hopCP: A Concurrent Hardware Description Language

VENKATESH AKELLA
GANESH GOPALAKRISHNAN

UUCS-91-021

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

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

November 25, 1991

Abstract

hopCP is a language for the specification, simulation, and synthesis of hardware systems. hopCP captures the behavior of a hardware system by specifying the causal relationships between actions *that the system can perform. No specific timing discipline is implied by a hopCP specification. Hence, hopCP specifications can be implemented as* synchronous, asynchronous, *or* mixed *synchronous and asynchronous circuits. Salient features of hopCP include nonatomic actions, synchronous and asynchronous styles of value communication, broadcast channels, a purely functional sublanguage to express the computational aspects of hardware behavior, and an efficient tool (called* parComp*) to infer the composite behavior of a collection of hopCP modules. Operational Semantics of hopCP in terms of labeled transition systems is presented. A few examples are described to illustrate the expressive power of hopCP. A summary of the implementation is also presented.*

hopCP: A Concurrent Hardware Description Language

VENKATESH AKELLA* (*akella@cs.utah.edu*)
GANESH GOPALAKRISHNAN† (*ganesh@bliss.utah.edu*)

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract. *hopCP is a language for the specification, simulation, and synthesis of hardware systems. hopCP captures the behavior of a hardware system by specifying the causal relationships between* actions *that the system can perform. No specific timing discipline is implied by a hopCP specification. Hence, hopCP specifications can be implemented as* synchronous, asynchronous, *or* mixed *synchronous and asynchronous circuits. Salient features of hopCP include nonatomic actions, synchronous and asynchronous styles of value communication, broadcast channels, a purely functional sublanguage to express the computational aspects of hardware behavior, and an efficient tool (called* parComp*) to infer the composite behavior of a collection of hopCP modules. Operational Semantics of hopCP in terms of labeled transition systems is presented. A few examples are described to illustrate the expressive power of hopCP. A summary of the implementation is also presented.*

*Supported in part by a University of Utah Graduate Research Fellowship
†Supported in part by NSF Award MIP-8902558

1 Introduction

With the ability to integrate many thousands of transistors on a single VLSI chip, the task of producing error-free and high-performance circuits is an increasingly challenging one. Today, entire VLSI chips are seldom specified in terms of circuit diagrams or truth tables: they are specified using Hardware description languages (HDLs). HDL descriptions are increasingly being used to document the intended behavior of the system, capture design refinements, support formal verification, and are the main input for provably correct high-level synthesis algorithms.

No consensus has been reached on the constructs that an HDL should include. This is despite ongoing standardization efforts [VHD85, SST90]. There are also notable omissions in the HDLs in use today. These include lack of a well specified and simple semantic definition for the HDL, inadequate notations for specifying concurrency, and inadequate support for *system design* issues. We now elaborate on these points, and present our HDL hopCP that is designed to rectify some of these omissions.

Most VLSI circuits are implemented as *synchronously clocked* sequential circuits. In fact, synchronous design is so widespread that many HDLs as well as high-level synthesis algorithms based on these HDLs heavily rely on the synchronously clocking assumption.

It is widely recognized that it is not very practical to design systems beyond a certain size as synchronous systems. The problem of *reliable* (skew free) clock distribution is the most widely cited reason for not building large synchronous systems. There are, however, many more reasons. Most large digital systems behave less like *slaves* that help implement operations such as *add* or *factorial*; they behave more like *concurrent processes* that synchronize, communicate, and coordinate their computations with similar units, to achieve a common computational goal. Therefore, whether a system is built using synchronous circuits or asynchronous circuits, from the point of view of *conceptual modeling* it is often important to be able to specify the system as a collection of concurrent processes. Doing so allows design requirements to be clearly specified, and *ad hoc* design decisions to be avoided early in the design. After all, the decision to *synchronously clock* is only an *implementation decision*, whereby synchronization between concurrent processes is *implicitly* achieved (*i.e.*, not explicitly, using hand-shaking signals.)

Virtually all the popular HDLs available today either omit concurrent process modeling primitives altogether, or provide only very low level primitives. Without the support of high-level concurrency primitives, specifying concurrent behavior can be a nightmare. This is clearly evident from the experience of researchers in Operating Systems in the 60's and early 70's, who faced difficulties such as the ease of introducing deadlocks, the difficulty of

analyzing concurrent programs, etc. In the same vein, descriptions of complex VLSI chips written HDLs that provide only low level concurrency primitives are unreadable, and overall very unreliable.

The *communicating process* paradigm pioneered by Milner [RM89] and Hoare [Hoa85], among other researchers, has become one of the most popular ways of organizing concurrent programs. This view is also gaining in popularity as an HDL primitive [Mar89, BS89, May90]. We also adopt this view in hopCP. Our work provides convenient extensions in the form of *multiway rendezvous* [Cha87] wherein a collection of processes can include one sender and more than one receiver, and they all rendezvous on a channel.

As an alternative to synchronous design, many researchers have proposed a fully *asynchronous* design approach [Mar89, Sut89, BS89, GJ90, Ebe89]. Though highly promising, asynchronous design will, in all likelihood, not entirely supercede synchronous design. In the near future, we can also expect *mixed synchronous/asynchronous* designs [MC80, Cha84] to be built.

Thus, concurrent processes that communicate through rendezvous can either be compiled directly into asynchronous circuits [Mar89, BS89, May90, AG91a] in a fully synchronously clocked style [Ku91, PL91], or in a mixed style. Whichever way concurrent processes are implemented in circuits, it involves the *refinement* of high level actions into partial orders of simpler actions. For example, the rendezvous communication action is refined into a partial order of signal transitions on wires. As another example, in the design of instruction processors, high level operations (*multiply*) get refined into partial orders of simpler operations (*a shift-add loop*). *Hierarchical refinement of time is one of the crucial aspects of VLSI design.* Thus, in addition to including high level concurrency primitives, the HDL of choice for designing large systems must be a language that also permits *actions to be hierarchically refined.* hopCP is such an HDL.

It is very constraining to adhere to just the rendezvous style of communication: the hardware designer may prefer to write specifications directly using other forms of communication and synchronization such as *spin waits* on "status signals", etc. In order to support such styles of specification, hopCP offers the feature of *asynchronous ports.* Another use of asynchronous ports is the following. HDL descriptions using rendezvous communications are eventually compiled into circuits whose components support signal assignments (= signal transitions). A natural organization of such a compiler, especially with a view to facilitate proving its correctness, is to present the compilation algorithm as a collection of *hierarchical action refinement* rules whose correctness can be established separately. Since these rules effect source-to-source transformations, the HDL used has to include both rendezvous actions and signal assignment actions. An example of an asynchronous port is a status signal,

such as the "*buffer empty*" signal for a buffer. This signal is raised by the sender when the buffer actually becomes empty. Once set, the sender can proceed in its computation without waiting for the receiver to sample the signal. Also notice that the receiver can sample *buffer empty* several times; each time it samples the signal, it *does not* involve the sender. Modeling this behavior using rendezvous is awkward because rendezvous is a higher level communication primitive where the sender and the receiver have to wait for each other before they can proceed.

One assumption we make of asynchronous ports is that the value written onto the port by a writer is "instantaneously" seen by all the readers of this port. In an asynchronous hardware implementation, the notion of "instantaneous" translates into some form of *data bundling* constraint [Sut89]. In a synchronously clocked implementation, on the other hand, "instantaneous" may mean 'all readers see the data within the cycle in which the write occurs'.

In the hopCP compilation system, we can often determine through static analysis that the receiver will begin sampling an asynchronous port only after the sender has finished writing on the port. In this case, asynchronous ports can be realized very cheaply, using just a register to hold the data. If we cannot rule out that the *receive* will be concurrent with the *send*, then suitable circuits (*e.g.* an arbiter-based circuit such as an ATS module described in [Kel74]) can be employed. (If the inputs to the ATS module are signaled concurrently, the module effectively *serializes* the inputs and then takes action based on the result of serialization.)

The above is just one of the static analysis techniques that we have developed for synthesizing efficient/simple circuits from hopCP specifications. We use a very similar same static analysis technique to derive efficient circuits for the *choice* construct of hopCP, and also to permit concurrent, and speculative guard evaluation. These details are reported in [AG91d].

To summarize, a system-design oriented HDL must include the following features.

- **Temporal abstraction mechanisms:** By temporal abstraction, we mean "lack of commitment to any specific timing discipline". As actions are temporally refined, specific timing details (*e.g.* two-phase clocking, mixed synchronous/asynchronous) can be incorporated.

- **Flexible Styles of Communication:** Much diversity in VLSI circuits arises due to the communication mechanisms that they use, such as explicit synchronization (via handshake signals), implicit synchronization (via clocks), polling, broadcast, etc. It is important that we do not rule out some of these useful styles of communication in hardware by committing to any one of them in the HDL itself.

- **Functional View of Computation:** There are several styles for specifying computation, such as procedural (*e.g. C*), functional (*e.g.* Pure Lisp), relational (*e.g.* Pure Prolog), etc. Functional languages are particularly appealing for describing computational aspects of hardware because their *referential transparency* (the ease of replacing equals by equals). The implicit parallelism in a functional description helps specify concurrency in computations naturally. This can reduce the expense of data flow analysis.

- **Formal Semantic Description:** Having a simple and well specified formal semantics helps in the following ways:

  - **Provide language reference:** One of the primary objectives of specification-driven design is to establish correctness of the design before indulging in the expensive activity of actually building the circuit. Verification becomes very difficult, if not impossible, if the specification language does not have a clear formal definition.

  - **Assist in Formal Verification:** Hardware verification using theorem proving tools is wasteful to apply to circuit descriptions that contain errors that can be detected more cheaply and automatically through other means; for example, *process composition tools* have been shown to be very useful in detecting many errors automatically [CPS89, GFAM89, GF91]. If no errors are detected, process composition returns an abstracted version of the design which can, then, be subjected to more rigorous verification.

  - **Help Synthesize Efficient Circuits:** Process composition is also employed in hopCP to help synthesize area-efficient circuits. We first obtain an abstracted process. We then analyze it to determine whether its asynchronous ports are used in an exclusive read/write mode, whether its *alternative* commands are deterministic, etc. This information helps us select more efficient circuits as reported in [AG91d].

  - **Assist in Testing:** In addition to establishing correctness, formal reasoning may reveal useful information in the form of *invariants* which can be exploited for test case generation during simulation, incorporation of self-diagnostic features during synthesis, etc.

Organization of the paper

In section 2, we briefly examine related work. In section 3, we briefly describe the hopCP design environment. In section 4, we provide an informal introduction to the language,

with examples. In section 5, we informally describe the structural operators of hopCP. In section 6 and 7, we present the formal semantics of hopCP. In section 8, we discuss the hopCP implementation and in section 9 we provide concluding remarks.

Salient Features and Primary Application Domain

hopCP can be best characterized as a functional language augmented with constructs to specify concurrency and communication explicitly. One of the significant features of hopCP is its lack of adherence to any specific timing discipline. hopCP retains the fundamental state-transition system view of hardware, but is equipped with features to specify computation in a purely functional style, specify concurrency without interleaving, and offer flexible communication schemes.

In hopCP, only the causal relationships between a set of *actions* which capture the behavior of the hardware are specified. Clocking and absolute timing relationships are delayed till the synthesis phase. This makes hopCP eminently applicable for the specification of systems that are comprised of loosely coupled communicating modules, each obeying non-trivial timing protocols. At present, it may not be quite natural to use hopCP for the specification of systolic designs or designs that decompose into regular array structures; this may, however, be a possible future extension, if we incorporate the results of [Joh84] and [She85].

2 Related Work

We have a wide spectrum of HDLs with different objectives and emphasis. The following are some of the successful hardware description formalisms currently in use: trace theory [Ebe89],CSP+probe [Mar89], Occam [BS89] used for the synthesis of pure asynchronous circuits, trace theory [Dil89] for verification of asynchronous circuits, HardwareC [KM90] and ISPS (with variations) [TLW+90] for synthesis of synchronous circuits, functional calculus [Joh84] for synthesis and verification of synchronous circuits, and Verilog HDL [SST90] and VHDL for modeling and simulation of synchronous and asynchronous circuits. It is difficult to place hopCP in this spectrum because it is more like an amalgamation of the essential ideas from the above formalisms. In hopCP the emphasis is in *accurate* modeling of hardware phenomenon and providing facilities for functional simulation and high-level debugging of specifications. We also support synthesis from hopCP specifications [AG91a]. In this paper we focus on the language design criteria and the formal semantics.

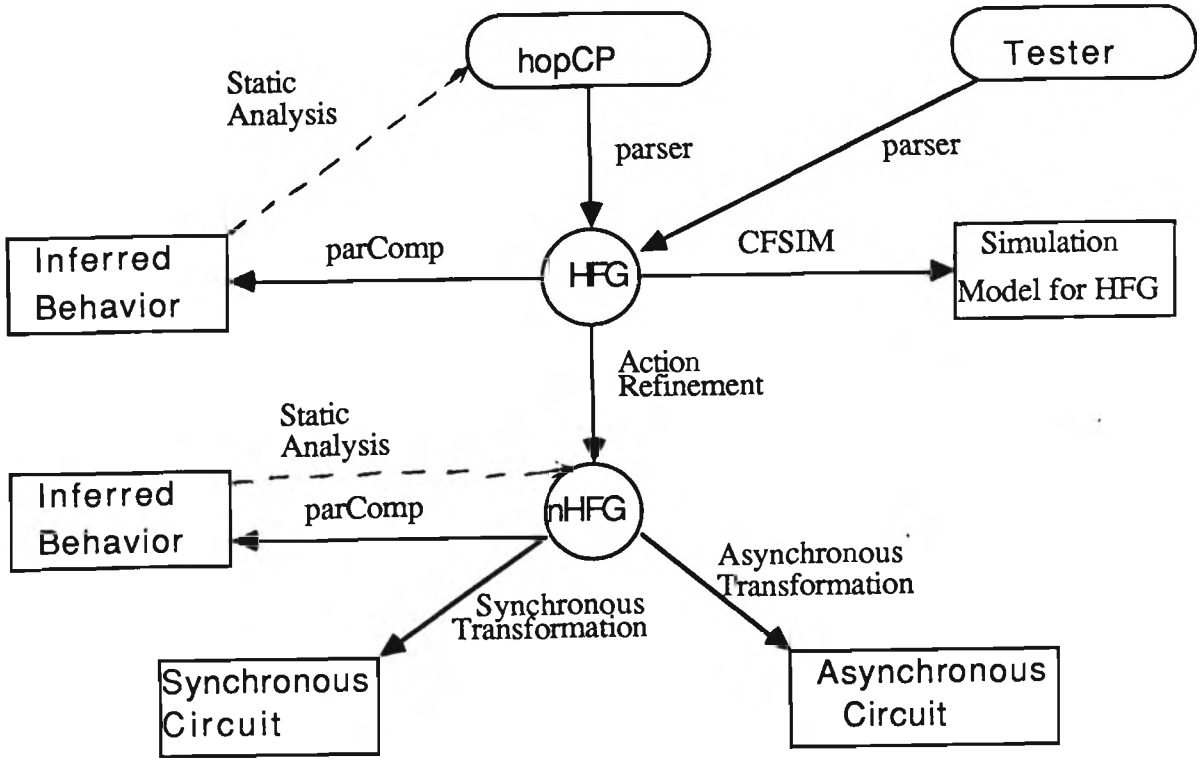

Figure 1: Flow Diagram of the hopCP Design Environment

3 hopCP Design Environment

Figure 1 outlines the VLSI design environment planned around hopCP. The hopCP specifications are parsed and compiled into an intermediate representation called HFG (hopCP Flow Graph) which will be discussed in detail in this paper. Using a *compiled-code concurrent functional simulator generator* called CFSIM, we derive a simulation model for the HFG (currently CFSIM generates a Concurrent ML program module). $parComp$ is a tool which operates on HFGs to *infer* the composite behavior of a collection of hardware modules specified in hopCP. *Action Refinement* is a procedure by which HFGs are translated into hardware. This procedure handles the familiar synthesis tasks of control/data decomposition and resource-allocation. hopCP specifications can be *refined* into purely asynchronous circuits (similar to [BS89]), purely synchronous circuits, or a mixture of both. Static Analysis techniques can be used for performance optimization, at different levels of hierarchy. In this paper we will focus on the specification language hopCP, and its formal semantics.

4 What is hopCP?

hopCP is a notation to describe a concurrent state-transition system called hopCP Flow Graphs (henceforth referred to as a HFGs), augmented with features to model computation in a purely functional style, and mechanisms to support synchronous and asynchronous

communication. In hopCP, hardware is modeled by a structural entity called a *module* which contains two parts (i) a *behavioral* entity called a HFG (hopCP Flow Graph) which captures the state-transition system describing the hardware in question, and (ii) a set of *communication* ports with which the hardware *interacts* with its environment. A hopCP specification has six sections described below out of which only the MODULE and the BEHAVIOR section are mandatory.

(i) The MODULE section introduces the name of the module being described.

(ii) The TYPES section introduces the datatypes of the communication ports used in the module. Currently we support two primitive types namely *bit* and *bitvector*. More complex types can be built using the primitive types. For example, we could define a *byte* as follows: $byte : vector\ 8\ of\ bit$.

(iii) The SYNCPORT section declares all the *synchronous* communication ports used in the specification. A *synchronous* port allows *rendezvous* style communication as in CSP. *Rendezvous* style of communication is characterized by the sender and receiver synchronizing (waiting for each other) before exchanging the information. Synchronous ports could be inputs or outputs which are distinguished by the last character of the portname. For example, $p?$ denotes a port p being used in the input sense while $p!$ denotes the same port being used in output sense. All ports in hopCP are unidirectional i.e. they are either input or output.

(iv) The ASYNCPORT section declares all the *asynchronous ports* used in the specification. An asynchronous port is a *shared variable* which provides value communication without synchronization. Asynchronous ports can be used either as input or output and this can be distinguished syntactically.

(v) The FUNCTION section contains the *user-defined* functions used in the specification. The functions are written in a *first-order* functional language. The syntax of Standard ML of New Jersey is used.

(vi) The BEHAVIOR section describes the state-transition system which captures the behavior of the hardware system being specified. The state-transition system being described is called HFG and is discussed in detail in the following section.

Informal Semantics of HFG

First, we define *State*, which models the state of one *thread* of control flow, and *Transition*, which defines a triple: (i) a collection of *States* (analogous to the input places to a transition

of a Petri net), (ii) a "transition" (analogous to a Petri net transition), and (iii) a collection of post *States* (analogous to the output places of a Petri net transition).

$$\begin{aligned} State &= ProcName \times LocalStore \\ Transition &= \mathcal{P}^{+}(State) \times CompoundAction \oplus Guard \times \mathcal{P}^{+}(State) \end{aligned}$$

where $Guard, CompoundAction$ are syntactic objects which will be defined later and $\mathcal{P}$ *and* $\oplus$ denotes power set and disjoint sum operations on sets respectively.

A hopCP flow graph is formally defined as a record with two fields:

$$HFG = (istate \subseteq \mathcal{P}(State),\ \ trel \subseteq Transition)$$

A record is represented as a 2-tuple. Each component of the record is denoted by a field name and its associated type. For example, *istate* is one field name, and its type (set of allowed values) is any subset of $\mathcal{P}(State)$.

istate denotes the set of *initial* states of the specification, and *trel* denotes the set of *transitions* in the HFG . A *transition* $tr \in Transition$ is a triple $(pre(tr), act(tr), post(tr))$ where $pre(tr)$ denotes a set of states called *precondition* of the transition, $post(tr)$ denotes a set of states called *postcondition* of the transition, and $act(tr)$ denotes the *action* of the transition.

The *execution semantics* of a HFG are similar to that of a *Petri net.* Let $tr \in Transition$; if tr is *enabled* (i.e. execution reaches $pre(tr)$) then the system performs actions $act(tr)$ and the execution reaches $post(tr)$. Note that no notion of clocks or time is being associated with the performance of the actions $act(tr)$. Also note that if more than one $tr \in Transition$ is enabled, they can perform their respective actions *concurrently*, subject to synchronization rules to be discussed later.

Example 1

We illustrate the features of hopCP using a example of a pipeline stage. Figure 2 shows a complete hopCP specification of the pipeline stage. It does not have a ASYNCPORT section. It declares an input synchronous channel a and and output synchronous channel b of type *byte.* Figure 3 denotes the HFG corresponding to the hopCP specification shown in Figure 2 and is textually described as follows:

$$hfg_1 = \{istate = \{(P,[x])\}, trel = \{((P,[x]), a?y, (Q,[x,y])), ((Q,[x,y]), b!(f\ x\ y), (P,[y]))\}\}$$

The above is the syntax of a record literal. Listed within braces ('{,}') are $field\_name_i$ and $field\_value_i$ (separated by =), for every applicable i.

```
MODULE ex1
TYPES
        byte : vector 8 of bit
SYNCPORT
        a?,b! : byte
FUNCTION
        fun f a b =  if (index(a,0)=1) then update(b,2,0) else b;
BEHAVIOR
        P [x] <= a?y ->  b!(f x y) -> P [y])
END
```

Figure 2: hopCP Specification of a Simple Pipeline Stage

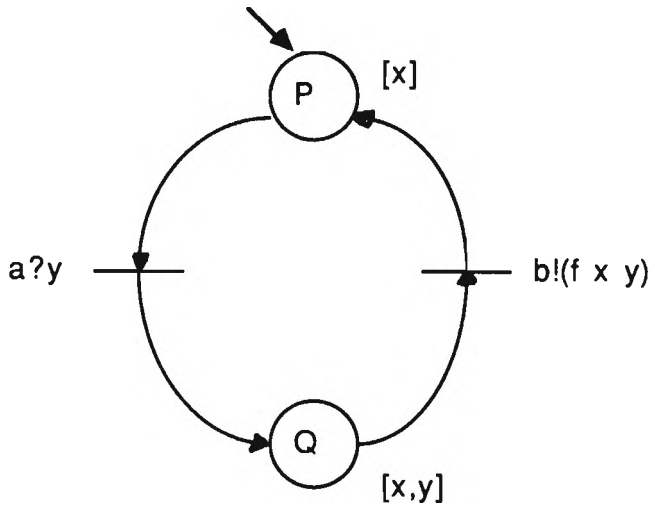

Figure 3: hopCP Flow Graph (HFG) of hopCP Specification in Example 1

It is more convenient to draw pictures to denote HFGs where circles denote the control state names ($ProcName$) and horizontal lines denote the actions. The HFG is interpreted (read) as follows: Module $ex1$ is *initially* in a state $(P, [x])$ where P is the control state (known as $ProcName$) which is analogous to *program counter* in a conventional computer while x is the *datapath state* (also known as $LocalStore$) which is a *snapshot* of its relevant *internal* state. In the state $(P, [x])$ the system can engage in a communication action $a?y$ which will be henceforth referred to as a *data query* and go to a state denoted by $(Q, [x, y])$. Data query $a?y$ denotes a rendezvous on channel a.

Note that by performing the action $a?y$ the internal state of the system is modified to include the value received on channel a which is reflected by the presence of the variable y in the state $(Q, [x, y])$. We could have a synchronous communication action without value communication, i.e. merely $a?$ which is referred to as *input control action.*

We have just described the execution of the system via the transition $((P, [x]), a?y, (Q, [x, y]))$. As a consequence of this execution we find that the transition $((Q, [x, y]), b!(f\ x\ y), (P, [y]))$ is *enabled.* In the state $(Q, [x, y])$, the module can perform the communication action $b!(f\ x\ y)$ and proceed to the state denoted by $(P, [y])$. $b!expr$ where $expr \in EXPR$ is said to be a *data assertion* and represents the *synchronous* communication action of *outputting* the value denoted by the expression $expr$ on the channel b. A data assertion without value communication, for example just $b!$ is called an *output control action.*

In the example, $expr$ is the application of user-defined function f on arguments x (original internal state) and y (received as a consequence of the action $a?y$). The function f could involve arbitrary computation and is expressed in a purely first-order functional language. hopCP has a wide repertoire of bit-level manipulation routines commonly used in hardware systems like $lshift$, $rshift$, $exor$, $subvector$, $index\text{-}vector$, $update\text{-}vector$, $parity$ etc. $(P, [y])$ denotes the fact that the system goes back to the same control state P (as the initial state) but the datapath state is now y instead of x. In a programming language sense, this could be viewed as invoking a function P with an *actual* parameter y for the *formal* parameter x.

Example 2

The specification in Figure 4 illustrates some more features of hopCP. Figure 4 describes a module $ex2$ which declares $TxRDY$ as an output *asynchronous* port (shared variable) of *bit* type. It uses the same function definition f as in the previous example. Informally, module $ex2$ starts in a state $(Q, [x])$, engages in an data query $a?y$, and depending on whether the input value y is even or odd it proceeds to perform the data assertion $b!(f\ x\ y)$ and an asynchronous output action $TxRDY := 1$ and goes back to its initial control state Q with

```
MODULE ex2
SYNCPORTS
        a?,b! : byte;
        c! : byte
ASYNCPORT
         TxRDY! : bit
FUNCTION
        fun f a b =  if (index(a,0)=1) then update(b,2,0) else b;
BEHAVIOR
Q [x] <= a?y ->  ((even y) -> (b!(f x y), TxRDY := 1) ->  Q [y+1])
                 |((odd y) -> c! (subvector(y,0,4)) -> Q [y])
END
```

Figure 4: Illustrating Alternate Behavior and Assignment Actions

its datapath state modified to the value denoted by $y + 1$ or performs $c!(subvector(y, 0, 4))$ and returns to the initial control state Q with y as its datapath state. The behavior has the following new features

1. *Assignment* $apo := expr$ where $apo \in AsyncPorts$ is an assignment action. In module $ex2$, $TxRDY := 1$ is an assignment action which denotes the evaluation of the expression $expr$ (which is 1 in our example) and *transmitting* the value on the asynchronous channel $TxRDY$. An assignment action does not have to *synchronize* with a *receiver* before transmitting the value. In this sense, it is *asynchronous*. Applications of this style of communication include outputting *status* information and modeling system initialization (reset). Misuse of asynchronous communication via shared variables could lead to undesired behavior like metastability and proclivity to deadlock. The details of asynchronous communications and syntactic restrictions to avoid these problems is discussed in section 6.

2. *Compound Actions* A tuple of actions $(a_1, a_2, \ldots, a_m)$ constitutes a compound action and is characterized by the following features:

   (i) $a_1, a_2, \ldots, a_m$ could denote data queries, data assertions, input control actions, output control actions or assignment actions with the restriction that all a_i and a_j should be *non-interfering*, i.e. no two a_i and a_j should use the same channel or try to update the same variable. For example the compound actions $(a?x, a?y, \ldots)$ and $(a?x, b?x, \ldots)$ are not permitted.

```
MODULE ex3
SYNCPORTS
        a?,b! : byte;
        b?,c! : byte
FUNCTION
        fun f a b =  if (index(a,0)=1) then update(b,2,0) else b;
        fun g a b =  if (index(a,0)=0) then update(b,2,0) else a;
BEHAVIOR
        (P [x1] <= a?y1 -> b!(f x1 y1) -> P [y1])
                        ||
        (Q [x2] <= b?y2 -> c!(g x2 y2) -> Q [y2])
END
```

Figure 5: hopCP Specification Illustrating Parallel Behavior

(ii) Let $(s, (a_1, a_2, \ldots, a_m), s') \in Transition$, the execution of the system in a state s corresponds to performing actions $(a_1, a_2, \ldots, a_m)$ concurrently and going to state s'. The execution of the system via a compound action is analogous to that of the *cobegin/coend* statement of concurrent programming languages.

In $ex2$, $(b!(f\ x\ y), TxRDY := 1)$ denotes a compound action.

3. *Choice* In hopCP conditional behavior is captured by *guards* and *choice* construct (represented by '|' in the textual syntax of hopCP). Guards are either boolean expressions, data queries (or input control actions) or both. We do not allow data assertions, output control actions, or assignment actions in guards. The informal semantics of the choice construct is as follows: all the guards are evaluated in parallel; the guard which succeeds (a guards succeeds if its boolean expression evaluates to true and if the input communication action succeeds) is picked and the execution moves to the corresponding state. If none of the guards succeeds, it denotes a *error* in the specification, and, the system halts. If more than one guard succeeds, any one of them can be picked. This introduces *nondeterminism* in hopCP.

   In example $ex2$, $(even\ y)$ and $(odd\ y)$ are the guards which control the system behavior. Expression guards can be specified with the help of user-defined functions in the $FUNCTION$ section of the specification.

Example 3

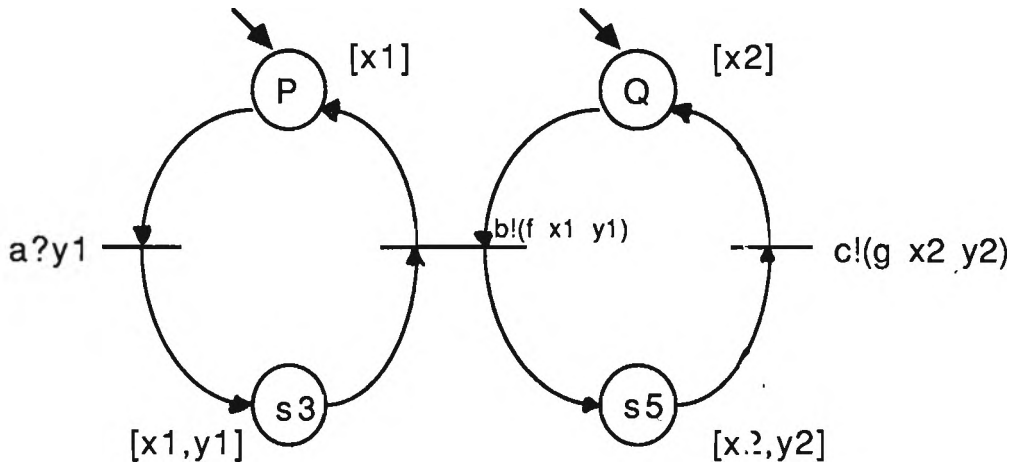

Figure 6: HFG of hopCP Specification in Example 3

The last two examples were basically *sequential* in nature except for the restricted form of concurrency introduced by compound actions. Figure 5 is a hopCP specification of a concurrent system with synchronization and value communication. It captures two *independent* threads of activities corresponding to two stages of a pipeline *coupled* by a rendezvous on the synchronous communication channel b. The stage described by P is capable of performing an data query $a?y1$ and a data assertion $b!(f\ x1\ y1)$ while the stage described by Q can first engage in a data query $b?y2$ and then perform a data assertion on channel c. The actions $a?y1$ and $c!(g\ x2\ y2)$ can be performed independently (hence concurrently) while the actions $b?y2$ and $b!(f\ x1\ y1)$ have to be performed synchronously. This is captured in the HFG shown in figure 6.

The initial states $(P, [x1])$ and $(Q, [x2])$ are marked by arrows. Initially, $a?y1$ can be performed by stage P while Q waits on action $b?y2$. Once $a?y1$ is completed, both stages P and Q can engage in $b?y2$ and $b!(f\ x1\ y1)$ which results in the datapath variable $y2$ in stage Q getting a value denoted by the expression $(f\ x1\ y1)$ (referred to as value communication). This is depicted by the annotation on the arc leading to control state $s2$. Once this synchronous activity is complete, stage Q can engage in $c!(g\ x2\ y2)$ and stage P can engage in $a?y2$ concurrently.

This illustrates synchronization and value communication between two agents and is known as *two-way rendezvous*. In this example, the no compound actions are involved. If the actions involved are compound actions, things become slightly more complicated and we ascribe a semantics to it in the later part of this paper.

```
 BEHAVIOR
(P [x1] <= a?y1 -> b!(f x1 y1) -> P [y1])
||
  ((Q [x2] <= b?y2 -> c!(f x2 y2) -> Q [y2])
   ||
   (R [x3] <= b?y3 -> d!(f x3 y3) -> R [y3])
  )
END
```

Figure 7: hopCP Specification Illustrating Multiway Rendezvous

Example 4

Finally we present an example which illustrates the notion of *multiway rendezvous* in hopCP. Synchronization and value communication shown in the previous example involved two agents (one performing a data query, and one capable of the corresponding data assertion). Multiway rendezvous is said to occur when there is more than one agent willing to perform a data query corresponding to a data assertion. (note that the converse of the situation that of more than one agent asserting a value on the same channel does not make much sense) Multiway rendezvous is a powerful notion which facilitates the specification of a wide variety of concurrent algorithms very naturally [Cha87]. It subsumes *broadcast* style of communication (point to multipoint communication) which is very natural in hardware, but not supported by many popular HDLs currently being used for synthesis. It does not mean that these things are *impossible* to specify without multiway rendezvous; but, it becomes awkward to model them in terms of two-way rendezvous. Figure 7 shows a hopCP specification (just the behavior section is shown for convenience). It illustrates multiway rendezvous on channel b.

Initially, only the stage P can make any progress by engaging in $a?y1$. Once this is complete, a multiway rendezvous on channel b is possible. This involves agents P, Q and R waiting for each other and once all the of them *arrive*, agent P transmits the value denoted by the expression $(f\ x1\ y1)$ on channel b which is received by agents Q and R and bound to their internal variable $y2$ and $y3$ respectively and then P, Q and R proceed to perform their next actions.

The multiway rendezvous advocated in hopCP is simpler than that in the protocol specification language *LOTOS* [LOJF88], in the sense that the multiway rendezvous and its participants can be *statically* determined by a simple analysis. This is because we do not

have dynamic process creation in hopCP.

5 Structural Descriptions in hopCP

In the previous section we said that the basic entity being modeled in hopCP is a *module* which consists of a set of channel names and a concurrent state-transition system (HFG). In this section, we will describe ways to specify an *interconnection* of such modules and a mechanism for *abstraction* in hopCP. Interconnection of two modules M_1 and M_2 is specified by a *renaming* function on the channel names and is expressed by the *rename* operator. Abstraction in the sense of controlling the visibility of actions is achieved by the *export* operator. *connect* operator *puts* together modules (whose interconnection is specified by the rename operator). After performing *basic* safety and realizability checks it *infers* the composite behavior of the whole system. The behavioral inference is done by *composing* the HFGs of the constituent modules by appealing to a procedure called *parComp*. Details of *parComp* will be described shortly.

6 Formal Semantics of hopCP

In this section we will present the *operational semantics* of hopCP in the style advocated in [Hen90]. The semantics presented below serves two objectives: (i) defines an *interpreter* for the language, by defining all possible *labeled-moves* of the underlying state-transition system, and (ii) defines a *compiler* for the language, by constructing the *underlying* HFG which is the intermediate representation for the hopCP design framework.

The semantics of each major feature of the language is presented separately by specifying the constituent syntactic categories, abstract syntax and the underlying transition relation. The organization of the semantic definition is as follows: We use three transition relations $\xrightarrow{ca^+}_{CA}$, $\xrightarrow{ca^+}_{proc}$, $\xrightarrow{ca^+}_{beh}$ and two expression evaluation functions $\Longrightarrow_e$, and $\Longrightarrow_{be}$, in the discussions below. The type signatures and formal definition will also be given shortly. $\longrightarrow_{beh}$ is the top-level transition relation which takes an abstract syntax tree denoting a hopCP behavioral expression and the associated process and function declarations and returns a HFG . In doing so, it invokes $\xrightarrow{ca^+}_{proc}$, which *incrementally* constructs the HFG by enumerating all possible moves in the HFG . $\xrightarrow{ca^+}_{proc}$, in turn, appeals to $\xrightarrow{ca^+}_{CA}$ transition relation to *reduce* the compound actions. Compound actions in hopCP contain expressions, which are evaluated by $\Longrightarrow_e$ *and* $\Longrightarrow_{be}$. The semantics will be presented in a bottom-up fashion. To facilitate reading, we present the entire abstract syntax, list of syntactic categories, type signatures of the various transition relations and helping-functions used in the

semantic definition in the appendix.

6.1 Expression Language

The expression language underlying hopCP is a first-order functional language with integer (treated as bitvector) and boolean as the primitive types. Notable omissions with respect to a standard functional languages are user-defined datatypes and higher-order functions. The expression language is now defined.

Primitive Syntactic Categories

In the following, VAL is the category of values, VAR of non-boolean variables (e.g. integer), $BVAR$ of Boolean variables, $AsyncPorts$ of asynchronous ports, and d of data path state variables, which are part of VAR. All other categories are explained in the BNF abstract syntax.

$$
\begin{array}{llll}
v & \in VAL & e & \in EXPR \\
x & \in VAR & bx & \in BVAR \\
be & \in BEXPR & FD & \in FunDecl \\
vop & \in VectorOp & op & \in ArithOp \\
bop & \in BooleanOp & F & \in FunName \\
ap & \in AsyncPorts & d & \in DPSVAR \subset VAR
\end{array}
$$

Abstract Syntax of Expressions

The (abstract) syntax of function declarations is given by the production rules for FD, and that for expressions is given by the rules for e. Expressions (e) can be values, variables, expressions using a binary operator op, let expressions, primitive function applications, if, user-defined function applications (F), and an asynchronous port identifier (ap) standing alone. The remaining expression categories are obvious.

$$
\begin{array}{rcl}
FD & ::= & F(x_1, x_2, \ldots, x_k) = e \mid F(x_1, x_2, \ldots, x_k) = e, FD' \\
e & ::= & v \mid x \mid e' \; op \; e'' \mid \underline{let}\; x = e \;\underline{in}\; e' \mid vop(e', e'', e''') \\
 & & \mid \underline{if}\; be \;\underline{then}\; e \;\underline{else}\; e' \mid F[e_1, e_2, \ldots, e_k]\,(where\ arity(F) = k) \mid ap \\
be & ::= & bx \mid True \mid False \mid be' \; op \; be'' \mid Not\ be \mid equal(e, e') \\
op & ::= & + \mid - \mid * \mid / \mid rshift \mid lshift \mid index \\
bop & ::= & and \mid or \mid nand \mid nor \mid exor \\
vop & ::= & update \mid subvector
\end{array}
$$

Semantic Domains

Using the above syntactic objects, we will present the semantics of expression evaluation in hopCP. Note that we have augmented a standard first-order functional language with primitive operators to manipulate bitvectors. *LocalStore* provides the environment for the evaluation of expressions involving datapath variables (denoted by $DPSVAR$), while *GlobalStore* gives the binding of the asynchronous channels (which are treated as variables). Asynchronous channels can be used in expressions with the restriction that they are used strictly in the *read-only* sense. They cannot be bound inside an expression.

$$\begin{aligned} \sigma_l &\in LocalStore = DPSVAR \mapsto VAL \\ \sigma_g &\in GlobalStore = AsyncPorts \mapsto VAL \end{aligned}$$

Evaulation Semantics of Expressions

$\Longrightarrow_e$ and $\Longrightarrow_{be}$ take an expression and its environment, and return a integer (bitvector) or a boolean, respectively, corresponding to the value of the expression. Their definition is fairly routine and is omitted to conserve space.

$$\Longrightarrow_e : (FunDecl \times LocalStore \times GlobalStore \times EXPR) \mapsto VAL$$

$$\Longrightarrow_{be} : (FunDecl \times LocalStore \times GlobalStore \times BEXPR) \mapsto Boolean$$

6.2 Actions and Compound Actions

In this section we will present the syntax and semantics of the various *actions* used in hopCP.

Additional Syntactic Categories

Synchronous output ports, synchronous input ports, and compound actions have the following syntactic categories:

$$\begin{aligned} spo &\in SyncOutputPorts \quad spi \in SyncInputPorts \\ ca &\in CompoundAction \end{aligned}$$

Abstract Syntax of Actions and Compound Actions

The syntax of actions and compound actions is presented below. A compound action *ca* is interpreted as a set of actions, each action being of type *a*. For example, the *compound action*

written syntactically as $(p?, q!)$ denotes the set of actions $\{p?, q!\}$, and the *compound action* written syntactically as $a?$ denotes the set of actions $\{a?\}$. dq denotes receiving a value on synchronous input port spi, da denotes asserting the value denoted by the expression e on the synchronous output port spo, and aa denotes asserting the value denoted by expression e on asynchronous output port ap.

$$\begin{aligned} a &::= dq \mid da \mid aa \mid spi? \mid spo! \\ ca &::= a \mid a, ca \\ dq &::= spi?x \\ da &::= spo!e \\ aa &::= ap := e \end{aligned}$$

Semantics of Actions and Compound Actions

The semantics of actions and compound actions is captured through the transition relation

$$\begin{aligned} \xrightarrow{ca^+}_{CA} \; : \; & (FunDecl \times LocalStore \times GlobalStore \times CompoundAction) \\ & \mapsto (LocalStore \times GlobalStore) \end{aligned}$$

Here,

$$ca^+ = ca \cup \{\epsilon\}.$$

Also define

$$a^+ = a \cup \{\epsilon\}$$

which will be used later.

Next we present the inductive definition of $\xrightarrow{ca^+}_{CA}$ for each action category.

6.2.1 Synchronous Communication Actions

There are two types of synchronous communication actions in hopCP: actions with value communication and synchronization and those with only synchronization (no value communication). For each action, we have both the input (query) and output (assertion) counterparts and we will present separate rules for both. The following are rules for synchronous communication action *without* value communication:

(i) Synchronous Input without value communication

$$\frac{}{(FD, \sigma_l, \sigma_g, spi?) \xrightarrow{spi?}_{CA} (\sigma_l, \sigma_g)}$$

(ii) Synchronous Output without value communication

$$\frac{}{(FD, \sigma_l, \sigma_g, spo!) \xrightarrow{spo!}_{CA} (\sigma_l, \sigma_g)}$$

Note that these actions do not affect the environment (σ_l, σ_g). We retain the labels $spi?$ and $spo!$ to record the fact that these communications were engaged in.

Next we present rules for actions which reflect value communication. Note that only σ_l is modified. The effect of a data query $spi?x$ is to undergo a synchronization with an agent producing a value v on the channel spi, and modify the localstore σ_l to reflect this value. On the other hand, a data assertion $spi!e$ involves first evaluating the expression e, and then undergoing a rendezvous with an agent willing to accept a value on the channel spi.

(i) Rule for synchronous value input:

$$\frac{}{(FD, \sigma_l, \sigma_g, spi?x) \xrightarrow{spi?v}_{CA} (\sigma_l[v/x], \sigma_g)}$$

(ii) Rule for synchronous value output:

$$\frac{(FD, \sigma_l, \sigma_g, e) \Longrightarrow_e v}{(FD, \sigma_l, \sigma_g, spo!e) \xrightarrow{spo!v}_{CA} (\sigma_l, \sigma_g)}$$

The synchronous communication actions outlined in this section need the cooperation of other agents to *succeed.*

6.2.2 Asynchronous Communication Actions

Motivation and Definition: An output asynchronous action in hopCP is written as $aa := e$. The complementary (*i.e. input*) action is indicated by the use of the asynchronous port identifier (*e.g.* aa in the above example) in hopCP expressions. It denotes the fact that an asynchronous input is first done through the asynchronous channel, and the value thus obtained is used in place of the channel name. Assignment actions provide communication without explicit synchronization. (Synchronization must be indirectly guaranteed.) Asynchronous communication in hopCP is characterized by *nonblocking* senders and *nonblocking* receivers. The sending party deposits the value denoted by the expression e on port ap whenever called upon to do so, and the receiver picks up the value whenever it is asked to. A sender can also write two values in succession without the receiver reading in between. The receiver can read the same value twice, without the sender writing in between.

Naive implementations of the asynchronous communication mechanism can lead to well-known synchronization problems. For example, since the sender and receiver are not synchronized, it is possible to *attempt* a read while the asynchronous port is being written. In a naive hardware implementation, this can result in the sampling of a voltage in between that for logic 0 and that for logic 1, which can lead to metastability. Similarly, since the read and write may happen in any order, it is possible to read the "*old* value" from the port, which could possibly lead to *deadlock*.

An asynchronous action (read/write) is said to be *safe* if all *accesses* i.e. (read/writes) to the asynchronous port are *serial*. Usages of asynchronous communication actions are checked for safety by the hopCP analyzer. This is done by performing a *reachability analysis* on the *inferred behavior* of the constituent hopCP modules [AG91d]. The inferred behavior is obtained very efficiently via a tool called *parComp* which will be discussed in detail later in this section. In the graph of reachable markings, it is then seen whether the asynchronous actions can be enabled simultaneously, or not. Basically, asynchronous actions are employed in hopCP specifications that use synchronous communication actions also. It is these synchronous communication actions that help define causal orderings among all the actions in the specification, and, indirectly, make certain asynchronous actions serial. For example, in the following specification fragment,

```
P[x] = (asyncport:=E) -> p! -> ...
||
Q[y] = p? -> (  <guard1 using asyncport> -> ...
              | <guard2 using asyncport> -> ... )

(* Here, guard1 and guard2 must be mutually exclusive and exhaustive. *)
```

the use of the synchronous communication actions `p!` and `p?` help order the write and read on `asyncport`.

If we cannot rule out that the *receive* will be concurrent with the *send*, then suitable circuits (*e.g.* an arbiter-based circuit such as an ATS module described in [Kel74]) can be employed. If the inputs to the ATS module are signaled concurrently, the module effectively *serializes* the inputs and then takes action based on the result of serialization. Another precaution to be obeyed is in connection with guards. If an asynchronous port is employed in guard $g_i, 0 \leq i \leq N-1$ of a choice construct, then the collection of guards g_0 through g_{N-1} must together account for all possible values on the asynchronous port; in other words, the guards must be exhaustive with respect to the values. Usually this situation arises when spin-waits are implemented, as shown in the boxed specification fragment that comes next.

Uses of Asynchronous Communication: Asynchronous communication is useful in situation where explicit synchronization (sender and receiver waiting for each other) is either awkward or unnecessary. Asynchronous communication can be used in the following scenarios which are quite common in hardware description:

- *Status Signals:* Status signals are typically set once and *optionally* read many times by different modules. It is awkward to enforce synchronization on status signals. They are best modeled as asynchronous actions.
- *Busy Waiting:* Busy waiting is characterized by a *nonblocking* sender and a *blocking* receiver (*waiting for a condition*). Busy waiting can be implemented in hopCP using asynchronous actions as shown in the specification fragment, below:

```
ASYNCPORT
        TxRDY? : bit
BEHAVIOR
       P [] <=  (IsTrue TxRDY) -> Q []
              |(not (IsTrue TxRDY)) -> P []
END
```

In state P, the system waits for `TxRDY` input to become *high*. Busy waiting *inherently* violates *safety* since there is a possibility of the writer accessing the asynchronous port while the read is in progress. (Unless this is permitted, the spin-wait condition, once found to be true, will remain true forever. While one process is spin-waiting, the other process performs an asynchronous port output, thus causing the spin-wait to exit.) As mentioned before, the guards must be *exhaustive*.

Restrictions: We impose the following restrictions on asynchronous communication in hopCP, mainly to *tune* the construct to hardware applications and facilitate a direct circuit implementation.

1. Only one agent can *own* an asynchronous channel; *i.e.*, only one agent is allowed to perform an *assignment* action (use it in the output sense) on a given asynchronous channel. However, there could be several agents using the channel in an *input* sense, *i.e.*, willing to read the asynchronous port. Just as in the synchronous communication scenario, the agent which is the sender (assignment) and the agents which are receivers (input) are *statically* determined.

2. The usage of asynchronous actions in choice constructs will be made only under the condition that the guards are exhaustive, as mentioned above.

3. An asynchronous channel is assumed to have a capacity of *one* which means that an asynchronous channel is modeled as a *assignable* location. Every assignment overwrites the existing value (much like the assignment in an imperative language).

Semantics: The dynamic semantics of the assignment action are given by the following transition rule

$$\frac{(FD, \sigma_l, \sigma_g, e) \Longrightarrow_e v}{(FD, \sigma_l, \sigma_g, (ap := e)) \xrightarrow{\epsilon}_{CA} (\sigma_l, \sigma_g[v/ap])}$$

The global store (σ_g) is being explicitly carried around precisely to capture the asynchronous value communication via assignment actions. An assignment action merely updates the corresponding location. Since, σ_g is shared by the whole system, anyone interested can *read* it without undergoing a synchronization (hence the asynchrony). Notice that the "label" of this transition is ϵ, meaning that there are no observable effects (other than the modification of σ_g, of course).

Implementation Hints: If *safety* is guaranteed, an asynchronous port can be implemented by a conventional latch. Otherwise it can be implemented using an ATS module (fully described in [Kel74]). One possible implementation of an ATS module supports the following operations: *write_zero*, *write_one*, and *test*. It has the property that if more than one of its commands are requested simultaneously, the module *serializes* the requests using an internal arbiter.

6.2.3 Compound Action

Compound actions introduce a restricted form of parallelism in the form of local fork-join activity. Synchronization or value communication among the constituent actions of the compound action is not permitted. The components of a compound action are restricted to be primitive actions.

The chief motivation for introducing compound actions in hopCP was to have something analogous to *synchronous parallelism* which is common in clocked systems. Also, we find that granularity of a primitive action (data query, data assertion) was too fine to make high-level modeling too cumbersome. Since a compound action is merely a collection of primitive which can all be done *concurrently*, its semantics is captured by by the $\xrightarrow{ca^+}_{CA}$ relation with the following additional rules.

$$(FD, \sigma_l, \sigma_g, a) \xrightarrow{a^+}_{CA} (\sigma_l', \sigma_g'),$$

$$\frac{(FD, \sigma_l, \sigma_g, ca) \xrightarrow{ca^+}_{CA} (\sigma_l'', \sigma_g'')}{(FD, \sigma_l, \sigma_g, (a, ca)) \xrightarrow{reduce(a^+, ca^+)}_{CA} (\sigma_l' \cup \sigma_l'', \sigma_g' \cup \sigma_g'')}$$

: Side conditions:

1. The variable assigned by a is *not* assigned by *any other* $a' \in ca$.
2. The intersection of the domains of σ_l' and σ_l'' is equal to the $\emptyset$. Therefore the union of the mappings σ_l' and σ_l'' is a well-defined operation that yields a new mapping.
3. The intersection of the domains of σ_g' and σ_g'' is equal to the $\emptyset$.

Here, $reduce(a^+, ca^+)$ computes the resultant label of the transition as follows: if there are non-ϵ actions in either a^+, or ca^+, then form a compound action out of the actions in a^+ and ca^+, after eliminating all the ϵs; else, return an ϵ. Note, that the same environment (σ_l, σ_g) is being distributed to all the components of a compound action which means the components of the compound action should all be non-interfering; *i.e.*, they should not share any channels and should do not access (read/write) the same asynchronous channels.

6.3 Sequential and Alternate Behavior

In this section, the semantics of sequential ($\rightsquigarrow$) and alternate ($[\!]$) constructs of hopCP will be presented. (in the textual syntax they are `->` and `|` respectively).

Additional Syntactic Categories

$$\begin{array}{llll} PD & \in & ProcDecl & P \in ProcName \\ g & \in & Guard & cproc \in ChoiceProc \\ proc & \in & Proc & \end{array}$$

The following is the abstract syntax of the alternate and sequential behavior in hopCP.

Abstract Syntax

One process declaration consist of the process name P followed by formal parameters, the $\Leftarrow$ symbol, and the process body, *proc*. PD consists of several process declarations. Process body, *proc*, is a sequential process consisting of *ca*, the $\rightsquigarrow$ symbol which denotes sequencing, and *proc*; or it is a *process call*, written $P[e_1, e_2, \ldots, e_m]$. A *choice process*, *cproc*, is a guard followed by *proc*, or several *cproc*s separated by $[\!]$. Finally, guards, g, are described.

$$
\begin{aligned}
PD &::= (P(d_1, d_2, \ldots, d_m) \Leftarrow proc) \mid P(d_1, d_2, \ldots, d_m) = proc, PD' \\
proc &::= ca \rightsquigarrow proc' \mid P[e_1, e_2, \ldots, e_m](where\ arity(P) = m) \mid cproc \\
cproc &::= g \rightsquigarrow proc \mid cproc \,[\!]\, cproc' \\
g &::= be \mid dq \mid spi \mid be, dq \mid be, spi
\end{aligned}
$$

Additional Semantic Domains

The domains of *Transition* and *State* are now introduced.

$$
\begin{aligned}
Transition &= \mathcal{P}^+(State) \times (CompoundAction \oplus Guard) \times \mathcal{P}^+(State) \\
State &= ProcName \times LocalStore
\end{aligned}
$$

A hopCP flow graph is formally defined as a record with two fields:

$$HFG = (istate \subseteq \mathcal{P}(State),\ trel \subseteq \mathcal{P}(Transition))$$

Given $h \in HFG$, $h.istate$ denotes the initial states of h and $h.trel$ its set of transitions.

The semantics of *proc* is given in terms of the transition relation $\xrightarrow{ca^+}_{proc}$ and helping functions $csName$ and $addtr2hfg$, whose type signatures are:

$$
\begin{aligned}
\xrightarrow{ca^+}_{proc} &: (ProcDecl \times FunDecl \times State \times Proc \times GlobalStore \times HFG) \mapsto \\
&\quad (ProcDecl \times FunDecl \times State \times Proc \times GlobalStore \times HFG) \\
csName &: Proc \mapsto ProcName \\
addtr2hfg &: HFG \mapsto (\mathcal{P}^+(State) \times CompoundAction \times \mathcal{P}^+(State)) \mapsto HFG
\end{aligned}
$$

$$(addtr2hfg\ \ hfg\ \ tr) = hfg[(hfg.trel \cup \{tr\})/hfg.trel]$$

Function $csName$ is any one-to-one mapping from the domain of abstract syntax trees of hopCP to $ProcName$. It is used to merely assign a unique name to each abstract syntax tree. In our implementation, $csName$ extracts the "control state name" of the root node of the abstract syntax tree given to it as an argument. Function $addtr2hfg$ adds a transition to a given HFG, returning the resulting HFG.

6.3.1 Rules for Sequential Behavior

(i) This rule defines the semantics of a *process call.* It says that a process call is processed similar to how a procedure call is processed in conventional programming languages. The actual parameters $e_1, e_2, \ldots, e_m$ are evaluated (using the evaluation relation for expressions) and substituted for the formal parameters $d_1, d_2, \ldots, d_m$ and the body of the process declaration *proc* is invoked. The notation $x[new_i/old_i, \ldots]$ says that x must be subject to the substitutions indicated in $[new_i/old_i, \ldots]$ and the result must be returned. Substitution lists always have $old_i \neq old_j$ for $i \neq j$.

$$\frac{(FD, \sigma_l, \sigma_g, e_i) \Longrightarrow_e v_i, 1 \leq i \leq m}{(PD, FD, (P'', \sigma_l), P[e_1, e_2, \ldots, e_m], \sigma_g, hfg) \xrightarrow{\epsilon}_{proc} (PD, FD, (P, \sigma_l'), proc, \sigma_g, hfg[P/P''])}$$

where

$$\begin{aligned} (P[d_1, d_2, \ldots, d_m] &\Leftarrow proc) \in PD \\ \sigma_l' &= \sigma_l[v_1/d_1, v_2/d_2, \ldots, v_m/d_m] \end{aligned}$$

Notice that $addtr2hfg$ adds one transition to hfg. This shows how we use the operational rules to capture the *compilation* of hopCP descriptions into HFGs. In addition to defining a labeled move, this rule is also showing how HFGs (the result of compiling hopCP programs) are incrementally built. All uses of $addtr2hfg$ in the rest of the paper will be to support the compilation.

(ii) This rule captures the sequential progress of the system by the execution of an compound action *ca*.

$$\frac{(FD, \sigma_l, \sigma_g, ca) \xrightarrow{ca}_{CA} (\sigma_l', \sigma_g')}{(PD, FD, (P, \sigma_l), (ca \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{ca}_{proc} (PD, FD, (P', \sigma_l'), proc, \sigma_g', (addtr2hfg \;\; hfg \;\; tr))}$$

where

$$\begin{aligned} P' &= (csName \;\; proc) \\ tr &= (\{(P, \sigma_l)\}, ca, \{(P', \sigma_l')\}) \end{aligned}$$

6.3.2 Rules for *Alternate* Behavior

Alternate behavior is expressed in hopCP using guards. It is clear from the abstract syntax (shown above) that guards could be Boolean expressions, data query (with or without value communication) or a combination of both. Multiple data queries are not allowed in guards. Output actions (assertions and assignments) are not allowed in guards, in the current version of hopCP, either, because of their propensity to induce *deadlocks* if used carelessly. However, input asynchronous actions can be used in guards. The semantic rules for alternate behavior are straightforward: the antecedent of the rule evaluates the guard; if the ith guard succeeds, the program coming after the ith guard is (which is a $proc \in PROC$) is chosen for execution.

The following rules capture the semantics of alternate behavior for the various cases that arise. Note that evaluation of guards does not change σ_g *global store*. Local store σ_l changes only if there is a data query involving value communication.

(i) This rule explains how a guard containing only a Boolean expression is handled.

$$\frac{(FD, \sigma_l, \sigma_g, be) \Longrightarrow_b T}{(PD, FD, (P, \sigma_l), (be \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{\epsilon}_{proc}}$$

$$(PD, FD, (P', \sigma_l), proc, \sigma_g, (addtr2hfg \;\; hfg \;\; tr))$$

where

$$\begin{aligned} P' &= (csName\ proc) \\ tr &= (\{(P, \sigma_l)\}, be, \{(P', \sigma_l')\}) \end{aligned}$$

(ii) This rule shows how a single input communication action in a guard is handled. Note that after the communication, the local store is updated.

$$\frac{}{(PD, FD, (P, \sigma_l), (spi?x \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{spi?v}_{proc}}$$

$$(PD, FD, (P', \sigma_l[v/x]), proc, \sigma_g, (addtr2hfg \;\; hfg \;\; tr))$$

where

$$\begin{aligned} P' &= (csName\ proc) \\ tr &= (\{(P, \sigma_l)\}, spi?x, \{(P', \sigma_l')\}) \end{aligned}$$

(iii) This rule shows how a single input synchronization action is handled. Notice that the stores are not updated. The communication action is recorded as a label of the transition.

$$\frac{}{(PD, FD, (P, \sigma_l), (spi? \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{spi?}_{proc} (PD, FD, (P', \sigma_l), proc, \sigma_g, (addtr2hfg\ \ hfg\ \ tr))}$$

where

$$\begin{aligned} P' &= (csName\ proc) \\ tr &= (\{(P, \sigma_l)\}, spi?, \{(P', \sigma_l')\}) \end{aligned}$$

(iv) This rule specifies how guards containing one Boolean expression followed by an input communication action are handled. The guard is first checked, and only if found true will the communication action be tried.

$$\frac{(FD, \sigma_l, \sigma_g, be) \Longrightarrow_b T}{(PD, FD, (P, \sigma_l), ((be, spi?x) \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{spi?v}_{proc} (PD, FD, (P', \sigma_l[v/x]), proc, \sigma_g, (addtr2hfg\ \ hfg\ \ tr))}$$

where

$$\begin{aligned} P' &= (csName\ proc) \\ tr &= (\{(P, \sigma_l)\}, (be, spi?x), \{(P', \sigma_l')\}) \end{aligned}$$

(v) This rule is similar to the previous, except that value communication does not occur; only synchronization occurs.

$$\frac{(FD, \sigma_l, \sigma_g, be) \Longrightarrow_b T}{(PD, FD, (P, \sigma_l), ((be, spi?) \rightsquigarrow proc), \sigma_g, hfg) \xrightarrow{spi?}_{proc} (PD, FD, (P', \sigma_l), proc, \sigma_g, (addtr2hfg\ \ hfg\ \ tr))}$$

where

$$\begin{aligned} P' &= (csName\ proc) \\ tr &= (\{(P, \sigma_l)\}, (be, spi?), \{(P', \sigma_l')\}) \end{aligned}$$

(vi) This shows that if the alternate command has two guarded *cproc*s, then they can be tried in any order.

$$\frac{PD, FD, s, cproc, \sigma_g, hfg) \xrightarrow{ca}_{proc} (PD, FD, s', proc, \sigma_g, hfg')}{(PD, FD, s, (cproc \parallel cproc'), \sigma_g, hfg) \xrightarrow{ca}_{proc} (PD, FD, s', proc, \sigma_g, hfg')}$$

$$\frac{(PD, FD, s, cproc', \sigma_g, hfg) \xrightarrow{ca}_{proc} (PD, FD, s', proc, \sigma_g, hfg')}{(PD, FD, s, (cproc \parallel cproc'), \sigma_g, hfg) \xrightarrow{ca}_{proc} (PD, FD, s', proc, \sigma_g, hfg')}$$

6.4 Parallel Composition

The "||" operator specifies concurrent behavior in hopCP. It defines the interaction of independently specified hopCP specifications. In this section we will define interaction of hopCP specifications by describing a tool called *parComp*. *parComp* helps *inferring* the composite behavior of a collection hopCP modules. We will conclude this section by describing some uses of *parComp*.

6.4.1 Synchronization and Behavioral Interaction

hopCP modules interact via communication actions. The interaction could be synchronous (via handshake or rendezvous) or asynchronous (via global store). As stated earlier, the latter interaction is potentially hazardous and its safety is guaranteed by checking for the serial usage of all asynchronous channels. Synchronous interaction is possible when modules are willing to perform complementary actions (query/assertion) on the same channel. When the number of modules willing to perform a *query* is equal to one, for a given assertion, we have a *two-way rendezvous* (similar to synchronous communication in CSP or CCS, for example). When the number of modules willing to perform a query is *more than one* for a given assertion, we have a *multiway rendezvous*. Multiway rendezvous is a powerful construct. It helps express several hardware-oriented algorithms very naturally.

Parallel composition is complicated in hopCP due to the presence of compound actions. This is because a *proper subset of the compound action set* can synchronize. We call this *partial synchronization*. In the next section we will provide the formal definition of *parComp* and illustrate it with an example.

6.4.2 Formal Definition of parComp

First we define an auxiliary function *conjugate* which checks if two primitive action can synchronize or not. Two primitive actions synchronize when they are complementary and both use the same channel. For example, $conjugate(a?x, a!(p+1))$ yields *true* while $conjugate(a?x, b!(p+1))$ or $conjugate(a?x, a?y)$ or $conjugate(a!x, a!(p+1))$ yields *false*.

$$
\begin{aligned}
conjugate(x, y) \triangleq\ & (isDquery\ x)\ \wedge\ (isDassert\ y)\ \wedge \\
& (channel(x) = channel(y)) \\
& \vee \\
& (isDquery\ y)\ \wedge\ (isDassert\ x)\ \wedge \\
& (channel(x) = channel(y))
\end{aligned}
$$

where $(isDquery\ x)$ and $(isDassert\ x)$ are predicates which check if the action x is a data query or a data assertion respectively and $channel(z)$ extracts the synchronous port associated with the action z.

In hopCP, parallel composition is complicated by the presence of compound actions. To assist in the definition of parallel composition, we define the following relations, which determine whether a pair of *compound actions* a and b are "*synchronous*" (a and b completely synchronize) or "*asynchronous*" (a and b do not synchronize at all):

$$\begin{aligned} synchronous(a,b) \triangleq\ & x \in a \Rightarrow (\exists y \in b.conjugate(x,y)) \\ & \wedge \\ & x \in b \Rightarrow (\exists y \in a.conjugate(x,y)) \end{aligned}$$

$$asynchronous(a,b) \triangleq \neg(\exists x \in a \wedge \exists y \in b.conjugate(x,y))$$

$parComp$ is a function which composes two concurrent state-transition systems ($HFGs$). It uses auxiliary functions $Valuecomm$ to perform value communication, and $RetainAsOutput$ to facilitate multiway rendezvous. The auxiliary functions are defined as follows:

$$\begin{aligned} RetainAsOutput\ (a,b) \triangleq\ \{\ x \mid\ & ((x \in a \wedge (isDquery\ x) \wedge (\exists y \in b.conjugate(x,y))) \\ & \bigvee\ (x \in b \wedge (isDquery\ x) \wedge (\exists y \in b.conjugate(x,y)))\ \} \end{aligned}$$

$ValueComm$ takes a pair of states denoting preconditions of the transitions being composed (s_1, s_2), a pair of *conjugate* actions (a, b) and a pair of states denoting the postconditions of transitions being composed (s_1', s_2') and *updates* the local stores of s_1' or s_2' depending on the actions a and b. It is defined recursively as follows:

$ValueComm\ (s_1, s_2, a, b, s_1', s_2') =$

let

$$\begin{array}{llll} a = \{a_1, \ldots, a_n\} & b = \{b_1, \ldots, b_n\} \\ s_1 = (P_1, \sigma_1) & s_2 = (P_2, \sigma_2) \\ s_1' = (P_1', \sigma_1') & s_2' = (P_2', \sigma_2') \\ a_i = c_i?x_i & a_j = d_j!e_j \\ b_i = c_i!e_i & b_j = d_j?x_j \end{array}$$

in

if $((a_i \in a) \wedge (isDquery\ a_i) \wedge (\exists b_i \in b.conjugate(a_i, b_i)) \wedge (FD_2, \sigma_2, \sigma_{g_2}, e_i) \Longrightarrow_e v_i)$
then $ValueComm\ (s_1, s_2, a \setminus a_i, b \setminus b_i, (P_1', \sigma_1'[v_i/x_i]), s_2')$
else if $((a_j \in a) \wedge (isDassert\ a_j) \wedge (\exists b_j \in b.conjugate(a_j, b_j)) \wedge (FD_1, \sigma_1, \sigma_{g_1}, e_j) \Longrightarrow_e v_j)$
then $ValueComm\ (s_1, s_2, a \setminus a_j, b \setminus b_j, s_1', (P_2', \sigma_2'[v_j/x_j]))$
else (s_1', s_2')
end if

end

Basically, function $ValueComm$ examines the constituent actions of a and b. It picks one action from a and one from b such that they are conjugates. It removes them from a and b, and performs the intended communication between them symbolically, and updates the stores appropriately. Function $ValueComm$ terminates when one of the sets a or b becomes empty, or when they cease to have conjugate actions.

Using the auxiliary functions defined above, $(parComp\ hfg_1\ hfg_2) = hfg_3$ where

$$\begin{aligned} hfg_1 &= \{istate = is_1, trel = tr_1\} \\ hfg_2 &= \{istate = is_2, trel = tr_2\} \\ hfg_3 &= \{istate = is_1 \cup is_2, trel = tr_3\} \end{aligned}$$

and tr_3 is inductively defined by the following rules. In defining tr_3, we shall build two "temporary" sets of transitions tr_1' and tr_2' also. Rules for building tr_1' and tr_2' are also given. All three sets (tr_1', tr_2', and tr_3) are inductively defined by the following rules.

(i) : One rule for building tr_1'

$$\frac{t \in tr_1}{t \in tr_1'}$$

(ii) : Another rule for building tr_2'

$$\frac{t \in tr_2}{t \in tr_2'}$$

(iii) : One rule for building tr_3: Case "Total Synchronization"

$$\frac{(s_1, a, s_1') \in tr_1' \wedge (s_2, b, s_2') \in tr_2' \wedge synchronous(a, b)}{(s_1 \cup s_2,\ RetainAsOutput(a, b),\ ValueComm\ (s_1, s_2, a, b, s_1', s_2')) \in tr_3}$$

This rule is applied when the compound actions a and b synchronize completely. $ValueComm$ performs the value communication across the HFGs . Function $RetainAsOutput$ retains the output counterparts of the synchronized actions, thus facilitating multiway rendezvous. In other words, if $p!e$ and $p?x$ synchronize, $RetainAsOutput$ retains $p!e$, which can thereafter synchronize with another $p?y$, if there happens to be one. (That is, it does not turn the communication between $p!e$ and $p?x$ into an ϵ, as in CCS, which supports only point-to-point rendezvous.)

(iv) : The other rule for building tr_3: Case "No Synchronization"

$$\frac{(s_1, a, s_1') \in tr_1' \wedge (s_2, b, s_2') \in tr_2' \wedge asynchronous(a, b)}{\{(s_1, a, s_1'), (s_2, b, s_2')\} \subseteq tr_3}$$

This rule is applied when none of the constituents of a and b can synchronize. It reflects the fact that both the transitions can be done concurrently. Note that we are not *interleaving* the transitions. This rule plays a very key role in keeping the space and time complexity of parallel composition linear in terms of the number of states. To give an example of its significance, if HFG h_1 has m states and HFG h_2 has n states, and if all the actions of h_1 and h_2 are *different*, then the total number of states in $parComp(h_1, h_2)$ is $O(m+n)$ as opposed to $O(m \times n)$ which a *interleaved* rule would give.

(v) : The last rule for building tr_1' and tr_2': Case "Partial Synchronization"

Let $a = a_1 \cup a_2$ *and* $b = b_1 \cup b_2$
Then,

$$\frac{((s_1, a, s_1') \in tr_1') \wedge ((s_2, b, s_2') \in tr_2') \wedge synchronous(a_1, b_1) \wedge asynchronous(a_2, b_2)}{prefine(s_1, a, a_1, a_2, s_1') \subseteq tr_1' \wedge prefine(s_2, b, b_1, b_2, s_2') \subseteq tr_2'}$$

where

$$\begin{aligned}
prefine(s_1, a, a_1, a_2, s_1') &= \{(s_1, a^i, \{s^i_{a_1}, s^i_{a_2}\}), (s^i_{a_1}, a_1, s^c_{a_1}), \\
&\quad (s^i_{a_2}, a_2, s^c_{a_2}), (\{s^c_{a_1}, s^c_{a_2}\}, a^c, s_1')\} \\
prefine(s_1, a, \emptyset, a_2, s_1') &= \{(s_1, a, s_1')\} \\
prefine(s_1, a, a_1, \emptyset, s_1') &= \{(s_1, a, s_1')\}
\end{aligned}$$

This rule handles the remaining case which is not handled by (iii) and (iv), *i.e.*, when compound actions a and b synchronize partially. The definition is based on the interpretation of compound actions as *sets* of primitive actions.

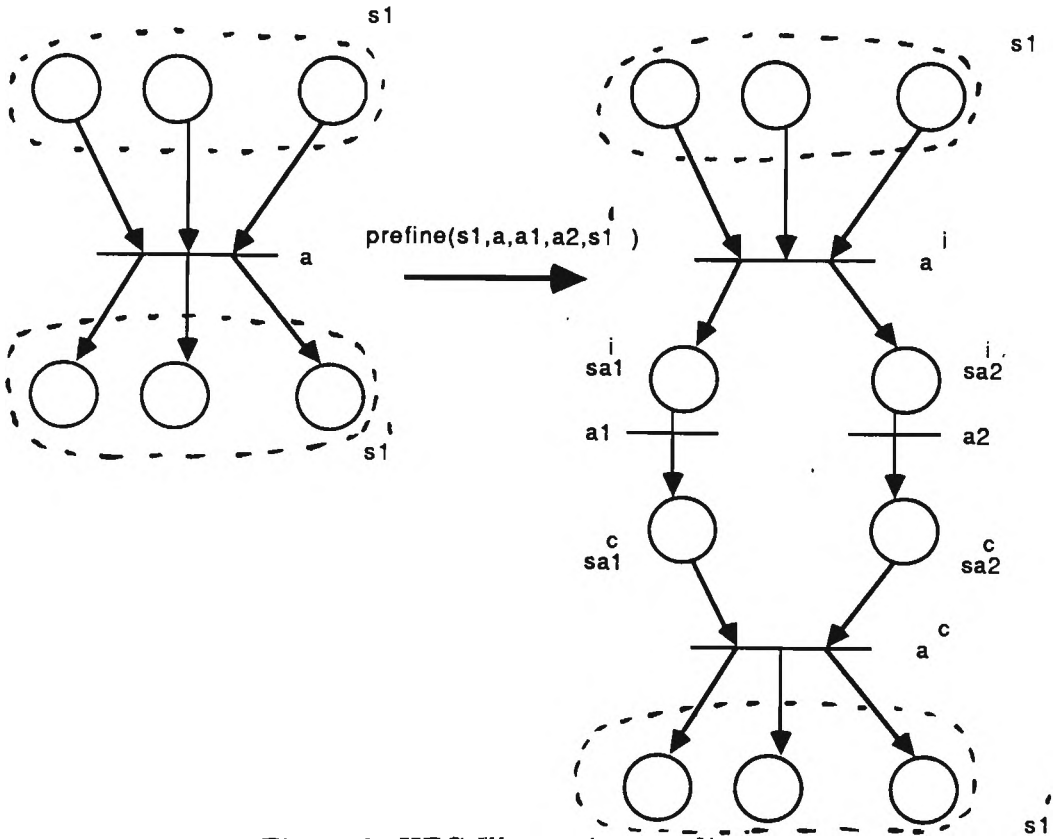

Figure 8: HFG Illustrating *prefine*

We pattern-match on the structure of the compound actions. This pattern matching process partitions a and b to extract components a_1,b_1,a_2, and b_2 which are, themselves, compound actions. The components that synchronize are a_1 and b_1, while the components that do not synchronize are a_2 and b_2.

The transitions added to tr_1' and tr_2' have the crucial property that they have one of the compound action sets a_1,b_1,a_2, or b_2 labeling them. This way we are introducing transitions that completely synchronize, or do not synchronize. This makes it possible for rules (iii) and (iv) to be applicable.

Also notice that transitions bearing the actions a^i and a^c are added to tr_1' and tr_2'. This is to facilitate the partitioning of a and b sets, by appealing to the fact that a compound action can be refined into its corresponding HFG that describes its *fork-join structure*, as shown in figure 8. We introduce intermediate states $s^i_{a_1}, s^c_{a_1}, s^i_{a_1}, s^c_{a_1}$ and actions a^i, a^c These states and actions help detail the execution following the partial synchronizations, They also have significance in the synthesis of circuits from hopCP specifications [AG91a]. Additional details are illustrated in figure 8.

To facilitate the understanding of the definition of $parComp$ we present the following example: Let, $s_1 = (P, \sigma_l^p), s_1' = (P', \sigma_l^{p'}), s_2 = (Q, \sigma_l^q), s_2' = (Q', \sigma_l^{q'})$,

Total Synchronization will occur, if for example $a = (c?x, d!z)$ and $b = (c!y, d?u)$ where $(x,u) \in VAR, (z,y) \in EXPR, (c?,d?) \in SyncInputPorts, (c!,d!) \in SyncOutputPorts$. The resultant transition will be $(\{s_1, s_2\}, (c!y, d!z), \{(P', \Sigma_l^{p'}), (Q', \Sigma_l^{q'})\})$ where

$$\begin{aligned} RetainAsOutput(a,b) &= (c!y, d!z) \\ ((P', \Sigma_l^{p'}), (Q', \Sigma_l^{q'})) &= ValueComm(s_1, s_2, a, b, s_1', s_2') \end{aligned}$$

No Synchronization will occur, if for example $a = (c?x, d!z)$ and $b = (f!y, g?u)$ where $(x,u) \in VAR, (z,y) \in EXPR, (c?,g?) \in SyncInputPorts, (f!,d!) \in SyncOutputPorts$. The resultant transitions will be (s_1, a, s_1') and (s_2, b, s_2')

Partial Synchronization will occur, if for example $a = (c?x, d!z)$ and $b = (c!y, g?u)$ where $(x,u) \in VAR, (z,y) \in EXPR, (c?,g?) \in SyncInputPorts, (c!,d!) \in SyncOutputPorts$. It is a simple exercise to rewrite each transition into the set of four transitions using the definition of *prefine* above.

6.4.3 Uses of parComp

A tool like *parComp* which infers composite behavior, is extremely important in a specification driven design environment. It can be used in the following ways: We will not go into their details because they have been published elsewhere.

1. *Verification:* It can be used in formal verification wherein the implementation is compared against the inferred behavior (after algebraic simplification). An example of such verification effort can be found in [GF91].

2. *Realizability Checks:* The absence of deadlock, safe usage of asynchronous communication action, and liveness are formulated on the HFG corresponding to the inferred behavior. Most of these checks involve doing a *reachability* analysis on the inferred behavior [AG91d].

3. *Simulation:* Inferred behavior can be used to derive a behavioral simulator for high-level specifications as shown in [Man89].

4. *High-Level Test Generation:* In the past we have shown [AG90] how a behavioral inference tool can be used in high-level test generation.

6.5 Behavior

Finally, we present the semantics of the top level syntactic object in hopCP namely *Behavior*.

Additional Syntactic Categories

$$beh \in Behavior$$

Abstract Syntax

The abstract syntax that specifies the behavior of a single hopCP module involves the following BNF rule:

$$beh ::= P\ [] \quad | \quad P[v_1, v_2, \ldots, v_m] \quad | \quad beh \parallel beh'$$

The semantics of *beh* is given by the transition relation $\longrightarrow_{beh}$ whose type signatures are as follows.

$$\xrightarrow{ca^+}_{beh}: (ProcDecl \times FunDecl \times Behavior) \mapsto (GlobalStore \times HFG)$$

This states that $\xrightarrow{ca^+}_{beh}$ takes the set of process and function declarations in a hopCP specification, and the behavioral description *beh* (in the form of a process call or a parallel composition of several process calls), and returns an HFG and a global store. The HFG returned represents the result of *compiling proc* into an HFG. The global store returned represents final global store attained after running all the process calls, beginning with an empty global store (*i.e.* completely undefined global store, denoted by $\emptyset$).

$\xrightarrow{ca^+}_{beh}$ is defined inductively by the following rules:

(i) This rule captures the semantics of *sequential* hopCP specifications with initial datapath state being empty. In other words, we are defining the behavior of P [].

$$\frac{(PD, FD, (P, \sigma_l), proc, \sigma_g^0, hfg) \xrightarrow{ca^+}_{proc} (PD, FD, (P', \sigma_l'), proc', \sigma_g, hfg')}{(PD, FD, P\ []) \xrightarrow{ca^+}_{beh} (\sigma_g, hfg)}$$

where

$$\begin{aligned} \sigma_g^0 &= \emptyset \\ (P[] &\Leftarrow proc) \in PD \\ \sigma_l &= \emptyset \\ hfg &= \{istate = \{(P, \sigma_l)\}, trel = \emptyset\} \end{aligned}$$

This rule first appeals to the $\xrightarrow{ca^+}_{proc}$ relation and obtains the final global store σ_g and final HFG hfg' obtained by "running" P []. The final HFG is built up, action by action, from the initial HFG, hfg, given to the rule.

(ii) This rule captures the semantics of *sequential* hopCP specifications with non-empty initial datapath state.

$$\frac{(PD, FD, (P, \sigma_l), proc, \sigma_g^0, hfg) \xrightarrow{ca^+}_{proc} (PD, FD, (P', \sigma_l'), proc', \sigma_g, hfg')}{(PD, FD, P[v_1, v_2, \ldots, v_m]) \xrightarrow{ca^+}_{beh} (\sigma_g, hfg')}$$

where

$$\begin{aligned} \sigma_g^0 &= \emptyset \\ (P[d_1, d_2, \ldots, d_m] &\Leftarrow proc) \in PD \\ \sigma_l &= \{d_1 \mapsto v_1, d_2 \mapsto v_2, \ldots, d_m \mapsto v_m\} \\ hfg &= \{istate = \{(P, \sigma_l)\}, trel = \emptyset\} \end{aligned}$$

(iii) This rule captures the semantics of *concurrent* hopCP specifications by appealing to *parComp* to implement parallel composition. The HFGs obtained by compiling *beh* and *beh'* are first obtained (hfg and hfg' respectively). The global stores obtained by "running" *beh* and *beh'* are also obtained. Then, the global stores are united, banking on the assurance that there will be no domain conflicts (only one process owns every global port). Finally, the HFGs are subject to *parComp* to obtain the resultant HFG.

$$\frac{((PD, FD, beh) \xrightarrow{ca^+}_{beh} (\sigma_g, hfg)),\ \ ((PD, FD, beh') \xrightarrow{ca^+}_{beh} (\sigma_g', hfg'))}{(PD, FD, (beh \parallel beh')) \xrightarrow{ca^+}_{beh} (\sigma_g \cup \sigma_g', (parComp\ \ hfg\ \ hfg'))}$$

7 Semantics of Structural Operators

rename, *connect*, and *export* are the structural operators in hopCP. Structural operators help specify an *interconnection* of hopCP modules. $PORT$ and $MODULE$ are defined as follows and denote the domain of all communication channels in hopCP and modules in hopCP

$$\begin{aligned} PORT &= SyncOutputPorts \oplus SyncInputPorts \oplus AsyncPorts \\ MODULE &= \{behavior \subseteq HFG, ports \subseteq \mathcal{P}^+(PORT)\} \end{aligned}$$

Let m be a hopCP module, i.e. $m \in MODULE$. We define selectors $m.p$ and $m.b$ to denote the set of ports used in m, and the HFG denoting the behavior of m, respectively.

rename does alpha conversion of channel names (specified by f) to facilitate interconnection i.e. channels with same names and complementary direction are assumed to be

electrically connected. Let $m_1 \in MODULE$ and $f : PORT \mapsto PORT$ denote a bijection, then $rename : MODULE \mapsto PORT \mapsto PORT \mapsto MODULE$ is defined as follows:

$$rename\ m_1\ f\ =\ \{behavior = m_1.b, ports = (map\ f\ m_1.p)\}$$

Let $m_1, m_2 \in MODULE$. $connect : MODULE \mapsto MODULE \mapsto MODULE$, returns the composite hopCP module denoted by interconnection of separately specified modules m_1 and m_2 after performing static checks like unconnected ports, and illegal connections etc.

$$connect\ m_1\ m_2\ = \{behavior = (parComp\ m_1.b\ m_2.b), ports = m_1.p\ \cup\ m_2.p\}$$

Note, that *connect* simply invokes *parComp* to infer the *composite* behavior.

$export : MODULE \mapsto PORT \mapsto MODULE$, removes a set of ports $p \in PORT$ f rom the sort of a module $m_1 \in MODULE$, so that the actions on those ports cannot be *shared by other modules* when assembled together by *connect* operator. The module m_1 though can still engage in actions involving ports in p.

$$export\ m_1\ p\ =\ \{behavior = m_1.b, ports = (m_1.p) \setminus p\}$$

8 Implementation

The design environment centered around hopCP (and shown in figure 1) is being implemented in Standard ML of New Jersey. The implementation has been partitioned into four independent tasks which are linked by HFG which is the common intermediate form for hopCP specifications. First, hopCP textual syntax is parsed using SML yacc/lex and compiled into a SML data structure denoting HFG . This is a direct implementation of the transition rules described in the previous section. The algorithm *parComp* has been implemented which composes HFGs and returns a new HFG . We have also implemented a *compiled-code concurrent functional simulator* called CFSIM for hopCP specifications which involves *translating* HFGs into *CML* source code in such way that the semantics of hopCP outlined in the previous sections are preserved. The details of the simulator will be described in [AG91b]. CML is an extension to SML to support synchronous message passing and is described in [Rep91].

Several realistic examples including Intel 8251 programmable communication interface chip have been specified in hopCP and simulated for functional correctness. We specified the Intel 8251 has a collection of four concurrent processes and inferred its behavior using *parComp* quite efficiently. The details are presented in [AG91c].

9 Conclusions and Future Work

hopCP was developed based on what we felt were some of the important features to be present in an HDL and its support system. The combination of features that hopCP and its implementation currently offer are:

- a simple HDL, a complete operational semantics that describes the hopCP interpreter and the compiler that generates hopCP flow graphs;
- addition of asynchronous ports to a message passing language, and a semantic characterization of asynchronous ports; a catalogue of uses of asynchronous ports;
- compound actions, and formal characterization of *partial synchronization*;
- functional notation to minimize the effort of data flow analysis and also for clearly specifying computations;
- an implementation using Standard ML, following a compiled-code approach; this approach enforces ML's strong type-checking on hopCP descriptions also, thereby helping avoid many bothersome errors;
- concurrency simulator using Concurrent ML, that supports strong type-checking, and the simulation of concurrency;
- the ability to simulate asynchronous descriptions using asynchronous "tester" processes, instead of forcing the designer to pore over waveforms (which are of even lesser value for asynchronous designs);
- parComp, that infers succinct behavioral descriptions, retaining concurrency (*i.e.*, not interleaving), thereby not suffering from combinatorial explosion of possible interleavings;
- static analysis procedures that help detect whether a pair of actions are serial or not, thereby supporting efficient compilation of asynchronous ports and also the choice construct.

Preliminary results indicate that the above combination of features are quite useful in developing and debugging the design specification as well as design refinements of large application specific ICs. Of course, we are aware of the omission of many other useful features from hopCP; for example: parameterized modules, proper exception handling, and the incorporation of real-time constraints. (We are working on some of these omissions, but clearly it is impossible to have every construct imaginable and have a semantically tractable HDL.)

The next major task we shall undertake will be *hierarchical action refinement*, described in [AG92, AG91a]. One of the problems with most existing high level synthesis systems is that they give very little clues on how they operate. This makes it nearly impossible to understand, evaluate, and *prove correct* the optimizations they make to user's source descriptions. We plan to rectify some of these problems by developing a rule based compilation system, where the rules encapsulate how *time* is hierarchically refined, and how resources are *incrementally allocated.*

Future modifications of parComp will help it detect simple errors (*e.g.* dead states, and loops of infinite chatters). It is a very easy matter to modify CFSIM to incorporate checks for real-time constraint violations, thanks to the features offered by CML; this will be done in the near future. Other remaining work includes the development of a synchronous compilation system (actually using an available system such as Olympus [Ku91]) and the investigation of mixed synchronous/asynchronous compilation. Effort is also underway to specify several large designs, including the Roll Back Chip [GF91].

For information on the availability of the hopCP code for experimental assessment, contact `akella@cs.utah.edu`.

References

[AG90] Venkatesh Akella and Ganesh Gopalakrishnan. High Level Test Generation via Process Composition. In *Designing Correct Circuits, Oxford, 1990*, pages 99–119. Springer Verlag, 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

[AG91a] Venkatesh Akella and Ganesh Gopalakrishnan. Hierarchical Action Refinement: A Methodology for Compiling Asynchronous Circuits from a Concurrent HDL. In *Proceedings of the Tenth International Symposium on Computer Hardware Description Languages and their Applications, Marseille, France*, April 1991.

[AG91b] Venkatesh Akella and Ganesh Gopalakrisnan. CFSIM: A Compiled-Code Concurrent Functional Simulator for VLSI Systems. Technical report, Department of Computer Science, University of Utah, 1991. In preparation; available upon request from the authors.

[AG91c] Venkatesh Akella and Ganesh Gopalakrisnan. Specification and Validation of a USART in hopCP. Technical report, Department of Computer Science, University of Utah, 1991. In preparation; available upon request from the authors.

[AG91d] Venkatesh Akella and Ganesh Gopalakrisnan. Static Analysis Techniques for the Synthesis of Efficient Asynchronous Circuits. Technical Report UUCS-91-018, Department of Computer Science, University of Utah, October 1991.

[AG92] Venkatesh Akella and Ganesh Gopalakrishnan. From Process-Oriented Functional Specifications to Efficient Asynchronous Circuits. In *To Appear, In Fifth International Conference on VLSI, Bangalore, India*, January 1992.

[BS89] 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.

[Cha84] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Department of Computer Science, Stanford University, October 1984.

[Cha87] Arthur Charlesworth. The Multiway Rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, July 1987.

[CPS89] Rance Cleveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tood for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, Univ of Edinburgh, August 1989.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56.*

[GF91] Ganesh Gopalakrishnan and Richard Fujimoto. Design and verification of the rollback chip using hop: A case study of formal methods applied to hardware design. Technical Report UU-CS-TR-91-015, University of Utah, Department of Computer Science, 1991.

[GFAM89] Ganesh C. Gopalakrishnan, Richard Fujimoto, Venkatesh Akella, and Narayana Mani. HOP: A process model for synchronous hardware. semantics, and experiments in process composition. *Integration: The VLSI Journal*, pages 209–247, August 1989.

[GJ90] Ganesh Gopalakrishnan and Prabhat Jain. Some Recent Asynchronous System Design Methodologies. Technical Report UU-CS-TR-90-016, University of Utah, Department of Computer Science, 1990.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley & Sons, 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Joh84] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

[Kel74] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.

[KM90] David Ku and Giovanni De Micheli. HardwareC - A Language for Hardware Design, Version 2.0. Technical Report CSL-TR-90-419, Computer Science Laboratory, Stanford University, April 1990.

[Ku91] David Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications.* PhD thesis, Department of Computer Science, Stanford University, June 1991.

[LOJF88] L. Logrippo, A. Obaid, J.P.Briand, and M.C. Fehri. An Interpreter for LOTOS, a Specification Language for Distributed Systems. *Software—Practice and Experience*, 18(4):365–385, April 1988.

[Man89] Narayana Mani. Behavioral Simulation from High Level Specifications. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, May 1989.

[Mar89] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. Technical Report Caltech-CS-TR-89-1, Department of Computer Science, California Institute of Technology, 1989.

[May90] David May. Compiling Occam into Silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication.* Addison-Wesley, 1990.

[MC80] C. A. Mead and L. Conway. *An Introduction to VLSI Systems.* Addison Wesley, 1980. *Chapter 7, entitled "System Timing".*

[PL91] Ian Page and Wayne Luk. Compiling Occam into Field-Programmable Gate Arrays. In *International Workshop on Field Programmable Logic and Applications*, September 1991. *September 4-6, 1991, Oxford University, UK.*

[Rep91] John H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[RM89] Robin Milber. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.

[She85] Mary Sheeran. Design of regular hardware structures using higher order functions. In *Proceedings of the Functional Programming and Computer Architecture Conference.* Springer-Verlag, LNCS 201, September 1985. Nancy, France.

[SST90] Eliezer Sternheim, Rajvir Singh, and Yatin Trivedi. *Digital Design with Verilog HDL.* Automata Publishing Company, Cupertino, CA, 95014, 1990. ISBN 0-9627488-0-3.

[Sut89] Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

[TLW+90] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, Boston, 1990.

[VHD85] VHDL Language Reference Manual, August 1985. *Intermetrics Report IR-MD-045-2; See also IEEE Design and Test, April 1986.*

10 Appendix

Syntactic Categories

v	$\in$	VAL	d	$\in$	$DPSVAR$
e	$\in$	$EXPR$	P	$\in$	$ProcName$
FD	$\in$	$FunDecl$	PD	$\in$	$ProcDecl$
beh	$\in$	$Behavior$	$proc$	$\in$	$PROC$
apo	$\in$	$AsyncPorts$	ca	$\in$	$CompoundAction$
g	$\in$	$Guard$	$cproc$	$\in$	$ChoiceProc$
spo	$\in$	$SyncOutputPorts$	spi	$\in$	$SyncInputPorts$
x	$\in$	VAR	bx	$\in$	$BVAR$
vop	$\in$	$VectorOp$	op	$\in$	$ArithOp$
bop	$\in$	$BooleanOp$	F	$\in$	$FunName$
be	$\in$	$BEXPR$			

Semantic Domains

$$
\begin{aligned}
\sigma_l \in LocalStore &= DPSVAR \mapsto VAL \\
\sigma_g \in GlobalStore &= AsyncOutputPorts \mapsto VAL \\
Transition &= \mathcal{P}^+(State) \times CompoundAction \oplus Guard \times \mathcal{P}^+(State) \\
State &= ProcName \times LocalStore \\
PORT &= SyncOutputPorts \oplus SyncInputPorts \oplus AsyncPorts \\
HFG &= \{istate \subseteq \mathcal{P}(STATE), trel \subseteq TRANSITION\} \\
MODULE &= \{behavior \subseteq HFG, ports \subseteq \mathcal{P}^+(PORT)\}
\end{aligned}
$$

Abstract Syntax

$$
\begin{aligned}
beh &::= P\,[\,] \mid P[v_1, v_2, \ldots, v_m] \mid beh \parallel beh' \\
PD &::= P(d_1, d_2, \ldots, d_m) \Leftarrow proc \mid P(d_1, d_2, \ldots, d_m) = proc, PD' \\
FD &::= F(x_1, x_2, \ldots, x_k) = e \mid F(x_1, x_2, \ldots, x_k) = e, FD' \\
proc &::= ca \rightsquigarrow proc' \mid P[e_1, e_2, \ldots, e_m](where\ arity(P) = m) \mid cproc \\
cproc &::= g \rightsquigarrow proc \mid cproc \,[\!]\, cproc' \\
g &::= be \mid dq \mid spi \mid be, dq \mid be, spi \\
a &::= dq \mid da \mid aa \mid spi \mid spo \\
ca &::= a \mid a', ca \\
e &::= v \mid x \mid apo \mid e'\ op\ e'' \mid \underline{let}\ x = e\ \underline{in}\ e' \mid vop(e', e'', e''') \\
&\quad \mid \underline{if}\ be\ \underline{then}\ e\ \underline{else}\ e' \mid F[e_1, e_2, \ldots, e_k](where\ arity(F) = k) \\
be &::= bx \mid True \mid False \mid be'\ op\ be'' \mid Not\ be \mid equal(e, e') \\
op &::= + \mid - \mid * \mid / \mid rshift \mid lshift \mid index \\
bop &::= and \mid or \mid nand \mid nor \mid exor \\
vop &::= update \mid subvector
\end{aligned}
$$

Type Signatures of Transition Relations

$$
\xrightarrow{ca^+}_{proc} : (ProcDecl \times FunDecl \times State \times Proc \times GlobalStore \times HFG) \mapsto
$$

$$
\begin{aligned}
& (ProcDecl \times FunDecl \times State \times Proc \times GlobalStore \times HFG) \\
\xrightarrow{ca^+}_{CA} \; : \; & (FunDecl \times LocalStore \times GlobalStore \times CompoundAction) \\
& \mapsto (LocalStore \times GlobalStore) \\
\xrightarrow{ca^+}_{beh} \; : \; & (ProcDecl \times FunDecl \times Behavior) \mapsto (GlobalStore \times HFG)
\end{aligned}
$$

Type Signatures of Auxillary Functions

$$
\begin{aligned}
csName \; & : \; Proc \mapsto ProcName \\
parComp \; & : \; HFG \mapsto HFG \mapsto HFG \\
addtr2hfg \; & : \; HFG \mapsto (\mathcal{P}^+(State) \times CompoundAction \times \mathcal{P}^+(State)) \mapsto HFG
\end{aligned}
$$

Type Signatures of Structural Operators

$$
\begin{aligned}
rename \; & : \; MODULE \mapsto (PORT \mapsto PORT) \mapsto MODULE \\
connect \; & : \; MODULE \mapsto MODULE \mapsto MODULE \\
export \; & : \; MODULE \mapsto PORT \mapsto MODULE
\end{aligned}
$$