CFSIM: A Concurrent Compiled-Code Functional Simulator for hopCP

VENKATESH AKELLA
GANESH GOPALAKRISHNAN

UUCS-92-002

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

January 22, 1992

Abstract

Control intensive ICs pose a significant challenge to the users of formal methods in designing hardware. These ICs have to support a wide variety of requirements including synchronous and asynchronous operations, polling and interrupt-driven modes of operation, multiple concurrent threads of execution, complex computations, and programmability. In this paper, we illustrate the use of formal methods in the design of a control intensive IC called the "Intel 8251" Universal Synchronous/Asynchronous Receiver/Transmitter (USART), using our formal hardware description language 'hopCP'. A feature of hopCP is that it supports communication via asynchronous ports *(distributed shared variables writable by exactly one process), in addition to* synchronous *message passing. We show the usefulness of this combination of communication constructs. We outline static analysis algorithms to determine safe usages of asynchronous ports, and also to discover other static properties of the specification. We discuss a compiled-code concurrent functional simulator called* CFSIM, *as well as the use of* concurrent testers *for driving CFSIM. The use of a semantically well specified and simple language, and the associated analysis/simulation tools helps conquer the complexity of specifying and validating control intensive ICs.*

CFSIM: A Concurrent Compiled-Code Functional Simulator for hopCP

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. We present a hardware description language (HDL) called hopCP for writing system-level specifications of hardware, and a simulation environment called CFSIM to validate hopCP behavioral specifications. HopCP is a process-oriented language based on the Communicating Sequential Processes (CSP) paradigm with a few extensions: computations are specified in a purely functional style, and a restricted form of ***distributed shared variables*** is provided to support asynchronous value communication. HopCP addresses high level protocol modeling as well as low level signaling details. A simulator in the CFSIM environment is generated by compiling a hopCP description into a Concurrent ML (CML) module. This CML module can then be compiled using the native code generator of Standard ML (SML) and executed. Since the CFSIM implementation is based on a polymorphically typed functional language (SML) and a concurrent language (CML), strong type-checking is automatically carried out before the simulation begins. CML offers a simple and extensible scheme for realizing hopCP's constructs. The compiled nature of the simulator guarantees much more efficiency over interpreted simulators. Support for design verification and circuit synthesis is also provided in the same framework. The implementation of CML as well as our experience with it are described.

1 Introduction

Of the tools that are used to design large VLSI systems, functional simulators play a major role. Functional simulators are used for debugging high level hardware descriptions of the system being designed. Designing effective functional simulation tools for large and complex VLSI systems is a non-trivial task. In the very same VLSI system, one is forced to describe – and simulate – both high level control/algorithmic aspects as well as low level signaling details. Most system level descriptions of VLSI are large, and therefore require high efficiency. Flexibility is required in the way the simulator is implemented so that HDL extensions can be easily accommodated, or experimented with. Strong type checking before simulation is another desirable feature to avoid simulating descriptions that have a type conflict. Last, but not the least, effective ways have to be developed to select simulation vectors, apply them, and observe the simulation results.

We present a hardware description language (HDL) hopCP for writing system-level specifications of hardware, and a simulation environment called CFSIM to validate hopCP behavioral specifications. HopCP is a multi-paradigm HDL. Computational aspects of hardware behavior are specified in a purely *functional* style and the communication and synchronization aspects are described *ex-*

plicitly in a process-oriented style like CSP. hopCP is equipped with constructs to model hardware phenomenon like *busy waiting*, *asynchronous message passing* and *broadcast*, and *barrier synchronization* directly. It has a simple operational semantics [2]. It addresses high level protocol modeling as well as low level signaling details.

We describe a simulation environment called CFSIM to validate hopCP behavioral specifications. A simulator in the CFSIM environment is generated by compiling a hopCP description into a Concurrent ML (CML) module. This CML module can then be compiled using the native code generator of Standard ML (SML) and executed. Since the CFSIM implementation is based on a polymorphically typed functional language (SML) and a concurrent language (CML), strong type-checking is automatically carried out before the simulation begins. CML offers a simple and extensible scheme for realizing hopCP's constructs. The compiled nature of the simulator guarantees much more efficiency over interpreted simulators. We have developed effective simulation schemes using *tester processes* in which the simulator is driven by well chosen vectors that try to establish high level properties. Finally, support for design verification and circuit synthesis is also provided in the same framework.

This paper deals with the design and implementation of CFSIM. Simulation based design validation is not new. Hardware simulation can be performed at the device level (e.g. SPICE), switch level with timing (e.g. IRSIM) and without timing (e.g. COSMOS), register transfer level (e.g. ISPS), or the functional level (e.g. the VHDL behavioral simulator). Whereas low-level simulators serve the purpose of modeling the underlying physics of devices, and reduce the time and expense needed to build the part and test it, behavioral simulators help debug the designer's understanding own of the hardware, and also help correct inconsistencies between two levels of an HDL description.

System-level behavioral specification of integrated circuits is often characterized by synchronous and asynchronous operations, polling and interrupt-driven modes of behavior, multiple concurrent threads of execution, and complex computation. Simulating such specifications using scalar inputs (i.e. vectors of 0's and 1's) to obtain output waveform traces is not a very satisfactory approach because of the large number of scenarios that have to be simulated. CFSIM allows the designer to write *tester* processes that can *animate* the environment of the process being simulated. For example, if a communications chip C with a send and a receive channel is being simulated, two tester processes T_1 and T_2 can be written, one to continuously send messages into the send channel, and another to continuously receive messages from the receive channel. T_1 and T_2 can then be run in parallel with C, thereby getting the effect of concurrently sending messages and reading messages from C. This effect is virtually impossible to achieve using traditional scalar simulation. This proved to be quite valuable; for example, in debugging a communications chip (called the Intel 8251) described in [3]. The testers that we wrote actually *proved to be very readable and succinct specifications of the system being debugged.*

Overview of CFSIM

A hopCP behavioral specification has three components: (i) the set of user-defined functions which capture the computational aspects of the module, (ii) the set of communication ports of the module, and (iii) the HFG (hopCP Flow Graph). hopCP Flow Graph is a concurrent state-transition system (analogous to a Petri net) which serves as the intermediate representation for the hopCP specifications.

The CFSIM simulation environment consists of tools to compile the hopCP specifications into *executable* CML (Concurrent ML) [6] source code. CML is a concurrent extension of Standard ML of New Jersey which supports first class synchronous operations. The major steps in CFSIM are translating the user-defined functions in a hopCP specifications into SML function definitions, the communication ports into CML channels and the HFG into a set of *communicating threads* in CML. The crux of this translation process is to *preserve* the semantics of hopCP. This requires building abstractions in CML to support hopCP constructs. For example, we built abstractions to support hopCP style barrier synchronization, shared variables and compound actions (restricted fork-join construct). A compiled simulator in CFSIM can be driven by tester processes, as explained in section 5.

Salient Features of CFSIM

CML is a strongly-typed, polymorphic, and higher-order concurrent language that facilitates building several concurrency abstractions. It is very easy to build abstractions to simulate hardware-specific constructs. The strong-typing facilitates debugging hopCP specifications for semantic errors like illegal port connections and inconsistent (and incorrect) usage of variables. The other advantages of the proposed simulation environment include the ability to simulate concurrency accurately and its efficiency with respect to time and space. CML capitalizes on the *continuation-passing* style implementation of the SML of New Jersey compiler [4] and supports concurrency with very little overhead [6]. In addition, the preemptive scheduling feature of CML makes CFSIM fairly responsive during interactive usage.

Organization of the Paper

In the next section we will illustrate hopCP with an example and bring out its salient features. In section 3 we will introduce CML briefly (sufficient to understand this paper) and motivate its advantages. Section 4 deals with the design and implementation of CFSIM while section 5 deals with an illustration of CFSIM to validate a pipelined stack specification in hopCP. We conclude by highlighting the principal advantages of CFSIM and presenting a brief sketch of our plans to extend CFSIM.

2 hopCP

Overview of hopCP

hopCP is a notation for describing concurrent-state transition systems based on a functional language augmented with features to express synchronous and asynchronous value communication. The basic unit of description is a $MODULE$ which consists of a set of communication ports, an optional set of user-defined functions, and a behavioral description called hopCP Flow Graph (or HFG). A HFG consists of a set of *states*, a set of *actions* and a set of *transitions*. States in hopCP are $(control, data)$ state pairs where control states are like finite-state machine (FSM) states, and data states capture the contents of internal storage locations. An action in hopCP is either a communication action or the evaluation of an expression. There are three types of communication actions:

1. Data Query and Data Assertion: These involve value communication *and* synchronization. For example, $p?x$ called *data query* denotes synchronizing on input port p and *receiving* a value denoted by x while $p!e$ called *data assertion* denote synchronization on the output port p and *sending* the value denoted by expression e.

2. Synchronous Control Actions: These involve only synchronization *no* value communication. For example, $p?$ denotes an input synchronization action on input port p while $p!$ denotes an output synchronization on output port p.

3. Assignment Actions: Assignment actions provide asynchronous communication via shared variables. For example, $a := e$ is an assignment action which involves writing the value denoted by expression e into the shared variable denoting the asynchronous port a.

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.* Next we will illustrate hopCP with an example.

Specification of a Pipelined Stack in hopCP

Figure 1 the structural specification of the pipelined stack as an interconnection of three modules namely, the address register (`AddrReg`), the memory unit (`Memory`) and the controller (`CTRL`).

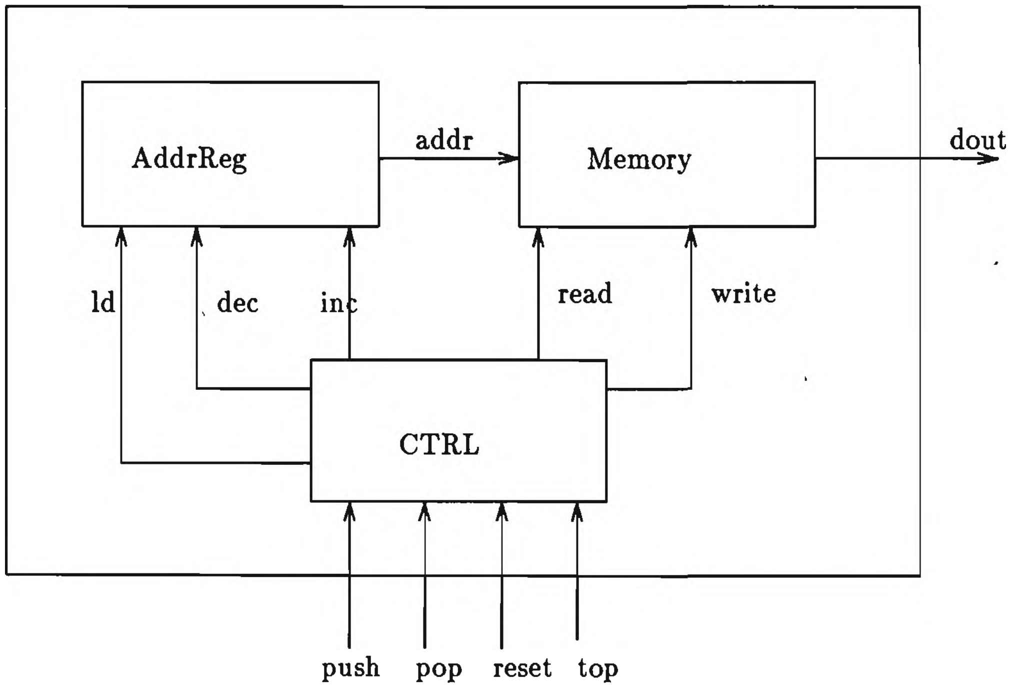

Figure 1: Structural Description of Pipelined Stack

```
MODULE AddrReg
TYPE
  addrType : vector 16 of bit;

SYNCPORTS
  ld?, inc?, dec? : bit;

ASYNCPORT
  addr!   : addrType;

FUNCTION

BEHAVIOR
    AREG [cs] <= (ld?inp -> addr := inp -> AREG [inp])
              | (inc? -> (addr := (cs + 1)) -> AREG [cs +1])
              | (dec? -> (addr := (cs - 1)) -> AREG [cs -1])
END
```

Let us examine the hopCP specification of the address register in detail. **addrType** is a user-defined type. `ld,inc` and `dec` are the three *input* synchronous communication channels. Communication on these channels is via ***handshake***. **addr** is declared as a *shared variable* (asynchronous port). The user-defined function definition section is empty. The behavior section describes a

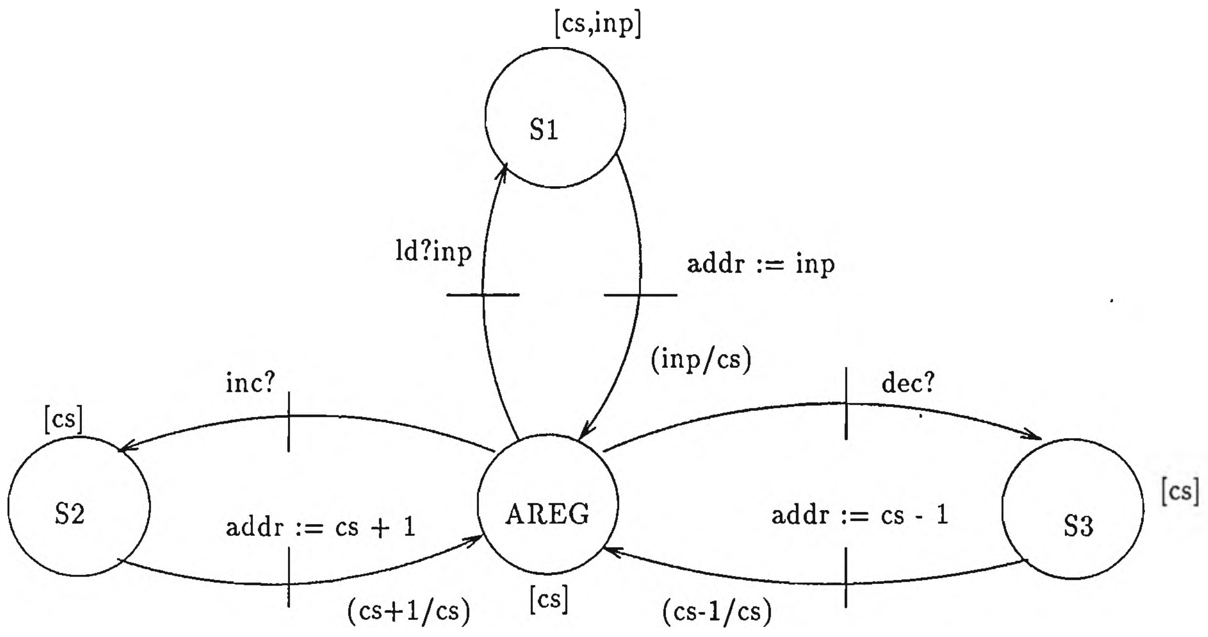

Figure 2: hopCP Flow Graph (HFG) for the Address Register

state-transition system or HFG shown in figure 2. Here the notation a/b means *replace*(update) b by a.

The address register module exhibits *alternate* (or conditional) behavior. `AREG` is the initial control state and *cs* is the initial *datapath state* of the module. In the initial state, the module has three *modes of behavior*: the address register can be loaded with a new value via the `ld` channel, or the address register can be *incremented* or *decremented* by corresponding commands on the `inc` and `dec` channels. The current value the address register is always available on the asynchronous port `addr`.

The specification of the memory system is shown in figure 3. In the top-level state, denoted by `MEM`, the module can accept a read or write request. A write request comes with the corresponding data to be written, denoted by `din` and the address is denoted by the asynchronous port `addr` (which is shared by the address register and the memory modules). Note that the write operation is captured by an expression action `update(s,addr,din)` where *update* is a predefined operation in hopCP. The read operation is more interesting. The result of the read operation is not delivered immediately; instead the module proceeds to an intermediate state, denoted by `MEM_AUX` wherein it has the capability of entertaining another read or write operation while delivering the result of the previous read. The output operation is done via a data assertion, namely, `dout!(index(s,a))` which informally means, output the value denoted by the expression `index(s,a)` on the output synchronous channel `dout`. The specification of the controller is notationally similar, and is shown in the appendix for completeness.

```
MODULE Memory
TYPE
  word : vector 16 of bit;
  addrType : vector 16 of bit;
SYNCPORTS
  read?,write?,dout! : word;
ASYNCPORT
  addr? : addrType;
FUNCTION
BEHAVIOR
    (MEM [ms] <=  (write?din ->  MEM [ update(ms,addr,din)])
                 |(read? -> MEM_AUX [ms,addr]);

     MEM_AUX [s,a] <= (write?din  -> dout!(index(s,a)) ->  MEM [update(s,addr,din)])
                      | (read? -> dout!(index(s,a)) -> MEM_AUX [s,addr]))
END
```

Figure 3: Specification of a Pipelined Memory in hopCP

The example above illustrated most of the features of hopCP like synchronous and asynchronous value communication, expression actions, alternate behavior (or guarded commands) and concurrency (by the fact that address register and memory could operate independently). Some of the features of hopCP that could not be illustrated with the simple pipelined stack example are as follows:

- *Compound Actions:* A transition could be annotated with a tuple of actions $a_1, a_2, \ldots, a_n$ instead of a single action a as was the case in the above example. The actions $a_1, a_2, \ldots, a_n$ could be data queries, data assertions, input control action, output control actions or assignment actions with the restriction that all a_i and a_j should be *non-interfering*, i.e., not two a_i and a_j should use the same channel or try to update the same variable. The execution of the system via a compound action is analogous to that of the *cobegin/coend* statement of concurrent programming languages.

- *Multiway Rendezvous:* Multiway rendezvous is said to occur when there is more than one agent willing to perform a *input* operation (data query) corresponding to a given output operation (data assertion). The semantics of multiway rendezvous in hopCP subsumes *broadcast* (point to multipoint communication) style of communication and *barrier synchronization* (alignment of time). Multiway rendezvous is a powerful construct which facilitates the specification of a wide variety of concurrent algorithms very naturally.

In the next section we will introduce CML and bring out its salient features.

3 Concurrent ML

CML is a high-level, high-performance language for concurrent programming. It is derived by augmenting SML with concurrency primitives. CML inherits all the good features of SML like higher-order functions, strong static typing, polymorphism, datatypes and pattern matching, exception handling and state-of-the-art module facility. CML capitalizes on the continuation-passing style implementation of the SML of New Jersey compiler and is very efficient [6]. The salient features of CML include: (i) high-level model for concurrency with dynamic creation of threads and typed channels, and CSP style synchronous communication based on a distributed memory model, (ii) provides *events* (or synchronous operations) as *first-class* values, which facilitates building new abstractions tailored to specific applications, (iii) provides an integrated I/O support, (iv) uses preemptive scheduling to guarantee responsiveness and (v) it is practical language tested in a variety of large-scale projects like *eXene* (a muti-threaded interface to X protocol), distributed ML and distributed Nuprl implementations. These features make CML a natural choice for the hopCP simulator.

4 Design of CFSIM

In this section we will present the details of the algorithm underlying CFSIM. We first outline the major steps in the algorithm and then illustrate the simulation of each construct of hopCP in CML.

Algorithm underlying CFSIM

Let $M_1, M_2, \ldots, M_n$ be the submodules of a hopCP module M.

Initialization:

Let, $M = M_1 \parallel M_2 \parallel \ldots \parallel M_n$, s_i = synchronous ports in M_i, a_i = asynchronous ports in M_i, f_i = user-defined functions in M_i, $(s_i, a_i, f_i$ could be empty)

Modules $M_1 \ldots M_n$ are parsed to extract the synchronous ports s_i, asynchronous ports a_i and user-defined functions f_i. The simulation environment is pre-loaded with the function definitions before commencing the simulation. Synchronous ports s_i are analyzed to detect *multiway rendezvous*. If a port does not have *exactly* one writer (outputting module) an error message is issued. If a port has more than one receiver, it is marked as a *barrier channel*, and is implemented by a `Barrier` abstraction. Ports with exactly one producer and one receiver is implemented directly by the `channel ()` construct in CML. Asynchronous ports (or shared variables) are simulated by the the structure `AsyncBarrier` and its associated operations. The details of the implementation of `Barrier` and `AsyncBarrier` will be presented later in this section.

Decomposition:

Let H denote the composite HFG denoted by the module M. Decompose H into $S_1, S_2, \ldots, S_m$

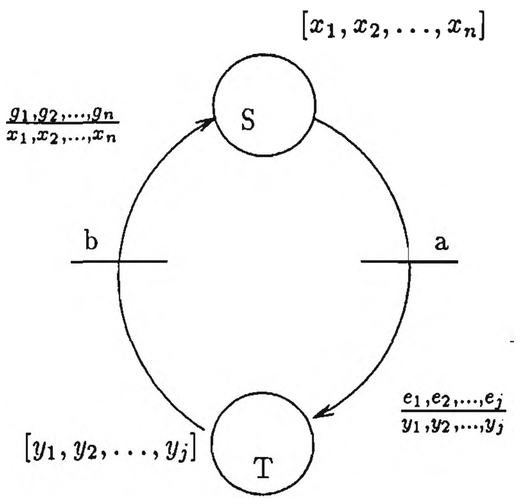
$[x_1, x_2, \dots, x_n]$
S
$\frac{g_1,g_2,\dots,g_n}{x_1,x_2,\dots,x_n}$
b
a
$\frac{e_1,e_2,\dots,e_j}{y_1,y_2,\dots,y_j}$
$[y_1, y_2, \dots, y_j]$
T

Figure 4: A Simple Sequential HFG

where $S_1, S_2, \dots S_m$ are *sequential* HFGs . A *sequential* HFG is one in which all transitions are of the form (S, a, S') where $\mid S \mid = \mid S' \mid = 1$. In otherwords, sequential HFG is one in which every *action* has exactly *one* predecessor state and one *successor* state. Each sequential HFG S_i is translated into a CML *thread.* Execution-wise sequential HFGs are not strictly sequential. They could *spawn* new threads which could execute in parallel. An example of such a scenario is the implementation of *compound-actions.* A separate thread is spawned to execute each component of a compound-action. This will be illustrated in an example later in this section.

Translating Sequential HFGs:

Each sequential HFG is translated into a CML thread which consists of a set of mutually recursive function definitions (one function for every transition).

Let $((S, [x_1, x_2, \dots, x_n]), a, (T, [e_1, e_2, \dots, e_j]))$ be a transition in a sequential HFG . This is translated into a function definition in CML as follows: The control state name S (in the precondition of the transition) becomes the name of the function, the datapath variables $x_1, x_2, \dots, x_n$ become the formal parameters of the function, the action a becomes the *body* of the function and the postcondition of the transition is translated into a function call, with the control state name T being the name of the function and the expression $e_1, e_2, \dots, e_j$ being the *actual* parameters.

Figure 4 shows a simple sequential HFG and the CML code fragment below illustrates the translation scheme outlined above.

```
fun S x1 x2 ... xn =  ( body a; T e1 e2 ...ej)
and
    T y1 y2 ... yj =  ( body b; S g1 g2 ...gn)
```

Here body a and body b denote the CML code implementing actions a and b. Generating CML code corresponding to particular hopCP action will be discussed next.

Translating Actions

Expression actions are directly *compiled* into CML code since they have a similar evaluation semantics. The rest of the action categories are translated as follows:

- Data Query: A data query $p?x$ is implemented as a **receive** or a **accept** operation on a synchronous channel p in CML. Note that we have declared p as a synchronous channel in STEP 1. For example, $((S,[x_1,x_2,\ldots,x_n]),p?x,(T,[e_1,e_2,\ldots,e_j]))$ will be simulated as

```
fun S x1 x2 ... xn =  let
                          val x = accept p
                      in
                            T e1 e2 ...ej
                      end
```

  Here we assumed a 2-way rendezvous on channel p. The implementation of a multiway rendezvous will be illustrated later.

- Data Assertion: A data assertion $p!e$ is implemented as a **transmit** or a **send** operation on a synchronous channel p in CML. For example, the transition $((S,[x_1,x_2,\ldots,x_n]),p!e,(T,[e_1,e_2,\ldots,e_j])$ will be simulated as

```
fun S x1 x2 ... xn =  (send(p,e); T e1 e2 ...ej)
```

Translating Special Constructs:

There are three special constructs in hopCP which do not have direct counterparts in CML. These are implemented as follows:

- Compound Actions:

  Compound actions are implemented by spawning a new thread to implement each component of the compound action and waiting for the completion of all the threads. For example, the transition $((S,[x_1,x_2,\ldots,x_n]),(a_1,a_2),(T,[e_1,e_2,\ldots,e_j]))$ has a compound action a with a_1 and a_2 as its components. This will be simulated as follows:

```
        fun S x1 x2 ... xn = let
                                val c__54 = channel ()
                                val c__55 = channel ()
                                fun s__56 x1 x2 ... xn = (body a1; send(c__54,0))
                                fun s__57 x1 x2 ... xn = (body a2; send(c__55,0))
                             in
                                (spawn (fn () => s_56 x1 x2 ... xn);
                                 spawn (fn () => s_57 x1 x2 ... xn);
                                let
                                    val _ = accept c__54
                                    val _ = accept c__55
                                in
                                   T e1 e2 ...ej
                                end
                             end
```

s__56 and s__57 are the new threads spawned to implemented a_1 and a_2. c__54 and c__55 are temporary channels to implement the synchronization between the threads s__56 and s__57. Note that a1 and a2 could be performed concurrently.

- Asynchronous Ports:

```
(*
 * An Asynchronous Multicast Channel abstraction.
 *)

signature ASYNCMULTICAST =
  sig
    structure CML : CONCUR_ML
    type 'a mchan
    val mChannel : '1a -> '1a mchan
    val newPort : 'a mchan -> 'a CML.event
    val multicast : ('a mchan * 'a) -> unit
  end (* ASYNCMULTICAST *)
```

Asynchronous ports are implemented by a *one-place buffer* abstraction whose signature is shown above. The operation mChannel creates a new asynchronous port, the operation multicast is used to *write* a value on the asynchronous port and the operation newPort is used to read from the asynchronous port. The value on the asynchronous port can be read without synchronizing. The assignment action in hopCP, a := e is implemented by the CML code fragment

```
let
    val a = AsyncBarrier.mchannel 0   (* initialized to 0 *)
    val a__9 = AsyncBarrier.newPort a
in
        AsyncBarrier.multicast(a,e)
end
```

- Multiway Rendezvous: Multiway rendezvous is implemented by an abstraction which implements a *busmaster*. The producer checks in the value to the busmaster and waits for the acknowledgement; the receivers wait for a value from the busmaster and an acknowledgement to proceed further; the busmaster receives the value from the producer and transmits it to all the receivers and then sends an acknowledgement to all the receivers and the producer. This implements both *broadcast* and *barrier synchronization*. If only broadcast is desired, the receivers need not wait for a final acknowledgement. They can proceed as soon as they receive the value from the busmaster. The code implementing the barrier abstraction is omitted to conserve space.

Translating Control Structure:

The control structure of hopCP is simulated in CML in the following manner: The sequencing construct (`->`) is implemented explicitly by the SML sequencing operator (`;`) or implicitly by a `let` construct. The choice construct or guarded commands (denoted by `|`) is implemented directly by the non-deterministic `choose` combinator of CML. The `||` construct is simulated by spawning independent threads for each sequential HFG as discussed above.

Unsynchronized Actions:

Some of the actions in a hopCP description do not have *corresponding* partners for rendezvous. In a hardware scenario these correspond to the external inputs and outputs. In CFSIM, unsynchronized inputs are *directed* to the standard input (i.e. keyboard) and unsynchronized outputs are *directed* to the standard output. This feature was found to be extremely useful in simulating concurrent specifications, because unsynchronized inputs could be used to *arrest* the progress of the system and obtain the effect of *single-stepping* through the behavior.

Salient Features of CFSIM

- Efficiency: The size of the simulator is proportional to the number of transitions in the HFG which is usually small because we initially eliminate the $||$ operator by *decomposing* the HFGs into sequential HFGs . The simulation is extremely fast because the CML code is directly executed as opposed to being interpreted which is common with most of the simulators. For example, it took less than 90 seconds (with garbage collection) to simulate the operation of a communication chip (Intel 8251) with more than 160 states and 6 concurrent processes, on a Sparc IPC with 24 Mb of memory. The operation involved approximately

32000 synchronizations and more than 60000 function calls.

- Static Checks: Since we translate the *HFG*s into CML source code and execute them in Standard ML environment, most of the static checks like consistency of types of the variables, name clashes, undefined variables and function names etc. are detected during compilation. This is facilitated by the strong typing offered by Standard ML.
- Interactive: CFSIM generates interactive simulators wherein the user can *step-through* the execution of the module by controlling the *input* to the system. This is facilitated by directing all the *unsynchronized* output actions in a *HFG* to the standard output and the *unsynchronized* input actions to the standard input. This is extremely useful for debugging *asynchronous* circuits where time is *continuous*. An unsynchronized action could be used to discretize the behavior.
- Flexibility: hopCP is an evolving language. It is being continuously modified to cater to new application scenarios. Since CFSIM is based on a higher-order applicative language with a sophisticated module facility, it is very easy to build abstractions to simulate the new constructs of hopCP and support the evolution of the HDL.

5 Tester Processes and Design Validation

In this section we will describe how hopCP specifications can be validated using CFSIM. Validation of specifications via CFSIM involves two phases: (i) identification of interesting *modes of behavior* of the system, and (ii) construction of high-level *simulation vectors*, which *enable* the chosen mode of behavior. These are now illustrated on the pipelined stack specification introduced earlier.

5.0.1 Tester Modules and Modes of Behavior

A *mode of behavior* of a system is a partial order of control/data related actions on the system that has a well defined and intuitive outcome. In terms of their overall execution effect, they correspond to an expression involving many operations of the system. For example, if one were to view the module M being specified in hopCP as an *abstract data type*, the various *axioms* which algebraically characterize M each describe their own *modes of behavior*. For example, let P be the pipelined stack, $x \in VAL$, then $top(push(P, x)) = x$ is an axiom of the stack datatype which P should satisfy; the corresponding mode of behavior is the partial order of actions captured by the following tester process:

```
TESTPUSH[] <= usergive?x -> push!x -> top?y -> (   (x=y)      -> EXECUTIONOK[]
                                                | (not(x=y)) -> ERROR[] )
```

There are several other axioms which a pipelined stack should satisfy. Each of these axioms could be encoded as a *mode of behavior*.

A *tester module* for a hopCP specification H is a hopCP description of a module which interacts with the system being tested (i.e. H) and *guides* the execution of H along a chosen *mode of behavior*. A tester module can be viewed as an *interface* between the user and the system under test. It receives the inputs from the user (via the CFSIM interactive environment) and provides the necessary stimulus to the system and it receives the responses from the system and channels them back to the user.

Testers can be written in such a manner that they check the results of the simulation themselves and give a "yes" or "no" answer. This is the reason why algebraic axioms (such as used above) are a good source for generating testers. This avoids the designer having to manually read and certify voluminous simulation outputs.

Illustrating Tester Modules

Let us consider validating the pipelined push operation in our example. To recapitulate, the memory module being used in the pipelined stack example, can entertain new read/write requests when the current read is still in progress. It delivers the result of the $read_i$ when it is processing $read_{(i+1)}$ or $write_{(i+1)}$, where i is the operation number. The controller pipelines the push operation by issuing a `write` request to the `Memory` module and while the `Memory` is being updated it can entertain the next operation.

A tester module called TEST which validates this operation is shown below

```
MODULE TEST
TYPE
        word : vector 16 of bit;
SYNCPORT
        reset!,top!,pop!,next? : bit;
        push! : word;
FUNCTION

BEHAVIOR
   (TEST [] <= reset! -> TEST_AUX [];
    TEST_AUX [] <=  push!1 -> push!0 -> pop! -> top! -> next? -> TEST [])
END
```

The tester module starts in a initial state denoted by TEST where it issues a *reset* operation to the pipelined stack module. Then it issues two push operations followed by a pop and top. The next operation is a dummy action which is an *unsynchronized* operation and facilitates the simulation process by providing a means to interact with the system through the keyboard. This is not necessary if the tester module was written as a *finite* process.

The CML code generated by CFSIM for the behavior part of the pipelined stack and the tester is presented in the appendix. Executing the code in CML environment validates the pipelined push operation of the stack. The executable CML code is quite compact and takes less than a second to validate the operation in question.

5.1 Discussion on Tester Processes and Design Validation

The stack example shown in this is paper is fairly straightforward and does not illustrate the full power of the tester process in the design validation task. The concept of tester processes could be extremely useful in the validation of hopCP specifications of control-intensive integrated circuits like the Intel 8251 USART [3], which depict concurrency and complex protocols. Each of the three submodules in the USART had over 50 states and 30 transitions. Simulating all possible interactions of such a specification is neither efficient nor meaningful. Generating simulation vectors and understanding the simulation output for such specifications is quite challenging. Tester processes and the concept of identifying modes of behavior enables one to *filter* out most of the irrelevant details and focus on the input/output relationships for the particular facet of the behavior being simulated. This is achieved by making the tester process more *intelligent* than a simple *sequence* of inputs and outputs. The full hopCP specification language could be used to write the tester modules. This gives us the ability to write testers which can have *conditional* behavior and concurrency. In fact, the tester used to simulate the Intel 8251 USART was itself a collection of three concurrent processes capturing the behavior of the CPU, the serial receiver and the serial transmitter.

One apparent drawback with our approach is the necessity to use an independent tester for each mode of behavior. In general, for complex circuits there could be scores of modes of behavior. So, a naive approach to building tester processes could be inefficient. In practice, one need not do that. Actually something similar to conventional *fault simulation* could be employed, Again, this feature was very widely used in simulating the USART specification. A single tester module was programmed with appropriate *control* words to check for more than one mode of behavior.

Finally some of the attractive features of this style of validation process which could be investigated further are:

- Testers are specified in the same HDL. This opens up several promising avenues for further research like extracting BIST (Built-In-Self-Test) hardware by *synthesizing* the tester module just like the rest of the hopCP specification, including DFT (Design For Testability) ideas in the specification i.e. writing a specification which when synthesized becomes more easily testable.

- *Tester modules* provide a systematic and elegant approach to functional simulation, since most of the details of the simulation are *buried* within the tester module. The user does not have to deal directly with the specification.

- By expressing the *tester module* in hopCP and simulating it via CFSIM we are actually *validating* our system in *truly concurrent* environment.

6 Conclusions and Future Work

We introduced a concurrent HDL called hopCP for system-level specification of VLSI circuits. hopCP is characterized by process-oriented features to specify communication and synchronization, a functional sublanguage to specify computation, and distributed shared variables. We then presented a simulation environment for hopCP called CFSIM. CFSIM is a compiled-code concurrent functional simulator and is obtained by translating the intermediate representation of hopCP specification (HFGs) into CML source code. Several advantages of using a strongly typed, polymorphic, higher-order applicative language for behavioral simulation were pointed out. Finally, we introduced a notion of tester processes to support efficient simulation and preliminary validation of hopCP specifications.

CFSIM is efficient and flexible enough to cater to the simulation requirements for a evolving HDL like hopCP. Currently, CFSIM is being extended in two ways: A high-level synthesis system based on *hierarchical refinement* of actions is being implemented [1]. CFSIM currently simulates specifications at the level of *rendezvous* (or handshakes). CFSIM is being extended to simulate circuits at the level of signal-transitions so that it could be used as a simulator during the synthesis phase. This will be done by treating every wire as an asynchronous port and implementing every signal transition as a pair of *assignment* actions on the corresponding asynchronous port. The other extension planned for CFSIM is to incorporate the ability to verify simple *timing constraints.* This will be done by extending hopCP to specify timing constraints as shown in [5] and using the `waitUntil` construct of CML.

Acknowledgements: We wish to thank John Reppy of Cornell University for providing CML and the SML of New Jersey implementation team for providing and supporting SML.

References

1. AKELLA, V., AND GOPALAKRISHNAN, G. 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* (Apr. 1991).

2. AKELLA, V., AND GOPALAKRISHNAN, G. hopCP: A Concurrent Hardware Description Language. Tech. Rep. UUCS-91-021, Department of Computer Science, University of Utah, Oct. 1991.

3. AKELLA, V., AND GOPALAKRISNAN, G. Specification and Validation of a USART in hopCP. Tech. rep., Department of Computer Science, University of Utah, 1991. In preparation; available upon request from the authors.

4. APPEL, A. W. *Compiling with Continuations.* Cambridge Univ. Press, 1992. ISBN 0-521-41695-7.

5. NESTOR, J. A., AND E.THOMAS, D. Behavioral Synthesis with Interfaces. In *Proceedings of the International Conference on Computer-Aided Design* (Nov. 1986), pp. 112–115.

6. REPPY, J. H. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (June 1991).

7 Appendix

```
MODULE CTRL
TYPE
  word : vector 16 of bit;
  addrType : vector 16 of bit;

SYNCPORTS
  reset?, push?, pop?, top?, dec!, inc!, read!, : bit;
  ld!, write!: addrType;

FUNCTION

BEHAVIOR
    (PCTRL [] <= (reset? -> ld!0 -> PCTRL [])
                 |(push?v1 -> inc! -> write!v1 -> PCTRL_AUX [])
                 |(pop? -> dec! -> PCTRL [])
                 |(top? -> read! -> PCTRL []);

     PCTRL_AUX [] <= (reset? -> ld!0 -> PCTRL [])
                    |(push?v2 -> inc! -> write!v2 -> PCTRL_AUX [])
                    |(pop? -> dec! -> PCTRL [])
                    |(top? -> read! -> PCTRL[]))
END
```

hopCP Specification of the Pipelined Stack Controller

```
fun pstack () =
    let
        val addr = AsyncBarrier.mChannel 0
        val addr__49 = AsyncBarrier.newPort addr
        val top = channel ()      val pop = channel ()
        val push = channel ()     val reset = channel ()
        val write = channel ()    val ld = channel ()
        val read = channel ()     val inc = channel ()
        val dec = channel ()      val next = channel ()
        val dout = channel ()

        fun s__21 v1 = (( send (write,v1 ) ; PCTRL_AUX ()))
        and
            s__28 () = (( send (read,0); PCTRL () ))
        and
            s__25 () = (( send (dec,0); PCTRL () ))
        and
            s__22 v1 = (( send (inc,0); s__21 v1 ))
        and
            s__19 () = (( send (ld,0 ) ; PCTRL ()))
        and
            PCTRL () = sync(choose [wrap (receive reset, fn (_u_)  =>( s__19 () )),
                                    wrap (receive push, fn (v1)  => (s__22 v1  )),
                                    wrap (receive pop, fn (_u_)  =>( s__25 () )),
                                    wrap (receive top, fn (_u_)  =>( s__28 () ))])
        and
            PCTRL_AUX () = sync(choose [wrap (receive reset, fn (_u_)  =>( s__31 () )),
                                        wrap (receive push, fn (v2)  => (s__34 v2  )),
                                        wrap (receive pop, fn (_u_)  =>( s__37 () )),
                                        wrap (receive top, fn (_u_)  =>( s__40 () ))])
        and
            s__31 ()  = ( ( send (ld,0 ) ; PCTRL () ))
        and
            s__34 v2  = ( ( send (inc,0); s__33 v2  ))
        and
            s__37 ()  = ( ( send (dec,0); PCTRL () ))
        and
            s__40 ()  = ( ( send (read,0); PCTRL () ))
        and
            s__33 v2  = ( ( send (write,v2 ) ; PCTRL_AUX () ))

        fun TEST ()   = ( ( send (reset,0); TEST_AUX () ))
        and
            TEST_AUX ()  = ( ( send (push,1 ) ; s__47 () ))
        and
            s__47 ()  = ( ( send (push,0 ) ; s__46 () ))
        and
            s__46 ()  = ( ( send (pop,0); s__45 () ))
        and
            s__45 ()  = ( ( send (top,0); s__44 () ))
        and
            s__44 ()  = (CIO.print( "Waiting for Input on Channel next? \n");
                         let
                             val _ =  input_int(sync(CIO.input_line std_in))
                         in
                             CIO.print("Synchronized on channel next?\n");TEST_AUX ()
                         end)
```

.... Continued On Next Page

.... Continued From Previous Page

```
      fun MEM ms   = sync(choose [ wrap (receive write, fn (din)  =>
                                          (MEM (let
                                                  val addr =  CML.sync  addr__49
                                               in
                                                  (update(ms,addr,din))
                                               end))),
                                    wrap (receive read, fn (_u_)  =>
                                         (MEM_AUX ms (let
                                                        val addr =  CML.sync  addr__49
                                                     in
                                                        addr
                                                     end)))])
      and
          MEM_AUX s a  =
          sync(choose [wrap (receive write, fn (din)  => (s__14 s a din)),
                       wrap (receive read, fn (_u_)  =>( s__16 s a  ))])
      and
          s__14 s a din = ( (CIO.print (" Output on Channel dout!"^Integer.makestring(s)^"\n");
                              MEM (let
                                      val addr =  CML.sync  addr__49
                                   in
                                      (update(s,addr,din))
                                   end)))
      and
          s__16 s a  = ((CIO.print (" Output on Channel dout!"^Integer.makestring(s)^"\n");
                          MEM_AUX s (let
                                        val addr =  CML.sync  addr__49
                                     in
                                        addr
                                     end)))

      fun AREG cs = sync(choose [wrap (receive ld, fn (inp)  => (s__3 cs inp )),
                                 wrap (receive inc, fn (_u_)  =>( s__5 cs  )),
                                 wrap (receive dec, fn (_u_)  =>( s__8 cs  ))])
      and
          s__3 cs inp   = ((AsyncBarrier.multicast(addr,inp);  AREG inp  ))
      and
          s__5 cs   = ((AsyncBarrier.multicast(addr,(cs + 1));  AREG (cs + 1)  ))
      and
          s__8 cs   = ((AsyncBarrier.multicast(addr,(cs - 1));  AREG (cs - 1)  ))
   in
      spawn (fn () => PCTRL () );
      spawn (fn () => TEST () );
      spawn (fn () => MEM 0  );
      spawn (fn () => AREG 1  );
      ()
   end;
```

CFSIM Generated CML code for the Pipelined Stack Example
(Memory || CTRL || AddReg || TEST)