hopCP: Language Definition, Semantics and Examples

Venkatesh Akella

UUCS-90-010

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

August 27, 1990

Abstract

We describe a formalism for high level modeling of hardware based on flow graphs and nonatomic actions called hopCP. A module is the description of a hardware system in hopCP, which contains a flow graph to model the behavioral aspects and ports which represent the communication links. Operations are provided to manipulate modules and flow graphs. Nonatomic actions provide the necessary functional and temporal abstraction to model hardware and action refinement is introduced to bridge the abstraction gap for high level synthesis. Examples are provided to elucidate the semantics of hopCP and illustrate the expressive power of the language.

Formal Aspects of VLSI Research Group
University of Utah, Department of Computer Science

hopCP: Language Definition, Semantics and Examples

VENKATESH AKELLA (*akella@cs.utah.edu*)

University of Utah
Dept. of Computer Science
2490 Merrill Engineering Building
Salt Lake City, Utah 84112

Keywords: Hardware description languages, behavioral modeling, action refinement, asynchrony

Abstract. *We describe a formalism for high level modeling of hardware based on flow graphs and nonatomic actions called hopCP. A module is the description of a hardware system in hopCP, which contains a flow graph to model the behavioral aspects and ports which represent the communication links. Operations are provided to manipulate modules and flow graphs. Nonatomic actions provide the necessary functional and temporal abstraction to model hardware and action refinement is introduced to bridge the abstraction gap for high level synthesis. Examples are provided to elucidate the semantics of hopCP and illustrate the expressive power of the language.*

1 Introduction

In this report we describe the language hopCP and its semantics. hopCP is a high level formalism to capture the behavior of hardware systems in a precise mathematical notation and facilitate their synthesis and validation. hopCP is the descendant of the language HOP [4] which is basically a process model to capture the behavioral aspects of *synchronous* VLSI circuits.

hopCP specification take a *sequence domain* view of hardware where the behavior of a system is captured by the *causal relationship* between a set of *actions* which the hardware can perform. The specification formalism does not prescribe any *timing discipline*. The designer is free to adhere to any *timing discipline* during synthesis, as long as it does not violate the *causal relationship* between actions noted in the specification. In this sense, hopCP specifications are *asynchronous*. The benefits of this *asynchronous* view of hardware include the flexibility to specify and reason about circuits with different timing disciplines like speed independent, delay insensitive, synchronous (with various clocking schemes) in a *uniform* framework and also refrain from introducing any unnecessary constraints, such as sequentiality in the specification domain itself.

In hopCP, the computational and communication aspects of hardware behavior are captured by *actions*. The principal reason for success of the sequence domain view of hardware

is the *nonatomicity* of actions. We can impart structure to actions by means of *action refinement rules.*

Significance of Refinable Actions

Every behavioral description formalism should have the ability to model (and reason about) a hardware system at different levels of hierarchy. In hopCP, we capture this notion of hierarchical modeling using *nonatomic actions.*

For example, at the architecture level description of a system, it is perfectly reasonable and is in fact sufficient to consider the *add instruction* as just an action. But, reasoning at the gate level soon reveals that an execution of the *add instruction* actually involves several microinstructions like instruction fetch, decode, operand fetch etc. spread across an *interval of time* (hence nonatomic). Going further down to the circuit level we see that each of the microinstruction itself needs certain signals (voltages) going high or low. In fact, even at the level of signals, it is still not very obvious whether they should be considered atomic, since there is always the notion of *rise time* and *fall time* of specific signals. This is especially important in clocked synchronous systems.

The example above motivates the usefulness of considering actions to be nonatomic to begin with and later *refining* them to facilitate detailed modeling of the practical implementation schemes.

This gives us several advantages:

1. In the context of the conventional hierarchical view of hardware design, these action refinement rules *bridge* the *abstraction gaps* between different levels of hierarchy and thereby promote an *incremental* approach to hardware design.

2. Action refinement rules provide a mechanism for circuit synthesis from hopCP specifications, because when the actions have been refined to the level of electrical signals we get an implementation for our specification.

3. The incremental approach to hardware synthesis via action refinement lends itself to easy validation and performance analysis procedures.

2 Overview of hopCP

In this section we shall give an overview of the language hopCP by highlighting its salient features and presenting the goals of this research.

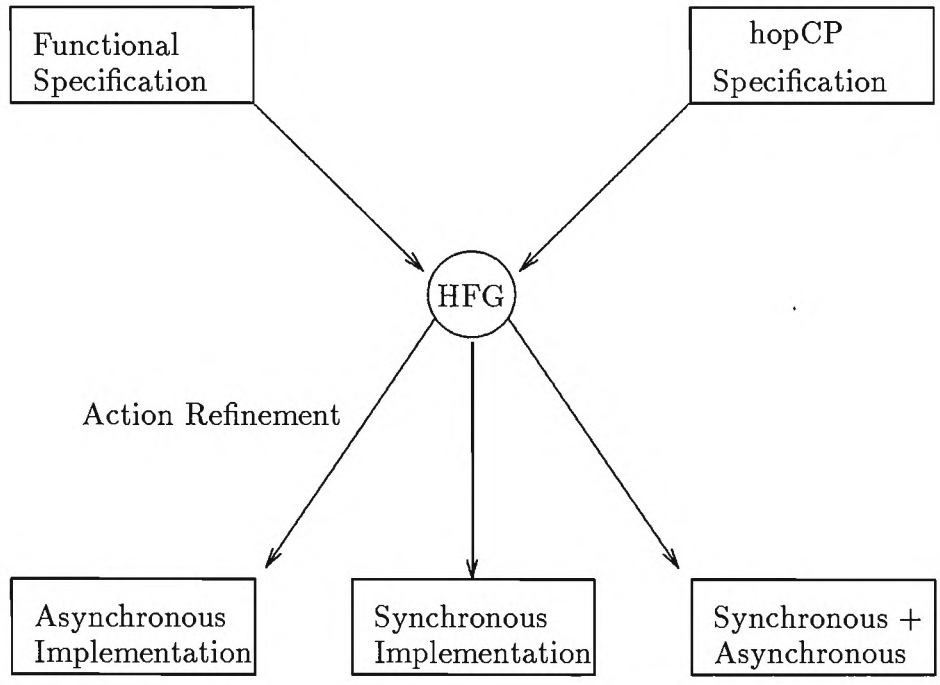


Figure 1: Overview of hopCP based Design Environment

hopCP Flow Graph Model

In the past we advocated a process oriented view of hardware [4] which is appropriate to model the communication and concurrency aspects but does not lend itself to describing algorithmic computation very well. Experience with hardware design quickly reveals that sometimes it is more natural to go with *functional* or an *abstract data type* view of hardware which is evident from Johnson's research at Indiana [2,3].

In general, we would like to keep both the options open when we are specifying a behavior. This has prompted the *flow graph* based model for hopCP, based on actions. The category of expression actions (to be described in detail later) provides the capability to start with an abstract data type view of hardware.

Objectives of the hopCP design effort

Figure 1 shows the current plan for a VLSI design environment centered around hopCP. The user will have the option of specifying the behavior either in hopCP directly or in a purely functional language (a subset of Standard ML), both of which will be converted into the hopCP flow graph notation which will form the *canonical* representation of hardware in our system. (A graphical specification tool to directly describe and manipulate the flowgraphs

is also under investigation)

Tools will be provided for *simulation* of these behaviors and systematic *translation* into either *synchronous*, *speed independent* or *hybrid* (partly synchronous and partly asynchronous) circuits.

This report will focus on the definition and semantics of hopCP. In the next section we provide the methodology of behavioral description by describing the flow graph model and its operations. It will be followed by techniques for structural specification and a detailed study of the various action categories and the synchronization and interaction schemes. Finally, we provide three examples of increasing complexity and distinctly different requirements to illustrate the expressive power of our language.

3 Behavioral Descriptions in hopCP

Specification of a hardware systems in hopCP consists of a *behavioral description* and a *structural description.* The behavior of a hardware system is captured by a *state transition system* called HFG (hopCP Flow Graph) which is defined as follows: [1]

Definition 1 *A HFG is a 4-tuple,* $\langle S_i, S, Act, \rightarrow \rangle$ *where* S *is the set of* states, $S_i \subseteq S$ *is the set of* initial *states, Act is the set of* actions *and* $\rightarrow \subseteq (\mathcal{P}^+(S) \times Act \times \mathcal{P}^+(S))$. *(*$\mathcal{P}^+(S)$ *is the* non empty *power set of S.)*

S is typically a pair $\langle cs, ds \rangle$ with cs denoting the *control state* and ds representing the *data state. Act* is the set of *actions* which capture the *computational* and *communication* aspects of the hardware in hopCP.

Let $P = \langle S_i, S, Act, \rightarrow \rangle$ be a HFG denoting a behavioral description in hopCP. Then S_{i_p}, S_p, Act_p and $\rightarrow_p$ denote the set of initial states, set of states, set of actions and set of transitions in P. These are selector functions which facilitate the manipulation of HFGs .

Definition 2 *A* transition *or* occurrence *of an action* $a \in Act_p$ *in a HFG P, is defined as a triple* $(s, a, s') \in \rightarrow_p$.

Note that a given action may have more than one *transitions* associated with it.

Compound Actions

Act which represents the actions in a HFG is basically a set of *nonatomic* actions, in the sense that they have *structure* and are amenable to *refinement.* So, they are different from the conventional notion of actions in process calculi and transitions in net theory. In fact,

[1] More precisely, it is a *hypergraph*, if the states are viewed as vertices and the actions as edges.

the basic modeling power and ease of synthesis in hopCP is derived from these *nonatomic* actions. Moreover, there are three types of actions which can be defined using the following syntactic domains:

$$\begin{aligned} A &= C + D + E \\ C &= Set\ of\ Control\ Actions \\ D &= Set\ of\ Data\ Actions \\ E &= Set\ of\ Expression\ Actions \\ Act &= \mathcal{P}(A)\ Set\ of\ Compound\ Actions \end{aligned}$$

Let P be a HFG and $a = \{a_1, a_2, \ldots, a_n\} \in Act$. Then, a transition $(s, a, s') \in \rightarrow_p$ means that if the system (being modeled) is in a state s, then it performs $a_1, a_2, \ldots a_n$ *concurrently* (in an unspecified order) and reaches the new state s'. Hence the following definition for compound actions:

Definition 3 *A compound action is defined as a set of actions (potentially nonatomic and compound) which are performed concurrently i.e. in an undetermined order. The set of actions defining a compound action are called* subactions *of a compound action.* $\{a_1, a_2, \ldots, a_n\}$ *are called* subactions *of* a.

Now, let us briefly explain the individual constituents of the domain A of actions.

- Control Actions (C): dictate the control flow of a hardware module and typically describe the control wire encodings. They capture the *synchronization* aspects of hardware behavior in our formalism. These are *boolean* signals and are also referred to as *events* in our formalism. Events are directional i.e. they could be *input* (passive) or *output* (active).

  Example

  $p!$ denotes an output event on *port* p while $q?$ denotes an input event on *port* q. (Note: *ports* are structural entities in a hopCP specification which will explained shortly.)

- Data Actions (C): capture *synchronization and value communication* aspects of hardware behavior. Data actions are further classified into *data queries* which denote *input* or passive actions and *data assertions* which refer to active actions.

  Example

  $p!exp$ denotes asserting an expression exp on *port* p (output data action) while $q?x$ denotes receiving a value x on *port* q (input data action).

- Expression Actions (C): capture the *computation* aspects of a hardware module and are described in a purely functional subset of Standard ML.

Definition 4 *Let P be an HFG and $(s, a, s') \in \rightarrow_p$. The elementary flow graph corresponding to a transition (s, a, s') is defined by*

$$efg : \mathcal{P}^+(S) \times Act \times \mathcal{P}^+(S) \mapsto HFG$$

such that $efg(s, a, s')$ is a HFG described by $\langle \phi, \{s \cup s'\}, \{a\}, \{(s, a, s')\}\rangle$.

Operations on Flow Graphs

hopCP provides three combinators (or functions) to manipulate HFGs and construct HFGs denoting more complex hardware behaviors from HFGs describing simpler ones. The behavior of a hardware system is captured via these combinators which are formalized as follows: [2]

progress

Syntax: $P \Leftarrow a \rightsquigarrow Q$ where $Q = \langle S_{i_q}, S_q, Act_q, \rightarrow_q \rangle$ and $a \in Act$.

Semantics: The *progress* combinator, $\rightsquigarrow$ defines a new HFG, P denoted by the four tuple $\langle S_{i_p}, S_p, Act_p, \rightarrow_p \rangle$ where

$$\begin{aligned} S_{i_p} &= \{P\} \\ S_p &= \{P\} \cup S_q \\ Act_p &= \{a\} \cup Act_q \\ \rightarrow_p &= \rightarrow_q \cup \{(P, a, S_{i_q})\} \end{aligned}$$

Hence, $\rightsquigarrow$ can be regarded as a function with the following signature:

$$\rightsquigarrow : HFG \times Act \mapsto HFG$$

Points to Remember:

1. $\rightsquigarrow$ is similar to the *sequencing* operator found in several languages.
2. If a is empty then HFG (P) and HFG (Q) are identical
3. The name of the HFG is usually taken to be synonymous with the name of the *control state*. It may be recalled that the set of states in a hopCP behavioral descriptions are pairs of *control* and *data* states.

[2] $\Leftarrow$ stands for *is defined by* in the hopCP semantic definition

choice

Syntax: $P \Leftarrow a \rightsquigarrow Q \;[\!]\; b \rightsquigarrow R$

Semantics:The *choice* ([]) operator describes the construction of a new $HFGP\ \langle S_{i_p}, S_p, Act_p, \rightarrow_p \rangle$ from Q and R defined as follows:

$$\begin{aligned} S_{i_p} &= \{P\} \\ S_p &= \{P\} \cup S_q \cup S_r \\ Act_p &= \{a\} \cup \{b\} \cup Act_q \cup Act_r \\ \rightarrow_p &= \rightarrow_q \cup \rightarrow_r \cup \{(P, a, Q), (P, b, R)\} \end{aligned}$$

Choice combinator could be regarded as a function with the following signature:

$$[\!] : HFG \times HFG \times Act \times Act \mapsto HFG$$

From the above definition, it can be easily shown that *choice* ([]) is both *associative* and *commutative*. So, in general we could have an *m-ary choice* operation defined over HFGs.

Choice is used to model *nondeterministic* behavior in hopCP. In our above definition, a hardware module in a configuration (state) described by P could nondeterministicly choose to participate in an action a or an action b.

Restrictions on *choice*

1. Actions a and b are *distinct*. Two actions are said to be *distinct* if they are either not the same or if they do not have any common subactions. This disallows *output nondeterminism* which is said to occur when the module makes a nondeterministic choice by itself. However, this does not rule out *input nondeterminism* wherein the environment can make the choice.

2. a and b should not contain any output actions i.e. (output control actions or data assertions)

3. If a or b contain any *expression actions*, they should be simple predicate tests to facilitate a direct implementation.

4. If the environment of the system being modeled *guarantees* that either only a occurs or only b occurs but never both, then we get a deterministic behavior of the system.

parcomp

Syntax: $P \Leftarrow Q \parallel R$

This is the familiar *composition* operator defined in various process calculii which captures the *interaction* between independent behaviors (which in our case are captured by different

HFGs). It deals with both *synchronization* (temporal constraints on the causal behavior) and *value communication* which reflects the *exchange* of data. *Parcomp* is a function with the following signature

$$\|: HFG \times HFG \mapsto HFG$$

Semantics: Let $Q = \langle S_{i_q}, S_q, Act_q, \rightarrow_q \rangle$ and $R = \langle S_{i_r}, S_r, Act_r, \rightarrow_r \rangle$. $\|$ is defined by three inference rules which capture the different synchronization scenarios (which will be explained in detail shortly). It is interesting to note that $\|$ is actually defined in terms of the *elementary flow graph* associated with each action occurrence in HFGs Q and R.

$$\frac{(s_1, a, s_1^{'}) \in \rightarrow_q, \quad (s_2, b, s_2^{'}) \in \rightarrow_r}{(s_1 \cup s_2, a, s_1^{'} \cup s_2^{'}) \in \rightarrow_p} \qquad a = b \qquad (1)$$

$$\frac{(s_1, a, s_1^{'}) \in \rightarrow_q, \quad (s_2, b, s_2^{'}) \in \rightarrow_r}{(s_1, a, s_1^{'}) \in \rightarrow_p, \quad (s_2, b, s_2^{'}) \in \rightarrow_p} \qquad a \cap b = \phi \qquad (2)$$

$$\frac{(s_1, a, s_1^{'}) \in \rightarrow_q, \quad (s_2, b, s_2^{'}) \in \rightarrow_r}{(s_1 \cup s_2, \quad SYNC(tr_a, tr_b), \quad s_1^{'} \cup s_2^{'}) \in \rightarrow_p} \qquad a \cap b \neq \phi \qquad (3)$$

where $tr_a = (s_1, a, s_1^{'})$ and $tr_b = (s_2, b, s_2^{'})$

Synchronization with Nonatomic Actions

The major challenge to defining the *parcomp* operator in hopCP is the *nonatomicity* of the actions. To recapitulate, we said that typically an action $a \in Act$ is actually a collection of *concurrent* actions $\{a_1, a_2, \ldots a_n\}$ (which was also defined as an *compound action*, definition 3). When we consider *interaction* of two HFGs then, we should discuss the interaction of such compound actions. Let $a = a_1, a_2, \ldots a_n$ and $b = b_1, b_2, \ldots b_m$ be two actions in Act.

Given the nature (structure) of the compound actions, there could be the following three styles of *interactions* between actions a and b:

Total Synchronization is said to occur when $a = b$; the equality is the well known equality relation on sets. It is interesting to note that, in this case (i.e. when $a = b$), the nonatomicity of the individual actions is unimportant. To the external world it appears as though a single *atomic* action has taken place.

No Synchronization is said to occur when $a \cap b = \phi$. This is fairly intuitive, since hopCP is centered around an *asynchronous* nature of computation and control flow. When two actions have nothing in common, then they can taken place independently. No *temporal constraint* is imposed on when an action is initiated and when it is completed.

Partial Synchronization is said to occur when $a \cap b \neq \phi$. This occurs when a and b share some common subactions. An action x typically *occurs* in a HFG as a transition triple $tr_x = (s_1, x, s_1^{'})$. Let, $tr_a = (s_1, a, s_1^{'}) \in \rightarrow_q$ and $tr_b = (s_2, b, s_2^{'}) \in \rightarrow_r$ be the occurrences of a and b in HFGs Q and R.

Definition 5 *Partial synchronization of two actions a and b with respect to their* occurrences tr_a *and* tr_b *is defined as an action or a HFG*

$$sync(tr_a, tr_b) = prefine(tr_a) \parallel prefine(tr_b)$$

where prefine is described in definition 7.

Action Refinement

We have found that a very general class of causal constraints between actions can be captured by just two refinement rules(where $\Longrightarrow$ denotes the refinement of a action):

- Series Refinement: $a \Longrightarrow a_1 \rightsquigarrow a_2$

  The *series refinement* strategy denotes the execution of the action a being tantamount to the execution of action a_1 followed by the execution of action a_2. Note that we have intentionally used the *progress* combinator, '$\rightsquigarrow$' to depict the series refinement because they are similar semantically.

- Parallel Refinement: $a \Longrightarrow a_1, a_2$

  The *parallel refinement* strategy denotes the execution of action a being tantamount to the execution of action a_1 and a_2 *concurrently.* Any action following a in a HFG will not be started till both a_1 and a_2 are completed.

In the HFG model for behavioral descriptions in hopCP, we see that there is a dual role for actions: they can either be viewed as just denoting transitions from one configuration to another or can be viewed as HFGs themselves. We shall characterize the *series* and *parallel* refinement rules in terms of occurrences of actions within a HFG . Let, P denote an HFG , $a \in Act_p$ and $tr_a = (s, a, s')$ be an occurrence of a in P (note that there could be many occurrences of a in P.)

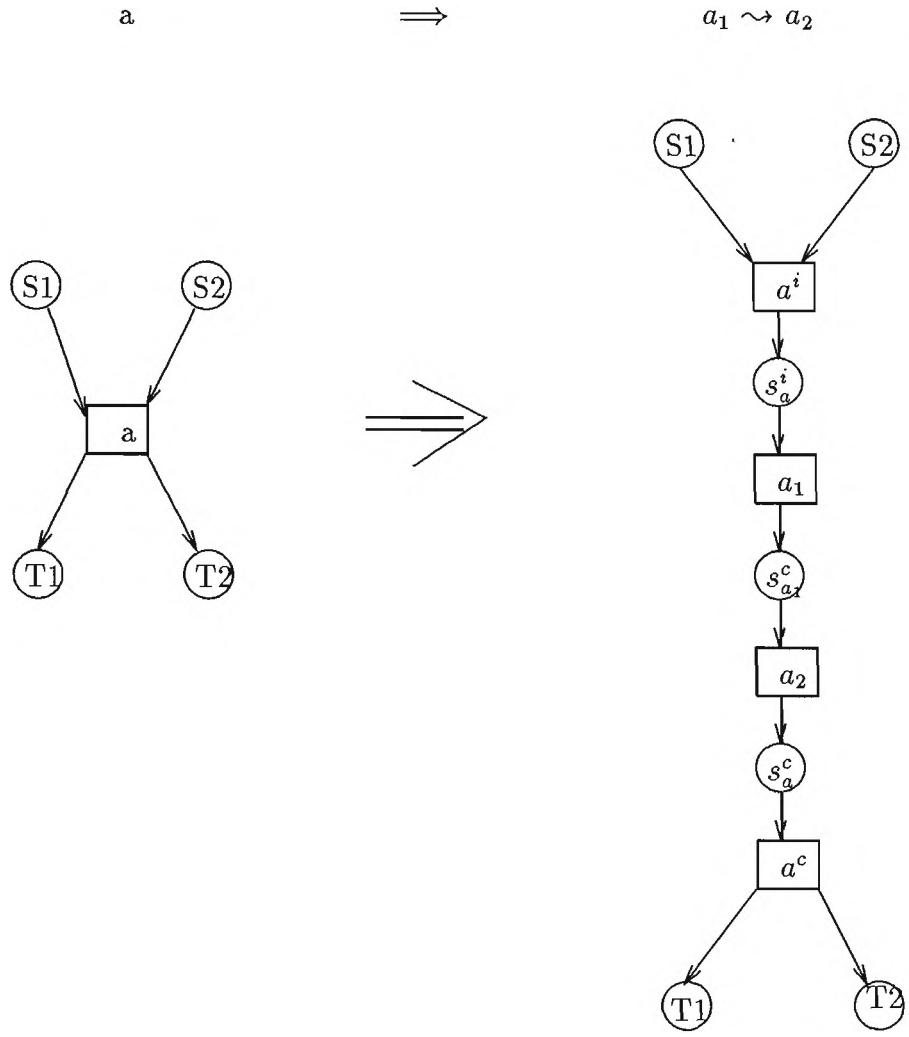


Figure 2: Illustration of Series Refinement of Actions in hopCP

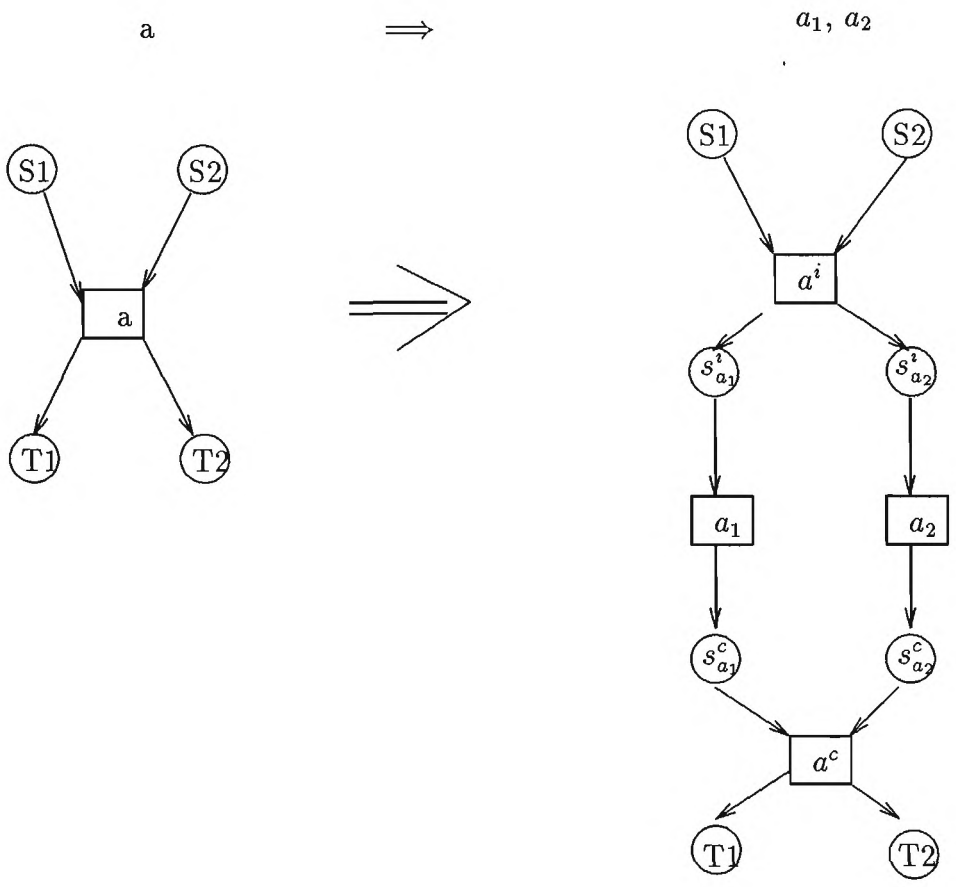


Figure 3: Illustration of Parallel Refinement of Actions in hopCP

Then, *series refinement* (see figure 2) and *parallel refinement* (see figure 3) of a into actions a_1 and a_2 is defined as follows:

Definition 6 *The* series refinement *of a into actions a_1 and a_2 is defined by the function srefine*

$$srefine : \mathcal{P}(S) \times Act \times Act \times Act \times \mathcal{P}(S) \mapsto HFG$$

where

$$\begin{aligned} srefine(tr_a) \;=\; & \langle \phi, \\ & s \cup s' \cup \{s^i_a, s^c_{a_1}, s^c_a\}, \\ & \{a^i, a_1, a_2, a^c\} \\ & \{(s, a^i, s^i_a)(s^i_a, a_1, s^c_{a_1}), (s^c_{a_1}, a_2, s^c_a), (s^c_a, a^c, s')\} \\ & \rangle \end{aligned}$$

Definition 7 *The* parallel refinement *of a into actions a_1 and a_2 is given by the function prefine*

$$prefine : \mathcal{P}(S) \times Act \times Act \times Act \times \mathcal{P}(S) \mapsto HFG$$

where

$$\begin{aligned} prefine(tr_a) \;=\; & \langle \phi, \\ & s \cup s' \cup \{s^i_{a_1}, s^i_{a_2}, s^c_{a_1}, s^c_{a_2}\}, \\ & \{a^i, a_1, a_2, a^c\}, \\ & \{(s, a^i, s^i_{a_1})(s, a^i, s^i_{a_2}), (s^i_{a_1}, a_1, s^c_{a_1}), (s^i_{a_2}, a_2, s^c_{a_2}), \\ & (s^c_{a_1}, a^c, s'), (s^c_{a_1}, a^c, s')\} \\ & \rangle \end{aligned}$$

Note that we have illustrated the refinement of a action into *two* actions to simplify the presentation. In general an action could be refined into m serial or parallel actions. The construction of the new HFG is identical. The power of the action refinement strategy becomes apparent in the verification process and the synthesis of circuits from hopCP descriptions. However, verification and synthesis are not addressed in this report. [3]

[3]Note that the names of the actions and states introduced in definitions are not arbitrary, they have very vital significance which will be apparent once you consider the abstract time semantics of the hopCP behaviors

Illustrating Behavioral Modeling in hopCP

We have introduced sufficient machinery to model interesting hardware applications in hopCP. To facilitate textual descriptions in hopCP, we will slightly abuse the syntactic notions developed so far, but one can get used to it fairly quickly. In the long run, we would actually encourage a *graphical input mechanism*, wherein HFGs could be directly described using graphics primitives, pretty much like the conventional *schematic capture* tools.

1. A piece of hardware which does nothing

$$\begin{aligned} \textit{Textual Description}: &\quad STOP \\ \textit{HFG Representation}: &\quad \langle \phi, \{STOP\}, \phi, \phi \rangle \end{aligned}$$

Actually $STOP$ represents an interesting piece of hardware, namely, a *finite* behavior. The utility of this hardware by itself maybe questionable but it certainly is helpful in describing *computation* oriented hardware, like the *expression actions* we discussed earlier.

2. A cyclic behavior

$$\begin{aligned} \textit{Textual Description}: &\quad P \Leftarrow a \rightsquigarrow P \\ \textit{HFG Representation}: &\quad \langle \{P\}, \{P\}, \{a\}, \{(P, a, P)\} \rangle \end{aligned}$$

The behavior is self-explanatory. It illustrates the *progress* combinator.

3. A piece of hardware with two actions

$$\begin{aligned} \textit{Textual Description}: &\quad Q \Leftarrow b \rightsquigarrow c \rightsquigarrow P \\ \textit{HFG Representation}: &\quad \langle \{Q\}, \{Q, Z, P\}, \{a, b, c\}, \{(Q, b, Z), (Z, c, P), (P, a, P)\} \rangle \end{aligned}$$

Q denotes a piece of hardware which does actions b and c *sequentially* and then behaves like the hardware described in the previous example. Notice that in behavioral modeling using hopCP we do not commit to any *absolute time*. Actions b and c need not take the same amount of time to execute; in fact, they will not because, they are typically *nonatomic* actions wherein b could merely be the assertion of a signal while c is a function computation like a *multiply* (expression action).

Note: We have introduced an arbitrary state Z to facilitate the HFG description and used P as a state and also as a HFG . The former merely reflects that Z is an *internal state* of the module, which we do not worry about while the latter is merely a *notational* convenience to save retyping the whole behavior of P once again.

4. Illustrating compound actions

$$\begin{aligned} \textit{Textual Description}: &\quad R \Leftarrow x, y \rightsquigarrow P \\ \textit{HFG Representation}: &\quad \langle \{R\}, \{P, R\}, \{a, x, y\}, \{(R, \{x, y\}, P), (P, a, P)\} \rangle \end{aligned}$$

R is piece of hardware which engages in actions x and y *concurrently* (in an unspecified order) and after the *completion* of both the actions proceeds to behave like the hardware described by P.

Note: $\{x, y\} \subset \mathcal{P}(Act_r)$ denotes a compound action, but the set notation for the action has been dropped to avoid cluttering the textual descriptions.

5. Illustrating Alternate Behaviors

$$\begin{aligned} \textit{Textual Description}: &\quad M \Leftarrow b \rightsquigarrow P [] c \rightsquigarrow R \\ \textit{HFG Representation}: &\quad \langle \{M\}, \{P, M, R\}, \{a, b, c, x, y\}, \\ &\quad \{(M, b, P), (M, c, R), (R, \{x, y\}, P), (P, a, P)\} \rangle \end{aligned}$$

M denotes hardware which has two alternate *modes of behavior*. It can *either* engage in action b and proceed to behave like the hardware described by P *or* perform action c and behave like the hardware described by $HFG\ R$. If we assume that only one of b or c is feasbile at anytime then we get *deterministic* behavior otherwise we get *nondeterminism*.

6. Illustrating Parallelism and Synchronization

$$\begin{aligned} \textit{Textual Description}: &\quad S1 \Leftarrow a \rightsquigarrow c \rightsquigarrow S1 \\ &\quad T1 \Leftarrow b \rightsquigarrow c \rightsquigarrow d \rightsquigarrow T1 \\ &\quad N \Leftarrow S1 \parallel T1 \\ \textit{HFG Representation}: &\quad \langle \{S1, T1\}, \{S1, S2, T1, T2, T3\}, \{a, b, c, d\}, \\ &\quad \{(S1, a, S2), (\{S2, T2\}, c, \{S1, T3\}), (T1, b, T2), (T3, d, T1)\} \rangle \end{aligned}$$

Notice, that we have two initial states now, which immediately reflects that there are two threads of parallel activity. Also, the transition $(\{S2, T2\}, c, \{S1, T3\}) \subset \rightarrow_n$ denotes the *total synchronization* on action c

7. Modeling Fork Behavior

$$\begin{aligned} \textit{Textual Description}: &\quad M \Leftarrow (b \rightsquigarrow G) \parallel (b \rightsquigarrow F) \\ \textit{HFG Representation}: &\quad \langle \{M\}, \{M, G, F\}, \{b\} \cup Act_g \cup Act_f, \\ &\quad \rightarrow_g \cup \rightarrow_f \cup \{(M, b, \{G, F\})\} \rangle \end{aligned}$$

M denotes a piece of hardware which performs action b and then *forks* two independent threads of behavior described by $HFG\ G$ and $HFG\ F$. Typically, the transition (occurrence of action) $(M, b, \{G, F\})$ denotes *forking* behavior.

4 Structural Specifications in hopCP

In hopCP the basic structural entity being modeled is a *module* which is defined as follows:

Definition 8 *A module is defined in hopCP by a behavioral description and the set of ports (channels).*

$$MODULE : HFG \times PORT$$

where $PORT \subset IDENT(Set\ of\ Identifiers)$

A structural description in hopCP, primarily denote how modules are connected to each other and what the visible connections are to the external world. To facilitate such description we propose three *operators* on modules:

rename

Let $\mathcal{R}$ be the set of bijections from $PORT \mapsto PORT$
rename is a function on hopCP modules defined as follows:

$$rename : MODULE \times \mathcal{R} \mapsto MODULE$$

Let $M = \langle hfg_M, ports_M \rangle$ and $f \in \mathcal{R}$

$rename(M, f) = \langle hfg_M, f(ports_M) \rangle$

export

export is a function which takes a module and a set of ports and makes the latter visible to the external world, and is defined as follows:

$$rename : MODULE \times PORT \mapsto MODULE$$

Let $M = \langle hfg_M, ports_M \rangle$ and $vis_{ports} \subseteq port_M$

$export(M, vis_{ports}) = \langle hfg_M, vis_{ports} \rangle$

(*export* is similar to the hide operator in CSP)

connect

connect is a function which takes two modules and returns a new module which reflects the *composite* behavior of both the modules when the common ports are connected and after verifying the *well-formedness* of the connections, which will be described shortly

connect can be viewed as the structural counterpart of the *parcomp* behavioral combinator. It is defined as follows:

$$connect : MODULE \times MODULE \mapsto MODULE$$

Let $M_1 = \langle hfg_{M_1}, ports_{M_1} \rangle$ and $M_2 = \langle hfg_{M_2}, ports_{M_2} \rangle$

Then, $connect(M_1, M_2) = \langle hfg_{M_1} \parallel hfg_{M_1}, ports_{M_1} \cup ports_{M_2} \rangle$

Conditions for well-formedness of connections

- No two output ports are connected (avoids contention).
- Every port in the modules being connected is either connected to some other port or made available to the external world. Of course, connected ports could be exported to the external world too (avoids unconnected wires).
- No port is connected to another port within the same module (avoids self loops).

These could be formulated mathematically using the ports and modules but gets cumbersome.

5 Synatatic and Semantic issues of Actions in hopCP

In section 3 we introduced the notion of actions to capture the communication and computation aspects of hardware behavior. In this section, first we shall take a closer look at *ports* and then document the syntactic and semantic issues involving actions in general and *expression actions* in particular. Let us define the following syntactic domains (some of them are being repeated for easy reference):

Primitive Domains:

$$\begin{aligned} INT &= Domain\ of\ Integers \\ BOOL &= Domain\ of\ Booleans \\ IDENT &= Domain\ of\ Identifiers \end{aligned}$$

Derived Domains:

$$\begin{aligned} VAL &= INT \cup BOOL \cup \perp \\ VAR &= IDENT \\ PORTS &= IDENT \end{aligned}$$

Ports or Communication Channels

In section 3 we introduced *control and data actions* to capture the communication aspects of hardware behavior. *Ports* are the *structural* counterparts of the control and data actions. They are concrete in the sense that they are directly realized in hardware via wires, though the actions associated with them are abstract in the HFG (behavioral model).

- Ports could can be annotated with *types*, to denote the values they carry. In the present version of hopCP we will restrict the types of values to booleans and integers.

- Ports can be annotated with directions, i.e. they can be *input*,*output* or *bidirectional*. If $p \in PORT$, then $p!$ classifies p as a *output* port while $p?$ describes p as an *input* port. Bidirectionality of ports is only an abstraction because for a particular action the port can either be an *input* or an *output* (so it should be rightly called time-multiplexed bidirectionality).
- Output control actions are associated with output ports and input control actions are associated with input ports. Similarly with data actions.

Control Actions

Control actions (also referred to by events) are the behavioral models for *control wire encodings* in a hardware system. They are boolean value (either a control action has occurred or has not). The domain of control actions is analogous to the set of ports i.e.

$$C = PORT \text{ (Set of ports)}$$

This means that an input port, $p?$ is an input control action and output port $q!$ is an output control action.

Data Actions

Data actions are the behavioral models for value transfer mechanism in a hardware system. The domain of data actions D has an internal structure:

$$\begin{aligned} D &= DQ + DA \\ DQ &= PORT \times VAR \\ DA &= PORT \times E \end{aligned}$$

DQ represents the domain of input data actions which are also known as *data queries*. Let $p \in PORT$ and $x \in VAR$, then, $p?x$ denotes a data query on the port p. The execution of the action $p?x$ implies, variable x contains the value received from the port p. DA represents the domain of output data action or *data assertions*. Let $p \in PORT$ and $exp \in E$, then, $p!exp$ denotes the assertion of the value denoted by the expression exp on the port p. We use CSP syntax to facilitate easy understanding.

Expression Actions and Modeling Computation in hopCP

In the definition of a HFG to model hardware in hopCP we have introduced the notion of state which was described by control and datapath states. The datapath state and the expression actions together describe the computational aspects of hardware behavior in hopCP.

Formally,

$$\begin{aligned} CS &= IDENT(Set\ \ of\ \ Control\ \ States) \\ DS &= VAR^{*}(List\ \ Domain\ \ of\ \ Variables) \\ S &= CS \times DS(Set\ \ of\ \ States) \end{aligned}$$

We impose the restriction that the identifiers used for control states be *distinct* from the identifiers used for datapath variables.

Abstract Syntax of Expression Actions: Let $e \in E$ (the domain of expression actions) and $x \in VAR$ and $v \in VAL$,

$$e ::= v \mid x \mid \lambda v.e \mid ee \mid e \rightarrow e, e \mid fix\ \ e$$

Note that, we have decided to use the very basic subset of lambda calculus to specify the expression action. In the first version of hopCP we will allow only *tail recursion.* It is found that a very large number of interesting applications can be captured without resorting to *general* recursion.

Also, though λ *abstraction* has been introduced as a basic construct, we do not intend expression actions to be *higher-order* (at least not in the current version of hopCP). It is primarily introduced to elegantly capture *local computation* via *let expressions.* In the actual behavioral description we will use the corresponding subset of Standard ML to describe expression actions.

We provide rules for translating expression actions into HFGs . This once again gives the designer the flexibility to model the behavior either directly in a purely *functional language* and reason about it in terms of HFGs or to specify the behavior in the graphical notation (i.e. HFG) using the expression actions to model functional computation.

6 Interaction of hopCP Flow Graphs and Synchronization

In this section we will take a closer look at the mechanics of synchronization and value communication when HFGs interact. We defined *parcomp* as the behavioral composition operator on flow graphs in section 3 and described three kinds synchronization scenarios which characterize flow graph interaction.

Multiway Synchronization:

Synchronization is said to occur when two modules which are making independent progress, wait for each other to undergo a *rendezvous* on a particular action. Synchronization invariably results in the flow of information from one module to another, either explicitly when the action involved is a data action or implicitly when the action involved is a control action.

Obviously, the question of synchronization does not arise in the case of expression actions, which are used to merely model the computational aspects.

When, more than two modules (represented by their flowgraphs) interact, it is left upto the designer (of the language) to suggest the interaction mechanism. In *point to point* synchronization schemes, advocated in CCS like languages, two reciprocating partners are chosen randomly to undergo synchronization and others are rejected. This brings about nondeterminism.

We adopt the synchronization strategy proposed in CSP and Trace Theory, called *multiway synchronization* with slight modification of our own. In multiway synchronization all the modules which share a common action (control or data) rendezvous and then resume their individual activity. Hence it is deterministic. We insist in hopCP that whenever there is a rendezvous involving more than one module, there be atmost one *active* partner (module willing to perform an output data action or an output control action) and any number of *passive* partners (module willing to engage in an input action). We ensure this by restricting the interconnection of modules in such way that no two output ports are connected.

Many communication architectures in hardware are naturally captured by multiway synchronization: for example communication via a electrical bus between a central controller and several recipients illustrates the utility in SIMD architectures. However, the main attraction for multiway synchronization in hopCP is in using it to derive *lockstep synchronous* behavioral descriptions from HFGs . This is because, multiway synchronization can also be viewed as *barrier* synchronization, which suggests the existence of certain *specialized* actions wherein a subset of the participating modules stop and exchange information (regarding their control state) and make further progress. A *lockstep synchronous* behavior differs from that described by a HFG only in the absence of a *global synchronizing action*, namely the *clock*. So, we find that one can generalize multiway synchronization (or barrier synchronization) in such a way that we get global information transfer via a common action, namely the clock. However, it will not work by just introducing a common clocking action with every action occurrence in a HFG , since we have the potential of *refinement of actions*. This is being currently addressed in our research.

7 Examples in hopCP

Specification of a Two Place Buffer

Define a module $M_1 \in MODULE$ which is a one place buffer whose behavioral description is given by $1PB$ and whose ports are $?p$ and $q!$. The module is shown in figure 4.

$$M_1 \Leftarrow \langle 1PB, \{p?, q!\} \rangle$$

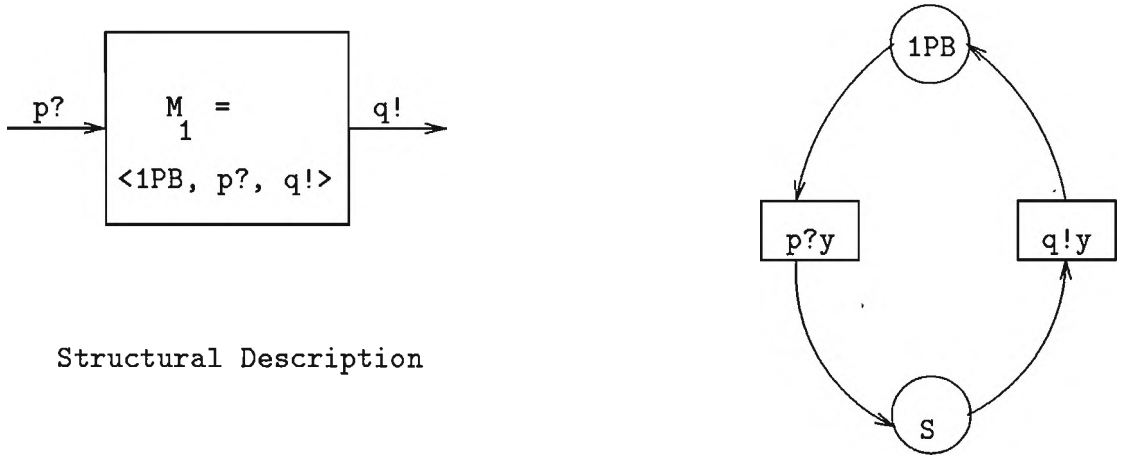


Figure 4: Specification of a one-place buffer in hopCP

The behavioral description is captured by the HFG 1PB which is shown in figure 4.

$$1PB\ [x] \Leftarrow p?y \rightsquigarrow q!y \rightsquigarrow 1PB\ [y]$$

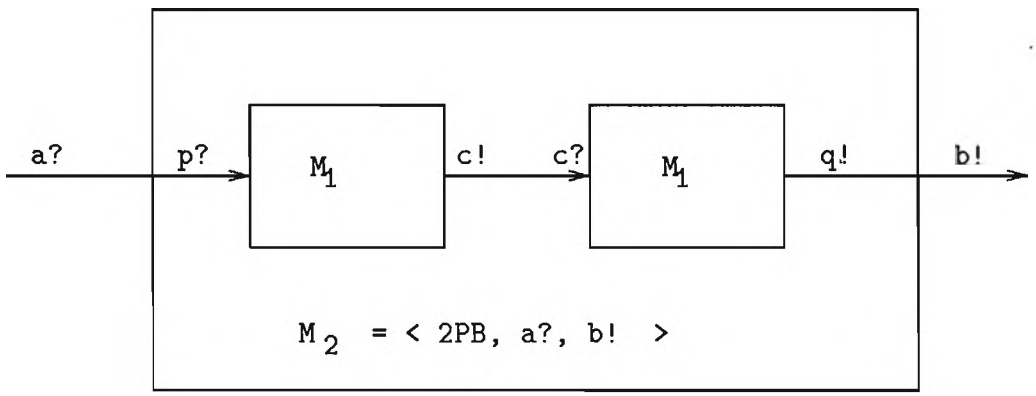
The notation $P\ [x]$ denotes a state, that is the control state data state pair, $\langle P,\ x \rangle$ where $P \in CS$ and $x \in VAR$. As explained in section 5, $p?y \in DQ$ and $q!y \in DA$ denote input data action on port p and output data action on port q respectively.

The structural description of the two place buffer is given by $M_2 \in MODULE$ which is defined as follows

$$M_2 \Leftarrow export(connect(rename(M_1, f), rename(M_1, g)), \{a?, b!\})$$

where f and $g \in \mathcal{R}$ and are defined by the following mapping

$f = \{(p?, a?), (q!, c!)\}$ and $g = \{(p?, c?), (q!, b!)\}$



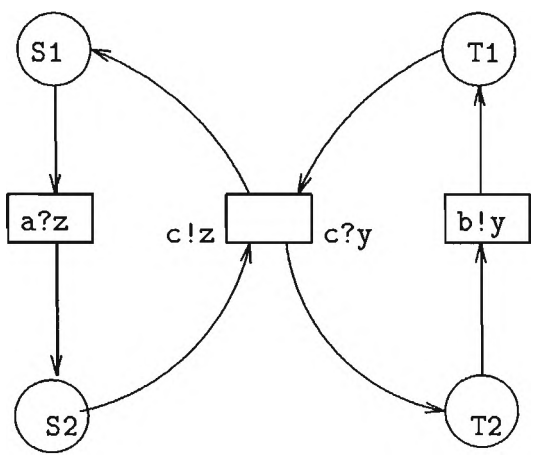


Figure 5: Specification of a two-place buffer in hopCP

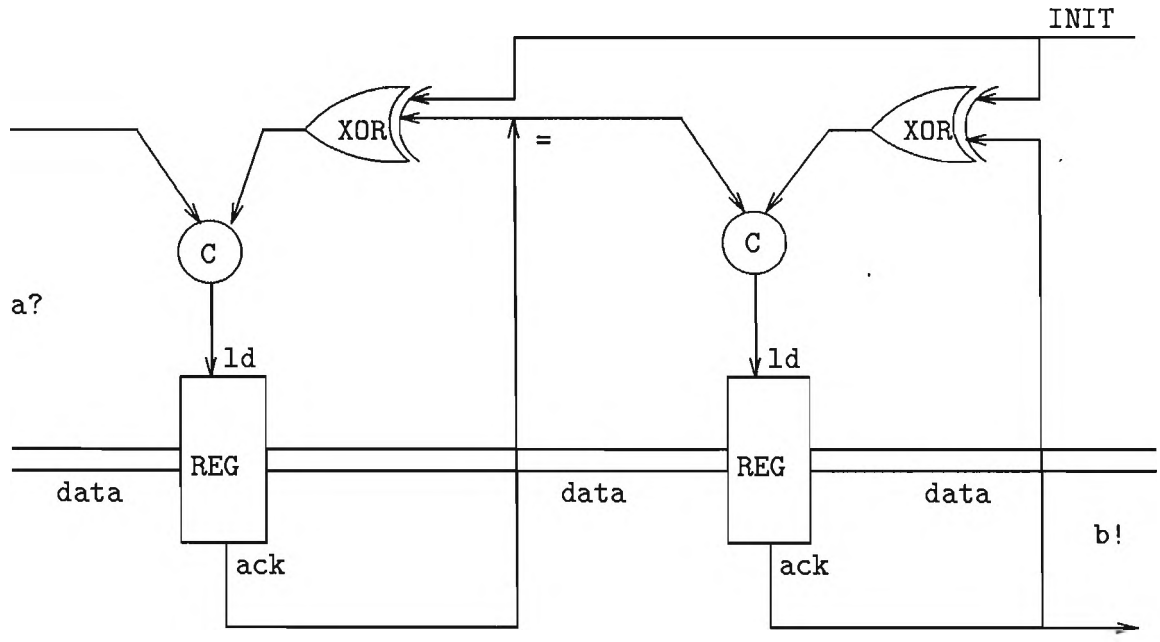


Figure 6: Implementation of a two-place buffer via Action Refinement

Applying the definitions of *export*, *connect* and *rename* from section 4 and the semantics of *progress* and *parcomp* from section 3 we get the *inferred behavior* for the two place buffer as the four tuple $\langle S_{i_{2pb}}, S_{2pb}, Act_{2pb}, \rightarrow_{2pb} \rangle$ (which is also illustrated in figure 5) where

$$\begin{aligned} S_{i_{2pb}} &= \{S1, T1\} \\ S_{2pb} &= \{S1, T1, S2, T2\} \\ Act_{2pb} &= \{a?z, c!z, c?y, b!y\} \\ \rightarrow_{2pb} &= \{(S1, a?z, S2), (\{S2, T1\}, c!z, \{S1, T2\}), (T2, b!y, T1)\} \end{aligned}$$

A speed independent implementation of the circuit using two-phase transition signaling with bundled data based on *action refinement* is shown in figure 6. We have used the circuits (components) described in [1]. Note that details of the compilation strategy are not the focus of this report (they will be described in a forthcoming report).

Specification of an Iterative Multiplier

This example illustrates the specification of a computation oriented piece of hardware. The structural specification is a module $M \in MODULE$ described as follows:

$$M \Leftarrow \langle mult, \{a?, b?, c!\} \rangle$$

where *mult* is a HFG which captures the behavior of the iterative multiplier and is specified as follows:

$$mult\ [x,y] \Leftarrow a?x,\ b?y \rightsquigarrow c!\ multiply(x,y,0) \rightsquigarrow mult\ [x,y]$$

where

```
fun multiply x y z = if (y = 0) then z
                     else case (odd y)
                          true ⇒ multiply x (y-1)(z+x)
                          false ⇒ multiply 2*x (y div 2) z;
```

Note, that this example illustrate the use of *compound actions* and the use of the iterative multiply function as a *expression action.* Translating the above specification into a HFG is fairly routine.

8 Conclusions

In this report we presented the basic motivation behind the design of the language hopCP and described its operational semantics. We illustrated the expressive power of the language by presenting several commonly occurring hardware scenarios in hopCP. Currently, we are focussing on the formal semantics of *action refinement* in hopCP and deriving asynchronous hardware from HFGs .

References

1. Erik Brunvand and Robert F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. In *International Conference on Computer-aided Design, ICCAD 89*, April 1989.

2. Steven D. Johnson. Applicative Programming and Digital Design. In *Eleventh Annual ACM Symposium on Principles of Programming Languages, pages 218-227, 1984*

3. Steven D. Johnson. Digital Design in a Functional Calculus. In *Formal Aspects of VLSI Design* Edited by G.J. Milne and P.A. Subrahmanyam, Elsevier Science Publishers B. V.(North-Holland), Amsterdam, 1986.

4. Ganesh C. Gopalakrishnan, Richard M. Fujimoto, Venkatesh Akella and Narayana Mani. HOP: A Process Model for Synchronous Hardware: Semantics and Experiments in Process Composition. In *Integration: The VLSI Journal, pages 209-247, August 1989*