Provably Correct Systems*

Jifeng He and C. A. R. Hoare[1]
Martin Fränzle and Markus Müller-Olm[2]
Ernst-Rüdiger Olderog and Michael Schenke[3]
Michael R. Hansen, Anders P. Ravn and Hans Rischel[4]

[1] Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
[2] Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstraße 1–9, D-24105 Kiel, Germany
[3] FB Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
[4] Department of Computer Science
Technical University of Denmark, bldg. 344
DK-2800 Lyngby, Denmark

Abstract. The goal of the Provably Correct Systems project (ProCoS) is to develop a mathematical basis for development of embedded, real-time, computer systems. This survey paper introduces the specification languages and verification techniques for four levels of development: Requirements definition and control design; Transformation to a systems architecture with program designs and their transformation to programs; Compilation of real-time programs to conventional processors, and Compilation of programs to hardware.

1 Introduction

An embedded computer system is part of a total system that is a physical process, a plant, characterized by a state that changes over real time. The role of the computer is to monitor this state through sensors and to change the state through actuators. The computer is simply a convenient device that can be instructed to manipulate a mathematical model of the physical system and state. Correctness means that the program and the hardware faithfully implement the control formulas of the mathematical model for the total system, and nothing else. However, the opportunities offered by the development of computer technology have resulted in large, complex programs which are hard to relate to the objective of systems control.

* This work is partially funded by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Project No. 7071: "**ProCoS II**: Provably Correct Systems". The hardware compilation work is partially funded by the UK Science and Engineering Research Council (SERC) under the Information Engineering Directorate SAFEMOS project (IED3/1/1036).

Activity	Documents	Language
Concept	Expectations	Natural (informal)
Requirements analysis	Requirements	RL
System design	System specs	SL
Program design	Program source	PL
Either:		
Hardware synthesis	Circuit diagram	Netlist
Or:		
Compilation	Machine code	ML

Fig. 1. Languages in **ProCoS**.

In the following, we describe a particular approach to mastering the complexities of such systems. The approach is the result of over 5 years of work within the **ProCoS** project, and represents a collection of techniques which have been successfully applied to the different aspects of development of real-time computer systems.

First of all there is a need for a short and precise specification of the desired control *requirements*, independent of the actual hardware and software system. For that purpose **ProCoS** has investigated a real-time logic, Duration Calculus [82, 22, 81, 80], that formalizes dynamic systems properties. This logic also provides a calculus such that a specification of controlling state machines can be verified to be a refinement of the requirements [66, 71, 67, 17, 31].

In order to use the logic as a specification language for requirements and design, we need a module concept and notations for standard concepts of discrete mathematics. Here we have decided to adapt the Z language [73] and embed the Duration Calculus [12] such that the type checking facilities and other tools of Z are available after expansion of the special Duration Calculus symbols. The resulting *requirements language* offers an interface to control theory [83], but it is not intended in any way to replace well known notations and procedures of control engineering.

A specification of controlling state machines is a step towards an implemented program. However, it neither defines the communication protocols among the state machines nor does it define the structure of the sequential programs. Here, **ProCoS** has chosen to build on the paradigm of synchronous communication and transform the state machines to a network of communicating processes [70]. The specification language SL [58, 59, 60] is used to specify such a network in a constraint oriented style, where protocols are defined through *trace assertions* that are regular expressions over communication channels; computations are specified by *communication assertions* that define the relation between the pre and post state of a program component and a communicated value; and *timing assertions* that constrain the time between communications. The SL language

also supports transformations [57, 69] to an occam-like programming language PL with real-time facilities.

These programs are then the basis for compilation to hardware or machine code, cf. [11].

Each step has to be correct, and unless we have blind faith in the developer, we expect to see documents for a rational development process, cf. [63]. Using a fairly standard division of a development into major activities, these documents can be organized as shown in Figure 1.

Each major activity layer uses its own specially tailored languages and verification techniques. It shall also link to the next lower layer. The inspiration for such a layered approach has been the CLIinc. "stack", see e.g. [8, 21]. A detailed technical account of work during the first 3 years of **ProCoS** is given in [5] and a previous survey of the results up till early 1993 is found in [9].

Overview. The following sections focus on central topics of each of the layers. In section 2 we use one of the **ProCoS** case studies to introduce requirements analysis and specification of a top level design. This design is used in section 3 to illustrate transformations to a systems architecture with program specifications and transformations to programs. Section 4 surveys work on developing a compiler for the real-time programming language that guarantees timing constraints for the generated machine code. Section 5 outlines some work on deriving a hardware description from a program.

In a final section 6, the underlying mathematical structures and the general approaches of the project are discussed and put into the perspective of a science of programming.

2 Requirements and Control Design

The first step in formalizing the requirements of a system is to construct a system model. Our basis is the well-known *time-domain model*, where a system is described by a collection of *states* which are functions of time (the real numbers).

Example: We illustrate the state concept through our running example: a version of a computer controlled gas burner [67].

The gas burner is controlled through a thermostat, and can directly control a gas valve and monitor the flame. This physical system or plant is illustrated by the diagram in Figure 2. For the gas burner we use the following discrete (Boolean valued) states *Heatreq*, *Gas* and *Flame* to model the state of the thermostat, the gas valve and the flame. In illustrations we assume that Boolean values are represented by 0 (*false*) and 1 (*true*). □

States or other quantities might be introduced informally as usually done in mathematics by phrases like 'let *Gas* denote ...and let 0 denote the Boolean value *false*'. This works very well for a small set of quantities used in a delimited context; but in development of a larger system with a modular structure and

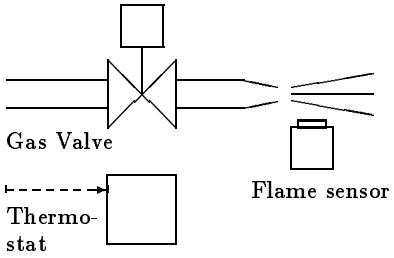


Fig. 2. Physical components – plant – for the gas burner.

going through several stages of refinement it is essential to be able to structure declarations corresponding to the specifications of systems and subsystems. Therefore we use a concrete syntax for specifications based on the Z-schema notation [73].

A state is a function from time to some value domain, but we do not want to refer to time explicitly. Thus **state** is used in front of a type of a name which denotes a state.

Example: The states for the gas burner example can thus be introduced as follows:

GB

Heatreq : **state Bool**
Flame, *Gas* : **state Bool**

where **state Bool** is an abbreviation for *Time* $\rightarrow$ **Bool** with *Time* denoting the set of non-negative reals. The input state *Heatreq* thus denotes a Boolean function of time. The controlled states are *Flame* and *Gas*. □

2.1 From Expectations to Requirements

Properties of systems are expressed by constraining the states over time. For that purpose we use the Duration Calculus [82].

Example: We introduce the logic while formalizing the following expectations for the gas burner.

Safe: Gas must not leak for more than 4 seconds[5] in any 30 second period.
Stop: The gas must not burn when heat request has been off for 40 seconds.
Start: The gas must burn when heat request has been on for 40 seconds, provided the gas ignites and burns satisfactorily.

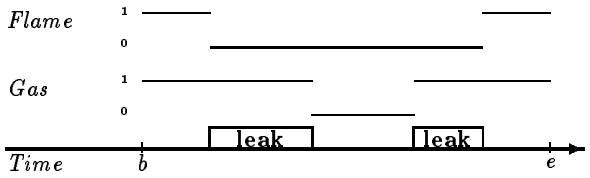
[5] We shall use second as a unit of time throughout the example.

Safe. We assume that a leak occurs whenever the state assertion

$$Leak == Gas \wedge \neg Flame$$

holds.

When we consider some bounded interval $[b, e]$ of time, we can measure the *duration* of *Leak* within the interval by $\int_b^e Leak(t)\,dt$, cf. the following timing diagram:



The symbol $\int Leak$ denotes the duration of *Leak*; for each particular interval it is a real number. An *atomic duration formula* is a predicate over intervals built from durations and real valued constants by a relation on real numbers. E.g. the 4 second constraint on leaks is the atomic formula $\int Leak \leq 4$.

The duration of the constant state 1 will be the length of the interval under consideration, and we abbreviate

$$\ell == \int 1$$

Thus the fact that an interval is not longer than 30 seconds is specified by $\ell \leq 30$. Formulas can now be generated from atomic formulas using the logical connectives. The safety expectation is thus

$$\ell \leq 30 \Rightarrow \int Leak \leq 4$$

i.e. if the given interval is not longer than 30 seconds then the duration of *Leak* in that interval is less than 4 seconds.

Start. For this expectation, consider a non-point interval where *Heatreq* holds; such an interval is described by

$$\lceil Heatreq \rceil == (\int Heatreq = \ell) \wedge (\ell > 0)$$

A counterexample for *Start* is, for a given interval, expressed with the binary "chop" operator [53]

$$(\lceil Heatreq \rceil \wedge \ell = 40)\ ;\ \lceil \neg Flame \rceil$$

The formula $\mathcal{F}_1\ ;\ \mathcal{F}_2$ (which reads $\mathcal{F}_1$ "chop" $\mathcal{F}_2$) holds on the interval $[b, e]$ just when this interval can be divided into an initial subinterval $[b, m]$ where $\mathcal{F}_1$ holds and a final subinterval $[m, e]$ where $\mathcal{F}_2$ holds. The "chop" operator is associative and monotone in both arguments. We assign it a priority higher than implication and lower than conjunction and disjunction.

The "somewhere" modality ($\Diamond$), defined by $\Diamond \mathcal{F} == true \ ; \ \mathcal{F} \ ; \ true$, can express the commitment

$$Start == \neg \Diamond ((\lceil Heatreq \rceil \wedge \ell = 40) \ ; \ \lceil \neg Flame \rceil)$$

I.e. there is not an interval that starts with *Heatreq* for 40 seconds and continues with $\lceil \neg Flame \rceil$.

Note that "always" ($\Box$) as usual can be defined in terms of "somewhere" (and thus "chop") by $\Box \mathcal{F} \Leftrightarrow \neg \Diamond (\neg \mathcal{F})$.

Stop. The last expectation is analogous to *Start*, so it can be given by the commitment

$$Stop == \neg \Diamond ((\lceil \neg Heatreq \rceil \wedge \ell = 40) \ ; \ \lceil Flame \rceil)$$

□

Standard forms In principle, commitments can be formulated using the full generality of Duration Calculus, but for the purpose of design some standard forms are useful [45].

Progress can be defined in terms of an elementary operator, called "leads-to" which expresses that a system which is in some initial region of its state space, specified by a formula $\mathcal{F}$ for some time t will continue to a goal state, specified by a state formula $\lceil P \rceil$.

Definition 2.1 (Leads-to)

$$\mathcal{F} \xrightarrow{t} \lceil P \rceil \ == \ \Box ((\mathcal{F} \wedge \ell = t) \ ; \ \ell > 0 \Rightarrow \ell = t \ ; \ \lceil P \rceil \ ; \ true)$$

The following law gives the relation to specification by counterexamples

$$\neg (\mathcal{F} \xrightarrow{t} \lceil P \rceil) \Leftrightarrow \Diamond ((\mathcal{F} \wedge \ell = t) \ ; \ \lceil \neg P \rceil)$$

The leads-to operator has the monotonicity properties of an implication, distributes over conjunction, and is monotone with respect to time for state assertions, i.e. when $t \leq t'$

$$(\lceil P_1 \rceil \xrightarrow{t} \lceil P_2 \rceil) \Rightarrow (\lceil P_1 \rceil \xrightarrow{t'} \lceil P_2 \rceil)$$

Leads-to is also transitive in the following sense

$$(\lceil P_1 \rceil \xrightarrow{t_1} \lceil P_2 \rceil) \wedge (\lceil P_2 \rceil \xrightarrow{t_2} \lceil P_3 \rceil) \Rightarrow (\lceil P_1 \rceil \xrightarrow{t_1 + t_2} \lceil P_2 \wedge P_3 \rceil)$$

It is easy to check that both the *Stop* and the *Start* commitments can be expressed using leads-to.

$$Stop == \lceil \neg Heatreq \rceil \xrightarrow{40} \lceil \neg Flame \rceil$$
$$Start == \lceil Heatreq \rceil \xrightarrow{40} \lceil Flame \rceil$$

Two derived forms will be used later, but are introduced here for convenience. In the "followed-by" form, the initial region does not have to be stable for a particular period of time.

Definition 2.2 (Followed-by)

$$\mathcal{F} \longrightarrow \lceil P \rceil == \forall r \bullet \mathcal{F} \xrightarrow{r} \lceil P \rceil$$

An instance is a state *transition* when a system is known to move from a state specified by P_1 to a state given by P_2 or remain stable

$$\lceil P_1 \rceil \longrightarrow \lceil P_1 \vee P_2 \rceil$$

A second form makes transition constraints time bounded.

Definition 2.3 (Time bounded transition)

$$\mathcal{F} \xrightarrow{\leq t} \lceil P \rceil == (\mathcal{F} \wedge \ell \leq t) \longrightarrow \lceil P \rceil$$

An example is conditional stability or *latching* where a condition P' latches the state P for a given period

$$\lceil \neg P \rceil \ ; \ \lceil P \wedge P' \rceil \xrightarrow{\leq t} \lceil P \rceil$$

Latching is also expressed by the law

$$(\lceil \neg P \rceil \ ; \ \lceil P \wedge P' \rceil \xrightarrow{\leq t} \lceil P \rceil) \Leftrightarrow \Box (\lceil \neg P \rceil \ ; \ \lceil P \wedge P' \rceil \ ; \ \lceil \neg P \rceil \Rightarrow \ell > t)$$

Another example is that a state P' is kept stable at least for some time t, when some other state P is entered. This is expressed by *framing*

$$(\lceil \neg P \rceil \ ; \ \lceil P \wedge P' \rceil) \xrightarrow{\leq t} \lceil P' \rceil$$

As for latching, the time condition can be omitted.

The *Safe* commitment can be formulated as a time bounded transition

$$Safe == (\int Leak = 4) \xrightarrow{\leq 30} \lceil \neg Leak \rceil$$

Assumptions Commitments are not the final requirements. During the design phase, the commitments may be weakened by assumptions about the plant.

Example: Being a bit clairvoyant about the gas burner design, we postulate the assumption

$$NoFlicker == (\lceil Flame \rceil \ ; \ \lceil \neg Flame \rceil) \xrightarrow{0.5} \lceil \neg Flame \rceil$$

which means that when the flame disappears, this state will be stable for half a second. This assumption is needed to detect a flame out situation which might violate *Safe*.

For *Stop*, it is assumed that no gas leads to no flame within a short time

$$Noflame == \lceil \neg Gas \rceil \xrightarrow{1} \lceil \neg Flame \rceil$$

and *Start* has as assumption that *Gas* ignites and burns, say within 3/4 of a second

$$GasOk == \lceil Gas \rceil \xrightarrow{0.75} \lceil Flame \rceil$$

□

In principle, any commitment can be taken as an assumption. Since assumptions are global, they have to be introduced from start. I.e. although they may be detected at a later stage of refinement, they shall in the final design document be introduced from the start. They are introduced as preconditions to the commitments. I.e. the assumptions and commitments define the requirements. A commitment C with assumption A thus gives the requirement

$$A \Rightarrow C$$

Given assumptions A, commitment C and a design D, the verification of the design demonstrates $D \Rightarrow (A \Rightarrow C)$ or equivalently

$$A \wedge D \Rightarrow C$$

An assumption thus replaces part of a design.

Assumptions are not for free. At the *end* of the design activity it must be proven that the assumptions in conjunction with the design is feasible. I.e. that the conjunction $A \wedge D$ is consistent with some assignment of state functions.

While feasibility is a formal property, there is still the question of whether the assumptions and the whole model is reasonable. This calls for *validation*, i.e. careful experiments to check the mathematical model against reality, a topic outside the scope of the current **ProCoS** project.

Requirements The total requirements for the gas burner can now be given by a schema with constraints

GBReq

GB

$NoFlicker \Rightarrow Safe$
$NoFlame \Rightarrow Stop$
$GasOk \Rightarrow Start$

In principle, the constraints of this schema are just a predicate which can be reached by unfolding the semantic definitions of the Duration Calculus.

2.2 Control design

A control design consists of two main parts:

1. Finite state machines or automatons describing how controllers progress through a number of *phases*. These are specified by formulas over phase control states. The formulas determine the conditions for a phase to be either stable or progress to a new phase.
2. A set of phase commitments that are local for each phase. They determine the behaviour of the system whenever the control is in the given phase.

Some phase commitments are considered elementary, and are interpreted as specifications of sensors or actuators. They will in general observe or control a single component of the plant state for a specific phase. With sequential and iterative decomposition of phases composite commitments can be refined till they are elementary for a finer set of phases [45].

When $\pi, \pi_1, \ldots$ denote single *phase* states, and $\varphi, \varphi_1, \ldots$ denote assertions on either the plant state or the phase states, the elementary forms are:

Sequencing: $\lceil \pi \rceil \longrightarrow \lceil \pi \vee \pi_1 \vee \cdots \vee \pi_n \rceil$, where $n = 0$ means that the controller is stable in phase π, while $n > 1$ means that there is a nondeterministic choice of a successor phase.

Progress: $\lceil \pi \wedge \varphi \rceil \stackrel{c}{\longrightarrow} \lceil \neg\pi \rceil$, where the phase π is left when φ holds for c time units. A progress form may also express *active stability* of the phase: $\lceil \pi \wedge \varphi \rceil \stackrel{c}{\longrightarrow} \lceil \pi \rceil$.

Selection: $\lceil \neg\pi \rceil$; $\lceil \pi \wedge \varphi \rceil \stackrel{\leq c}{\longrightarrow} \lceil \pi \vee \pi_1 \vee \ldots \vee \pi_n \rceil$, where the sequencing of phase π is constrained under the condition φ for c time units (or forever, if the time bound is omitted). If $n = 0$ the formula defines conditional, time bounded *stability* of the phase. Note that c is a lower bound, a design may keep the selection open for a longer time, but not for a shorter.

Synchronization: $\lceil \pi_1 \vee \ldots \vee \pi_n \rceil \stackrel{c}{\longrightarrow} \lceil \varphi \rceil$, where the combined phase $\pi_1 \vee \ldots \vee \pi_n$ will cause φ to hold after c time units. Note that c is an upper bound, a design is allowed to cause φ sooner but not later.

Framing: $\lceil \neg\pi \rceil$; $\lceil \pi \wedge \varphi \rceil \stackrel{\leq c}{\longrightarrow} \lceil \varphi \rceil$, is dual to phase stability. It is a commitment that the state φ will remain stable for c time units when the phase is entered.

Example: A gas burner control system satisfying the above requirements may be designed in many ways. The following design consists of a controlling state machine given informally by the diagram in Figure 3.
The system proceeds cyclically through the following *phases*:

idle: Initially and after a completed cycle. The gas is turned off. Heat request terminates the phase.
purge: The gas remains off for 30 seconds.
ignite: The gas valve is opened and ignition should occur. After 1 second the phase is left.
burn: Go to *Idle* if Flame or Heat request goes off.

This controlling automaton can be derived systematically from the requirements [45]. Here, we give only the final result which is specified by a controller state *main* ranging over the phases and a parameter δ denoting a reaction time

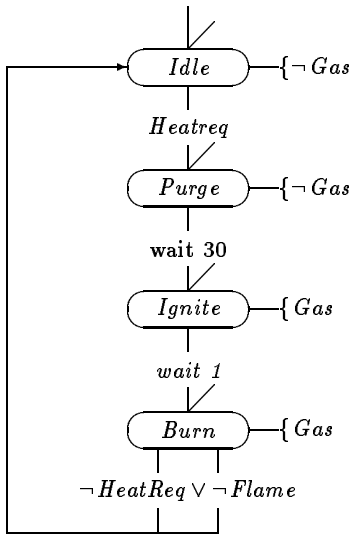


Fig. 3. Main controller of Gas burner.

GBCon

GB

$main$: **state** $\{Idle, Purge, Ignite, Burn\}$

$\delta : \mathbf{R}$

$0 < \delta < \frac{1}{9}$

The phases are given by abbreviations $idle == (main = Idle)$ etc.

The *sequencing* of the phases is given by a schema with simple transitions. The initial phase is *idle*.

GBSeq

$GBCon$

$\ell > 0 \Rightarrow \lceil idle \rceil \ ; \ true$

$\lceil idle \rceil \longrightarrow \lceil idle \vee purge \rceil$

$\lceil purge \rceil \longrightarrow \lceil purge \vee ignite \rceil$

$\lceil ignite \rceil \longrightarrow \lceil ignite \vee burn \rceil$

$\lceil burn \rceil \longrightarrow \lceil burn \vee idle \rceil$

For each of the phases, a collection of simple phase commitments can be defined. The idle phase is under *Heatreq* left within $2 \cdot \delta$ seconds, it is stable under $\neg Heatreq$, and *Gas* is turned off.

GBIdle

GBCon

$\lceil idle \wedge Heatreq \rceil \xrightarrow{2\cdot\delta} \lceil \neg idle \rceil$
$\lceil \neg idle \rceil \ ; \ \lceil idle \wedge \neg Heatreq \rceil \longrightarrow \lceil idle \rceil$
$\lceil idle \rceil \xrightarrow{2\cdot\delta} \lceil \neg Gas \rceil$

The purge phase is stable for 30 and left after $30 + \delta$ seconds. *Gas* is turned off.

GBPurge

GBCon

$(\lceil \neg purge \rceil \ ; \ \lceil purge \rceil) \xrightarrow{\leq 30} \lceil purge \rceil$
$\lceil purge \rceil \xrightarrow{30+\delta} \lceil \neg purge \rceil$
$\lceil purge \rceil \xrightarrow{2\cdot\delta} \lceil \neg Gas \rceil$

The *ignite* phase turns on the gas and is stable for 1 second before it is left for the burn phase.

GBIgnite

GBCon

$(\lceil \neg ignite \rceil \ ; \ \lceil ignite \rceil) \xrightarrow{\leq 1} \lceil ignite \rceil$
$\lceil ignite \rceil \xrightarrow{1+\delta} \lceil \neg ignite \rceil$
$\lceil ignite \rceil \xrightarrow{2\cdot\delta} \lceil Gas \rceil$

The *burn* phase is under $\neg Flame$ left within the noflicker period ($2 \cdot \delta \leq 0.5$); under $\neg Heatreq$ it is left within $38 - 5 \cdot \delta$ seconds; it is stable under $Heatreq \wedge Flame$; and finally, *Gas* is kept on.

GBBurn

GBCon

$\lceil burn \wedge \neg Flame \rceil \xrightarrow{2\cdot\delta} \lceil idle \rceil$
$\lceil burn \wedge \neg Heatreq \rceil \xrightarrow{38-5\cdot\delta} \lceil idle \rceil$
$\lceil \neg burn \rceil \ ; \ \lceil burn \wedge Heatreq \wedge Flame \rceil \longrightarrow \lceil burn \rceil$
$(\lceil \neg burn \rceil \ ; \ \lceil burn \wedge Gas \rceil) \longrightarrow \lceil Gas \rceil$

It can be verified using laws for combination of phases [45] that

$$GBSeq \wedge GBIdle \wedge GBPurge \wedge GBIgnite \wedge GBBurn \Rightarrow GBReq$$

□

2.3 Refinement towards an architecture

The above design does not consider a particular technology or architecture. However, for a program implementation with distribution it is necessary to formulate the design in terms of control states only, as pointed out by Schenke [70]. The plant states are mapped to control states by *simple* sensors and actuators (A/D and D/A converters) of the following shapes.

Example:

GBConP

$\delta : \mathbf{R}$
$hr, fl, gas : \mathbf{state\ Bool}$
$main : \mathbf{state}\ \{Idle, Purge, Ignite, Burn\}$

$0 < \delta < \frac{1}{9}$

hr, fl and gas are the phase states of the simple sensors and actuator.

GBHeat

GBConP
$Heatreq : \mathbf{state\ Bool}$

$\lceil Heatreq \rceil \xrightarrow{\delta} \lceil hr \rceil$
$\lceil \neg Heatreq \rceil \xrightarrow{\delta} \lceil \neg hr \rceil$

GBFlame

GBConP
$Flame : \mathbf{state\ Bool}$

$\lceil Flame \rceil \xrightarrow{\delta} \lceil fl \rceil$
$\lceil \neg Flame \rceil \xrightarrow{\delta} \lceil \neg fl \rceil$

GBGas

GBConP
$Gas : \mathbf{state\ Bool}$

$\lceil gas \rceil \xrightarrow{\delta} \lceil Gas \rceil$
$\lceil \neg gas \rceil \xrightarrow{\delta} \lceil \neg Gas \rceil$

The requirement can be refined using transitivity to communicate through these components. This reduces the allowed reaction times by δ. The phase constraints are reduced similarly, and are presented below, rearranged according to the elementary forms.

Sequencing is unchanged, while the progress constraints become

GBProg

GBConP

$$\lceil idle \wedge hr \rceil \xrightarrow{\delta} \lceil \neg idle \rceil$$
$$\lceil purge \rceil \xrightarrow{30+\delta} \lceil \neg purge \rceil$$
$$\lceil ignite \rceil \xrightarrow{1+\delta} \lceil \neg ignite \rceil$$
$$(\lceil burn \wedge \neg fl \rceil \xrightarrow{\delta} \lceil \neg burn \rceil$$
$$(\lceil burn \wedge \neg hr \rceil \xrightarrow{\delta} \lceil \neg burn \rceil$$

Stability constraints are

GBStab

GBConP

$$\lceil idle \rceil \ ; \ \lceil idle \wedge \neg hr \rceil \longrightarrow \lceil idle \rceil$$
$$(\lceil \neg purge \rceil \ ; \ \lceil purge \rceil) \xrightarrow{\leq 30} \lceil purge \rceil$$
$$(\lceil \neg ignite \rceil \ ; \ \lceil ignite \rceil) \xrightarrow{\leq 1} \lceil ignite \rceil$$
$$(\lceil \neg burn \rceil \ ; \ \lceil burn \wedge hr \wedge fl \rceil \longrightarrow \lceil burn \rceil$$

and synchronizations with actuators

GBSynch

GBCon

$$\lceil idle \vee purge \rceil \xrightarrow{\delta} \lceil \neg gas \rceil$$
$$\lceil ignite \vee burn \rceil \xrightarrow{\delta} \lceil gas \rceil$$

□

Despite the systematic refinement techniques used in the design, it would obviously be nice to have a tool to check constraints, e.g. on the design parameter δ [72].

This refinement forms a link to the architecture given by the SL specification in section 3.

2.4 Related work

The development of requirements for a model of a dynamic system presented above is similar in spirit although not in the chosen formalism to work by Leveson [40, 34] and Parnas [64]. The Duration Calculus builds on Moszkowski's interval temporal logic [53, 54, 75]. Time may also be handled explicitly as in TLA [38, 39] or within a conventional temporal logic [36, 65, 30]. In a proof assistant, model checking [20, 7] might be very useful. The refinement approach outlined above is inspired by the hierarchical state machines of Harel [23]. Designs can also be subjected to reliability analysis [79].

3 Architecture and Programs

The formalisms of duration calculus, even in the rather restricted standard forms, and programs in an occam-like [32] language PL are on very distant abstract levels. In particular at some step the description must change from the state based world of the duration calculus to the event based world of PL. That is where an intermediate stage comes in, the specification language SL [61] for reactive systems with real-time constraints. This language will be described now. We outline the transformation of an RL architectural design to an SL specification and outline a transformational approach for the systematic design of programs in the programming language PL.

The purpose of a reactive system is to react to stimuli from its environment so that this is kept in a certain desirable condition. To this end, the system may communicate with its environment via directed channels. As our running example we use the gas burner introduced in the previous section.

For reactive systems a variety of specification formalisms have been developed, among them Temporal Logic [42], iterative programs like action systems [1] or UNITY programs [16], input/output automata [41], process algebra [48, 2], and stream processing functions [14]. However, it remains a difficult task to design correct programs from such specifications.

In our approach we formulate transformation rules for the stepwise design and implementation of both sequential and concurrent systems.

3.1 Specification language SL

An SL specification begins with the description of an *interface* declaring the communication channels *ch* of the component, for example

`INPUT OF` *type ch*

Then the desired behaviour of the system components is described. This description is split into a *trace part*, a *state part* and a *timing part*.

Trace part The trace part specifies the sequencing constraints on the system channels but ignores the communication values. This is done by means of *trace assertions*. Each trace assertion is a regular expression over a communication alphabet. The trace part thus uses ideas from path expressions [15] and regular trace logic [84, 57].

The syntactical form of a trace assertion *ta* is

`TRACE` α `IN` *re*

where the *alphabet* α is a subset of the interface channels and *re* is a *regular expression* over these channels. Informally, *ta* describes the order in which the channels in α can occur in the sequences or *traces* of communications between component and environment: at any moment this order must correspond to a word in the language denoted by the regular expression *re*, the set of possible sequences of channel communications. If there are several trace assertions, all of them must be satisfied simultaneously.

Example: Consider the specification of a simple controller for the gas burner, cf. figure 3 and the schemas of section 2.3.

A systematic transformation introduces channels corresponding to all phase changes of the architecture and synchronizations, cf. schema *GBConP*. This is the interface for the main controller, one of several parallel components.

```
INPUT OF Signal yesheat, noheat, noflame
OUTPUT OF Signal idle, purge, ignite, burn,
                 gason, gasoff
```

The sequencing constraints on the phase automaton are recorded in a trace assertion of the main controller

```
TRACE alphabet IN pref cycle*
```

with `cycle` = `gasoff.yesheat.purge.ignite.gason.burn.(noheat+noflame).idle`

The alphabet here is the set of all events appearing in the regular expression. In regular expressions . and + and * are the usual operators that denote concatenation, choice and Kleene star. The operator `pref` denotes prefix closure; it specifies the stepwise evolution of a system where one communication occurs after the other.

For the heat and the flame controller we get the trace assertions

```
TRACE heaton, yesheat, heatoff,noheat
      IN pref (heaton.yesheat.heatoff.(noheat+ε))*
TRACE flon, floff, noflame
      IN pref (flon.floff.(noflame+ε))*
```

The choices with the empty word reflect the fact that in the main controller the burn phase can be left either with normal end of a heat request or a flame failure. In the first case the signal `noheat` will occur, but the signal `noflame` has to be dropped in the flame controller. In case of a flame failure it is the other way round. □

The trace part defines the control behaviour of a system without consideration of any internal state, communicated values or timing.

State part The state part specifies the communication values that can be exchanged over the interface channels. To this end local state *variables* may be introduced. Changes of these variables are recorded by a set of *communication assertions* in a pre-post style assertional specification. In the communication assertions the values and channels are linked together.

A variable x is declared as follows:

`VAR` *type* x `INIT` e

The expression e represents the initial value of x. The local variables constitute the state space of the specification but need not appear in an implementation of the specified system nor necessarily mirror the states of an RL specification.

A communication assertion for a channel *ch* has the form

`COM` *ch* `WRITE` $\overline{w}$ `READ` $\overline{r}$ `WHEN` *wh* `THEN` *th*

and describes the state transition that each communication on *ch* induces. The lists $\overline{w}$ and $\overline{r}$ record the state variables that may be modified or only read during the transition. The `WHEN` predicate *wh* and the `THEN` predicate *th* describe the precondition and the effect of the state transition. In *th*, a primed variable x' refers to the value of variable x at the moment of termination. The communication value on the channel *ch* is specified by @*ch*. In communication assertions, empty variable lists and predicates being `true` can be omitted. A `true` in the `WHEN` predicate shows that the communication is never forbidden by the state part, a `true` in the `THEN` predicate shows that the effect of the communication is not determined by the state part.

Example: In the gas burner specification above, communications are simple input and output signals, and we do not need a state part. However, consider a slightly changed scenario, where the sensors for heat request and flame denote the presence or absence of heat request or flame respectively by a variable. Their values are transmitted by the `yesheat`, `noheat`, `noflame` signals and changed by `burn`

```
VAR Boolean  flame, heatreq  INIT false :
```

The communication assertions that define the right flame and heatreq phases are

```
COM yesheat  WRITE heatreq  THEN heatreq' = true
COM noheat   WRITE heatreq  THEN heatreq' = false
```

which sets and resets the heatreq and

```
COM burn     WRITE flame  THEN flame' = true
COM noflame  WRITE flame  THEN flame' = false
```

The variable `flame` is not set by an external signal, but by `burn` because of the assumption that after that signal the flame is burning. In `idle` the variables can be tested. There we have only read variables. In this assertion there is even a `WHEN` predicate.

`COM idle READ heatreq, flame WHEN` $\neg$`heatreq`$\vee$ $\neg$`flame`

□

The example illustrates the flexibility of SL. If a design has a simple regular control structure, this can be recorded immediately in the trace assertions. If the design has a more involved control structure, the control can be encoded in the variables and the communication assertions. This is unavoidable, if the language of possible channel sequences is not regular. In the extreme, when no trace assertions are left, the specification is an action system. Although all

information can be encoded in the communication assertions, the trace assertions are useful for the task of developing implementations in a methodological way.

How to cope with a system which has infinitely many states, is shown in the specification of a railway crossing in [70].

3.2 Timing part

In the timing part it is specified when channels are ready for communication. Lower bounds are expressed by

`AFTER` *re* `WAIT` *ch r*

which means that after communication of a trace belonging to the language defined by the regular expression *re*, the system will not communicate on *ch* before time *r* has elapsed.
Upper bounds are expressed by

`AFTER` *re* `READY` *ch r*

which means that after communication of a trace in *re*, the system becomes ready to communicate on *ch* within time *r*.

In order to guarantee the timing restrictions we shall assume that for each channel *ch* there is a *latency* `lat(ch)` with the property that, if both communication partners are ready for a communication via *ch* for `lat(ch)` time units, the communication takes place. These latency constants will be used in our intermediate step, the SL specification. We do not assume maximal progress since this would prohibit implementation on real hardware.

Example: For the gas burner we get the following time restrictions:

```
AFTER cycle*.gasoff.yesheat.purge        WAIT ignite 30
AFTER cycle*.gasoff.yesheat.purge.ignite WAIT burn 1
```

for the main controller. The heat controller has the time restriction

```
AFTER cycle1*.heaton.purge.heatoff  WAIT heaton lat(noheat)
```

with `cycle1` = `heaton.purge.heatoff.(noheat+`ε`)`.
This requirement is necessary to enable a `noheat` signal, if such a signal is required. Otherwise the main controller would not necessarily get this information if a `heaton` follows too quickly. Then the main controller could not communicate on `purge`, and a system deadlock would only be prevented by a flame failure! So the `heaton` must be delayed for this extremely short time and the communication on `noheat` will be performed, if it becomes necessary.
The flame controller has the time restriction

```
AFTER cycle2*.flon.floff  WAIT flon lat(noflame)
```

with `cycle2` = `flon.floff.(noflame+`ε`)`.
This requirement is similar to the heat requirement. □

With the timing assertions, the timing constraint part allows tight control of real-time properties of a component.

3.3 System specifications

Component specifications can be named in SL as illustrated by the following aggregation of the example for the main controller.

Example:

```
SPEC GBMaincontrol
  INPUT OF Signal yesheat, noheat, noflame
  OUTPUT OF Signal idle, purge, ignite, burn,
                   gason, gasoff

  TRACE alphabet IN pref (gasoff.yesheat.purge.ignite.gason
                          .burn.(noheat+noflame).idle)*

  AFTER cycle*.gasoff.yesheat.purge         WAIT ignite 30
  AFTER cycle*.gasoff.yesheat.purge.ignite  WAIT burn 1
END
```

with `alphabet` and `cycle` as above.

The specifications `GBHeat` and `GBFlame` for the heat and the flame controller are even simpler and omitted here. □

Component specifications can in SL be combined with other components in a parallel construct denoted by the operator `SYN` and channels can be hidden by a `HIDE` operator.

Example:

```
SYSTEM GB
  HIDE yesheat, noheat, noflame
  SYN
    GBMaincontrol
    GBHeat
    GBFlame
END
```

This interplay between `SYN` and `HIDE` is also called `PAR` in the programming language. □

Altogether SL is a specification language that is close to the programming level. As such it extends Z specifications [73], UNITY programs [16] or action systems [1] by explicit communications and regular expressions to control their occurrence.

3.4 Programming language PL

We consider an occam-like [32] programming language PL where parallelism is allowed at the outermost level only. Programs are constructed using *programming operators* like

- `PAR` for the parallel composition of sequential components,
- `SKIP` to continue immediately (the neutral element of sequential composition),
- assignment $x := e$, input $ch?x$ and output $ch!e$, and
- `WHILE`, `SEQ`, `IF`, `ALT` for loops, sequential, conditional, and alternative composition.

In inputs and outputs the input destination x or output value e are omitted if the channel value type is `Signal`.

An upper bound on the computation time of a program segment is specified by prefixing the segment with

`UPPERBOUND` r

where r is a non-negative real constant. This applies only for the time needed for active computations. Times which are spent in a passive state during a delay are not included.

The wait time of a program segment is specified by an exact delay statement of the form

`DELAY` r

where r is a non-negative real constant.

3.5 Compilation of SL to PL

The simple structure of SL can be exploited in the development of PL programs. The trace part is transformed into a communication skeleton and the state part completes this skeleton to a program by adding purely sequential parts. Using this idea a large subset of SL specifications can be transformed fully automatically into PL by the so-called *syntax-directed transformation* SDT [69, 61]. The name "SDT-rule" refers to the idea to guide the development of the control structure by the syntactic structure of regular expressions. SDT is applicable to specifications containing only one trace assertion with a regular expression re which is of the form

$$\sum_i w_i.(\sum_j x_{i,j})^* + \sum_k y_k$$

with words $w_i, x_{i,j}, y_k$ over the channel alphabet such that certain conditions on the initial events are fulfilled. Then the program construction proceeds by induction on this structure:

- every letter is transformed into an input or output,
- every . is transformed into a `SEQ` construct,
- every $+$ is transformed into an `ALT` construct,
- every $*$ is transformed into a `WHILE true` loop,

where we have to take the time restrictions into consideration.

In order to apply the SDT-rule, in general we have to apply a number of transformations to the SL specification before. For instance the trace assertions have to be merged into a single assertion. This can be done because the semantics of trace assertions depends not on their particular form but on the language defined by them.

Example: Now the main controller of our above SL specification translates to the PL program

```
CHANNEL OF Signal gason, gasoff, yesheat, noheat, noflame,
                  idle, purge, ignite, burn :
WHILE true
  SEQ
    gasoff_u !
    yesheat_u ?
    purge_u !
    DELAY 30
    ignite_u !
    gason_u !
    DELAY 1
    burn_u !
    ALT MAXDELAY min(lat(noheat),lat(noflame))
      noheat ?
        idle_u !
      noflame ?
        idle_u !
```

For a communication `com` the expression `com_u ?` is an abbreviation for

```
UPPERBOUND lat(com)
  com ?
```

and analogously for `com !`. The `MAXDELAY` value (the minimum of the two latency constants) is a uniform upper bound for all initial actions of the `ALT`. □

More sophisticated concurrent programs can be developed using transformations on the full SL language mixed with PL so-called *mixed terms* [58]. These are constructs that mix the programming and specification notation. The individual transformations represent refinement steps from specifications via mixed terms to programs. This is an extension of refinement calculi devised for sequential programs [51].

3.6 Related work

In general we pursue a transformational approach where a given specification is transformed stepwise into a program. Our work is in the tradition of Dijkstra's approach to refinement, and the work originated by Burstall and Darlington and pursued further to practical application in projects like CIP (standing for

Computer-aided Intuition-guided Programming) [3] and PROSPECTRA (standing for PROgram development by SPECification and TRAnsformation) [37] but our emphasis is on concurrency and communication, as in [1].

In specific cases, several transformation rules can be combined to *strategies*, i.e. recipes how to apply them systematically or even automatically. SDT is one such strategy. More widely applicable is an *expansion strategy* [61]. Both SDT and expansion yield only sequential programs, i.e. without any concurrent composition. No fixed strategy is given for designing concurrent implementations. It is an "intuition-guided" activity where transformation rules are selected and applied [60]. A strategy by which parallelism can be introduced in the timed setting at some stage between the standard forms of duration calculus and SL is given in [70].

The correctness of all transformations and hence of the resulting occam-like programs is ultimately based on a combined *state-trace-readiness model* [61] for reactive systems which has been extended to a timed semantics. However, a user of the transformations will not be concerned with such semantic details, but will deal only with the syntactic rules.

4 Programs to Machine Code

Defective development software, in particular compilers, can result in incorrect machine code, even when a correct high-level program has been developed with the methods shown in the previous section. Inspection of generated machine code is a possible solution but is tedious and error-prone. It can be avoided by constructing reliable compilers.

The traditional approach to increase confidence in a compiler is to validate it by compiling test suites, and inspecting the results of executing the object programs. It is questionable whether this can replace target code inspection in safety-critical software development, as test programs normally exhibit rather simple behaviour and will not catch most of the timing and synchronization errors. Hence, development of a reliable compiler for a real-time programming language should include formal verification of its vital constituents, in particular of its code generator.

In order to verify a code generator one has to show that it maps source programs to semantically adequate target programs. The intricacy about this verification process is that syntactically and semantically different layers must be related, namely a structured source program and a flat list of machine language instructions. A correctness proof may thus easily become monolithic, aimed at a narrow source language with a specific code generator for a given target processor. Such a proof would have little interest beyond the particular application, and might still require a large effort. In **ProCoS** we have pursued a modular approach that should adapt to modifications of both the source and the target, and thus justify the effort. The approach [27] is based on defining the effect of machine programs in terms of the high-level language and using algebraic reasoning in verifying code generation. Assuming that code generators adhere

to the structures of the high-level language, the proof becomes modular. In the case of highly optimizing code generators this might be a limitation, but for the application area that **ProCoS** has in mind, this certainly is acceptable.

It is natural to think of instructions of von Neumann machines as denoting assignments to machine components like accumulators and store. Hence, the effect of machine instructions can be described by imperative programs. E.g., the transputer [33] instruction $\mathtt{ldc}(1)$, which loads the accumulator called $\mathtt{A}$ with 1, and moves $\mathtt{A}$'s contents to accumulator $\mathtt{B}$, as well as $\mathtt{B}$'s contents to accumulator $\mathtt{C}$, can be represented by the multiple assignment

$$\mathcal{E}(\mathtt{ldc}(1)) \mathrel{\widehat{=}} \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B} \ .$$

Similarly, the transputer instruction $\mathtt{stl}(x)$, writing $\mathtt{A}$'s contents to variable x, moving $\mathtt{B}$'s value to $\mathtt{A}$, $\mathtt{C}$'s value to $\mathtt{B}$, and an unspecified value to $\mathtt{C}$, can be represented by

$$\mathcal{E}(\mathtt{stl}(x)) \mathrel{\widehat{=}} x, \mathtt{A}, \mathtt{B}, \mathtt{C} := \mathtt{A}, \mathtt{B}, \mathtt{C}, - \ ,$$

where the assignment of $-$ to $\mathtt{C}$ abbreviates the nondeterministic choice between all possible assignments to $\mathtt{C}$. Clearly, the transputer instructions reference memory locations, not variable identifiers. We shall see in section 4.4 how program variables are mapped to memory locations.

For the purpose of this survey we idealize from some details of the transputer, namely the prefixing used to build large operands, the loading of programs to memory etc. A proper treatment of jump instructions is also omitted here.[6]

If the semantics of machine instructions is defined with source level programs as above, the source language's refinement algebra can be used to derive that certain machine instruction sequences refine, i.e. implement, certain source programs. E.g. the following calculation shows that the code sequence $\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle$ has the same effect as the assignment $x := 1$, if the additional effect of the machine instructions on the accumulators is taken to be irrelevant.

$$\begin{array}{ll} & \mathcal{E}(\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle) \\ = & \{\text{Instruction list}\} \\ & \mathcal{E}(\mathtt{ldc}(1)) \ ; \ \mathcal{E}(\mathtt{stl}(x)) \\ = & \{\text{Definitions above}\} \\ & \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B} \ ; \ x, \mathtt{A}, \mathtt{B}, \mathtt{C} := \mathtt{A}, \mathtt{B}, \mathtt{C}, - \\ = & \{\text{Combine assignments, Identity assignment}\} \\ & x, \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B}, - \\ = & \{\text{Identity assignment}\} \\ & x, \mathtt{C} := 1, - \end{array}$$

[6] Papers describing the code generator verification in more detail can be obtained via anonymous ftp from host `ftp.informatik.uni-kiel.de` (net address 134.245.15.114) in directory `/pub/kiel/procos`.

We have used the assignment laws

$$\begin{array}{rcll} (x := e) & = & (x, y := e, y) & \text{(Identity assignment)} \\ (x := e \ ; \ x := f) & = & (x := f[e/x]) & \text{(Combine assignments)} \end{array}$$

where $f[e/x]$ denotes substitution of e for x in expression f, and the property

$$\mathcal{E}(\langle i_1, \ldots, i_n \rangle) \ = \ \mathcal{E}(i_1) \ ; \ \ldots \ ; \ \mathcal{E}(i_n) \qquad \text{(Instruction list)}$$

of machine code sequences without jump instructions.

The little calculation above is a proof that $\langle \texttt{ldc}(1), \texttt{stl}(x) \rangle$ is correct target code for $x := 1$ when timing is not taken into account. But of course execution times of the source and target processes must be related in a proof of correct implementation of a real-time program. Therefore, we will now take a look at the mechanisms provided by PL, the programming language used in the **ProCoS**-project, to control execution times.

4.1 Real-time in programs

A PL *program* is a parallel composition of *sequential programs*. As in occam [32] an environment can observe a PL program only through communications on its external channels. Internal actions like assignments, particularly their execution times, can only indirectly be observed by their effect on succeeding communications. Semantically, the value of a program variable is not an observable of the entire programs [19, 56]. It is only visible inside sequential programs. Therefore, for sequential programs convenient abstractions about the timing of actions can be used since they need not be reflected directly by the corresponding machine code. Only the timing of communications must be preserved. More specifically the following abstractions have been built into the semantics of PL.

- The atomic internal actions are immediate, taking no execution time. This convention applies to assignments and the process `SKIP`.
- The control structures (sequential composition, conditional and loops) do not spend time beyond the time spent by their component processes.
- In contrast, communications spends time. Their time consumption is divided into two different classes: *Active time* is the time used by preceding internal actions preparing for the communication, while *waiting time* is time spent waiting for its communication partner to also engage into the communication. The amount of *real time* consumed by a communication command is always the *sum* of the two. The reason for distinguishing the two portions of real-time is compilability: Only the active time can be checked by a compiler whereas the waiting time depends on the synchronization structure of the algorithm.

These assumptions greatly simplify reasoning about sequential processes since they result in a number of powerful and general laws facilitating algebraic calculation. But they imply a task for a compiler: The execution time of machine

code implementing internal processes and control structures must be shifted to succeeding communication commands. This technique is described in section 4.3.

Basically, the active time consumption of external processes is unconstrained. For real-time programming, PL offers an upper-bound construct to constrain the active time. In this section we use the notation $\mid P \mid \leq t$ instead of the occam-style notation used in section 3. $\mid P \mid \leq t$ confines the enclosed sequential process P to spend at most t units of active time.

In order to reason about active time spent by internal activity of the machine we extend the programming notation by an *active delay*: $\Delta\ d$. It is a process that neither communicates nor changes the state and terminates after at most d units of active time.

4.2 Adding timing information to the effect processes

The description of the untimed meaning of an instruction has been given by describing its effect on the transputer state by a source language-like process. The additional description of the *instruction timing* can be done by adding appropriate delays to these effect processes. For simplicity, we ignore in this survey the dependencies of instruction timing on the length of operands and on the latencies of different memory devices.

Assuming that c is the duration of one processor cycle we can describe the effect of a load-constant instruction, which takes one cycle, by

$$\mathcal{E}(\texttt{ldc}(n)) \mathrel{\widehat{=}} \Delta\ c\ ;\ \ \texttt{A},\texttt{B},\texttt{C} := n, \texttt{A}, \texttt{B}\ .$$

Similarly,

$$\begin{aligned}
\mathcal{E}(\texttt{stl}(x)) &\mathrel{\widehat{=}} \Delta\ c\ ;\ \ x, \texttt{A}, \texttt{B}, \texttt{C} := \texttt{A}, \texttt{B}, \texttt{C}, - \\
\mathcal{E}(\texttt{ldl}(x)) &\mathrel{\widehat{=}} \Delta\ 2\ c\ ;\ \ \texttt{A}, \texttt{B}, \texttt{C} := x, \texttt{A}, \texttt{B} \\
\mathcal{E}(\texttt{outword}) &\mathrel{\widehat{=}} \Delta\ 25\ c\ ; \\
&\quad \texttt{if}\ \texttt{B} = \mathit{MinInt} + 0 \rightarrow \texttt{Link0}\ !!\ \texttt{A} \\
&\qquad \vdots \\
&\quad\ \ \texttt{B} = \mathit{MinInt} + 3 \rightarrow \texttt{Link3}\ !!\ \texttt{A} \\
&\quad \texttt{fi}\ ; \\
&\quad \texttt{A}, \texttt{B}, \texttt{C} := -, -, -
\end{aligned}$$

where *channel* !! *expression* abbreviates $\mid$ *channel* ! *expression* $\mid \leq 0$, i.e. a communication that spends no active time.

4.3 Meeting hard real-time constraints with compiled code

Since timing is mirrored in the semantics it is clearly reasonable to consider a machine program m an implementation of source process P iff the (timed) effect of the generated machine code refines the source process, i.e. iff $P \sqsubseteq \mathcal{E}(m)$. Unfortunately, this is much too restrictive. As described in section 4.1 the compiler must distribute the execution time of machine code corresponding

to internal processes. Our argument that $\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle$ refines $x := 1$ in an untimed world, for example, breaks down when timing is taken into account. $x := 1$, in contrast to the machine code, does not take time to execute. But since $\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle$ is reasonable code for $x := 1$ — indeed, the most reasonable one can think of — we need a more liberal correctness predicate.

The idea is to shift excess time of code implementing internal activity to a sequentially successive process that is compiled to a machine program needing less active time than allowed by the source. This can be accomplished by adding two parameters L and R to the correctness predicate, where L states the excess active time of the sequential predecessor that is absorbed and R states the excess active time that is handed over to the sequential successor for absorption. A third new parameter E is introduced that can be used to constrain the active time allowed for the source process. This leads to the following definition.

Definition 4.1 *A machine program m implements source process P, absorbing excess active time L from its sequential predecessor, exporting excess active time R to its sequential successor, under time bound E iff*

$$| P | \leq E \ ; \ \Delta R \sqsubseteq \Delta L \ ; \ \mathcal{E}(m) \ .$$

Now, using the laws

$$\begin{array}{rcll} (\Delta d \ ; \ x := e) & = & (x := e \ ; \ \Delta d) & (:=\text{-}\Delta\text{-commute}) \\ (| x := e | \leq d) & = & (x := e) & (:=\text{-bound}) \\ (\Delta d_1 \ ; \ \Delta d_2) & = & (\Delta(d_1 + d_2)) & (\Delta\text{-additivity}) \end{array}$$

we can show that $\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle$ implements $x := 1$, absorbing an arbitrary amount L of excess active time from its sequential predecessor, provided its sequential successor is willing to absorb $L + 2\,c$ units excess active time. As the source process cannot spend any active time, its context may place the obligation to spend the least possible active time, namely 0. This is expressed by

$$| x, \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, -, -, - | \leq 0 \ ; \ \Delta\,(L + 2\,c) \sqsubseteq \Delta\,L \ ; \ \mathcal{E}(\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle)$$

where, again, the additional effect on the accumulators is considered irrelevant. The proof is

$$\begin{array}{ll} & \Delta\,L \ ; \ \mathcal{E}(\langle \mathtt{ldc}(1), \mathtt{stl}(x) \rangle) \\ = & \{\text{Instruction list}\} \\ & \Delta\,L \ ; \ \mathcal{E}(\mathtt{ldc}(1)) \ ; \ \mathcal{E}(\mathtt{stl}(x)) \\ = & \{:=\text{-}\Delta\text{-commute}, \ \Delta\text{-additivity}\} \\ & \Delta\,L \ ; \ \Delta\,2\,c \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, \mathtt{A}, \mathtt{B} \ ; \ x, \mathtt{A}, \mathtt{B}, \mathtt{C} := \mathtt{A}, \mathtt{B}, \mathtt{C}, - \\ = & \{\text{combine assignments, identity assignment}\} \\ & \Delta\,L \ ; \ \Delta\,2\,c \ ; \ x, \mathtt{C} := 1, - \\ = & \{\Delta\text{-additivity}\} \\ & \Delta\,(L + 2\,c) \ ; \ x, \mathtt{C} := 1, - \end{array}$$

$$
\begin{array}{ll}
= & \{:=\text{-}\Delta\text{-commute}\} \\
& \quad x, \mathtt{C} := 1, - \ ; \ \Delta\,(L + 2\,c) \\
= & \{:=\text{-bound}\} \\
& \quad |\, x, \mathtt{C} := 1, - \,| \leq 0 \ ; \ \Delta\,(L + 2\,c) \\
\sqsupseteq & \{\text{nondeterminism}\} \\
& \quad |\, x, \mathtt{A}, \mathtt{B}, \mathtt{C} := 1, -, -, - \,| \leq 0 \ ; \ \Delta\,(L + 2\,c) \ .
\end{array}
$$

Note that the excess active time L of the sequential predecessor is simply handed through to the sequential successor.

A similar calculation shows that $\langle \mathtt{ldc}(\mathit{MinInt}), \mathtt{ldl}(x), \mathtt{outword} \rangle$ correctly implements $\mathtt{Link0}\,!\,x$ and absorbs all excess time coming from its sequential predecessor if the source process is placed into a context that allows to spend $L + 28\,c$ units of active time, i.e.

$$
\begin{array}{l}
|\, \mathtt{Link0}\,!\,x \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := -, -, - \,| \leq (L + 28\,c) \ ; \ \Delta\,0 \\
\qquad\qquad \sqsubseteq \ \Delta\,L \ ; \ \mathcal{E}(\langle \mathtt{ldc}(\mathit{MinInt}), \mathtt{ldl}(x), \mathtt{outword} \rangle) \ .
\end{array}
$$

Note that this illustrates that excess execution time can be absorbed by unused active time of sequentially following communication commands. The inequality is proved by

$$
\begin{array}{ll}
& \Delta\,L \ ; \ \mathcal{E}(\langle \mathtt{ldc}(\mathit{MinInt}), \mathtt{ldl}(x), \mathtt{outword} \rangle) \\
= & \{\text{Instruction list}\} \\
& \Delta\,L \ ; \ \mathcal{E}(\mathtt{ldc}(\mathit{MinInt})) \ ; \ \mathcal{E}(\mathtt{ldl}(x)) \ ; \ \mathcal{E}(\mathtt{outword}) \\
\sqsupseteq & \{:=\text{-}\Delta\text{-commute, }\Delta\text{-additivity, combine assignments}\} \\
& \Delta\,L \ ; \ \Delta\,28\,c \ ; \ \mathtt{Link0}\,!!\,x \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := -, -, - \\
= & \{\Delta\text{-additivity}\} \\
& \Delta\,(L + 28\,c) \ ; \ \mathtt{Link0}\,!!\,x \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := -, -, - \\
\sqsupseteq & \{!\text{-bound}\} \\
& |\, \mathtt{Link0}\,!\,x \,| \leq (L + 28\,c) \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := -, -, - \\
= & \{\text{Internal actions may be moved across bounds}\} \\
& |\, \mathtt{Link0}\,!\,x \ ; \ \mathtt{A}, \mathtt{B}, \mathtt{C} := -, -, - \,| \leq (L + 28\,c) \ ,
\end{array}
$$

using the law

$$
(\Delta\,d \ ; \ c\,!!\,e) \ \sqsupseteq \ (|\, c\,!\,e \,| \leq d) \ . \qquad (!\text{-bound})
$$

Clearly, the most interesting question when compiling a real-time programming language is how the compiler is going to ensure that every deadline stated in the source program is met at run-time. E.g. consider an upper bound $|\, P \,| \leq t$ in the source process to be compiled, constraining the active time available to the enclosed process P and thus demanding a certain execution speed of the compiled code. Then from the law

$$
(|\, P \,| \leq t_1) \ \sqsupseteq \ (|\, P \,| \leq t_2) \ , \qquad \text{if } t_1 \leq t_2 \ , \qquad (\text{Bound-refinement})
$$

we get that $|P| \leq t$ is implemented by machine code m, absorbing excess time L from its predecessor and exporting excess time R to its successor (i.e. $|P| \leq t\ ;\ \Delta R \sqsubseteq \Delta L\ ;\ \mathcal{E}(m)$), if there is $E \leq t$ with $|P| \leq E\ ;\ \Delta R \sqsubseteq \Delta L\ ;\ \mathcal{E}(m)$. Therefore a compiler encountering an upper bound operator in the source program must only check whether the required time bound is more liberal than the one asserted upon the code generated for the enclosed process. If it is, then no further action is necessary, as the real-time requirement expressed by the bound is met. If it is less liberal, on the other hand, then this indicates that the source process cannot be adequately compiled for the given target hardware with the given code generation strategy, and it should be rejected (or perhaps another code generator should be activated).

4.4 Representing source program state by memory locations

In the previous examples, variables have been addressed by their name. This has allowed us to use simple refinement formulas when illustrating correctness of code generation. Clearly, the actual transputer can only access storage locations instead of named variables. As the environment only interacts with a program through its external channels whereas variables and internal channels of the program are completely hidden, the latter components can be arbitrarily represented.

But it would be too liberal just to forget about variables and internal channels as we want a compositional code generator verification that can deduce correctness of code for a sequential composition from correctness of code for its components. The approach is to ensure that an encoding of the state of the additional components is always available on the machine.

The well-known standard technique in code generation is to establish a symbol table assigning locations in the machine store to variable names of the source process. Suppose that $x, \ldots, z$ are the variables of the source process and that Ψ is a symbol table that maps each variable name of the source program to the workspace, `M`, location allocated to hold its value, so $\mathtt{M}[\Psi\, x]$ is the value of x. Clearly it is necessary to insist that Ψ is a total injection. We define a symbolic dump $\hat{\Psi}$; it assigns to each high level variable the value in the corresponding location [29], thus retrieving the source process state from the machine state.

$$\begin{aligned}\hat{\Psi} \stackrel{\text{def}}{=}\ & \mathtt{var}\ x, \ldots, z\ ; \\ & x, \ldots, z := \mathtt{M}[\Psi\, x], \ldots, \mathtt{M}[\Psi\, z]\ ; \\ & \mathtt{end}\ \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{M}\end{aligned}$$

Here, `var` $x, \ldots, z$ introduces the variables $x, \ldots, z$ of type `integer`. The corresponding `end` `A`, `B`, `C`, `M` forgets, i.e. ends the scope of variables `A`, `B`, `C` and `M`.

Vice versa, the part of the machine store which is allocated by the symbol table can be initialized such that it reflects the current values of $x, \ldots, z$ by reverse dumping with

$$\begin{aligned}\hat{\Psi}^{-1} \stackrel{\text{def}}{=}\ & \mathtt{var}\ \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{M}\ ; \\ & \mathtt{M}[\Psi\, x], \ldots, \mathtt{M}[\Psi\, z] := x, \ldots, z\ ; \\ & \mathtt{end}\ x, \ldots, z\end{aligned}$$

A notable difference between $\hat{\Psi}$ and $\hat{\Psi}^{-1}$ is that the dump $\hat{\Psi}$ determines the complete state of the source process, whereas the reverse dump $\hat{\Psi}^{-1}$ only determines part of the machine state. Therefore, they are not inverses, but an easy calculation shows that the pair $(\hat{\Psi}, \hat{\Psi}^{-1})$ is a *simulation* [29], i.e. that

$$\hat{\Psi} \; ; \; \hat{\Psi}^{-1} \sqsubseteq \texttt{SKIP} \quad \text{and} \quad \texttt{SKIP} \sqsubseteq \hat{\Psi}^{-1} \; ; \; \hat{\Psi} \; .$$

Now, using the dumps as a means of understanding state representation with respect to a symbol table Ψ, one can define

> *a source process P is implemented by machine program m relative to a symbol table Ψ, iff* $\hat{\Psi} \; ; \; P \sqsubseteq \mathcal{E}(m) \; ; \; \hat{\Psi}$,

which closely corresponds to the well-known notion of a commutative diagram formalizing state representation. Note that by ending the scope of `A`, `B` and `C` at the end of $\hat{\Psi}$ we also have formally expressed that the effect on the accumulators is irrelevant.

The above implementation condition is only half of the story, as it does not cover absorption of excess time as demonstrated in the previous section. Combining both state representation via symbolic dumps and move of excess time, we arrive at the following implementation condition

Definition 4.2 *A machine program m implements source process P with respect to the symbol table Ψ, absorbing excess active time L from its sequential predecessor, exporting excess active time R to its sequential successor, under time bound E, iff*

$$\hat{\Psi} \; ; \; (|\, P \,| \leq E) \; ; \; \Delta R \sqsubseteq \Delta L \; ; \; \mathcal{E}(m) \; ; \; \hat{\Psi} \; .$$

For brevity, this property is denoted $\mathcal{T}(P, m, \Psi, L, R, E)$.

As $\mathcal{T}$ expresses correctness of machine code fragment m relative to source process P it is the *implicit code generator specification* for process compilation used in the **ProCoS II** project.

4.5 Towards explicit code generator specifications

The simplest reasonable specification of a code generator, but also the most implicit, is to say that it assigns *correct* machine code fragments to source objects. Aiming at a compositional code generator specification that proceeds along the syntactic structure of source programs, it is furthermore desirable to state additional implementation properties for program components that imply correctness of code assigned to full programs when satisfied on all program components. That is exactly what has been done when defining the implementation property $\mathcal{T}$ in the previous section. Hence, $\mathcal{T}$ qualifies as a reasonable implicit specification of code generation for processes.

Now, the task of code generator design is to develop a complying explicit code generator specification from the implicit one. We accomplish this by proving theorems about the implicit specification $\mathcal{T}$ in the calculational style demonstrated

in the introduction to this section and in section 4.3. There we have shown theorems for special cases of assignment and output. For compound constructs $op(P_1, \ldots, P_n)$ the theorems take the form of an implication that establishes the correctness of code for the compound construct from correctness of code for the components $P_1, \ldots, P_n$ and syntactic conditions on the surrounding code. Taking a slightly different view, such a theorem describes how to construct correct code for a larger process if correct code for all its components is known. Thus, the collection of these theorems induces a syntactically defined compiling relation $\mathcal{C}$ that is a sub-relation of the correctness relation $\mathcal{T}$, i.e. an *explicit* specification of a correct code generator. If necessary $\mathcal{C}$ is further specialized to a function. This corresponds to code selection. More specifically, the specialized $\mathcal{C}$ is a function depending on the source process P the symbol table Ψ and the absorption capacity L.

In the framework of the **ProCoS**-project we develop a prototype compiler written in the functional language Standard ML [49, 76]. The code generator is simply given by expressing $\mathcal{C}$ in Standard ML syntax.

4.6 Related Work

The idea of specifying a machine by a high level program is old and is present already in the idea of micro-programming [77]. Alain Martin and others [43, 13, 25] use such descriptions as a starting point for hardware design (see also section 5). One of the contributions of Hoare et al. [27, 29] on which our work is based is the proposal to use such a description together with a refinement algebra related to the source language [28, 50] for the reasoning about compilers. From more classical work about compiler correctness [47, 52, 74, 35] we are distinguished by aiming at code for an actual processor and not for idealized hardware, a goal shared with E. Börger et. al. [6], who are also concerned with proving correct compilation of occam. We try to accurately reflect the restrictions imposed by the hardware. The additional complexity of aiming at actual hardware requires modularity to split the verification in relatively small independent steps. Another difference to classical methods is that we use refinement as the correctness notion instead of semantic equivalence (this is also borrowed from [29]). This allows a proper treatment of under-specification in the source language's semantics (e.g. of uninitialized variables) and accommodates modularization. We treat real-time and communication [55, 18]. Like the work at CLInc [4] on the 'verified stack' we try to interface to higher and lower levels of abstraction.

5 Programs to Hardware

In this section we outline a technique to compile programs written in PL directly into hardware via provably correct transformations. A PL program defines what a hardware circuit should achieve, while a hardware description language, on the other hand, provides a way to express the components of a hardware circuit and their interconnections. Hardware description languages are widely used in

many computer-aided systems, allowing libraries of standard checked hardware modules to be assembled.

Here a simple description language for globally clocked circuits will be given an observation-oriented semantics based on the states of the wires of a device. Algebraic laws based on this semantics permit circuit descriptions to be expressed in a *hardware normal form*. This form is designed to guarantee absence of such errors as combinational cycles and conflicts. The necessary link with the higher abstraction level of the programming language is provided by an *interpreter* written in the programming language itself.

A hardware normal form is a correct translation of a source program if its interpretation is a refinement of the source program. For the primitive components of the source language this is proved directly by giving corresponding circuits; and then a series of theorems shows how a correct circuit for a composite program can be constructed from circuits that implements its components. Such theorems can be interpreted as transformation rules, which can be used directly or indirectly in the design of an automatic compiler.

The hardware normal form is fairly close to the typical notation of a hardware *netlist* language describing the interconnection of basic digital components. These netlists can be implemented in hardware in many ways. Currently, we use FPGAs (Field Programmable Gate Arrays) which can be dynamically reconfigured by software. This enables us to build hardware implementations of modest-sized programs entirely by a software process. A significant feature of such a hardware implementation is that a *global clock* synchronizes the activity of subcomponents; i.e., updates on latches can only take place at the end of each clock cycle.

5.1 Programming language

The programming language used in this section is close to both occam and PL, as introduced in section 3. Here we only consider variables of type `BOOL` for simplicity, but more complicated data types can and have been handled [62]. At the target hardware level, communication is implemented using a synchronous handshaking protocol at discrete clock cycle time steps. Verification of the target hardware assigned to PL programs takes advantage of the source language's refinement calculus. The basic algebraic laws for occam programs are given in [68]. In [24] we have presented some algebraic laws specifically concerned with real-time properties of programs.

In order to model the behaviour of a clocked circuit we add the generalized assignment to the PL language.

Let $R(v,\ v')$ be a predicate relating the final value v' of the program variable v to its initial value v. For simplicity we assume that R is *feasible*, i.e. $\forall\, v \exists\, v' \bullet R$. The notation

$$v :\in R$$

is a generalized assignment which assigns v a value such that the post-condition R holds at its termination.

The following laws illustrate the relation between a generalized assignment and ordinary ones:

$$v := e \;=\; v :\in (v' = e)$$
$$(x := e;\; y := f) \;=\; x, y :\in ((x' = e) \wedge (y' = f[x'/x]))$$

5.2 Synchronous Circuits

A digital circuit has one or more input and output wires. Its behaviour is described by a predicate on the values of these wires. There are only two stable values for a wire: either 0 standing for connection to the ground, or 1 standing for the presence of electrical potential. A synchronous circuit is equipped with a global clock which runs slow enough such that all inputs of the circuit become stable before being latched at the end of each clock cycle. The unspecified duration of each cycle is here taken as the unit of time.

Digital Elements Let W be a Boolean expression, and w be a wire name not used in the expression W. The notation $w.\mathbf{Comb}(W)$ describes a combinational circuit where the value of the output wire w is defined by the value of W. This relationship holds only at the end of each clock cycle. Let w_t and W_t represent the values at time t of w and W respectively. Since only the states that devices assume at the ends of clock cycles are of interest, the behaviour of the combinational circuit $w.\mathbf{Comb}(W)$ is described by

$$w.\mathbf{Comb}(W) \mathrel{\widehat{=}} \forall t \bullet (w_t = W_t)$$

where the range of t is the natural numbers, denoting clock cycles.

Another elementary circuit is the latch $x.\mathbf{Latch}(B,\, E)$ where on each clock cycle, the output wire x takes the value of the expression E of the *previous* clock cycle when the condition B is true; otherwise the value of x remains unchanged. Initially x is 0.

$$x.\mathbf{Latch}(B,\, E) \mathrel{\widehat{=}} (x_0 = 0 \wedge \forall\, t \bullet x_{t+1} = (E_t \lhd B_t \rhd x_t)$$

A variation of a latch is the delay element $l.\mathbf{Delay}(E) = l.\mathbf{Latch}(1,\, E)$, where on each clock cycle, the output wire l takes the value of the expression E on the *previous* clock cycle.

Input and Output Let D be a Boolean expression and d a wire name not mentioned in D. The notation $d.\mathbf{Out}(D)$ stands for the combinational circuit $d.\mathbf{Comb}(D)$, where the output wire d is used to connect the circuit with its environment.

Similarly, the notation $c.\mathbf{In}$ represents an input wire c whose value is solely determined by the environment. Since the value of an input wire is arbitrary, the behaviour of $c.\mathbf{In}$ is described by the following nondeterministic assignment:

$$c.\mathbf{In} \mathrel{\widehat{=}} \forall t \bullet (c_t = 0 \vee c_t = 1)$$

Composition Circuit components can be connected by *parallel composition* to form larger circuits. Internal wires of such assemblies can be made invisible to the environment using *hiding*.

A pair of components ($C1$, $C2$) with distinct output wires can be assembled by connecting like-named wires, making sure that any cycle of connection is cut by a latch. Since the value observed on the input end of a wire is the same as that produced on the output end, the combinational behaviour of an assembly can then be described by the conjunction of their descriptions as follows:

$$C1 \,\&\, C2 \mathrel{\widehat{=}} C1 \wedge C2$$

To explain hiding, let w be a wire used in a circuit C. If the environment does not use w, we can hide w by existential quantification

$$\exists\, w \bullet C \ .$$

5.3 Hardware Normal Form

Let $\underline{w}$ be a list of wire names and $\underline{W}$ a list of Boolean expressions of the same length as $\underline{w}$. We will use the notation $\underline{w}.\mathbf{Comb}(\underline{W})$ to represent the network of combinational circuits

$$w_1.\mathbf{Comb}(W_1) \,\&\, .. \,\&\, w_{\#\underline{w}}.\mathbf{Comb}(W_{\#\underline{W}})$$

where $\#\underline{w}$ is the length of the list $\underline{w}$. Later we will adopt the same convention for networks of Delay elements and latches.

Let $\underline{B}$ and $\underline{E}$ be lists of Boolean expressions with the same length as the list $\underline{x}$ of latch names. For notational simplicity, we use the notation $\underline{E} \lhd \underline{B} \rhd \underline{x}$ to stand for the list of conditionals

$$\{(E_1 \lhd B_1 \rhd x_1),\ \ldots\ , (E_{\#\underline{x}} \lhd B_{\#\underline{x}} \rhd x_{\#\underline{x}})\}$$

Since we are aiming at a circuit structure where circuit activity is triggered by the environment through a signal on a *start wire* named s and termination of its activity is signalled to the environment through a signal on a *finish wire* named f, and where only input and output terminals and latch states are visible to the environment, we are interested in circuits of the following special form. Let $\underline{c}$ be a list of input wire names with $s \in \underline{c}$, and $\underline{d}$ a list of output wire names with $f \in \underline{d}$. Let $\underline{x}$ be a list of latch names, $\underline{l}$ a list of Delay names, and $\underline{w}$ a list of combinational gate names. The circuit

$$C(s, f) \mathrel{\widehat{=}} \exists\, l_1, \ldots, l_{\#\underline{l}},\ w_1, \ldots, w_{\#\underline{w}} \bullet \left(\begin{array}{l} \underline{l}.\mathbf{Delay}(\underline{L}) \\ \&\ \underline{w}.\mathbf{Comb}(\underline{W}) \\ \&\ \underline{x}.\mathbf{Latch}(\underline{B},\ \underline{E}) \\ \&\ \underline{c}.\mathbf{In} \\ \&\ \underline{d}.\mathbf{Out}(\underline{D}) \\ \&\ f.\mathbf{Delay}(F) \end{array} \right)$$

is a network where $s \in \underline{c}$ is an input wire by which the environment triggers the circuit and $f \in \underline{d}$ is an output wire where the circuit signals the end of its operation.

Definition 5.1 (Normal Form) *The circuit* $C(s, f)$ *is a* hardware normal form *if it satisfies the following conditions:*

(NF-1) *: The output wire* f *is not used as input by any component.*
(NF-2) *: All the latches* $x_i.\mathbf{Latch}(B_i, E_i)$
(NF-3) *: None of the Boolean expressions* $\underline{L}$, $\underline{W}$, $\underline{D}$, $\underline{B}$ *and* F *is true in the case when all wires* $\underline{l}$, $\underline{w}$ *and* s *have the value* 0.

The first condition states that the wire f acts only as a link between the circuit C with its environment. The second condition is technical, but can easily be lifted by adding circuitry. It is necessary for non-interference between parallel components sharing a latch. Condition 3 characterizes a specific kind of 'one-hot' synchronous circuit [62] where control signals are used to initiate activities.

Multiplexor When the circuits resulting from software translation are put together, we need some method of allowing them to share their latches, input wires and output wires. Let C_i $(i = 1, 2)$ be a network

$$C_i \mathrel{\widehat{=}} \exists\, \underline{v}_i \bullet \left(\begin{array}{l} \quad \underline{w}_i.\mathbf{Comb}(W_i) \\ \&\ \underline{l}_i.\mathbf{Delay}(\underline{L}_i) \,\&\, \underline{x}_i.\mathbf{Latch}(\underline{B}_i, \underline{E}_i) \\ \&\ \underline{c}_i.\mathbf{In} \,\&\, \underline{d}_i.\mathbf{Out}(\underline{D}_i) \end{array} \right)$$

where $\underline{v}_i \subseteq (\underline{w}_i \cup \underline{l}_i)$. Assume that C_1 and C_2 do not share wire names except latches, input and output devices, and also avoid combinational cycles. The notation $\mathtt{Merge}(C_1, C_2)$ represents a network produced by using multiplexors to merge the latches and outputs in C_1 and C_2, namely

$$\mathtt{Merge}(C_1, C_2) \mathrel{\widehat{=}} \exists\, \underline{v}_1, \underline{v}_2 \bullet \left(\begin{array}{l} \quad \underline{w}_1.\mathbf{Comb}(\underline{W}_1) \,\&\, \underline{w}_2.\mathbf{Comb}(\underline{W}_2) \\ \&\ \underline{l}_1.\mathbf{Delay}(\underline{L}_1) \,\&\, \underline{l}_2.\mathbf{Delay}(\underline{L}_2) \\ \&\ \underline{x}.\mathbf{Latch}(B, E) \\ \&\ \underline{c}.\mathbf{In} \\ \&\ \underline{d}.\mathbf{Out}(D) \end{array} \right)$$

where

$$x.\mathbf{Latch}(B, E) \mathrel{\widehat{=}} \left(\begin{array}{l} \quad \{x.\mathbf{Latch}(B_1, E_1) \mid x \in \underline{x}_1 \setminus \underline{x}_2\} \\ \&\ \{x.\mathbf{Latch}((B_1 \vee B_2), (E_1 \vee E_2)) \mid x \in \underline{x}_1 \cap \underline{x}_2\} \\ \&\ \{x.\mathbf{Latch}(B_2, E_2) \mid x \in \underline{x}_2 \setminus \underline{x}_1\} \end{array} \right)$$

and $\underline{c}$ represents the union of $\underline{c}_1$ and $\underline{c}_2$, and

$$\underline{d}.\mathbf{Out}(D) \mathrel{\widehat{=}} \left(\begin{array}{l} \quad \{d.\mathbf{Out}(D_1) \mid d \in \underline{d}_1 \setminus \underline{d}_2\} \\ \&\ \{d.\mathbf{Out}(D_1 \vee D_2) \mid d \in \underline{d}_1 \cap \underline{d}_2\} \\ \&\ \{d.\mathbf{Out}(D_2) \mid d \in \underline{d}_2 \setminus \underline{d}_1\} \end{array} \right)$$

From the definition it is easy to check that `Merge` is commutative and associative, allowing us to treat `Merge` as a unary operator with a set of networks as its argument.

One can also prove that when $C_1(s,\ h)$ and $C_2(h,\ f)$ are normal forms without shared wires except data latches and outputs, then the network

$$\exists\, h \bullet \texttt{Merge}(C_1,\ C_2)$$

is also a normal form.

5.4 Simulator

The normal form $C(s, f)$ described in the previous section starts its operation after being triggered by an input signal from the wire $s \in \underline{c}$. Initially, all its Delay elements are reset to 0, and the combinational components enter their stable states immediately afterward. The initialization phase of $C(s, f)$ can thus be described by a generalized assignment

$$\begin{array}{l} \mathit{Init} \mathrel{\widehat{=}} \\ \quad \underline{l}, \underline{w}, \underline{c}, \underline{d}, f \ :\in (s' = 1)\ \&\ (\underline{l} = \underline{0})\ \&\ (\underline{w'} = \underline{W'})\ \&\ (\underline{d'} = \underline{D'})\ \&\ (f' = 0) \end{array}$$

The activity of the normal form in one clock cycle is described by a delay and a generalized assignment

$$\begin{array}{l} \mathit{Step} \mathrel{\widehat{=}} \\ \quad \Delta 1\ ;\ \underline{l},\ \underline{w},\ \underline{x},\ \underline{c},\ \underline{d},\ f\ :\in \\ \qquad\qquad (s' = 0)\ \&\ (f' = F)\ \& \\ \qquad\qquad (\underline{l'} = \underline{L})\ \&\ (\underline{w'} = \underline{W'})\ \&\ (\underline{x'} = \underline{E} \lhd \underline{B} \rhd \underline{x})\ \&\ (\underline{d'} = \underline{D'}) \end{array}$$

where the delay statement $\Delta 1$ takes one clock cycle. The operation of C can then be modelled by iterating *Step* while $\neg f$.

The circuit C signals completion on the wire f. In order to do so, condition 3 of the normal form requires that the values of the wires $\underline{w}$ and $\underline{l}$ become 0 when f becomes active. This requirement is represented by an assertion *Final*

$$\mathit{Final} \mathrel{\widehat{=}} (\neg \underline{w}\ \&\ \neg \underline{l})\_$$

When both $\underline{l}$ and $\underline{w}$ are empty lists $\mathit{Final} = (\texttt{true})\_ = \texttt{SKIP}$.

In summary the behaviour of the normal form $C(s, f)$ can be described by the following simulator:

$$\begin{array}{l} < s,\ C,\ f > \mathrel{\widehat{=}} \texttt{VAR}\ s, f, \underline{l}, \underline{w}\ : \\ \qquad\qquad \texttt{SEQ} \\ \qquad\qquad\quad \mathit{Init} \\ \qquad\qquad\quad \texttt{WHILE}\ \neg f \\ \qquad\qquad\qquad \mathit{Step} \\ \qquad\qquad\quad \mathit{Final} \\ \qquad\qquad \texttt{END}\ s, f, \underline{l}, \underline{w} \end{array}$$

where the start and finish wires s and f are the only control wire connections to the outside world.

$C(s, f)$ is a correct implementation of a program P if

$$P \ \sqsubseteq\ < s,\ C, f >$$

In what follows we will only deal with hardware normal forms.

5.5 Hardware compilation

The rest of this section demonstrates how to convert a PL program into a hardware normal form.

Let b be a Boolean expression. The assignment $(\Delta 1);\ x := b$ assigns the value of b to x, and terminates after at most one time unit.

$$(\Delta 1;\ x := b) \ \sqsubseteq\ < s,\ \begin{pmatrix} x.\mathbf{Latch}(s,\ s \wedge b) \\ \&\ f.\mathbf{Delay}(s) \end{pmatrix},\ f >$$

The simplest *PL* construct is sequential composition, the implementation of which is given by

Theorem 5.1 *If* $(\{s\} \cup \underline{l1} \cup \underline{w1}) \cap (\{f\} \cup \underline{l2} \cup \underline{w2}) = \emptyset$, *then*

$$\begin{array}{l} \texttt{SEQ} \\ \quad < s,\ C_1,\ h > \\ \quad < h,\ C_2,\ f > \\ \sqsubseteq \\ < s,\ \exists\, h \bullet \texttt{Merge}(C_1,\ C_2),\ f > \end{array}$$

which means that sequential composition of two processes can be implemented by merging their hardware implementations.

Similar theorems can be given for the conditional statement and iteration.

When discussing communication, we will for simplicity assume that there are only two input channels $ch?$, $c?$ and one output channel $dh!$.

The input process $ch?x$ becomes ready to receive a message from the channel ch by rising $ch.inrdy$ immediately after it starts. It observes the readiness of the partner residing at the other end of the channel ch via $ch.outrdy$, and signals the synchronization on ch by rising the flag $ch.synch$. Once the communication takes place, the variable x will be assigned the data received from the channel ch.

$$ch?x \ \sqsubseteq\ < s,\ \mathbf{Input}(s,\ ch,\ x,\ f),\ f >$$

where $\mathbf{Input}(s,\ ch,\ x,\ f)$ represents the network

$$\exists\, l \bullet \begin{pmatrix} l.\mathbf{Delay}((s \vee l) \wedge \neg ch.outrdy.\mathbf{In}) \\ \&\ f.\mathbf{Delay}((s \vee l) \wedge ch.outrdy.\mathbf{In}) \\ \&\ ch.inrdy.\mathbf{Out}(s \vee l) \\ \&\ ch.synch.\mathbf{Out}((s \vee l) \wedge ch.outrdy.\mathbf{In}) \\ \&\ x.\mathbf{Latch}(\ (s \vee l) \wedge ch.outrdy.\mathbf{In}, \\ \qquad\qquad (s \vee l) \wedge ch.outrdy.\mathbf{In} \wedge ch.\mathbf{In}) \\ \&\ \mathbf{Idle}(\{c?,\ dh!\}) \end{pmatrix}$$

and

$$\mathbf{Idle}(Chan) \mathrel{\widehat{=}} \begin{pmatrix} \&_{c? \in Chan}\ c.inrdy.\mathbf{Out}(0)\ \&\ c.synch.\mathbf{Out}(0) \\ \&_{d! \in Chan}\ d.outrdy.\mathbf{Out}(0) \end{pmatrix}$$

A corresponding theorem can be given for an output process and for `STOP` and the internal actions.

The alternation construct makes a choice on its input guards, and executes one of those statements whose guard becomes ready. Assume that $H_i(h_i, f_i) \mathrel{\widehat{=}} \exists \underline{v_i} \bullet f_i.\mathbf{Delay}(F_i)\ \&\ C_i$ for $i = 1, 2$. If H_1 and H_2 do not share output wires except latches and delay devices, then

$$
\begin{array}{l}
\texttt{ALT}\ t: \\
\quad b?x \rightarrow < h_1,\ H_1,\ f_1 > \\
\quad c?y \rightarrow < h_2,\ H_2,\ f_2 > \\
\sqsubseteq \\
< s,\ \exists\, s_1,\ s_2,\ l,\ h_1,\ h_2 \bullet \\
\left(\begin{array}{l}
\quad \mathbf{ALT}(s,\ b,\ c,\ s_1,\ s_2) \\
\left. \begin{array}{l} \\ \\ \\ \end{array} \right. \\
\ \&\ \texttt{Merge} \left\{ \begin{array}{l}
\mathbf{Input}(s_1,\ b,\ x,\ h_1),\ C_1, \\
\mathbf{Input}(s_2,\ c,\ y,\ h_2),\ C_2, \\
b.inrdy.\mathbf{Out}(s \vee l), \\
c.inrdy.\mathbf{Out}(s \vee l)
\end{array} \right. \\
\ \&\ f.\mathbf{Delay}(F_1 \vee F_2)
\end{array} \right), f >
\end{array}
$$

where

$$
\mathbf{ALT}(s,\ b,\ c,\ s_1,\ s_2) \mathrel{\widehat{=}}
\left(\begin{array}{l}
\quad l.\mathbf{Delay}((s \vee l) \wedge \neg b.outrdy.\mathbf{In} \wedge \neg c.outrdy.\mathbf{In}) \\
\ \&\ s_1.\mathbf{Comb}((s \vee l) \wedge b.outrdy.\mathbf{In}) \\
\ \&\ s_2.\mathbf{Comb}((s \vee l) \wedge c.outrdy.\mathbf{In} \,\&\, \neg s_1)
\end{array} \right)
$$

Similar constructs can be made for time-in (delay) and for time-out. In the parallel construct, the incoming start pulse activates all statement in parallel. As PL features only outermost parallelism, a parallel composition never activates a sequential successor. Hence, there is no need for the target circuit to generate a finish signal.

5.6 Related Work

This work builds on results published by other researchers [10, 13, 25, 62]. In particular, there is a strong relationship between our method and that used by Hoare in software compilation [27]. However, our method handles parallel composition and preserves true concurrency in the implementations. Additionally, a simple FPGA description language is introduced to mimic the behaviour of a synchronous circuit, which can also be defined in the same semantical model used for the source language.

In related work, David May at Inmos has shown that a communicating sequential process [26] can be implemented as a set of special-purpose computers (one per process), each with just sufficient resources and microcode [46]. Alain Martin at CalTech has developed a method of compiling a concurrent program into a circuit using semantic-preserving program transformations [44]. Ian Page at Oxford has developed a prototype compiler in the functional language SML

which converts an Occam-like language [32], somewhat more expressive than the one presented here, to a netlist. After further processing by vendor software the netlist can be loaded into Xilinx FPGA chips [62, 78]. However, the algebraic approach presented here offers the significant advantages of providing a provably correct compiling method, and it is also expected to support a wide range of design optimization strategies.

6 Mathematical Foundations

The preceding sections have illustrated a top-down approach to the design of a simple real-time process control system. For complex embedded and safety critical systems, it is even more important to decompose the project into such a progression of related phases. These should start with an investigation of the properties and behaviour of the process evolving within its environment, and an analysis of requirements for its optimal or satisfactory performance, or at least for its safety. From these is derived a specification of the electronic or program-controlled components of the system. The project then may pass through an appropriate series of design phases, culminating in a program expressed in a high level language. After translation into the machine code of the chosen computer, it is loaded into memory and executed at high speed by electronic circuitry. Additional application-specific hardware may be needed to embed the computer into the system which it controls. Each of these phases relies on theories, concepts, and notations particularly suited to that phase. But the conceptual gaps between the phases present at least an equal challenge, since reliability of the delivered system requires that all the gaps be closed. Reliability is achieved not just by testing, but by the quality of thought and meticulous care exercised by analysts, designers, programmers and engineers in all phases of the design.

This has been a description of an ideal that is rarely achieved in any field of engineering practice. Nevertheless, an ideal forms the best basis for long-term research into engineering technology. The goal of this research is to discover and formalize methods which reduce the risks and simplify the routines of the design task, and give fuller scope for the exercise of human skill and invention in meeting product requirements at low cost and in good time. The goal of this survey has been to convey an impression of the methods and intermediate results of just one such research project.

In principle, the transition between one design phase and the next is marked by delivery of a document, expressed in some more or less formal notation. Each phase starts with study and acceptance of the document produced by the previous phase; and ends with the delivery of another document, usually formulated at a lower level of abstraction, closer to the details of the eventual implementation. Each designer seeks high efficiency at low cost; but is constrained by an absolute obligation that the final document must be totally correct with respect to the initial document for this design phase. Thus the requirements must be faithfully reflected in the specification, the specification must be fully achieved by the design, the design must be correctly implemented by the program, the

program must be accurately translated to machine code, which must be reliably executed by the hardware. Different words are used in English to describe the correctness relation at each different level of design. But in principle, it would be very much simpler, and therefore safer, to use the same correctness relation in all cases; and that is the principle adopted in the **ProCoS** project. Correctness in all phases is identified with simple relationship of logical implication, denoted by $\Leftarrow$.

When the system is eventually delivered and put into service, all that really matters is that the actual hardware and software delivered should meet the overall requirements of the system. This is guaranteed by a simple mathematical property of the implementation relation: it is transitive. If P is implemented by Q and Q is implemented by R then P is implemented by R

$$\text{If } P \Leftarrow Q \text{ and } Q \Leftarrow R \text{ then } P \Leftarrow R$$

However long the chain of intermediate documents, if each document correctly implements the previous one, the overall requirements will be correctly implemented by the delivered product.

This is a very simple account of the design process, and the reason why it can validly be split into any number of phases. The account is highly abstract: in concrete reality, complications arise from the fact that each of the design documents is written in a different notation, adapted to a different conceptual framework at a different level of abstraction. For example, this paper has illustrated the use of set theory (Z) at the highest level of abstract requirements capture. The requirements are then transformed by the Duration Calculus to the more implementation-oriented framework of state machines, which in turn are transformed into the regular expressions and communication assertions of the specification language. After further merging and adjustment, these are translated with machine assistance to the notations of an occam-like programming language. Finally these are compiled to a transputer-like machine language. The machine itself is defined in the same high-level language, which can also be translated by the same techniques to hardware notations, implementable directly by implantation on silicon. How can we be certain that a document serving as an interface between one of these design phases and the next has been correctly understood (i.e. with the same meaning) by the specialists who produced it as a design and the different specialists who accepted it as a specification for the next phase? The utmost care and competence in each individual phase of design will be frustrated if bugs are allowed to congregate and breed in the interfaces between them.

The solution is to interpret every one of the documents in the chain as a direct or indirect description at an appropriate level of abstraction of the observable properties and behaviour of some system or class of system or component that exists (or could be made to exist) in the real world. These descriptions can be expressed most precisely in the language which science has already shown to be most effective in describing and reasoning about the real world, namely the language of mathematics. Such descriptions use identifiers as free variables to

stand for observations or measurements that could in principle be made of the real world system.

The Z notation used in section 2 gives the most direct examples in the introduction of the free variables *Heatreq, Flame,* and *Gas*, and their use in predicates to describe the actual and desired behaviour of the gas burner. The other notations used in the project can be regarded as more specialized and more abbreviated descriptions of the behaviour of the ultimate product. For example the behaviour of a computer program can be described in terms of the initial and final values of global variables before and after a typical run of the program; for reactive systems, this is accompanied by a trace of its intermediate interactions with its environment; and for time-critical systems, these traces must be timed.

The use of free variables to stand for observations or measurements of timed trajectories lies at the basis of all reasoning in science and engineering. Consider for example the simplest mechanical system, involving movement of a physical point, say a plotter pen or a ship or a projectile. A simple requirement may be that its motion remain steady during a certain interval after its start. Let x_t be its displacement on the x axis at time t, and let a be the desired velocity. Then within some desired interval the difference between the actual and desired position should, within the relevant period, always be less than some permitted tolerance

$$\mid x_t - at - x_0 \mid \leq 0.4$$

for all $t \in (3 \ldots 5)$. If we also want steady motion on the y-axis, this is stated separately

$$\mid y_t - bt - y_0 \mid \leq 0.4$$

for all $t \in (3 \ldots 5)$. The additional requirement is just conjoined by "and" to the original requirement. The use of conjunction to compose complex requirements from simple descriptions is a crucial advantage of the direct use of logical notations at the earliest stage of a design project.

In this example, the requirements formalize the permitted tolerances on the accuracy of implementation. Of course an implementation is permitted to achieve even greater accuracy. For example, suppose the behaviour of a particular implementation is described by

$$(x_t - at - x_0)^2 + (y_t - bt - y_0)^2 < 0.1$$

for all $t \leq 5$. This implies both of the requirements displayed above; consequently the implementation correctly fulfills its specification.

This notion of correctness is perfectly general. Suppose a design document P and a specification S use consistent naming conventions for variables to describe observations of the same class of system; and suppose that P logically implies S. This means that every observation of any system described by P is also described by S, and therefore satisfies S as a specification. Certainly, no observation of the system described by P can violate the specification S. That is the basis of our claim that the relationship of correct implementation is nothing other than

simple logical implication, the fundamental and familiar transitive relation that governs every single step of all valid scientific and mathematical reasoning. It should therefore be no surprise that it is also fundamental to all stages and phases of sound engineering design.

Implication is a relationship between predicates describing observations. In the example of a mechanical system, the description of the observations is direct, as it is in the Z specification of the top level of requirements. But most of the notations used in the later phases of design contain no direct reference to any kind of observation. In order to use implication as the criterion of correctness, it is essential to interpret all these intermediate notations as abbreviations for predicates describing observations at the appropriate level of granularity and abstraction.

The example we have just described might have been part of the requirement on a control system, formulated at the start of a design project. Our next example describes the actual behaviour of the ultimate components available for its implementation, right at the final phase of electronic circuit design and assembly. As in all descriptions of the real world, we choose to model it at a certain level of abstraction, which is only an approximation of reality. We have chosen a level which (subject to reasonable constraints) is known to be generally implementable in a wide range of technologies.

Let the variables x_t, y_t and w_t stand for voltages observable at time t on three distinct wires connected to an OR-gate (Fig. 4). The voltage takes one of two values, 0 standing for connection to ground and 1 standing for presence of electrical potential. The specification of the OR-gate is that the value of the output wire w is the greater of the values of the input wires x and y. This relationship cannot be guaranteed at all times, but only at regular intervals, at the end of each operational cycle of the circuit. For convenience, the unspecified duration of each cycle is taken to be the unit of time. The behaviour of the OR-gate is described as an equation

$$w_t = x_t \vee y_t$$

for all $t \in (0, 1, 2, \ldots)$, where the range of t is here and later restricted to the natural numbers. This means that observations can be made only at discrete intervals, understood to be on the rise of the relevant clock signal.

Another example of a hardware circuit is the Delay element. On each cycle of operation, the voltage at its output c is the same as the voltage at its input p on the previous cycle of operation

$$c_{t+1} = p_t$$

The clock event which advances t is communicated to the Delay element by the global clock input signal (marked C1 in Fig. 4). Note that, we are unable to predict the initial value c_0, obtained when the hardware is first switched on. The correctness of any circuit using this component must not depend on the initial value. The description reflects a certain physical non-determinism, which cannot be controlled at this level of abstraction. As in the case of engineering

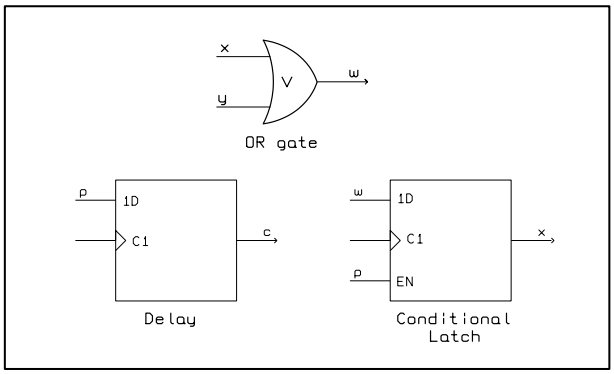


Fig. 4. Circuit elements.

tolerance, the specification must also allow for a range of possible outcomes; otherwise correctness will not be provable.

A useful variation of the Delay element is the Conditional Latch element (in engineering terms this is an edge-triggered flip-flop with a clock enable input). Here, the value of the output is changed only when a clock event occurs and the input control wire p is high; and then the new value of x is taken from the other input wire w. Otherwise the value of x remains unchanged. This behaviour is formally described

$$x_{t+1} = (w_t \lhd p_t \rhd x_t)$$

where $a \lhd b \rhd c$ is read as "a `if` b `else` c".

A pair of hardware components is assembled by connecting like-named wires of each of them, making sure that no two outputs become connected and that any cycle of connections is cut by a Delay. Electrical conduction ensures that the value observed at the input ends of each wire will be the same as that produced at the output end. As a result, the combined behaviour of an assembly of hardware components is described surprisingly but exactly by a conjunction of the descriptions of their separate behaviours. Imagine this done for the circuits of Fig. 4

In principle, the conjunction of the descriptions of all the hardware components of the system should imply the conjunction of all the requirements originally placed upon the system as a whole. In a simple system, this implication may be proved directly; otherwise it is proved through a series of intermediate design documents, perhaps including a program expressed in a high level language. A program also must be interpreted as an indirect description of its own behaviour when executed. We therefore need names to describe its observable features, and for reasons of our own we have chosen to reuse the names of the hardware wires.

Let p_t be an assertion true at just those times t when execution of the program starts, and let $c_{t'}$ be true at just those times t' at which it terminates. Let x_t be the value of a program variable x at time t, so $x_{t'}$ is the final value. With a slight simplification, the assignment statement

x := x OR y

can now be defined as an abbreviation for

$$p_t \Rightarrow \exists t' \geq t \bullet (x_{t'} = x_t \vee y_t) \quad \text{and} \quad c_{t'}$$

If the program starts at time t, then it stops at some later time t'; and at that time the final value of x is the disjunction of the initial values of x and y. Note that the execution delay $(t' - t)$ has been left deliberately unspecified. This gives design freedom for implementation in a variety of technologies, from instantaneous execution (achieved by compile time optimization) to the arbitrary finite delays that may be interposed by an operating system in execution of a timeshared program.

Now we confess why we have chosen the same names for the software variables as the hardware wires. It demonstrates immediately that our example hardware assembly is also a valid implementation of the software assignment: the description of one of them implies the description of the other. The proof is equally simple: take the software termination time as exactly one hardware cycle after the start. It is the translation of both hardware and software notations into a common framework of timed observations that permits a proof of the correctness of this design step as a simple logical implication, thereby closing the intellectual and notational gap between the levels of hardware and software.

Our example has been artificially simplified by use of exactly the same observation names at both levels. In general, it may be necessary to introduce a coordinate transformation to establish the link between them. The coordinate transformation is also a predicate, describing the relationship between the observations or measurements made at one level of abstraction and those made at the other. Its free variables therefore include those relevant at both levels; one of the levels can then be abstracted by quantification. A good example is the transformation $(\hat{\Psi})$ used in a compiler (4.4) to correlate the observations of initial and final values of the abstract program variables $x, \ldots, z$ with the concrete locations $\mathtt{M}[\Psi\, x], \ldots, \mathtt{M}[\Psi\, z]$, where their values may be observed in the store of the machine.

In principle, proof of correctness of a design step can always be achieved, as we have shown, by expanding the abbreviation of the relevant notations and constructing a proof in the predicate calculus. But for a large system this would be impossibly laborious. What we need is a useful collection of proven equations and other theorems expressed wholly in the abbreviated notations; it is these that should be used to calculate, manipulate, and transform the abbreviated formulae, without any temptation or need to expand them. For example, the power of matrix algebra lies in the collection of equational laws which express associative and distributive properties of the operators:

$$A \times (B \times C) = (A \times B) \times C$$
$$(A + B) \times C = (A \times C) + (B \times C)$$

The mathematician who proves these laws has to expand the abbreviations, into a confusing clutter of subscripts and sigmas

$$\sum_j A_{ij} (\sum_k B_{jk}\, C_{kl}) = \sum_k (\sum_j A_{ij} B_{jk})\, C_{kl}$$

But the engineer who uses the laws just does not want to know.

This survey article has given many examples of the informal use of algebraic reasoning within each design phase and in making the transition between them. For ultimate reliability, each of these transitions must be justified by explicit appeal to some valid mathematical law, preferably expressed as an equation, or possibly an inequation using logical implication. But how can the engineer be certain of the validity of the laws? That is a question that can be answered, not by an engineer but by a mathematician, who thereby makes an essential contribution to the success of the entire development and the reliability of the developed product.

This contribution starts with the design of a semantics for all the notations involved. Each formula is identified with a predicate, with free variables describing all the observations directly or indirectly relevant to its meaning. Now all that is needed is to prove the validity of all the laws within the rather tedious, low-level reasoning methods of the predicate calculus. The goal of the mathematician is to prove sufficient laws for all engineering purposes, so that never again will there be any need to expand the definitions in terms of predicates, or to use predicate level reasoning in carrying out the design task.

The final goal of research is to close all the gaps between all the conceptual levels and phases of a real-time engineering project. For ultimate security, it seems desirable to construct a single reference model of all the kinds of observations that may be relevant at any phase of the project. Then all the more specialized theories can be mathematically embedded in the single model, thereby ensuring identity of interpretation particularly at the interfaces between the phases and the theories.

The research reported in this survey concentrates on a very particular paradigm of parallel computing and synchronized communication, with its associated language and machine code, all based on the occam/transputer tradition. The case study used to illustrate the research is even more particular. So it is important to remind ourselves that the goals of any research project, especially a basic research action, are much more general than that; they are nothing less than an increase in scientific understanding and engineering skills, which can be communicated by education and enhanced by further research, with an accumulation of benefits into the indefinite future. These benefits are most highly concentrated in the formulation of the mathematical foundations within which a range of related theories can be embedded, and the proof of the mathematical laws as a secure foundation for engineering technology. It is these that we will pass on to the industrial beneficiaries of the research, so that they can be applied

to completely different projects, with different languages, machines, and requirements. It is only by aiming at such generality that we can justify the support given to our research, and all the work that we have put into it.

Acknowledgements. Thanks to Lennart Andersson, Kirsten M. Hansen, Zhiming Liu, Paulo Masiero, Jens U. Skakkebæk, E. V. Sørensen and Chaochen Zhou for collaboration on requirements and design. For collaboration and comments on compilation Bettina Buth, Karl-Heinz Buth, Burghard von Karger, Yassine Lakhneche and Hans Langmaack. For the hardware part Jonathan Bowen, Wayne Luk, Ian Page and in particular Zheng Jianping are thanked for comments and collaboration.

A special thanks to Jonathan Bowen for revising the bibliography.

References

1. R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 67–93, 1990.
2. J. C. M. Baeten and P. Weijland. *Process Algebra.* Cambridge University Press, 1980.
3. F. L. Bauer et al. *The Munich Project CIP, Volume II: The Transformation System CIP-S*, volume 292 of *LNCS*. Springer-Verlag, 1987.
4. W. R. Bevier, W. A. Hunt, Jr., and W. D. Young. Towards verified execution environments. Technical Report 5, Computational Logic, Inc., Austin, Texas, USA, February 1987.
5. D. Bjørner, H. Langmaack, and C. A. R. Hoare. ProCoS I final deliverable. ProCoS Technical Report [ID/DTH DB 13/1], Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, January 1993.
6. Egon Börger, Igor Durdanovic, and Dean Rosenzweig. Occam: Specification and compiler correctness — Part I: The primary model. unpublished note.
7. A. Bouajjani, R. Echahed, and R. Robbana. Verifying invariance properties of timed systems with duration variables. In *these proceedings*, 1994.
8. J. P. Bowen, editor. *Towards Verified Systems.* Real-Time Safety Critical Systems Series. Elsevier, in press.
9. J. P. Bowen, M. Fränzle, E.-R. Olderog, and A. P. Ravn. Developing correct systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 176–189. IEEE Computer Society Press, June 1993.
10. J. P. Bowen, He Jifeng, and I. Page. Hardware compilation. In Bowen [8], chapter 10, pages 193–207.
11. J. P. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.
12. S. Brien, M. Engel, He Jifeng, A. P. Ravn, and H. Rischel. Z model for Duration Calculus. ProCoS Technical Report [OU HJF 12/2], Oxford University Computing Laboratory, UK, September 1993.
13. G. M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, Workshops in Computing, pages 120–131. Springer-Verlag, 1991.

14. M. Broy. Specification and top-down design of distributed systems. *J. Comput. System Sci.*, 34:236–265, 1987.
15. R. H. Campbell and A. N. Habermann. The specification of process synchronisation by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems, International Symposium, Rocquencourt 1974*, volume 16 of *LNCS*. Springer-Verlag, 1974.
16. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
17. M. Engel et al. A formal approach to computer systems requirements documentation. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 452–474, 1993.
18. M. Fränzle and M. Müller-Olm. Towards provably correct code generation for a hard real-time programming language. In P. A. Fritzson, editor, *Compiler Construction '94, 5th International Conference, Edinburgh, UK*, volume 786 of *LNCS*, pages 294–308, 1994.
19. M. Fränzle and B. von Karger. Proposal for a programming language core for ProCoS II. ProCoS Technical Report [Kiel MF 11/3], Christian-Albrechts-Universität Kiel, Germany, August 1993.
20. C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, May 1990.
21. D. I. Good and W. D. Young. Mathematical methods for digital system development. In S. Prehn and W. J. Toetenel, editors, *VDM '91, Formal Software Development Methods: Volume 2*, volume 552 of *LNCS*, pages 406–430, 1991.
22. M. R. Hansen and Zhou Chaochen. Semantics and completeness of the Duration Calculus. In J. W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 209–225, 1992.
23. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
24. He Jifeng and J. P. Bowen. Time interval semantics and implementation of a real-time programming language. In *Proc. 4th Euromicro Workshop on Real-Time Systems*, pages 110–115. IEEE Computer Society Press, 1992.
25. He Jifeng, I. Page, and J. P. Bowen. Towards a provably correct hardware implementation of Occam. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *LNCS*, pages 214–225. Springer-Verlag, 1993.
26. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.
27. C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C. C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, Workshops in Computer Science, pages 33–48. Springer-Verlag, 1991.
28. C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–687, 1987.
29. C. A. R. Hoare, He Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.
30. J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *PARLE '89, Parallel Architectures and Languages Europe: Volume II*, volume 366 of *LNCS*, pages 424–441. Springer, 1989.
31. R. Inal. Modular specification of real-time systems. In *Proc. 6th Euromicro Workshop on Real-Time Systems*, pages 16–21. IEEE Computer Society Press, 1994.

32. INMOS Limited. *Occam 2 Reference Manual*. Prentice Hall, 1988.
33. INMOS limited. *Transputer Instruction Set: A Compiler Writer's Guide*. Prentice Hall, first edition, 1988.
34. M. S. Jaffe, N. G. Leveson, M. P. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. Software Engineering*, 17(3):241–258, March 1991.
35. J. J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 277–201, 1990.
36. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.
37. B. Krieg-Brückner. Algebraic specification and functionals for transformational program and meta program development. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT '89: Volume 2*, volume 352 of *LNCS*, 1989.
38. L. Lamport. The temporal logic of actions. Technical report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, USA, 25 December 1991.
39. L. Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 77–102, 1993.
40. N. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, February 1991.
41. N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th PODC*, pages 137–151, 1987.
42. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
43. A. J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 244–259, 1990.
44. A. J. Martin. Programming in VLSI: From communicating processes into delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 1. Addison-Wesley, 1990.
45. P. C. Masiero, A. P. Ravn, and H. Rischel. Refinement of real-time specifications. ProCoS Technical Report [ID/DTH PCM 1/1], Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, July 1993.
46. D. May. Occam and the Transputer. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 2. Addison-Wesley, 1990.
47. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. Schwarz, editor, *Proc. Symp. Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.
48. R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1989.
49. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
50. C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, 1988.
51. C. C. Morgan. *Programming From Specifications*. Prentice Hall International Series in Computer Science, 1990.

52. F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Proc. ACM Symp. Principles of Programming Languages, Boston, Mass.*, pages 144–152, 1973.
53. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
54. B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.
55. M. Müller-Olm. On translation of TimedPL and capture of machine instruction timing. ProCoS Technical Report [Kiel MMO 6/2], Christian-Albrechts-Universität Kiel, Germany, August 1993.
56. Markus Müller-Olm. A new proposal for TimedPL's semantics. ProCoS Technical Report Kiel MMO 10/1, Christian-Albrechts-Universität Kiel, Germany, May 1994.
57. E.-R. Olderog. *Nets, Terms and Formulas.* Cambridge University Press, 1991.
58. E.-R. Olderog. Towards a design calculus for communicating programs. In J. C. M. Baeten and J. F. Groote, editors, *Proc. CONCUR '91*, volume 527 of *LNCS*, pages 61–72, 1991.
59. E.-R. Olderog. Interfaces between languages for communicating systems. In W. Kuich, editor, *Automata, Languages and Programming*, volume 623 of *LNCS*, 1992.
60. E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 90–104, 1993.
61. E.-R. Olderog, S. Rössig, J. Sander, and M. Schenke. ProCoS at Oldenburg: The interface between specification language and Occam-like programming language. Technical Report Bericht 3/92, Univ. Oldenburg, Fachbereich Informatik, Germany, 1992.
62. I. Page and W. Luk. Compiling Occam into field programmable gate arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, pages 271–284, 15 Harcourt Way, Abingdon OX14 1NV, UK, 1991. Abingdon EE&CS Books.
63. D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Software Engineering*, 12(2):251–257, February 1986.
64. D. L. Parnas and J. Madey. Functional documentation for computer systems engineering (version 2). Technical Report CRL 237, TRIO, McMaster University, Hamilton, Canada, September 1991.
65. A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems (extended abstract). In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 331 of *LNCS*, pages 84–98. Springer, 1988.
66. A. P. Ravn and H. Rischel. Requirements capture for embedded real-time systems. In *Proc. IMACS-MCTS'91 Symp. on Modelling and Control of Technological Systems*, volume 2, pages 147–152. IMACS, May 1991.
67. A. P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Trans. Software Engineering*, 19(1):41–55, January 1993.
68. A. W. Roscoe and C. A. R. Hoare. Laws of Occam programming. *Theoretical Computer Science*, 60:177–229, 1988.
69. S. Rössig and M. Schenke. Specification and stepwise development of communicating systems. In S. Prehn and W. J. Toetenel, editors, *VDM '91, Formal Software Development Methods: Volume 1*, volume 551 of *LNCS*, pages 149–163, 1991.

70. M. Schenke. Specification and transformation of reactive systems with time restrictions and concurrency. In *these proceedings*, 1994.
71. J. U. Skakkebæk, A. P. Ravn, H. Rischel, and Zhou Chaochen. Specification of embedded, real-time systems. In *Proc. 4th Euromicro Workshop on Real-Time Systems*, pages 116–121. IEEE Computer Society Press, 1992.
72. J. U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In *these proceedings*, 1994.
73. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
74. J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–245, 1981.
75. Y. Venema. A modal logic for chopping intervals. *J. Logic of Computation*, 1(4):453–476, 1991.
76. A. Wikström. *Functional Programming using Standard ML.* Prentice Hall International Series in Computer Science, first edition, 1987.
77. M. W. Wilkes and J. B. Stringer. Micro-programming and the design of the control circuits in an electronic digital computer. *Proc. Cambridge Phil. Soc.*, 49:230–238, 1953. also *Annals of Hist. Comp.* **8**, 2 (1986) 121–126.
78. Xilinx Inc. The programmable gate array data book. Technical report, Xilinx Inc., San Jose, California, USA, 1991.
79. Zhiming Liu, A. P. Ravn, E. V. Sørensen, and Zhou Chaochen. Towards a calculus of systems dependability. *High Integrity Systems*, 1(1):49–75, January 1994.
80. Zhou Chaochen. Duration Calculi: An overview. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal Methods in Programming and their Application*, volume 735 of *LNCS*, pages 256–266, 1993.
81. Zhou Chaochen, M. R. Hansen, and P. Sestoft. Decidability results for Duration Calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proc. STACS 93*, volume 665 of *LNCS*, pages 58–68, 1993.
82. Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5), December 1991.
83. Zhou Chaochen, A. P. Ravn, and M. R. Hansen. An extended Duration Calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 36–59, 1993.
84. J. Zwiers. *Compositionality, Concurrency, and Partial Correctness: Proof Theories for Networks of Processes and their Relationship*, volume 321 of *LNCS*. Springer-Verlag, 1989.