PROVABLY CORRECT HARDWARE COMPILATION USING TIMING DIAGRAMS

Michael Schenke

Universität Oldenburg, Postfach 2503, D-26111 Oldenburg

Michael.Schenke@Informatik.Uni-Oldenburg.DE

Michael Dossis

The author has worked on the OMI/HORN ESPRIT project and is now an independent consultant.

Abstract In this paper we present a framework within which hardware implementations are proven correct from specifications given in an *OCCAM-like language* called *Handel* by the use of a robust set of mathematical transformational laws. The semantic basis for Handel and its hardware implementations are simple functions of time which are called *timing diagrams*. This basis allows us to denote the abstract properties of the Handel programs and hence the implementations in a modal logic, called *Duration Calculus*. The semantic treatment by one model for all three levels including the abstract properties, Handel and the level of gates, is one of the outstanding features of our approach. The delicate mathematical model which is used is able to cope with the complex form of parallelism used in Handel and with the detailed treatment of the relation between parallelism and timing. An immediate benefit of this approach is that Handel is a language already in use by hardware designers for specification in addition to the potential for easy integration of the proposed transformations with existing hardware compilation systems.

*This work was partially funded by the EPSRC in the project "Design and Transformational Implementation of Provably Correct Real Time Systems" and by the Leibniz Programme of the German Research Council (DFG) under Ol 98/1-1. A more complete version of the paper including more examples, proofs and drawings can be found at <ftp://theoretica.Informatik.Uni-Oldenburg.DE/pub/Papers/msmd97-p1.ps.gz> and ...-p2.ps.gz

The introduction of so-called *quick* operators utilises the non-determinism of implementations and enables to realise implementations of sequential and parallel Handel operations, which are improved over conventional code optimisation techniques.

Keywords: Semantic foundations, Real-time aspects, Parallel systems, Hardware compilation.

1. INTRODUCTION

The current availability of specification languages with higher levels of abstraction begins to blur the difference in the engineering practices for the design of software and hardware. However, for software implementations the semantic gaps between abstract specification code and its actual implementation seem to be smaller than for hardware implementations, if the many levels of abstraction in hardware are taken into account.

The eventual hardware model generated by almost all the current automatic or semi-automatic synthesis methods is usually given in the form of gate-netlists. The process of attempting to bridge the enormous abstraction gap between the specification and its direct implementation in hardware netlists is driven in practice by design optimisations. These optimisations are based on a very diverse range of implementation constraints. The optimisations introduce a large set of additional semantics in the delivered hardware netlist, and thus they generate different semantic spaces between the specification and implementation. This does not cause any significant problems, particularly when the details of the target architecture are well-known in advance and they are fixed for a large number of applications. Experience indicates, however, that the methodology has its problems, when formal verification of the coherence between the specification and implementation, as well as of the actual transformation taking place during automatic hardware synthesis, is required. The difficulty to apply verification then, is due to the semantic inconsistencies caused by the heuristic optimisation models.

An Occam-based notation has been used by the Oxford research group for hardware specification and synthesis. This computer language is called Handel. A set of derivatives for its hardware compiler have been developed, used and currently extended by member researchers or the industrial engineers who collaborate with the hardware compilation group in Oxford.

The bulk of the research work with Handel addresses the compilation process from Occam-like programs to the produced hardware netlists, which are normally prototyped on FPGA devices. FPGAs

(Field-Programmable Gate Arrays) utilise a systolic-type array of configurable logic blocks (CLBs) and a matrix of erasable interconnect nodes determining the connections between them. This, in conjunction with the one-to-one translation of the spatial and temporal attributes of the Handel programs into the hardware architecture, allows for total control of the intended architecture by the Handel designer.

In addition, the distributed mechanism used for the synchronous hardware designs generated by the Handel compiler, combined with the systolic-type architecture of the targeted devices, improve significantly the scalability of the Handel designs. Furthermore, the compilation approach is based on a reduction of the Occam programs used for specification into their normal form. In combination with the transformational algebra CSP ?, this supports the Handel paradigm. It makes the synthesis approach fertile for further exploitation and enhancement using formal techniques, such as the one presented in this paper.

2. PREVIOUS WORK AND CURRENT APPROACH

Programs written in Handel ? specify hardware implementations. Although Handel programs resemble *occam* programs ? a major difference concerns the use of the parallel operator: Like *occam*, Handel allows the possibility of a point to point communication, but additionally in the version we are using an asynchronous communication via access to variables can take place. The only restriction on this sort of communication is that a simultaneous write access of two different processes will lead to an undetermined value.

The compiler has been constructed in such a way that certain algebraic transformation and implementation rules are fulfilled. It is the ambition of the authors' work to supply a transformational framework in which an implementation relation can be defined between Handel programs and hardware which makes these rules true, thus proving the implementation calculus used in the Handel compiler correct. Handel programs will have as much timing nondeterminism as possible. This will allow several different implementations. Step by step this nondeterminism will be reduced, until a level is reached in which no timing nondeterminism is left: the so-called *intermediate terms*. From there obvious implementation rules lead directly to hardware implementations. The validity of these rules, on which the compiler relies, is then proven in our framework. Furthermore on top of Handel we set an interval temporal logic, the duration calculus (DC) ?; ?, which allows us to prove properties of specifications and implementations. All levels are based on the same

semantic model: so-called *timing diagrams*. This way is mathematically the most complex, but also the most satisfying one because all possible inconsistencies are avoided, which appear when one semantic model is changed to another one. We get the following picture of our approach:

Syntactic transformation and its semantic foundations

Another novelty of our approach is the treatment of a parallel operator in a denotational way with synchronous and asynchronous communication in a many read / many write fashion. Some implementation examples, kept reasonably small for the purpose of this paper but related to real hardware implementations, are also included.

The current approach achieves the necessary degree of delicacy in the mathematical analysis of the Handel language:

- 1.) It is very difficult to find a denotational semantics for a language which has a complex parallelism like the one in Handel, which combines the OCCAM-like point to point communication with communication via shared variables.
- 2.) If timing nondeterminism is retained and decisions for a parallel or sequential implementation of the specifications are employed at a later stage during the automatic design synthesis transformations, then as a consequence the interdependencies between the timing of the parallel and sequential composition inevitably become very complex.
- 3.) Any approach where the semantics of all abstraction levels is expressed in a uniform model leads inevitably to a certain degree of complexity in the description of the operators.

An approach similar to the one proposed has been taken in the ProCoS project [?], where hardware verification was only one issue among many other aspects. However, the formal analysis of Handel transformations presented in this paper was not an issue in that project. Transformation methods based on real-time refinement have attracted a considerable amount of interest. However, the present paper is among the first in which real-time refinement is used for hardware verification in a far-reaching approach from the very abstract requirement language level (DC) down to hardware implementation. Previous work using

transformation algebra was pursued by He Jifeng et al., e.g. [1]. The broader framework of our work is given by [2, 3]. As a complementary approach one might regard self-timed circuit implementation techniques as in [4, 5, 6].

3. THE MODEL

Our models will be based on *timing diagrams*. Intuitively these are functions which attribute to each point t of time the state the system is in at t . They are used to give a semantic foundation to the DC, to Handel and to hardware. In this paper we are using a discrete version. The semantics of a Handel process P is given as a transformation on sets of timing diagrams. The meaning of this is that we describe how P affects a system which has evolved already according to some timing diagram. For reasons which will be explained later our semantics is called *Quickstep-Semantics*. We have to introduce some general notation:

A.) In the semantics presentation we assume a fixed set of channels $Chan$ and a disjoint set of program variables Var . Additionally there are three distinguished variables W, R, C which are not contained in $Chan \cup Var$. Intuitively W, R describe the sets of all variables which are written to or read from at some point in time, C provides information about the readiness of the channels for communication. By this we remain close to the tradition of CSP [7], the scientific predecessor of *occam*, which requires that channel communications are dealt with by means of so-called refusal or ready sets, C will contain, however, slightly more information than just a ready set.

Then we introduce the following notations and general assumptions:

- Each program variable v is equipped with a type $ty(v)$, W, R are of type $\mathcal{P}(Var)$ and C is of a special type, which contains sets of elements of the form $ch?v$ or $ch!e$.
- The elements of the set $Var \cup \{W, R, C\}$ are called observables.
- A *state*, typically called σ , is a mapping from the set of all observables where the image of each observable obs is contained in $ty(obs)$.
- For a logical term e or a logical formula b over the variables in the process, let $\llbracket e \rrbracket(\sigma)$ and $\llbracket b \rrbracket(\sigma)$ be their values in the state σ as usually defined.

B.) In the following we shall define functions of time. As domains of such functions we need the following sets of closed intervals:

$$\begin{aligned} Intv_{fin} &= \{ \{0, \dots, e\} \mid e \in \mathbb{N}_0 \} \\ Intv &= Intv_{fin} \cup \{\mathbb{N}_0\} \end{aligned}$$

Our time domain is $Time \stackrel{\text{def}}{=} \mathbb{N}_0$.

We also remind the reader of some mathematical standard notation:

The functional composition $f \circ g$ is defined by $f \circ g(x) \stackrel{\text{def}}{=} f(g(x))$.

The repeated application of composition f^n is defined by $f^n(x) \stackrel{\text{def}}{=} f(\dots f(x))$, where f has been applied exactly n times.

C.) The following set of notations are connected with timing diagrams, on which our model is built:

- For a given set Obs of observables and $I \in Intv$ a *timing diagram* over I is a mapping, typically called td , with domain $Obs \times I$ such that $\forall obs \in Obs, t \in I : td(obs, t) \in ty(obs)$.

Throughout the paper we shall consider only timing diagrams with the property that $td(W, 0)$ is equal to the set of all variables ($td(W, 0) = Var(\alpha)$ in the notation of section 5.). This means that each variable must have been set to some initial value.

- Note that for each fixed $t \in I$ the function $td(\cdot, t)$ is a state. Here $td(\cdot, t)$ is the function which attributes to each variable v its value $td(v, t)$ at time t .
- Let $\ell(td)$, the *length of the timing diagram* td over I , be the length of the interval I , where the arithmetic on $\mathbb{N}_0 \cup \{\infty\}$ is defined as usual, and let $I(td) = I$ and $Obs(td) = Obs$. These are projection functions which extract either component from the domain of td .
- Let $TD(Obs)$ denote the set of all timing diagrams over Obs and over any $I \in Intv$.
- Let $TD_{fin}(Obs), TD_{inf}(Obs)$ denote the set of all timing diagrams which are defined for finite and for infinite intervals, respectively.

4. DISCRETE DURATION CALCULUS

As an assertional language we are using a discrete version of the duration calculus. From the hardware design point of view this is the form we shall need, because we assume the existence of a global synchronous clock scheme. For more detailed studies of DC the reader may refer to ?. That text introduces its continuous version.

Semantically, duration formulae F are evaluated in a given interpretation $\mathcal{I}$ and a given time interval $\{b, \dots, e\} \subseteq Time$. Such an interval may also be empty. For us interpretations are timing diagrams (modulo W, R, C).

State assertions P are obtained by applying propositional connectives like ' $\wedge$ ', ' $\vee$ ' or ' $\neg$ ' to elementary assertions of the form $var = val$ or the expression ' $true$ '. For a given interpretation $\mathcal{I}$, state assertions denote

functions

$$P_{\mathcal{I}} : Time \rightarrow \{0, 1\}$$

For example $P_{\mathcal{I}}(t) = (v = true \wedge w = false)(t)$ is 1, iff v is true and w is false at time t in the interpretation $\mathcal{I}$.

Then the basic syntax of duration formulae apart from propositional connectives is as follows:

- **Duration:** $\int P = k$ expresses that the *duration* of the state assertion P in $\{b, \dots, e\}$ is k . Semantically, duration is the measurement $\sum_b^e P_{\mathcal{I}}(t)$.
- **Chop:** The composite duration formula $F_1 ; F_2$ holds in $\{b, \dots, e\}$ if this interval can be divided into an initial subinterval where F_1 holds and an adjacent final subinterval where F_2 holds.
- **Next:** This operator is typical for discrete time calculi. It cannot be used in a continuous setting. A duration formula $\bigcirc P$ holds at time b , iff at time $b + 1$ the duration formula P holds, i.e. $\int P = 1$ in the one point interval $\{b + 1\}$.

Besides this basic syntax various abbreviations can be defined like *length* (ℓ) of an interval, *sometime* ($\Diamond F$), *always* ($\Box F$). They can be used as is common in temporal logic and explained in ?. Furthermore $\lceil P \rceil$ means $\int P = \ell \wedge \ell > 0$ and $\bigcirc var = exp$ means that in the next point of time the variable var gets the value determined by the expression exp .

A duration formula F holds in an interpretation $\mathcal{I}$ if F evaluates to true in $\mathcal{I}$ and every interval of the form $\{0, \dots, t\}$ with $t \in Time$. As usual in logic we can identify a formula F with the set of all interpretations which make F true.

5. THE SYNTAX OF HANDEL

An *alphabet* $\alpha = Chan(\alpha) \cup Var(\alpha)$ is associated with each process P . This alphabet must include all channel and variable names which occur in P with $Chan(\alpha) \cap Var(\alpha) = \emptyset$. Let $Chan(\alpha) = In(\alpha) \cup Out(\alpha)$, the sets of input and output channels and $Var(\alpha) = Wr(\alpha) \cup Re(\alpha)$, the sets of write and read variables. This does not exclude bidirectional wires or read-write variables.

The set of Handel processes P is generated by the following grammar:

$$P ::= \text{SKIP} \mid \text{STOP} \mid \text{DELAY} \mid ch?v \mid ch!exp \mid x_1, \dots, x_n := exp_1, \dots, exp_n \\ \text{SEQ}[P, P] \mid \text{IF}(cond, P, P) \mid \text{WHILE}(cond, P) \mid \text{PAR}[P, P]$$

Parallel processes:. A process $\text{PAR}[P_1, \dots, P_n] = \text{PAR}[P_1, \text{PAR}[P_2, \dots, P_n] \dots]$ consists of $n > 0$ processes where the interface of the subcomponents P_i is given by α_i . PAR is an associative operator in our setting. We distinguish two sorts of channels: *external channels* for which no internal communication partner exists: $\exists^1 i : ch \in \text{Chan}(\alpha_i)$ and *internal channels*, if there are exactly two i with $ch \in \text{Chan}(\alpha_i)$.

From now on we assume that all channels appearing in the interfaces of the sequential components are internal or external. A parallel process can be observed at its external channels and its variables. We shall assume the existence of external channels only in our inductive definitions; compilable Handel programs do not have them.

Intermediate terms:. In order to cope with the problems of a proper implementation of the timing, we shall need some additional syntactical constructs whose behaviour resembles the behaviour of the already known constructs but does not allow timing non-determinism. The reason for introducing intermediate terms will be made clear in the section on the transformation of Handel programs. The present section is only meant to exhibit the necessary syntax.

The set of intermediate terms is generated by the following grammar:

$$P ::= op \mid ch??v \mid ch!!e \mid x_1, \dots, x_n ::= exp_1, \dots, exp_n \mid \text{if}(cond, P, P) \mid \text{while}(cond, P)$$

where op is an arbitrary Handel operator, and the newly introduced operators are called *slow operators*.

6. THE SEMANTICS OF HANDEL

6.1 OPERATORS ON TIMING DIAGRAMS

In order to describe the semantics of a language which has an elaborate parallel operator like Handel and includes timing, it is not surprising that we need some auxiliary operators. It is rather more surprising that we need just so few.

- By $td[n / v := exp]$ we denote the timing diagram td after a *timing diagram update* at time n of the observable v by $\llbracket exp \rrbracket(td(., n-1))$, the value of the expression exp one time unit before.
- The delay operator del extends a given timing diagram by one time unit, retaining the last state. On infinite timing diagrams the delay has no effect.

$$del(td)(v, t) \stackrel{\text{def}}{=} \begin{cases} td(v, t) & \text{for } 0 \leq t \leq \ell \\ td[\ell / W, R, C := \emptyset, \emptyset, td(C, \ell)](v, \ell) & \text{for } t = \ell + 1 \end{cases}$$

where $\ell = \ell(td)$. At the newly created point of time $\ell + 1$ no write or read has taken place so far, so W, R are empty. The willingness for a communication has not changed, so C remains the same.

- In order to allow for a quicker update by an assignment than only at the end of a given timing diagram td , we give a definition for the earliest point of time where a process P writing on a variable set X and reading from a variable set Y can interfere with td . P must not interfere earlier than one of the variables from X or Y has been written upon and not earlier than one of the variables from X has been read for the last time:

$$time(td, X, Y) \stackrel{\text{def}}{=} \max(\{1+n \mid td(W, n) \cap (X \cup Y) \neq \emptyset\} \cup \{1+n \mid td(R, n) \cap X \neq \emptyset\})$$

An earlier interference of P would have disastrous consequences.

- The following *at* operator is a little more elaborate than the pure update. It is basically the semantic equivalent of a program assignment. So in contrast to the update, a changed variable retains its value until the next explicit update. Formally, *at* is defined by a repeated use of the timing diagram update. Also the distinguished observables W, R are updated:

$$at[n / v := exp](td) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{for } \ell(td) < n \vee n < time(td, vars(v), vars(exp)) \\ td[i / v := exp \mid i = n, \dots, \ell(td)] \\ [n / W := W \cup vars(v)] \\ [n - 1 / R := R \cup vars(exp)] & \text{otherwise} \end{cases}$$

The first line maintains that the timing diagram cannot be updated if it is not long enough or if the update is ruled out by the *time* function. The notation in the other three lines describes an update of variable v at time n by the expression exp . Its effect remains noticeable at all times in $n, \dots, \ell(td)$, expressed in the second line. The program variable v is written upon at time n , so W must be updated at n (third line). The program variables from exp are read at time $n - 1$, so R must be updated at $n - 1$ (fourth line).

In the semantic model we may want to use updates of the sets W, R, C . For W, R we use the same syntax, for example for W :

$$at[n / W := W \cup \{v\}](td) \stackrel{\text{def}}{=} td[i / W := W \cup \{v\} \mid i = n, \dots, \ell(td)]$$

We have to say how C should be treated by *at*:

$$\begin{aligned} at[n / C := C \cup \{ch?v\}](td) &\stackrel{\text{def}}{=} td[i / C := C \cup \{ch?v\} \mid i = n, \dots, \ell(td)] \\ at[n / C := C \setminus \{ch?v\}](td) &\stackrel{\text{def}}{=} td[i / C := C \setminus \{ch?v\} \mid i = n, \dots, \ell(td)] \\ &\quad [n / W := W \cup \{ch?\}] \end{aligned}$$

The first line is used when channel ch becomes ready for a communication at time n , the other two express a communication at n . Then readiness is withdrawn (line two) and the notation $ch?$ will indicate that the communication has taken place, similar to the way $v \in W$ means that a communication on v has taken place. By ' $\setminus$ ' we denote the set theoretic difference. Since a communication via ch can take place only after the last communication via ch , we have to make our definition more precise by

$$\begin{aligned} at[n / C := C \cup \{ch?v\}](td) &\stackrel{\text{def}}{=} \emptyset \\ \text{if } \exists m : n < m \leq \ell(td) \wedge \{ch?, ch!\} \cap td(W, m) &\neq \emptyset \end{aligned}$$

The expression $\{ch?, ch!\} \cap td(W, m) \neq \emptyset$ means that a communication via ch has taken place at m . New readiness for a communication is impossible at any earlier point of time (called here n). Note that we claim strict inequality between n and m . This corresponds to the idea that a channel which has just been communicated upon is immediately ready for a new communication. The operator for output is defined in an equivalent fashion.

The following abbreviation $at[n_1 / v_1 := exp_1, \dots, n_m / v_m := exp_m]$ marks a repeated use of the at operator at various points of time $n_1 < \dots < n_m$. This format will be called the *at-format* of a mapping. The following lemma establishes a somewhat weaker property than commutativity. It allows us to consider the *at-format* as some sort of normal form for mappings on timing diagrams.

Lemma 6..1 (*comassi*)

For $m \leq n$

$$at[m / v_m := exp_m] \circ at[n / v_n := exp_n] \in \{\emptyset, at[m / v_m := exp_m, n / v_n := exp_n]\}$$

So either the two mappings commute or their composition amounts to the empty mapping. $\square$

A similar law governs the interplay between the *del* and *at* operators:

Lemma 6..2 (*delassi*)

For $0 < n \in \mathbb{N}_0$ it is

$$del \circ at[n / v_n := exp_n](td) \in \{\emptyset, at[n / v_n := exp_n] \circ del(td)\}$$

So either the two mappings commute or the image of their composition is the empty set. On td the mappings do not commute, iff $n = \ell(td) + 1$. $\square$

In the following we shall inductively construct the mapping which Handel programs induce on a timing diagram td . Basically this happens

by repeated applications of the *at* and *del* operators. The *del* operator is used for extending the given diagram in the most trivial way by one time unit (if this is necessary) after the *at* operator modifications of the state take place.

6.2 SEMANTICS OF THE HANDEL OPERATORS

Now we are ready to describe the semantics of the Handel operators. They will be conceived as operators on sets of timing diagrams, which are given in the *at*-format, preceded by sufficiently many applications of the *del* function. The definition will be given on single diagrams and extended to sets TD of timing diagrams. The semantics will be given in a continuation style, saying how some process will act on some history. Therefore infinite timing diagrams will not be changed by the Handel operators. The slow operators are only mixed terms, whereas the quick ones, which are a part of Handel, can change the past, so it is justified to call them “quick”. This has led to the name “Quickstep-Semantics”.

For the slow assignment $ass \equiv$

$$v ::= exp$$

let the semantics be defined by

$$\llbracket ass \rrbracket(td) = at[\ell(td) + 1 / v ::= exp] \circ del(td)$$

So the update can take place only at the end of the timing diagram after the extension by one unit. In contrast to the quick assignment there is hence no non-determinism when the update takes place.

For the quick assignment $ass \equiv$

$$v ::= exp$$

let the semantics be defined by

$$\llbracket ass \rrbracket(td) = \bigcup_{i=0}^{\ell(td)} at[i / v ::= exp](td) \cup \llbracket v ::= exp \rrbracket(td)$$

So the update can take place at any point of time in the timing diagram or, after an extension, one time unit after the end of td . Note that in practice most of the $at(i / v ::= exp)$ contribute nothing to the semantics, because an update at time i is not allowed due to the first line in the definition of *at*.

$$\begin{aligned}
\llbracket \text{SKIP} \rrbracket(td) &= td \\
\llbracket \text{DELAY} \rrbracket(td) &= \text{del}(td) \\
\llbracket \text{SEQ}[P_1, P_2] \rrbracket(td) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(td))
\end{aligned}$$

For td let $\llbracket \text{STOP} \rrbracket(td)$ be the unique extension td' of td with

$$\ell(td') = \infty \wedge \forall obs, i > \ell(td) : td'(obs, i) = \text{del}^{i-\ell(td)}(td(obs, i))$$

So after $\llbracket \text{STOP} \rrbracket(td)$ no further change of the diagram is possible.

Semantically, each of the channel communications is, apart from the communication sets, treated almost like half an assignment, so that later their parallel composition amounts to a full assignment. For quick operators this is

$$\begin{aligned}
\llbracket ch?v \rrbracket(td) &= \bigcup_{m \in ty(v)} \bigcup_{i=0}^{\ell(td)} \left(\bigcup_{j=i+1}^{\infty} \text{at}[j / C, v := C \setminus \{ch?v\}, m] \circ \right. \\
&\quad \left. \text{at}[i / C := C \cup \{ch?v\}] \circ \text{del}^{j-\ell(td)} \cup \right. \\
&\quad \left. \llbracket \text{STOP} \rrbracket \circ \text{at}[i / C := C \cup \{ch?v\}] \right)(td) \\
\llbracket ch!exp \rrbracket(td) &= \bigcup_{i=0}^{\ell(td)} \left(\bigcup_{j=i+1}^{\infty} \text{at}[j / C, R := C \setminus \{ch!exp\}, R \cup \text{vars}(exp)] \circ \right. \\
&\quad \left. \text{at}[i / C := C \cup \{ch!e\}] \circ \text{del}^{j-\ell(td)} \cup \right. \\
&\quad \left. \llbracket \text{STOP} \rrbracket \circ \text{at}[i / C := C \cup \{ch!e\}] \right)(td)
\end{aligned}$$

Here i is the point of time where the channel becomes ready and j is the point of time where the communication actually happens. Here $j - \ell(td)$ extensions of the diagram by the del operator can become necessary if the communication happens after $\ell(td)$. For $j < \ell(td)$ the del operator is applied zero times.

The infinite timing diagram introduced by $\llbracket \text{STOP} \rrbracket$, stands for the case where a communication does not take place. For input additional non-determinism is added by the various input values m .

The definition for the slow operators $ch??v$ and $ch!!exp$ coincides with the one for the quick operators, except that i is confined to $\ell(td)$.

For parallel composition we relinquish the *occam* view that parallelism is an intersection of the behaviours of the parallel components. Instead we construct an operator on functions in the *at*-format:

We assume that $\llbracket P_j \rrbracket(td)$, $j = 1, 2$ are each given by a set of elements of the form $\text{at}[i / v_i := exp_i \mid i \in M \subseteq \mathbb{N}_0] \circ \text{del}^k$. (So M is the set of points in time where variables are changed.) Then we construct

$$\llbracket \text{PAR}[P_1, P_2] \rrbracket(td) = \{m_1 \parallel m_2 \mid m_j \in \llbracket P_j \rrbracket(td)\} \cap \text{SYN}$$

With SYN as below and the additional definition for a set S

$$at[n / v := S] = \bigcup_{m \in S} at[n / v := m]$$

we define

$$at[i / v_i := exp_i \mid i \in M_1 \subseteq \mathbb{N}_0] \circ del^k \parallel at[i / w_i := exp'_i \mid i \in M_2 \subseteq \mathbb{N}_0] \circ del^l = at[i / v_i := exp_i \parallel w_i := exp'_i \mid i \in M_1 \cup M_2] \circ del^{max}$$

Here $max = max(k, l)$ indicates that at first the shorter timing diagram is extended. Furthermore

$$v_i := exp_i \parallel w_i := exp'_i = v_i := exp_i \quad \text{for } i \in M_1 \setminus M_2$$

and analogously for $i \in M_2 \setminus M_1$. For $i \in M_1 \cap M_2$ let

$$v_i := exp_i \parallel w_i := exp'_i = v_i, w_i := exp_i, exp'_i$$

the normal parallel assignment for program variables for $v_i \neq w_i$. Simultaneous assignment to the same variable from both components leads to total non-determinism:

$$v_i := exp_i \parallel v_i := exp'_i = v_i := ty(v_i)$$

For the distinguished observables W, R the RHS contains their join. For the distinguished observable C the changes on both sides of the $\parallel$ take place simultaneously.

Example 6..3 ()

$$v_1, v_2, C := exp_1, exp_2, C \cup \{ch_1\} \parallel v_1, v_3, C := exp'_1, exp'_2, C \setminus \{ch_2\} = v_1, v_2, v_3, C := ty(v_1), exp_2, exp'_2, C \cup \{ch_1\} \setminus \{ch_2\}$$

Note that a simultaneous addition of some string by one component and its removal by the other is impossible. This ensures the well definedness for C . $\square$

The set SYN makes channel communication happen:

$$\begin{aligned} \text{SYN} = \{ & td \mid \forall ch \forall n, v, exp : (ch? \in td(W, n) - ch! \in td(W, n)) \wedge \\ & (\{ch?v, ch!exp\} \subseteq td(C, n) \Rightarrow \\ & \{ch?, ch!\} \subseteq td(W, n+1) \wedge td(v, n+1) = \llbracket exp \rrbracket(td(n))) \} \end{aligned}$$

The first line means that a communication takes place, only if both partners take part in that communication. The second and third lines mean that if an output and an input on a channel ch is possible at some

time n , the communication must take place as soon as possible, namely at time $n + 1$. This is indicated by the communication markers $ch?$, $ch!$ in W . Additionally the update on v must be performed.

Note that in this definition hiding is implicitly contained, because a parallel process can be observed at its external channels and its variables. If two parallel processes fitted together, their common channels are not longer external. This means that they are hidden.

The semantics for **IF** does not need a further explanation, and also **WHILE** is defined as usual by a least fix-point.

Now we can explain why an update as early as possible, by quick operators, is useful: For example for variables x, y, z we might want some implementation

$$\text{SEQ}[x := y, y := z] \prec \text{PAR}[x := y, y := z]$$

(defined below). If we had only the slow semantics, the LHS would inevitably take two time units and the RHS one unit, so an implementation would be impossible. In the Quickstep-Semantics the LHS takes one or two time units and the RHS still only one unit, which is a reduction of non-determinism.

In the version available by FTP (address on page 1), several examples are provided which show that this semantics can cope with the details of the timing behaviour and the cooperation of the various operators (sequence, parallelism, assignment).

6.3 IMPLEMENTATION RELATION

In order to compare DC formulae and Handel programs, we face three problems:

- For us DC formulae are timing diagrams, whereas Handel programs are operators on timing diagrams.
- The arbitrarily given initial values at time 0 must be somehow taken into account. Remember that assignment takes time. So each initialisation on the program level can become effective not earlier than at time 1.
- Whereas DC formulae are interpreted only over infinite timing diagrams, Handel programs are mostly dealing with finite ones.

The first two problems are overcome the same way: We apply programs to diagrams of length 0. These reflect exactly the implicit initial values at time 0. To a DC formula an additional conjunct must be added

which makes the initial values explicit. A diagram td of length 0 over variables var_i with values val_i can be conceived as a state assertion

$$\lceil td \rceil \equiv \lceil \forall i : var_i = val_i \rceil$$

Then the DC formula $\lceil td \rceil; true$ says that the observation must start in a state in which $\lceil td \rceil$ holds.

The last problem is fixed by extending finite timing diagrams to infinite ones using the STOP operator.

So for any Handel process P and for any DC formula F we define implementation $\prec$ (and analogously its converse $\succ$)

$$F \prec P \iff_{df} \forall td : \ell(td) = 0 \Rightarrow \llbracket STOP \rrbracket(\llbracket P \rrbracket)(td) \subseteq \lceil td \rceil; true \wedge F \text{ modulo } W, R, C$$

7. THE TRANSFORMATION FROM HANDEL TO HARDWARE

All hardware devices are described by DC formulae. We only give an example here.

A latch with input x , output y and latch condition $cond$ conforms to the following DC formula

$$\begin{aligned} Latch(x, y, cond) \equiv & \quad \Box \forall val \in ty(x) = ty(y) : \\ (x = val \wedge cond = true \vee y = val \wedge cond = false) \Rightarrow & \quad \bigcirc y = val \end{aligned}$$

Similar formulae apply to other devices like gates, multiplexers, etc. The aim of our transformations is to find an architecture, expressed in DC, which implements the Handel description. The transformation method starts with some algebraic laws for intermediate terms which are used for manipulations in the CSP tradition, for example associativity of SEQ or that SKIP is the neutral element of sequential composition. We will not go into more details here.

Besides laws for semantic identity there are also laws for implementation which reduce time non-determinism, most notably the fact that the slow variant of each operator implements the quick one. This allows us to finally do away with all quick operators in the last transformation step.

7.1 DISTRIBUTED CONTROL CIRCUITS

In this section we shall describe how an intermediate term, which does not contain any quick operators, is directly implemented in hardware. For such a given process P we need a data-multiplexing part $DM(P)$, a channel arbitration part $CA(P)$ and a control part $CTRL(P)$. Then

the implementation of P is given by the formula

$$IMPL(P) = DM(P) \wedge CA(P) \wedge CTRL(P)$$

Before explaining the details of these formulae, we make some assumptions:

- 1.) For each expression exp appearing in some assignment or output and for each condition $cond$ appearing in some control structure there is some circuitry (gates, adders, etc.) which implements exp , $cond$ and is described by some term $impl(exp)$, $impl(cond)$.
- 2.) For each channel ch there is only one variable v_{ch} in which an input can be written. So all inputs to ch are of the form $ch??v_{ch}$. Furthermore such a v_{ch} is not written in an assignment. These assumptions are not essential. However they make this part more readable.

By $DM(P)$ the change of data during one system tact is defined in terms of simple DC formulae for latches and multiplexers. Latches and multiplexers are triggered by expressions $ctrl(exp_i)$ (playing for example the role of $cond$ in the definition of a latch) which allow or prohibit data updates. The $ctrl(exp_i)$ are defined below. The full formulae can be found in the version available by FTP.

The channel arbitration part will be given as a conjunction over the channels

$$CA(P) = \bigwedge_{ch \in Chan(P)} CA(ch) \wedge Connect_{DM-CA}(P)$$

$Connect_{DM-CA}(P)$ describes the interplay between DM and CA . Via a channel ch performing output by one of the expressions $ch!!exp_1, \dots, ch!!exp_n$ in the program, a data transfer can take place if the input and one output are ready:

$$CA(ch) = \Box(transfer_{ch} = \bigvee_{i=1}^n transfer_{ch,i} \wedge \bigwedge_{i=1}^n (transfer_{ch,i} = rdy_{ch!exp_i} \wedge rdy_{ch?v}))$$

The rdy will be determined in $CTRL(P)$. The connection between DM and CA defines some of the $ctrl(exp_i)$ allowing data update (if the data transfer is possible):

$$Connect_{DM-CA}(P) = \Box(\bigwedge_{ch,i} ctrl(exp_i) = transfer_{ch,i})$$

where the conjunction varies over all channels ch and all output expressions $ch!!exp_i$ appearing in the program for ch .

The implementation of the control part $CTRL(P)$ of a process P is done inductively by means of two DC variables s_P, f_P , which indicate the start and the final control state (one-hot encoding). The following formulae will give

- 1.) the connections between the start and the final control states of the subprocesses of P , given by $Connect_{SF}(P)$,
- 2.) the connections between the control states of the subprocesses and, respectively, the control wires of the DM part and the not yet defined wires of the CA part, given by $Connect_{CTRL-DM}(P)$ and $Connect_{CTRL-CA}(P)$.

For $R = \text{DELAY}$ control at state s_R means next time control at final state f_R :

$$CTRL(R) = \Box (\bigcirc f_R = s_R) \quad (\equiv Connect_{SF}(R))$$

For $R = x ::= exp$, control changes the same way and a data update is allowed:

$$\begin{aligned} CTRL(R) &= Connect_{SF}(R) \wedge Connect_{CTRL-DM}(R) \\ Connect_{SF}(R) &= \Box (\bigcirc f_R = s_R) \wedge Connect_{CTRL-DM}(R) = \Box (s_R = ctrl(exp)) \end{aligned}$$

The last equality is another part of the delayed definitions of the $ctrl(exp_i)$ from DM . For the process $R = \text{SEQ}[Q_1, \dots, Q_n]$ let

$$\begin{aligned} CTRL(R) &= \bigwedge_i CTRL(Q_i) \wedge Connect_{SF}(R) \\ Connect_{SF}(R) &= \Box (s_R = s_{Q_1} \wedge f_R = f_{Q_n} \wedge \bigwedge_i s_{Q_{i+1}} = f_{Q_i}) \end{aligned}$$

For $R = \text{PAR}[P_1, \dots, P_n]$ simultaneous termination of the P_i at f_R is achieved by consideration of $wasf_{P_i}$ and $keepf_{P_i}$ (saying that P_i has reached f_{P_i}):

$$\begin{aligned} CTRL(R) &= \bigwedge_i CTRL(P_i) \wedge Connect_{SF}(R) \\ Connect_{SF}(R) &= \Box (\bigwedge_i (s_R = s_{P_i}) \wedge (f_R = \bigwedge_i keepf_{P_i}) \wedge \\ &\quad (\bigwedge_i (keepf_{P_i} = f_{P_i} \vee wasf_{P_i})) \wedge \\ &\quad (\bigwedge_i (\bigcirc wasf_{P_i} = keepf_{P_i} \wedge \neg f_R))) \end{aligned}$$

The formulae for **if** and **while** are dropped due to space limitations. For $R = ch??v$ (similarly for $R = ch!!e_i$) let $wasrdy_{ch?v}$ be a new internal DC variable showing readiness but no data transfer in the last step.

$$\begin{aligned} CTRL(R) &= \Box ((rdy_{ch?v} = wasrdy_{ch?v} \vee s_R) \wedge \\ &\quad (\bigcirc wasrdy_{ch?v} = rdy_{ch?v} \wedge \neg transfer_{ch}) \wedge \\ &\quad (\bigcirc f_R = transfer_{ch})) \end{aligned}$$

7.2 IMPLEMENTATION BY DISTRIBUTED CONTROL CIRCUITS

At the Handel level we only consider the data variables. The newly introduced control variables in the DC formulae describing the hardware implementation are not visible on the upper levels. So they will be hidden as usual by existential quantification. These are all variables of the form

$\text{mux}_v, \text{change}_v, \text{ctrl}(\text{exp}), \text{transfer}_{\dots}, \text{rdy}_{\dots}, \text{wasrdy}_{\dots}, s_P, f_P, \text{keep}f_P, \text{was}f_P$

Let the set of all these variables be called CtrVar . As we have done for the data variables we must specify an initial state for each control variable. Since we aim at a one-hot encoding for a given process P , initially only one variable in CtrVar can be true: the start variable s_P of course. Let initctr be the state with

$$[\text{initctr}] \equiv [s_P \wedge \bigwedge_{s_P \neq v \in \text{CtrVar}} \neg v]$$

In the following theorem initctr is used to define the initial state of those control variables which are not defined in the Handel program. Remember that the initialisation of the other variables is dealt with in the definition of “ $\prec$ ”.

Theorem 1 ()

For all Handel processes R involving only slow operators

$$R \prec \exists \text{CtrVar} : [\text{initctr}]; \text{true} \wedge \text{IMPL}(R)$$

□

The proof is full of mathematical details, see the version available by FTP.

8. CONCLUSIONS AND FURTHER WORK

In this paper we have described a formal transformation framework for provably-correct hardware compilation using Handel. The outstanding features of this method are the precise mathematical treatment of a complex parallel operator and its interdependencies with sequential composition with respect to time. Our method allows us to prove abstract requirements in a logical formalism, the DC, on the basis of a uniform model. Such a uniform model is also a novelty.

The introduction of the quick operators enables a powerful mechanism for exploiting operation re-scheduling to deliver optimal implementations. Although this model is adequately general to demonstrate

real applications in the near future, the clear and strict definition of the operators which specify and construct the timing diagrams, which in turn constitute the transformational basis, ensure the robustness as well as the further refinement of our model. There is ongoing work, with a number of real-world applications such as communication protocols and large coding blocks, formally designed using our framework. Results from these activities will be the subject of future publications.

References

- G.M.Brown. Towards truly delay-sensitive circuit realisation of process algebra. In: G.Jones and M.Sheeran, Eds., *Designing Correct Circuit*, Springer-Verlag, 1991, 120–131.
- M.R.Hansen, Zhao Chaochen. Lecture Notes on Logical Foundations of Duration Calculus. *Formal Aspects of Computing* 9, 1997, 283–330.
- He Jifeng, C.A.R. Hoare, M. Fränzle, M. Müller-Olm, E.-R. Olderog, M. Schenke, M.R. Hansen, A.P. Ravn, and H. Rischel. Provably correct systems. In: H. Langmaack, W.-P. de Roever and J. Vytupil, Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863 (Springer-Verlag, 1994), 288–335.
- He Jifeng, I.Page, J.Bowen Towards a Provably Correct Hardware Implementation of *occam*, CHARME93, LNCS 683 (Springer-Verlag, 1993), 214–226.
- He Jifeng, Zhang Jianping. Simulation Approach to Provably Correct Hardware Compilation. In: H. Langmaack, W.-P. de Roever and J. Vytupil, Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863 (Springer-Verlag, 1994), 336–350.
- C.A.R.Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- A.J.Martin. The design of a self-timed circuit for distributed mutual exclusion. In: H.Fuchs, Ed., *Proc 1985 Chapel Hill Conference VLSI*, 1985, 247–260.
- A.J.Martin. Compiling communicating processes into delay-insensitive circuits. *Distributed Computing* 1, 226–234.
- I.Page, W.Luk. Compiling *occam* into Field Programmable Gate Arrays. In: W.Moore, W.Luk, Eds., *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, 1991.
- M.Spivey, I.Page. *How to program in Handel*. Based on the Handel hardware compiler, ver. H 159.H. Revised by W.Luk, Dec.94. Oxford University Computing Laboratory, Hardware Compilation Research Group technical report, 1994.
- Zhou Chaochen, C.A.R. Hoare, A.P. Ravn. A Calculus of Durations. *IPL* 40/5 1991, 269–276.