2}{3}$-bisimulations and that we have added an extra condition concerning an index on the relations.

We next give a more detailed account for the two PhD. theses by Millington [39] and Barrett [4] which are most relevant to our work on correct nes of

translations.

Millington's thesis

Millington considers two translations from (a subset of) CSP to CCS. He presents two definitions of correctness and for each definition he conducts the corresponding correctness proof. The first definition is built on bisimulation ideas [42] and the second exploits the notion of testing [45].

The first approach in [39] improves upon previous work by Wei Li [37] and Astesiano and Zucca [3].

Li considers a translation from CSP to CCS. In his definition of correctness he focuses on aspects of termination. In order for a translation to be correct he requires that an execution of a translated program should be divergent just in case an execution of the source program is, and in case of convergent execution both should either end in a deadlocked or a terminal configuration; furthermore the final configurations must correspond: The final source configuration should "translate" into the final target configuration.

Astesiano and Zucca also consider a translation from CSP to CCS, but in addition to termination they also take communication behaviour into account. This is done essentially by requiring that each pair consisting of a source process and its translation should belong to some bisimulation. To express correspondence between final configurations of source and target executions they introduce auxiliary processes which communicate the process states just before termination.

Millington also takes communication behaviour into account by requiring that source processes and their translations should belong to a bisimulation. But he has a more clean approach for ensuring that final configurations correspond: For any relation P between source and target configurations he introduces the notion of a P-bisimulation; this is a bisimulation which is contained in P. By choosing a sufficiently narrow P he can express the desired correspondence.

Millington goes on to prove correctness: There exists a P-bisimulation to which each pair of configurations corresponding to source and target programs belongs. As in our $[III]$ Millington builds up the desired P-bisimulation

compositionally; for each process p he defines a relation $R(p)$; and if p has subconstructs, say, p_1 and p_2, then $R(p)$ is defined by some operation upon the relations $R(p_1)$ and $R(p_2)$. The desired relation is then the union of all these relations. This approach is slightly less general than the use of inference systems in $[III]$. In particular we demonstrate in $[III]$ the elegance of using an inference rule for iteratively building up the relation for the `WHILE`-construct. To deal with a similar problem Millington has to introduce a special operator ITER which when applied to a process term generates an infinite set of process terms. The proof that each $R(p)$ is a P-bisimulation is, of course, conducted by induction on the structure of p.

There are two major differences between the P-bisimulation approach of Millington and our approach in $[III]$.

First non-determinism is treated differently. In our approach non-determinism can be used as a means of underspecification which is not the case in Millington's approach. Consequently we have to give up the idea of establishing a symmetric simulation relation between source and target code. We can only hope for simulating each low level step by some (sequence of) high level step(s).

Next our correctness notion is stronger in that it ensures that silent divergence is simulated by silent divergence. As previously mentioned Li's and Millington's approaches ensure that if execution of the source code is divergent, then so is execution of the target code, and vice versa. But divergence here means engaging in an infinite sequence of communications. Thus silent (or internal) divergence is not treated as divergence, it is instead equated with the terminated process *Nil*. Millington does point out, however, that P-bisimulations could also be used to deal with divergence if the relation P is chosen such that related pairs are either both convergent or both have the possibility of internally diverging.

Millington also presents another, very interesting, approach to translation correctness build on the notion of testing [45]. The basic idea in testing is that the observable behaviour of a process p can be characterized by the possible outcomes of experiments *test* **par** p where *test* is a tester communicating with the process; the process may pass or fail a test (indicated by the experiments having outcome $\top$ or $\bot$) or it may be capable of doing both.

Millington notes that if the tester is expressed as a process in the source

language, then the translation can also be applied to the tester. As a consequence it becomes possible to speak meaningfully about correctness of a translation even if communication interfaces are refined during the translation: Instead of requiring that source and target code have same communication capabilities it is possible to require just that they behave the same when exposed to corresponding tests.

To demonstrate the applicability of this idea Millington considers a translation from a small subset of CSP (with only unguarded communication, assignments, and one-level parallelism) to a slightly different language where communication is based on shared variables instead of handshaking. He argues that the correctness of this translation cannot be expressed by means of bisimulations—both because of the interface refinement and because the amount of non-determinism may be different in the source and target code. Instead he expresses the correctness by the requirement that for each experiment *test* **par** p it is the case that $t($*test* **par** $p)$ "completely-implements" *test* **par** p. That experiment f completely-implements experiment e means that f and e have the same set of possible outcomes.

The main idea in the proof of correctness is for each execution of $f($*test* **par** $p)$ to construct an execution of *test* **par** p having the same outcome. Millington reduces this obligation to the obligation of proving four conditions. The most important condition says that if the low level experiment $f($*test* **par** $p)$ can evolve in a number of steps into a configuration of a special kind, then this configuration can also be obtained by first taking a number of steps from the high level experiment, then translating the result, and finally taking a few number of low level steps. The condition is proved by induction on the length of low level executions: First the notion of a "consistent" low level configuration is introduced; consistent low level configurations can be seen as natural images of high level configurations, so it is possible to define an inverse translation on these configurations. Then it is demonstrated that each low level step preserves consistency and that each low level step between consistent configurations can be simulated by some number of steps between their inverse images. It is worth noting that the last demonstration resembles the demonstrations required in $[I]$. It even suggests that Millington's proof could be simplified: The preservation of consistency could instead be obtained as an inherited property.

Much work has yet to be done to improve the testing approach. Most im-

portantly the source language has to be extended. In particular it should be considered how iterative or recursive constructs would be treated; in Millington's proof it is essential that there are no infinite executions. To deal with infinite executions Millington puts forward a fifth condition. He only suggests how it would be proved, however. Interestingly his suggestion employs the same idea as in [*IV*, *V*]—namely that sufficiently long low level executions can be matched with non-empty high level executions.

Barrett's thesis

In [4] Barrett presents "a rigorous, although not formal, proof of implementation of occam". Barrett concentrates on the treatment of parallelism, communications and priority. So his subset of occam is almost orthogonal to the subset PL_0 in [*IV*] which includes variables, declarations, expressions, and traditional sequential language constructs. As a result his "implementation of occam" contains more characteristics of a kernel development than of a translation; the development steps are concerned with the introduction of certain run-time data structures for representing running, waiting, and terminated processes etc. Nevertheless he employs a proof method which is very close to the method we use in [*III*]. And some of the other ideas he presents resemble ideas used in [*IV*] and [*V*].

Barrett's implementation of his occam-subset proceeds in three stages. In the first stage (the scheduler) a data structure for representing the parallel and sequential composition of processes is introduced. In the next (the synchronizer) the implementation of alternations by sequences of transputer instructions is introduced. Finally the third stage (the transputer) introduces explicit instructions for process creation and process destruction.

As in [*IV*, *V*] the overall goal for Barrett is to establish that the "external" semantics of the implementation is a refinement of the "external" semantics of the source program. As external semantics Barrett uses the infinite traces model of Roscoe [54]. The main difference between the infinite traces model and our external semantics is the treatment of internal divergence and fairness. In the infinite traces model internal divergence is considered to be disastrous and an internally divergent process nested in most larger process makes the larger processes divergent too. In our external semantics we have a

notion of fairness which ensures that a divergent process in parallel with some other process does not destroy the other process' behaviour. Consequently we have a less pessimistic view at internal divergence.

Also as in [*IV*, *V*] Barrett ensures "external" refinement by finding implementation conditions based on simulation techniques and by proving that these implementation conditions imply "external" refinement. Of course, to do this he has to give each stage an operational semantics and this is done by structural operational semantics. At the later stages this semantics contains mostly axioms because the occam structure of process terms is represented in specific data structures.

Barrett employs different simulation techniques for each of the three implementation stages. The aim is to give an accurate description of how the next stage implements the former. So each implementation condition is strictly stronger than what is needed to a ensure refinement.

The implementation condition for the first stage is very strong. It essentially expresses that the source occam program and its implementation are strongly bisimilar in the sense of Milner [42]. Such a strong implementation condition is possible because the stage merely changes one, syntactic, representation of composite processes into another based on explicit data structures.

The implementation condition for the next two stages are quite similar. They both require the existence of a simulation relation with properties almost identical to those put forward in [*III*]. As in our approach Barrett uses a natural number as index to the simulation relations (the index is called a variant in [4]). The reason for these indites are a bit different, however. In our approach they are required to ensure that low level divergence is simulated by high level divergence; the underlying problem is that there need not be a one-one correspondence between high and low level transitions. In [4] there *is* a one-one correspondence between the high level transitions and a subset of the low level transitions; but instead completely new, bookkeeping, transitions are introduced. The aim then is to ensure that no infinite sequence of these new transitions can ever occur. In the requirements to the simulation relations Barrett explicitly distinguishes between new and old transitions which is not possible for our translation. in [*III*].

To define simulation relations satisfying the implementation conditions Barrett introduces a number of quite complicated maps between sets of config-

urations at the different stages. Then, essentially, each relation consists of all pairs $\langle c, \bar{c} \rangle$ where $\bar{\cdot}$ is one of the maps. As in Millington's work Barrett conducts the proof that the implementation conditions hold by induction on the structure of c. Barrett has a recursive construct, but—in contrast to the `WHILE`-construct in [*III*]—the operational semantics he gives to this construct does not call for our more general technique of defining the simulation relation by a deduction system and proving the implementation condition by induction on the structure of deductions.

Another similarity between the approaches in [4] and [*V*] should be pointed out. Although the infinite traces model does not take fairness into account, Barrett *does* work with a notion of fairness in his description of the operational semantics of occam. To express that transitions from different processes in a parallel construct should be fairly interleaved Barrett needs to distinguish between labels of different processes. To do this he introduces a tree address as subscript to transition labels—exactly as it is done in [*V*] (in [*V*] we only have to subscript τ labels, however, because communication labels uniquely identify the participating processes in our approach).

Apparently Barrett only uses the restriction to fair processes at the top-level of his development, the chosen subset of occam. For this level he gives a proof of congruence between the infinite traces semantics and the operational semantics: He defines a mapping Φ from occam constructs to elements in the infinite traces model by deriving the needed observations from fair executions of the constructs, and then he proves that $\mathcal{M}(c) \sqsubseteq \Phi(c)$ holds for each construct where $\mathcal{M}(c)$ is the (compositionally defined) infinite traces semantics and $\sqsubseteq$ denotes refinement in the infinite traces semantics.

At the levels below Barrett hints at how similar maps Φ_i can be defined and he sketches the proof that if (c_1, c_2) is contained in one of his simulation relations, then $\Phi_i(c_1) \sqsubseteq \Phi_2(c_2)$. It is not clear, however, which role fairness plays at these levels, if any; Barrett does not mention it. And our experience in [*IV*, *V*] is that it is not trivial to establish that a constructed high level simulation is fair when the simulated low level execution is.

(A more detailed account of Barrett's work is given in [*III*].)

6.4 Future investigations

It is difficult to support a claim that some technique is easy or natural to use. To support such a claim one has to apply the technique to a substantial number of examples. Although the techniques in this thesis have already been successfully applied to many examples, there are still many interesting left.

The technique in [I] could be applied to the problem in [III, IV, V] and vice versa. Here one should note, however, that the technique in [I] has been specifically designed to allow inheritance of properties, whereas the techniques in [III, IV, V] have been designed to verify compiling specifications.

The languages in [III, IV, V] could be extended with additional constructs. The experience from [IV, V] seems to indicate that the chunk-by-chunk technique will also be successful for other traditional sequential programming constructs. As explained in the previous section the inclusion of nested parallelism will be very difficult to treat with the hybrid technique in [V]; it remains to be seen whether the generalizations of the chunk-by-chunk technique suggested in [IV] can form the basis of a proof for nested parallelism.

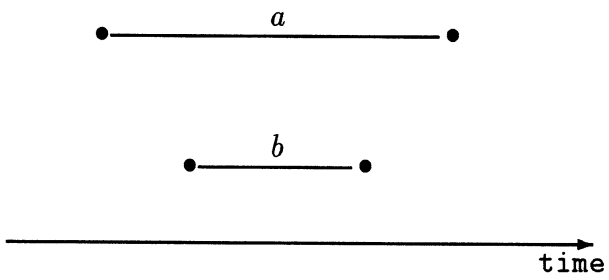

Figure 6.1: Two concurrent timed actions

Inclusion of constructs for real time presents a challenge to any verification method. Already the task of finding an adequate semantic model of real time presents a problem. To this end the concept of non-interleaved executions may prove useful. When using ordinary interleaved executions it is tempting to assume that actions are instantaneous. But consider e. g. the two concurrent actions a and b in figure 6.1. In an interleaving either a or b must come first. If a comes first, then the completion of a precedes the entire action b and this is clearly not the case if the actions have the duration

indicated in figure 6.1. If b comes first, then the beginning of a will succeed the entire action b, again in contrast to the real case. With non-interleaved executions there is no need for forcing a and b to occur in some order so the problem vanishes. This shows that non-interleaved executions may be useful in modelling actions which have duration.

One issue which has been carefully avoided in $[V]$ is interface refinement. In $[V]$ the `ALT`-construct is translated into an instruction sequence where a single machine instruction, `alt`, can perform each of the initial communications of the high level construct. In the real transputer the `ALT`-construct is implemented through the use of the instructions `enbc`, `disc`, and `altwt` each of which are capable of performing communications. So a single communication in occam is carried out through a sequence of communications on the transputer. Even worse, it seems that a shift in computational model is necessary to give a faithful description of the transputer; its functioning is most easily described by a shared variable model as in [27].

When interfaces are refined it becomes necessary to supply an interpretation of low level communications or variable assignments. These actions must be grouped and defined to correspond to distinct high-level communications. The paper $[I]$ hints at how to use abstraction functions for this purpose, even though the computational model in $[I]$ is a shared variable model for both the high and the low level. As pointed out in the discussion of $[I]$ the use of interpretations makes "correct implementation" a relative concept: Each interpretation gives a particular definition of correctness. Whether the interpretation has to satisfy some restrictions to be sensible is an open question. One candidate for a necessary restriction could be that no high level communication corresponds to an infinite number of low level actions. Another restriction can be derived from the testng approach as suggested by Millington [39]: Experiments and their translation should have the same set of possible outcomes.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Proc. 2. Symp. on Logic in Computer Science*, 1988, pp. 165-175.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, Vol. 21, 1985, pp. 181-185.

[3] E. Astesiano and E. Zucca. Semantics by translation of CSP and its equivalence with B-semantics. *Math. Found. of Comp. Science 1981*, LNCS 118, pp. 172-270.

[4] G. Barrett. *The Semantics and Implementation of* occam, PhD. Thesis, Oxford University 1988.

[5] G. Berry and G. Boudol. The Chemical Abstract Machine. *Proc. 17. an. ACM Symp. on Principles of Programming Languages*, 1990, pp. 81-94.

[6] S D. Brookes, C A. R. Hoare, and A. W. Roscoe. A theory of Communicating Sequential Processes. *Journal of the ACM*, Vol. 31, no. 3, 1984, pp. 560-599.

[7] S. D. Brookes and A. W. Roscoe. An Improved Failures Model for Communicating Processes. *Seminar on Concurrency*, LNCS 197, 1985, pp. 281-305.

[8] K. M. Chandy and J. Misra. *Parallel Program Design*, Addison Wesley 1988.

[9] R. Cleaveland and B. Steffen. When is "partial" adequate? A logic-based proof technique using partial specifications. *Proc. of LICS 1990*, pp. 440-449.

[10] P. Degano, R. De Nicola and U. Montanari. A Partial Ordering Semantics for CCS. *Theoretical Computer Science*, Vol. 75, no. 3, 1990, pp. 223-262.

[11] J. Despeyroux. Proof of Translation in Natural Semantics. *Proc. of LICS 1986*, pp.l93-205.

[12] E. D. Dijkstra et al. On-the-Fly Garbage Collection: An Exercise in cooperation. *Communications of the ACM*, Vol. 21, no. 11, 1978, pp. 966-975.

[13] P. Dybjer. Using domain algebras to prove the correctness of a compiler. *Proc. STACS 1985*, LNCS 182, pp. 98-108.

[14] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, PhD. Thesis, Eindhoven University of Technology, 1987. Implementation Conditions for Delay Insensitive

[15] A. Gammelgaard. Appendix A of submission [III].

[16] A. Gammelgaard. Appendix B of submission [III].

[17] A. Gammelgaard. Appendix C of submission [III].

[18] A. Gammelgaard. Appendix D of submission [III].

[19] A. Gammelgaard, H. H. Løvengreen (ed.), C. Ø. Rump, and J. F. Søgaard-Andersen. *Volume 4: Base System Verification*, Draft Monograph, ProCoS tech. rep. ID/DTH HHL 41.

[20] P. Halmos. *Naive Set Theory*, Van Nostrand, Princeton, 1960.

[21] M. R. Hansen, Z. Chaochen, and J. Staunstrup. A Real-Time Duration Semantics for Circuits. ProCoS tech. rep. ID/DTH MRH 7/1, Technical University of Denmark, 1991.

[22] J. He, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. *Proc. ESOP 86*, LNCS 213, pp. 187-196.

[23] J. He. Specification Oriented Semantics for ProCoS Programming Level 0. ProCoS tech. rep. OU HJF 5, Oxford University, 1989.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, Vol. 12, no. 10, 1969, pp. 576-580,583.

[25] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice Hall, 1985.

[26] INMOS Limited. occam 2 *Reference Manual.* Prentice Hall, 1988.

[27] INMOS Limited. *Transputer Instruction Set*, Prentice Hall, 1988.

[28] B. Jonsson. Modular Verification of Asynchronous Networks. *Proc. 6. An. ACM Symp. on Principles of Distributed Computing*, 1987, pp. 152-166.

[29] A. Kaldewaij. The translation of Processes into Circuits. *Proc. PARLE 1987*, LNCS 258, pp. 195-212.

[30] B. v. Karger. Distinguishing Divergence from Chaos. ProCoS tech. rep. Kiel BvK 6, Institut Für Informatik and Praktische Mathematik, Christian-Albrechts-Universitä Kiel, 1991.

[31] N. KlarIund. *Progress Measures und Finite Arguments for Infinite Computations*, PhD. Thesis, Cornell University 1990.

[32] L. Lamport. "Sometime" is Sometimes "Not Never". *ACM Transactions on Programming Languages and Systems*, 1980, pp. 174-185.

[33] L. Lamport. Specifying Concurrent Program Modules. *ACM Transections on Programming Languages and Systems*, Vol. 5, no. 2, 1983, pp. 190-222.

[34] L. Lamport. What Good is Temporal Logic?. *Information Processing 83*, pp. 657-667.

[35] L. Lamport. A simple approach to specifying concurrent systems. *Communicutions of the ACM*, Vol. 32, no. 1, 1989, pp. 32-47.

[36] K. G. Larsen and A. Skou. Bisimulation Through Probalistic Testing. *Proc. ACM POPL 1989*, pp. 344-352.

[37] W. Li. *An operational approach to semantics and translation for concurrent programming languages*, PhD. Thesis, University of Edinburgh 1983.

[38] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *Proc. 6. An. ACM Symp. on Principles of Distributed Computing*, 1987, pp. 137-151.

[39] M. Millington. *Theories of translation correctness for concurrent programming languages*, PhD. Thesis, University of Edinburgh, 1987.

[40] R. Milner and R. Weyhrauch. Proving Compiler Correctness in a mechanized Logic. *Machine Intelligence*, Vol. 7, 1972? pp. 51-72.

[41] R. Milner. A Modal Characterisation of Observable Machine Behaviour. *Proc. CAAP '81*, LNCS 112, pp. 25-34.

[42] R. Milner. *Communication and Concurrency*, Prentice Hall, 1989.

[43] F. L. Morris. Advice on structuring compilers and proving them correct. *Proc. ACT POPL 1973*, pp. 144-152.

[44] P. D. Mosses. A constructive approach to compiler correctness. *Proc. ICALP 1980*, LNCS 85, pp. 449-462.

[45] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Scince*, Vol. 34, 1984, pp. 83-133.

[46] F. Nielson and H. R. Nielson. Two-Level Semantics and Code Generation. *Theoretical Computer Science*, Vol. 34, 1988, pp. 59-133.

[47] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, 1982, pp. 455-495.

[48] D. M. R. Park, Concurrency and Automata on Infinite Sequences, *5th GI-Conference, 1981*, LNCS 104, pp. 167-183.

[49] G. Plotkin. A Structural Approach to Operational Semantics. DAIMI FN-19, Aarhus University, 1981.

[50] A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. *Proc. 12. Int. Coll. on Automata, Languages and Programming*, LNCS 194, 1985, pp. 15-32.

[51] A. Pnueli. Linear Time Temporal Logic. Course notes from REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, the Netherlands, 1988.

[52] V. Pratt. Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming*, Vol. 15, no. 1, 1986, pp. 33-71.

[53] W. Reisig.*Petri nets—An introduction*, Springer 1985.

[54] A. W. Roscoe. Recent Developments in CSP. Technical Monograph PRG-67, 1988.

[55] D. A. Schmidt. *Denotational Semantics*, Allyn and Bacon, 1986.

[56] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design.* LNCS 200, 1985.

[57] H. M. J. L. Schols. *A Formalisation of the Foam Rubber Wrapper Principle.* Master's Thesis, Eindhoven University of Technology, 1985.

[58] C. Seitz. "System Timing". Chapter 7 in *Introduction to VLSI Systems*, eds. C. Mead and L. Conway, Addison-Wesley Publishing Company 1980.

[59] E. Stark. Proving Entailment between Conceptual State Specifications. *Theoretical Computer Science*, Vol. 56, 1988, pp. 135-154.

[60] J. Staunstrup and M. R. Greenstreet. From High-Level Descriptions to VLSI Circuits. *BIT*, Vol. 28, 1988, pp. 620-638.

[61] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[62] J. W. Thathcer, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, Vol. 15, 1981, pp. 223-249.

[63] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits and system. *Distributed Computing*, Vol. 1, 1986, pp. 197-204.

[64] D. Walker. Bisimulations and Divergence. *Proc. of LICS 1988*, pp. 186-192.