Specification and Validation of Control Intensive ICs in 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

Keywords: asynchrony, behavioral simulation, formal methods, hardware description languages, formal specification and validation

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, non-trivial computational requirements, 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 hardware description language 'hopCP'. A feature of hopCP is that it supports communication via *asynchronous ports* in addition to *synchronous* message passing. Asynchronous ports are distributed shared variables writable by exactly one process. We show the usefulness of this combination of communication constructs. We outline 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.

1 Introduction

Over the last two decades, VLSI technology has advanced by leaps and bounds, and has contributed to a rapidly increasing performance/price ratio of hardware. With these improvements, however, have come a variety of new problems. Although the speed and the scale of VLSI systems continues to grow, their functional complexity may not scale at the same rate, unless some of the problems that have begun to creep up at the level of system design are properly tackled and solved.

There are many sources for the problems encountered at the system level of hardware design. Many of these are problems of scale akin to those found in the design of large software systems. The more serious of these problems are, however, due to the concurrent nature of hardware, and because of the large number of complex features that hardware designers are trying to support in VLSI systems they are currently building.

We can illustrate many of the above mentioned problems, as well as possible solutions, through one example: the Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART) [23]. Integrated circuits (ICs) such as the 8251 USART exhibit diverse behaviors. They typically possess independent threads of execution, have coexistent *synchronous* (clocked) and *asynchronous* (unclocked) subcomponents, support multiple modes of operation, such as the *interrupt-driven* and the *polled* modes. They are programmable to set the baud rate, the number of stop bits, start bits, error flags, and the synchronization scheme *etc.*. They can perform computations, such as error-checking, assembling and disassembling of data, and code-conversion. Such ICs are commonly classified as "control intensive". There are very many control intensive ICs in day-to-day use today; we selected the 8251 because it has been widely used in the past as a benchmark for comparing

high-level synthesis tools and their associated hardware description languages (HDLs).

We wish to contribute to the current state of the art of specifying, verifying, and ultimately, of synthesizing control intensive ICs. Currently, control intensive ICs are most commonly described through a combination of natural language descriptions, timing diagrams, state-charts, *etc.* These informal descriptions are prone to misinterpretations, and are not machine readable. They cannot be used as a basis for design validation.

Lately, such ICs are being specified in HDLs. For example, the 8251 USART has been specified in many HDLs such as ISPS [8], VHDL [44], and Verilog [40, 43]. However, when studied from a formal point of view, these HDLs have many shortcomings. None of the currently popular HDLs (such as referred to above) have a well specified and simple formal semantics. The advantages of providing formal semantic definitions needs no emphasis: it helps describe language constructs precisely, put the language concepts to precise tests (*e.g.* exhibiting semantic properties), and makes it easy to develop verification/validation procedures for the language. In this paper, we develop the specification of the 8251 in our HDL hopCP—a semantically well specified and simple language [1].

The design of control intensive hardware systems has a lot in common with the specification and verification of concurrent software systems. Several verification techniques for concurrent software have been widely studied [7]. Research prototypes embodying some of these techniques (such as the Concurrency Workbench [12] and COSPAN [21]) are also available. In addition, formalisms such as trace theory [13] and Temporal Logic [30] have been applied for the verification of speed independent asynchronous circuits [14] as well as for verifying concurrent protocols such as cache coherence protocols. These tools have also been used by us in our past work [18, 33]. However, few of these tools or techniques have been applied for the verification of real-world control intensive hardware systems where one has to address data dependent control flows, exception handling, and similar issues.

In this paper, we do not attempt to *formally verify* the requirements specification of the 8251 against the proposed design specification of the 8251 in hopCP; instead, we demonstrate the application of *formal methods* in the design of the 8251, centered around hopCP. We do this by presenting the hopCP language, outlining its operational semantics (detailed in [1]), and present tools and techniques that help validate hopCP descriptions. As Hall points out [20], even without conducting formal verification at all levels of design representation, the application of formal methods can enhance the degree of confidence in a design, help discover design flaws, and promote overall understanding of the design. Also, by presenting the description of the 8251 in a semantically well specified notation, we believe that some of the difficulties of formally verifying such systems will be brought to the surface.

Currently there is a growing trend towards applying formal methods in the design of real-world hardware systems [39, 16, 15]. Most of the current efforts do not address ICs that, in addition to exhibiting control intensive behavior also support non-trivial computations. Our contributions in this paper include the following: (i) the design of a semantically well specified and simple HDL tailored for the specification of ICs that exhibit control intensive behavior as well as support non-trivial computations; (ii) the design of tools for validating the design specifications, that can ultimately lead to the verification and "correct by construction" synthesis of these ICs. We now examine these features of our work.

1.1 HDL Features

1.1.1 Synchronous Communication Primitives

The task of specifying and designing complex control intensive ICs is as hard as that of writing parallel programs. It is well known that without the support of high-level concurrency primitives, concurrent programming can be a nightmare. Virtually all the popular HDLs available today either omit concurrent process modeling primitives altogether, or provide only very low level primitives. For example, none of the languages ISPS, VHDL, or Verilog provide a high level synchronization primitive. Synchronizations between various communicating processes are, in fact, implemented using explicit handshakes. This can make descriptions in these HDLs hard to follow. It also makes it easier to accidentally introduce deadlocks or other errors. A description of the 8251 in a language that provides a high level synchronization primitive is, on the other hand, much more readable.

hopCP supports multiway rendezvous. This means that the sender and all the receivers on a synchronous port wait for each other, and the value put out by the sender is copied by all the receivers before all the processes participating in the multiway rendezvous can proceed. Synchronous ports could be inputs or outputs which are distinguished by the last character of the portname: ? for input, and ! for output. It is possible to employ synchronous ports which deal only with control and do not have data.

Multiway rendezvous is a powerful notion which facilitates the specification of a wide variety of concurrent algorithms very naturally [11]. 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 situations are *impossible* to specify without multiway rendezvous, but it becomes awkward to model them in terms of two-way rendezvous.

1.1.2 A Multi Paradigm HDL

Many HDLs with a well specified and simple formal semantic definition are available today. These HDLs are designed based on a single paradigm; for example, many of these HDLs adopt a purely functional view of computations [24, 38], many adopt a formal process model [22, 36] while others are based on special theories like the Trace theory [15]. An HDL based on a single paradigm (*e.g.* functional view of computations) is well suited for a limited class of circuits (*e.g.* computation oriented), but fares poorly when it comes to dealing with both computations and control/communication activities. Our solution is to adopt a multi-paradigm language that amalgamates features from process oriented languages (*e.g.* CSP) to model control/communication and functional languages to model computations.

1.1.3 Use of Asynchronous Ports

Design seldom proceeds through top-down refinement, in practice. This is especially true in the area of hardware design where a designer often makes decisions based on his/her knowledge of the underlying circuitry or the geometry of the design. In other situations, a hardware designer is forced to design around existing parts. To express design intent in this richer domain, lower level primitives are often required.

As an example of the need for lower level primitives, consider an example. A binary counter supports the operations *load* and *increment* and its output is connected to a module M. The act of incrementing the counter causes the counter to put out a new value on its output port. This happens in the lower level implementation of the counter whether the external world is "interested" in this

value or not.

How do we model this detailed behavior using a CSP-like language? If the value production by the counter is modeled through rendezvous communication, we run into the following problem: the value may be of "no interest" to M at this time. We would thus be forced to write a specification in which module M picks up and then discards this value.

Another use of asynchronous ports is to model status signals that can be set many times before being read, as well as read many times after being set. This style of process interaction is very difficult—if not impossible to specify—using synchronous communication (i.e. rendezvous) alone. In short, hardware designers often wish to unbundle the synchronization and the value-communication aspects of a rendezvous. In hopCP, we provide a construct known as asynchronous ports to model such situations.

Asynchronous ports are distributed single-writer multiple-reader shared variables. Other HDL users also have felt the need for asynchronous ports. For example, the "first asynchronous microprocessor" [29] is generally considered to be specified in a CSP-like language; yet, asynchronous port assignments are *extensively* used in the specification. In short, without using asynchronous ports, many hardware systems become very difficult to describe.

1.2 Specification/design Validation Tools

1.2.1 Seriality Checking

Asynchronous ports must be used with caution. They must not be concurrently read and written by two different threads of execution. In hopCP we provide support for safe usage of asynchronous ports through seriality checking, as will be described later. Although it is in principle possible to allow asynchronous ports to be concurrently read and written through the use of special synchronizers such as Q-flops [37], we do not currently support this capability in hopCP. Though asynchronous ports are used in [29], no support for the safe usage of asynchronous ports is offered in their system, thus making it users' responsibility to use asynchronous ports safely.

1.2.2 Compiled Code Simulation

A high-level specification is not very useful unless it is supported by a methodology to *validate* it. We provide a simulation environment called CFSIM to validate hopCP specifications. CFSIM is compiled-code concurrent functional simulator obtained by translating hopCP specifications into CML (Concurrent ML) source code. CML facilitates building concurrent-programming abstractions and is implemented efficiently capitalizing on the continuation-passing style technology of the SML of New Jersey compiler [6, 35]. CML is a practical language tested in a variety of large-scale projects like *eXene* (a muti-threaded interface to X protocol), distributed ML and distributed Nuprl (a theorem proving environment) implementations.

1.2.3 High Level Validation using Testers

The behavioral specification of complex control intensive ICs can be hard to follow, even if such ICs are described in a modern concurrent HDL, such as Occam [34, 32, 28] or hopCP. Although a simulator can help "animate" the specification for selected scenarios, output waveform traces produced by typical simulation runs offer very little help in understanding or debugging complex control intensive ICs. In the hopCP system, high level validation is supported in two ways. In the first approach, the designer can write *tester* processes that can simulate the *environment* of the process being validated. 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 simulation methods. In the second approach to high level validation, C, T_1, and T_2 can be interconnected and subject to the *parComp* algorithm. *parComp* will obtain a single process, CT, whose behavior is equivalent to that of $C \parallel T_1 \parallel T_2$. Process CT can be analyzed to reveal general properties of the composite system (*e.g.:* "are two actions serial so that they can share a resource?"). These two methods of debugging specifications have proved to be quite valuable, for example, in debugging the Intel 8251 USART specification. The testers that we wrote actually *proved to be very readable and succinct specifications of the system being debugged.*

Organization

The remainder of this paper is organized as follows. In section 2, we provide an overview of hopCP. In section 3, we provide an informal description of the 8251. In section 4, we provide a formal description of the 8251 in hopCP. In section 5, we present details of the hopCP design environment, including details of the behavioral inference algorithm *parComp*, a static analysis tool to detect seriality, the compiled code functional simulator CFSIM, and the use of *tester* processes for debugging. We then provide concluding remarks and outline ongoing work in section 6. An Appendix is also provided, containing deferred details.

2 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 and a behavioral description called the *HFG* (hopCP Flow Graph).

A hopCP specification has six sections described below out of which only the MODULE and the BEHAVIOR sections are mandatory.

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

(ii) TYPES section introduces the datatypes of the communication ports used in the module. Types can be defined in terms of *bit* and *bitvector* which are primitive types.

(iii) SYNCPORT section declares all the *synchronous* communication ports used in the specification. A *synchronous* port allows *rendezvous* style communication, as in CSP.[1] hopCP's rendezvous is *multiway*: the sender and all the receivers on a synchronous port synchronize (waiting for each other), the sent value is copied by all the receivers, and they all proceed. Synchronous ports could be inputs or outputs which are distinguished by the last character of the portname: ? for input, and ! for output.

(iv) ASYNCPORT section declares all the *asynchronous* communication ports used in the specification. An *asynchronous* port is a *shared variable* which provides communication *without* explicit synchronization. Asynchronous ports can be written by only *one module*—its owner. They can be read in any number of other modules than the owner module. The ownership of asynchronous ports is fixed i.e. it cannot be changed dynamically during the execution of the system.

[1] we use "port" and "channel" interchangeably

(v) FUNCTION section contains the *user-defined* functions used in the specification. The functions are written in a *first-order* functional language.

(vi) 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 described next.

2.1 hopCP Flow Graph

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*.

We shall illustrate behavioral description in hopCP using the following examples.

Example 1

Figure 1 describes a module $ex1$ which declares $TxRDY$ as an output *asynchronous* channel and $a?$ and $b!$ as *synchronous* channels. f is a user-defined function used in the behavioral description. f is specified in a first-order functional language augmented with bit-vector manipulation routines. Informally, module $ex1$ 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 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 features:

1. *Assignment Action:* $apo := expr$ where $apo \in AsyncPort$ is an assignment action. In module $ex1$, $TxRDY := 1$ an assignment action which denotes the evaluation of the expression $expr$

```
MODULE ex1
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 1: Illustrating Alternate Behavior and Assignment Actions

(which is 1 in our example) and *updating* the asynchronous port $TxRDY$. 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). It is characterized by the absence of an *rendezvous* or handshake unlike synchronous communication. Indiscreet use of asynchronous communication could lead to undesired behavior like metastability and deadlock. In hopCP framework, unsafe usage of asynchronous communication actions is checked by static analysis of the underlying HFGs and appropriate warnings are issued.

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.

   (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 $ex1$, $(b!(f\ x\ y), TxRDY := 1)$ 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.

```
MODULE ex2
SYNCPORT
        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 2: hopCP Specification Illustrating Parallel Behavior

In the above example, $(even\ y)$ 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 2

The previous example was basically *sequential* in nature except for the restricted form of concurrency introduced by compound actions. Figure 2 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 3.

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 Q and P 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). 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 via *two-way rendezvous*. Multiway rendezvous is said to occur when there is more than one agent willing to perform a data query corresponding to a data assertion. (The converse of the situation—more than one agent asserting a value on the same channel—is not supported in hopCP.) Multiway rendezvous is a powerful notion which facilitates the specification of a wide variety of concurrent algorithms very naturally [11]. It subsumes *broadcast* style of communication (point to multipoint communication) which is very natural in hardware. Multiway rendezvous is not supported by most HDLs currently being used for high level synthesis. Without multiway rendezvous, many situations become awkward to model. Figure 4 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 $(f\ x1\ y1)$

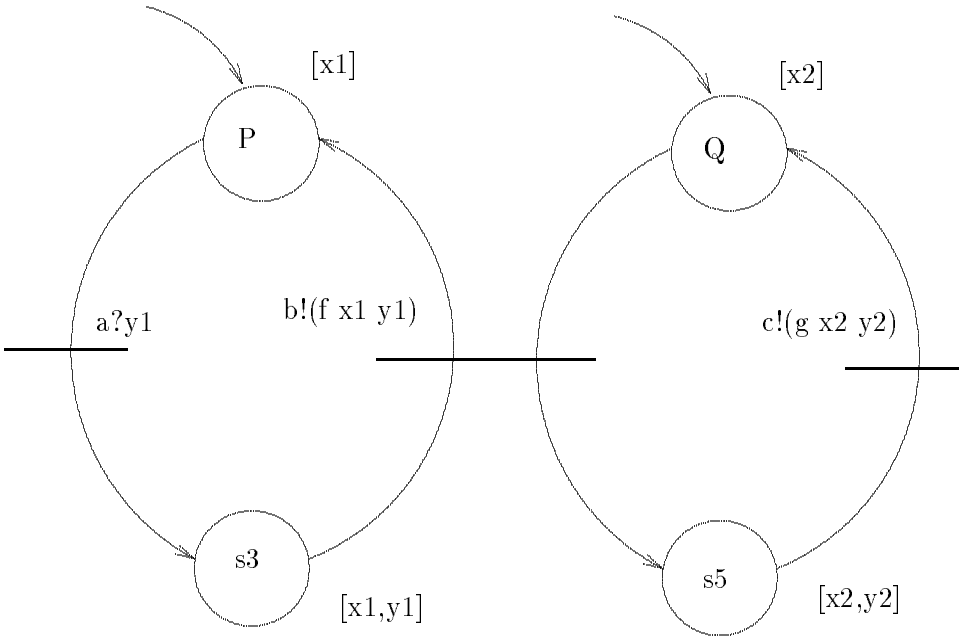

Figure 3: HFG of hopCP Specification in Example 2

```
 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 4: hopCP Specification Illustrating Multiway Rendezvous

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* [27], 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.

3 An Informal Description of 8251

In this section we present the functional description of Intel 8251 USART. We begin by pointing out some of the essential difficulties of specifying a system such as the 8251. We then provide the details, almost verbatim from the manual pages [23].

Intel 8251 is a USART designed for data communication with Intel's microprocessor families. It possess independent threads of execution, has coexistent *synchronous* (clocked) and *asynchronous* (unclocked) subcomponents, and supports multiple modes of operation, such as the *interrupt-driven* and the *polled* modes. It can be programmed for various baud rates, the number of start/stop bits, error conditions, as well as the synchronization scheme. It can perform computations such as error-checking, assembling and disassembling of data, and code-conversion. Such ICs are commonly classified as "control intensive". A single language *with a compositional formal semantics* that can specify all these aspects of control intensive ICs has not been designed to date. It may even be impossible to develop such a language because of the disparate modes of behavior embodied in control intensive ICs.

There are two approaches to specifying control intensive ICs. The most prevalent approach is to describe the detailed implementation of a control intensive IC in a language such as VHDL [44]. Such descriptions are well suited for simulation. Various aspects of the behavior of these ICs can be revealed by applying suitable simulation vectors and observing the responses, on a case by case basis. However such descriptions are not well suited for studying general properties of control intensive ICs because of the lack of a *compositional semantics* by means of which the overall behavior can be inferred from the behaviors of the parts and the interconnections among the parts. The alternative approach involves developing a language in which one can specify many (if not all) the aspects of control intensive ICs in a compositional manner. hopCP is a language of the latter type. Although a language such as hopCP cannot be used to describe all the operational aspects of a control intensive IC, it is well suited for writing high-level descriptions that make the global properties of interest quite explicit. Given the increasing prevalence of high-level synthesis tools [31, 4, 17], descriptions such as written in hopCP can be compiled to derive large portions of the silicon implementation of control intensive ICs.

We now proceed to describe the 8251 in great detail. The 8251 can be programmed by the CPU to operate under many serial data transmission schemes. The USART accepts data characters from the CPU in a parallel format and converts them into a continuous serial data stream for transmission. *Simultaneously*, it can receive serial data streams and convert them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the complete status of the USART at any time. These include data transmission errors, and control signals such as `SYNDET and TxEMPTY`. Figure 5 shows the block diagram and the pin configuration of the Intel 8251.

The 8251 is programmable by the system software. A set of control words (called MODE and COMMAND) must be sent out by the CPU to initialize the 8251 to support the desired communication format. These control words will program the baud rate, character length, number of stop bits,

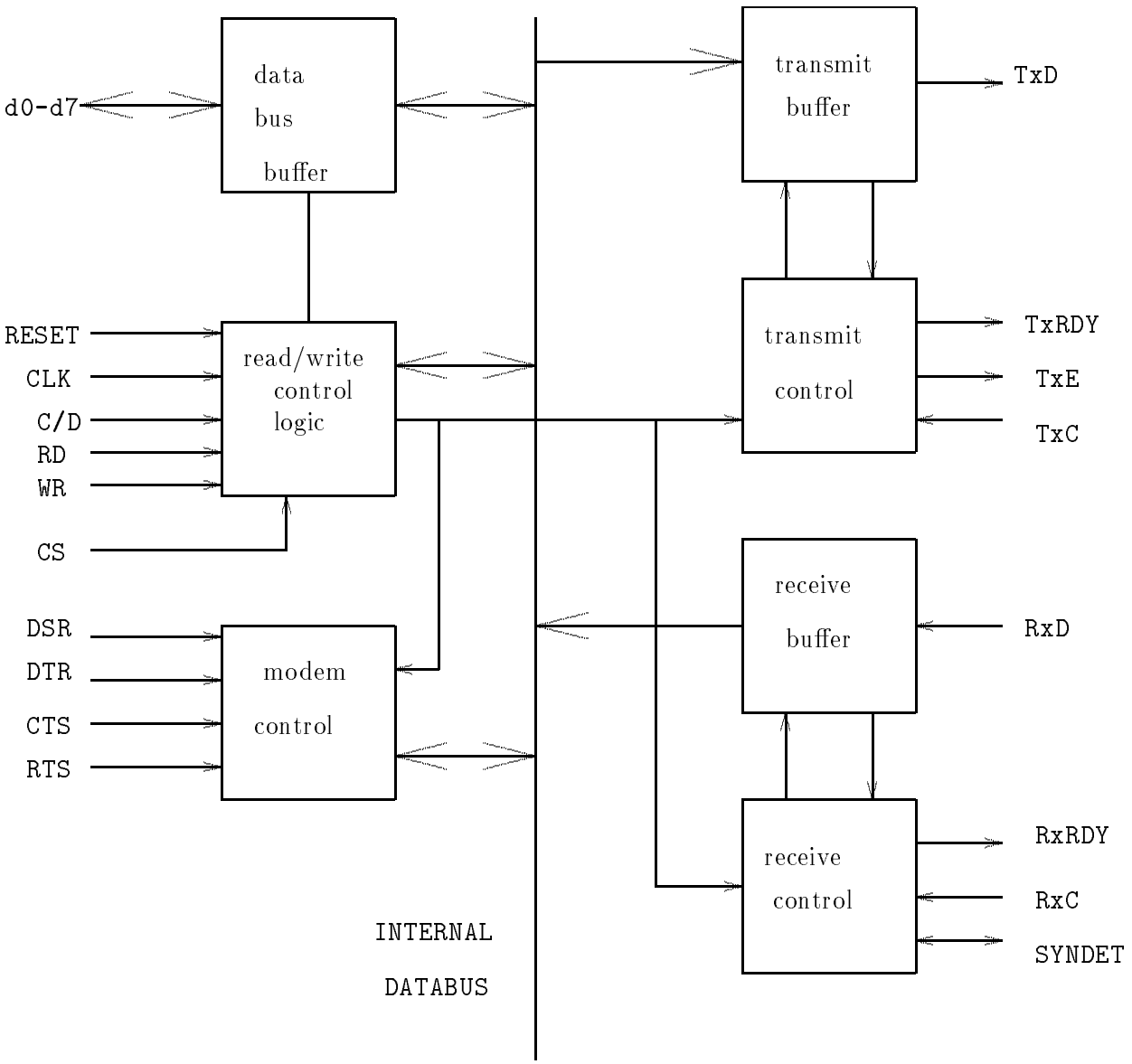

Figure 5: Intel 8251 Block Diagram and Pin Configuration

synchronous or asynchronous operation, even/odd/no parity, etc. In the *synchronous mode*, options are also provided to select between internal and external character synchronization.

Once programmed, the USART is ready to perform its communication functions. The `TxRDY` output is raised *high* to signal the CPU that the USART is ready to receive a data character from the CPU. This output(`TxRDY`) is reset automatically when the CPU writes a character into the 8251. On the other hand, the 8251 receives serial data from the MODEM or I/O device. Upon receiving an entire character, the `RxRDY` output is raised *high* to signal the CPU that the 8251 has a complete character ready for the CPU to fetch. `RxRDY` is reset automatically upon the CPU data read operation. The 8251 cannot begin transmission until the `TxEnable`(Transmitter Enable) bit is set in the COMMAND word and its has received a Clear To Send ($\overline{\texttt{CTS}}$) input. The `TxD` output will be held in the marking state upon *reset*. Next let us examine the detailed requirements of the synchronous and asynchronous modes of transmission and reception.

3.1 Asynchronous Mode (Transmission)

Whenever a data character is sent by the CPU the 8251 automatically adds a Start bit (low level) followed by the data bits (least significant bit first), and the programmed number of Stop bits to each character. Also, an even or odd Parity is inserted prior to the Stop bit(s) as defined by the MODE instruction. The character is then transmitted as a serial data stream on the `TxD` output. The serial data is shifted out on the falling edge of the $\overline{\texttt{TxC}}$ as defined by the MODE instruction. When no data characters have been loaded into the 8251, `TxD` output remains *high*.

3.2 Asynchronous Mode (Receive)

The `RxD` line is normally high. A falling edge on this line triggers the beginning of a START bit. The validity of this START bit is checked by again strobing this bit at its nominal center. If a low is detected again, it is a valid START bit, and the bit counter will start counting. The bit counter thus locates the center of the data bits, the parity bit (if it exists) and the stop bits. If a parity error occurs, the Parity Error flag is set. If a low is detected as the STOP bit, the Framing Error flag is set. The STOP bit signals end of a character. The character is then loaded into the parallel I/O buffer of the 8251 and the RxRDY pin is raised to signal the CPU that a character is ready to be fetched. If a previous character has not been fetched by the CPU, the present character replaces it in the I/O buffer and the Overrun Error flag is set (and the previous character is lost). The occurrence of any of these errors will not affect the operation of the 8251.

3.3 Synchronous Mode (Transmission)

The `TxD` output is continuously high until the CPU sends its first character to the 8251 which usually is a SYNC character. When the $\overline{\texttt{CTS}}$ line goes low, the first character is serially transmitted out. All characters are shifted out on the falling edge of the $\overline{\texttt{TxC}}$. Data is shifted out at the same rate as the $\overline{\texttt{TxC}}$. Once the transmission has started, the data stream at the `TxD` output must continue at the $\overline{\texttt{TxC}}$ rate. If the CPU does not provide a character before the transmitter buffer becomes empty, SYNC characters will be automatically inserted in the `TxD` output stream and `TxEMPTY` pin is set high to indicate the same.

3.4 Synchronous Mode (Receive)

In this mode, character synchronization can be internally or externally achieved. If the SYNC mode has been programmed, ENTER HUNT command should be included in the the first COMMAND word.

Data on the RxD pin is sampled on the rising edge of $\overline{\texttt{RxC}}$. The Receiver Buffer is compared at every bit boundary with the first SYNC character until a match occurs. If the 8251 is programmed with two SYNC characters, then the subsequent received character is also compared; when both SYNC characters match, the USART ends the HUNT mode and is in character synchronization. The SYNDET pin is set high and is reset by subsequent STATUS read operation. In the external SYNC mode, synchronization is achieved by applying a high level on the SYNDET pin, thus forcing the 8251 out of the HUNT mode. Parity and Overrun errors are checked in the same way as above.

The COMMAND word controls the actual operation of the 8251 by issuing commands like Enable Transmit/Receive, Error Reset and Modem Control and Internal Reset. The COMMAND instruction can be issued anywhere during data transmission while the MODE instruction can be issued only after an internal or external reset. In a data communication environment it is necessary to examine the "status" of the active device to ascertain if errors have occurred or other conditions that require the processor's attention. The 8251 has facilities that allow the programmer to "read" the status of the device at any time during the functional operation. Some of the bits of the STATUS word have identical meaning to the external output pins so that the 8251 can be used in a completely polled or interrupt-driven environment; the TxRDY signal is an exception.

4 Formal Description in hopCP

The above informal specification makes it clear that unless a precise and succinct notation is employed, the overall behavior of the 8251 will not be comprehensible for a user. We now discuss a suitable logical organization of a hopCP specification of the 8251.

4.1 Logical Organization of the Specification of 8251

The specification of the Intel 8251 in hopCP raises the following issues:

- Partitioning:

  It is not very useful to specify the whole 8251 as one *monolithic* hopCP module. It would not capture the concurrency in the behavior accurately. Therefore we model it as a collection of three independent processes: main which handles the CPU interface and the modem control, xmit which describes the transmitter section which includes both the synchronous and asynchronous transmission modes and the associated status information, and rcvr which describes the receiver section which includes both the synchronous and the asynchronous modes of behavior.

- Logical Channels:

  The xmit, rcvr and main execute concurrently and communicate with each other using synchronous and asynchronous ports. The communication channels used in the hopCP specification and the electrical pins of the 8251 (shown in figure 5) are *not* in direct correspondence. Hence, we call the communication ports used in hopCP specification as *logical* channels. Several logical channels can be mapped into the same set of *physical* wire(s). This is useful in two ways:(i) it helps us to model bidirectional buses as two separate unidirectional logical channels since bidirectional buses are not allowed in hopCP and (ii) A time-shared bus like $(D_0, D_1, \ldots, D_7)$ which is used to communicate data, status, and control from/to the CPU, is modeled as four separate logical (unidirectional) channels. This makes the specification clearer. Time-shared implementation can be derived in hopCP as an optimization. Derivation of circuits from hopCP is not discussed further in this paper.

- Handling Shared State:

  COMMAND, MODE, STATUS and SYNC characters are variables common to xmit,rcvr, and main processes. COMMAND, MODE, and SYNC characters are written by the CPU and read by all the three processes while STATUS is read by the CPU and written by xmit and rcvr processes. In hopCP, shared variables like COMMAND, MODE, and SYNC characters are handled by keeping local copies of each variable in all the processes which read it and maintaining the consistency of the data by using *multiway rendezvous*. Multiway rendezvous ensures data consistency because value is sent to all the processes participating in the multiway rendezvous at the same time. STATUS is handled by keeping only one copy in the main process and having xmit and rcvr processes send their individual status information to the main processes which does the update.

- Status Signals and Interrupt-driven Mode

  Status signals like `TxRDY` (which announces that the transmitter section is ready to receive the next character) and `RxRDY` (announcing the availability of next character) are modeled in hopCP using asynchronous channels. Asynchronous communication actions do not need synchronization, they involve asserting a value on the associated channel. This enables us to model *interrupt-driven* modes of behavior, because `TxRDY/RxRDY` could be connected to the interrupt lines of the CPU. If so, a status output on `TxRDY/RxRDY` could trigger the corresponding interrupt-handler in the CPU.

xmit,rcvr, and main are implemented by the hopCP modules XMIT, RCVR, and MAIN modules, which are discussed in detail next. The logical interconnection of the three modules is shown in figure 6.

4.2 Main Module

The specification has six sections as described earlier.

```
Module MAIN
Type
        byte: vector 8 of bit;
        Bit: vector 1 of bit
```

The first two sections shown above specify the name of the module and the types of the various communication channels.

```
SyncPort
        indata?, inctrl?,rcvrmain?, rcvrstatus? : byte;
        broadcastdata!, broadcastctrl!, out!, xmitmain! : byte;
        indata!, inctrl!, rcvrmain!, readdata!, readstatus!,rcvrstatus! : byte;
        readdata?, readstatus?, intReset! : Bit

AsyncPort
        reset? : Bit;
        reset! : Bit;
        TxRDY!, RxRDY! : Bit;
        XmitBufferEmpty!, ReceiverBufferEmpty!,extReset!  : Bit
```

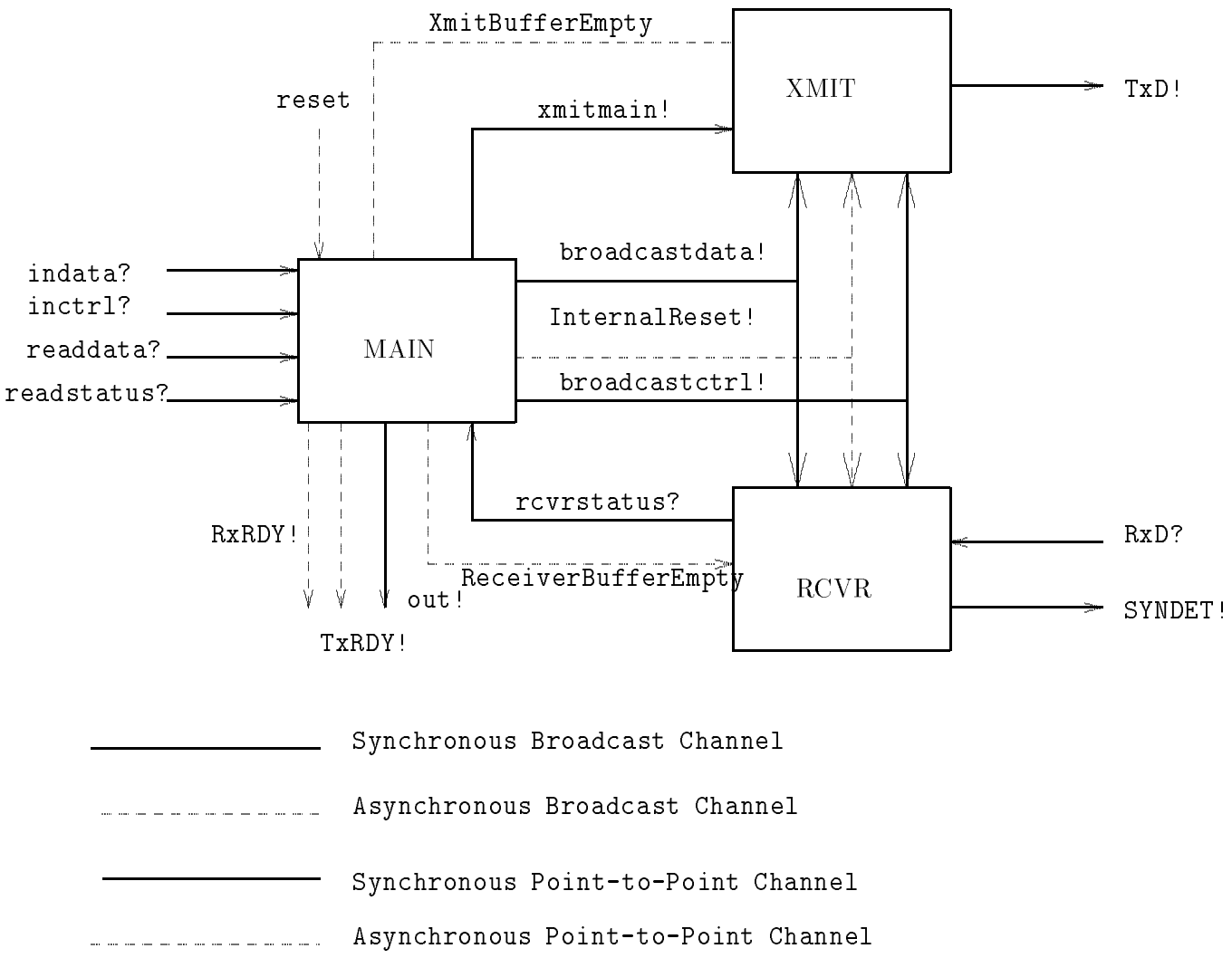

Figure 6: Logical Interconnection of XMIT, RCVR and MAIN modules

The `SyncPort` section describes the synchronous communication channels used in the specification. `indata?,inctrl?` are input channels which carry data and control information from the CPU while `out!` is the output channel which carries data and status information to the CPU. `readdata?` and `readstatus?` are input control channels (only synchronization, no value communication) through which the CPU initiates a data or status read operation. `broadcastdata` and `broadcastctrl` are internal channels which *broadcast* MODE and COMMAND words to XMIT and RCVR modules using multiway rendezvous. `rcvrmain` and `xmitmain` are internal channel to receive status information from RCVR module and send data to XMIT module respectively. The `AsyncPort` section describes the asynchronous channels (shared variables) used in the specification. `reset` and status outputs `RxRDY` and `TxRDY` are modeled as bit-valued asynchronous ports. `XmitBufferEmpty, intReset, extReset, ReceiverBufferEmpty` are internal asynchronous ports which are written by the MAIN module and read by the XMIT and RCVR modules.

```
Function

fun IsTrue x =  if (x=1) then true else false endif;

fun IsFalse x = if (x=0) then true else false endif;

fun SyncMode x =  if (subvector(x,0,1) = 0) then true else false endif;

fun ReadSync2 x =  if (subvector(x,7,7) = 0) then true else false endif;

fun InternalReset y = if (subvector (y,6,6) = 1) then true else false endif;

fun UpdateStatus status new_st = orb(status, new_st)
```

The `function` section describes the user-defined functions used in the specification. They capture the decoding of the MODE words and assembling the STATUS words based on their format in [23]. Note that the functions are expressed in a first-order functional language with built-in routines for bit-level manipulations. `subvector(x,y,z)` returns the value of the integer formed by the bits from y to z from the bitvector denoted by x while `orb` does a simple bitwise OR operation. For example, `subvector(63,0,3) = 15` and `orb(6,5) = 7`.

The `Behavior` section describes the underlying state-transition system whose initial control state is `MAIN_INITIATE`.

```
MAIN_INITIATE [] <= (((IsTrue reset) -> (extReset := 1) -> MAIN_IDLE [])
                  | (not(IsTrue reset)) -> MAIN_INITIATE []);
```

In `MAIN_INITIATE`, the module waits for the `reset` input to go *high*; once reset, the module asserts a 1 on the `extReset` output (an assignment action) which is a signal internal to the USART to reset the XMIT and RCVR modules and proceeds to an *idle* control state called `MAIN_IDLE`.

```
MAIN_IDLE [] <= indata?mode -> broadcastdata!mode -> READ_SYNC_CHAR [mode];
READSYNCCHAR [m] <= (((SyncMode m) -> indata?sync1 -> broadcastdata!sync1
                                                   -> OPTIONAL_SYNC_READ [m, sync1])
                   |((not(SyncMode m)) -> READ_CMD_WORD [m, 0, 0]));
OPTIONAL_SYNC_READ [m, s1] <= (((ReadSync2 m) -> indata?sync2 ->
                                    broadcastdata!sync2 -> READ_CMD_WORD [m,s1,sync2])
                            | ((not(ReadSync2 m))            -> READ_CMD_WORD [m,s1,0]));
READCMDWORD [m, s1, s2] <= inctrl?ctrl -> broadcastctrl!ctrl  -> MAINEXECUTE [0,0];
```

In `MAIN_IDLE`, the module receives the MODE word and broadcasts it to XMIT and RCVR modules and enters a state called `READ_SYNC_CHAR` where it checks if the current mode is synchronous or asynchronous by invoking the function `SyncMode`. If the USART has been programmed to operate in the synchronous mode it reads one or two SYNC characters depending on the MODE words(determined by the function `ReadSync2`) and branches to the control state `READ_CMD_WORD` to read the COMMAND word and broadcast it to the XMIT and RCVR modules using *multiway rendezvous* mechanism of hopCP. If the USART has been programmed to operate in the asynchronous mode it directly proceeds to `READ_CMD_WORD`. Note that, state `READ_CMD_WORD` is annotated with variables `m,s1,s2` which reflect the fact that the internal datapath of the module is updated to contain the MODE words and the SYNC characters. This describes the initialization sequence.

```
MAIN_EXECUTE [status, d] <= (indata?x -> (XmitBufferEmpty := 0, TxRDY := 0) ->
                                  xmitmain!x -> (XmitBufferEmpty := 1,TxRDY := 1)
                                        -> MAIN_EXECUTE [status,d])
                       | (inctrl?y -> (((InternalReset y) -> intReset! -> MAIN_IDLE[])
                                    |((not(InternalReset y)) ->   broadcastctrl!y ->
                                                          MAIN_EXECUTE [status,d])))
                       | (rcvrmain?data -> (ReceiverBufferEmpty := 0, RxRDY := 1) ->
                                  MAIN_EXECUTE [(update(status,1,1)),data])
                       | (readdata? -> out!d -> (ReceiverBufferEmpty := 1, RxRDY := 0) ->
                                  MAIN_EXECUTE [update(status,1,1),d])
                       | (readstatus? -> out!status -> MAIN_EXECUTE [status, d])
                       | (rcvrstatus?st -> MAIN_EXECUTE [(UpdateStatus status st),d])
                       | ((IsFalse reset) -> extReset := 0 -> MAIN_INITIATE [])
```

The fragment of hopCP code shown above, denotes the execution loop of the USART. It is expressed using the choice construct of hopCP. In the state `MAIN_EXECUTE`, the USART either receives the data from the CPU and transmits to the XMIT module, or receives a request to read status or data from the CPU wherein it sends the available data or status on `out` channel. It is also capable of receiving status updates from the RCVR module and receiving further COMMAND words from the CPU. If the internal reset command is issued by the CPU anytime, the module resets the XMIT and RCVR modules through the `intReset` channel and branches back to `MAIN_IDLE`.

4.3 Rcvr Module

The *behavior* section of the Rcvr module is described next. It captures the synchronous and asynchronous receive operations

```
RCVR_START [] <= ((IsTrue extReset) -> RCVR_INITIATE [])
                |((not(IsTrue extReset)) -> RCVR_START []);

RCVR_INITIATE [] <= (broadcastdata?m ->
        ((SyncMode m) -> broadcastdata?s1 -> RCVR_READ_SYNC2 [ m, s1])
        |((not(SyncMode m)) -> RCVR_EXECUTE [m, 0, 0] ));

RCVR_READ_SYNC2 [m,s1] <= ((ReadSync2 m) -> broadcastdata?s2 ->
                                RCVR_EXECUTE [m, s1, s2])
                        | ((not (ReadSync2 m)) -> RCVR_EXECUTE [m, s1, 0]);
```

Initially (when the power is switched on), the RCVR is in a state `RCVR_START` where it waits for the `extReset` asynchronous input to go *high* following which it goes to state `RCVR_INITIATE`. In this

state, it receives the MODE word from the MAIN module (via multiway rendezvous with XMIT and MAIN modules). If the mode is synchronous, RCVR module proceeds to receive one or two SYNC characters and proceeds to the control state RCVR_EXECUTE with the MODE and SYNC characters as the internal datapath state. If the mode is asynchronous it proceeds directly to RCVR_EXECUTE.

```
RCVR_EXECUTE [m, s1, s2] <=
        ((IsFalse extReset) -> RCVR_START [])
       |( intReset? -> RCVR_INITIATE [])
       |(broadcastctrl?ctrl ->
                ((ReceiveEnable ctrl) ->
                        ((SyncMode m)->(SYNDET:=0) -> ENTER_HUNT [m,s1,s2,ctrl,0])
                       | ((not (SyncMode m)) -> RCVR_ASYNC [m,(BitsPerChar m),0]))
                |((not(ReceiveEnable ctrl)) -> RCVR_EXECUTE [m, s1, s2]));
```

In RCVR_EXECUTE, it receives the COMMAND word (again via multiway rendezvous) and either enters a *synchronous* or an *asynchronous* receive mode. In the state RCVR_EXECUTE, RCVR module is capable of handling an internal reset (via the COMMAND word) or an external (hard) reset.

```
ENTER_HUNT [mode, syn1, syn2, ctrl, rxbuffer]  <=  RxD?din ->
  CHECK_FOR_SYNC_CHAR1 [mode, syn1, syn2, ctrl,
                        (AccumulateSerialData rxbuffer din)];

ENTER_HUNT2 [mode, syn1, syn2, ctrl, rxbuffer]  <=  RxD?din ->
               CHECK_FOR_SYNC_CHAR2 [mode, syn1, syn2, ctrl,
                          (AccumulateSerialData rxbuffer din)];

CHECK_FOR_SYNC_CHAR1 [m, s1, s2, ctrl, rxb] <=
                      ((IsFalse extReset) -> RCVR_START [])
                      |(intReset? -> RCVR_INITIATE [])
                      |((rxb=s1) ->
                                ((ReadSync2 m) -> ENTER_HUNT2 [m, s1, s2, ctrl, 0])
                                |((not (ReadSync2 m)) -> SYNDET := 1 ->
                                      RCVR_ASYNC [m, (BitsPerChar m), 0]))
                      |((not (rxb=s1)) -> ENTER_HUNT [m, s1, s2, ctrl, rxb]);

CHECK_FOR_SYNC_CHAR2 [m, s1, s2, ctrl, rxb] <=
                      ((IsFalse extReset) -> RCVRSTART [])
                      |(intReset? -> RCVRINITIATE [])
                      |((rxb=s2)-> SYNDET:=1-> RCVRASYNC[m,(BitsPerChar m), 0])
                      |((not (rxb=s1)) -> ENTER_HUNT2 [m, s1, s2, ctrl, rxb])
```

In the synchronous receive mode, the module first enters a *huntmode* where it scans the incoming data for the synchronization characters and then proceeds to the control state RCVR_ASYNC to receive the *serial* data. In the asynchronous receive mode, it directly proceeds to the control state RCVR_ASYNC.

```
RCVR_ASYNC [mo, size, data] <=
           ((not(size=0)) -> RxD?y ->
                 RCVR_ASYNC [mo, (Decrement size), (AccumulateSerialData data y)])
         |((size =0) -> RxD?pin ->
                 RCVR_PROCESS_DATA [mo, data, (CheckParityError data mo pin)]);

RCVR_PROCESS_DATA [mo, data, perror] <=
                 ((IsFalse extReset) -> RCVR_START [])
                |(intReset? -> RCVR_INITIATE [])
                |(RxD?sb ->((sb=0) -> SEND_DATA_TO_MAIN [mo, data, perror, 1])
                           |((sb=1) -> SEND_DATA_TO_MAIN [mo, data, perror, 0]));

SEND_DATA_TO_MAIN [mode, data, pe, fe] <=
                ((IsFalse extReset) -> RCVR_START [])
                |(intReset? -> RCVR_INITIATE [])
                |((IsTrue ReceiverBufferEmpty) ->
                        rcvrstatus!(MakeAsyncStatus pe fe 0 1)
                                -> rcvrmain!data -> RCVR_EXECUTE [mode,0,0])
                | ((not(IsTrue ReceiverBufferEmpty)) ->
                        rcvrstatus!(MakeAsyncStatus pe fe 1 1)
                                -> rcvrmain!data -> RCVR_EXECUTE [mode,0,0]);
```

In `RCVR_ASYNC`, RCVR module receives the specified number of bits *serially* on the `RxD` input. The number of bits is programmable by the CPU and is computed by the function *Bitsperchar*. The received serial data is checked for *framing* and *parity* errors as dictated by the COMMAND and MODE words. Then the data is sent to the MAIN module in *parallel* via a data assertion on the internal channel `rcvrmain!`. In the process it checks for *overrun* error. Note that we use the same mechanism to perform the synchronous and asynchronous receive operations. This is because in hopCP only the *sequence-domain* relationships between a set of actions is specified, no specific timing discipline (except causality) is advocated. This makes hopCP specifications smaller and more *abstract*. After transmitting the data to the MAIN module, the RCVR module assembles the status information (the state of `fe,oe,pe,RxRDY` bits). and sends it to the MAIN module using the `rcvrstatus` channel. The MAIN module can communicate the status information to the CPU.

4.4 Xmit Module

The detailed hopCP specification of the XMIT module is presented in the appendix (to conserve space). The behavior section resembles that of the RCVR module: Initially, XMIT module is in state `XMIT_START` where it waits for a reset signal (from the MAIN module) and then receives the MODE word and COMMAND (via multiway rendezvous with RCVR and MAIN modules). If the current operating mode is synchronous, XMIT module receive one or two synchronization (depending on the output of the `ReadSync2` function), receives the input character from the MAIN module (in parallel) on the `xmitmain` channel and transmits it *serially* on the output port `TxD`. If a new character is not received at the end of transmission of the current character, `TxEMPTY` pin is set high and SYNC characters are transmitted on `TxD`. In the asynchronous mode, the input character is received from the MAIN module, padded with *start* and *stop* bits and shifted out *serially* (least significant bit first) on the `TxD` output at a *rate* determined by the *baud rate* setting in the MODE word.

4.5 Comparison With Existing Work

Intel 8251 has been specified in HardwareC [25] and a variant of ISPS [42]. The specifications are available with the distribution of high-level synthesis benchmarks. In this section we will compare the hopCP specification of the 8251 with its HardwareC and ISPS specifications. We will also touch

upon the drawbacks of describing the 8251 in a language like Occam [10] which has been advocated for the specification of asynchronous circuits.

4.5.1 ISPS

ISPS is a procedural language augmented with constructs to describe synchronous hardware. The significant differences between hopCP and ISPS specifications are that the ISPS specification (i) lacks abstraction in the sense that it describes on particular implementation of the 8251, based on synchronization flip-flops (ii) does not have constructs to expressing parallel behavior explicitly. (iii) the computation is described in a imperative language.

4.5.2 HardwareC

The language hardwareC [25] comes closest to hopCP in terms of the communication constructs it uses. This is encouraging because starting from similar motivations about the real-world scenarios that we wish to model, we have independently ended up selecting the same set of communication constructs in our respective HDLs. However, hardwareC is currently used to capture synchronous computations only. In addition, hopCP is much simpler and is semantically well specified. Some of the key differences between hardwareC and hopCP are as follows:

(i) HardwareC is a synchronous hardware description language, so it does not provide the same *temporal* abstraction as hopCP. Specifications in hopCP can be implemented as purely synchronous circuits, purely asynchronous circuits or a mixture of both. In addition, in a hopCP specification we do not make any assumptions about the representation of the electrical signals i.e. we allow both *transition* based or *level-based* implementations—two popular styles of implementing asynchronous circuits [41].

(ii) HardwareC is based on an imperative language to specify computation where parallelism has to be extracted from sequential descriptions (during synthesis) while hopCP is based on a functional language the parallelism is implicit in the program (*i.e.* it is much easier to extract). In addition, the *referential transparency* of functional languages facilitates formal reasoning and proving properties about the system which are generally difficult in imperative languages. However, HardwareC has the ability to specify *resource* and *timing* constraints which are not provided in hopCP, at present.

4.5.3 CSP based Languages

CSP based languages like Occam used in [10] and Trace Theory used in [15] have the disadvantage of supporting only *synchronous* message passing. It is awkward to model asynchronous phenomena like *interrupts* and *status* and *reset* operations in such languages which makes them restrictive for hardware specification. There are operators suggested in [22] to correct this defficiency but they are yet to appear in a realistic HDL.

5 Tools for Analysis of hopCP Specifications

High-Level specifications of complex protocols are of little use if they are not adequately supported by tools to analyze them and reason about them. In the hopCP design environment we provide three different types of tools to support high-level specification:

- A suite of *static analysis* tools to perform *reachability* and *seriality* analysis on the HFGs .
- A behavioral inference tool called $parComp$ which *infers the composite* behavior of a collection of hopCP modules.

- A compiled-code behavioral simulator to establish functional correctness of the hopCP specifications.

In section 5.1, we introduce the algorithm *parComp*. In section 5.2, we will briefly introduce the seriality-checking algorithm. These algorithms have been detailed in [2]. Section 5.3 presents the compiled code functional simulator that can be used to debug hopCP descriptions. Section 5.4 presents how hopCP specifications are debugged using tester processes.

5.1 Behavioral Inference via Parallel Composition

In this section we will briefly introduce *parComp* and discuss its performance on the USART example. We specified the USART as a collection of three independent modules MAIN, XMIT and RCVR. It is useful to have the *composite* (also known as inferred behavior) of the complete USART for several reasons. Inferred behavior can be used in high-level simulation, flow analysis of the hopCP specifications, and in formal verification. In this section we will describe a tool called *parComp* to derive the composite behavior of a set of modules specified in hopCP. Modules in hopCP interact via communication actions (data assertions and data queries). *parComp infers* the behavior of a collection of hopCP modules by composing the individual transitions in the HFGs of the constituent modules. Composing transitions involves checking for synchronization and performing value communication. Transitions $t_1 = (\{s_1\}, a_1, \{s_1^{'}\})$ and $t_2 = (\{s_2\}, a_2, \{s_2^{'}\})$ are said to *synchronize* if (i) a_1 and a_2 are mutually complementary (i.e. one is a data assertion and the other is a data query) and (ii) they use the *same* communication port. For example, if $a_1 = b?x$ and $a_2 = b!e$, t_1 and t_2 will synchronize and the resultant transition is $t_3 = (\{s_1, s_2\}, a_2, \{s_1^{''}, s_2^{'}\})$ where the $s_1^{''} = s_1^{'}[E[e]/x]$ (E[e] denotes the value of the expression e evaluated in s_2). The latter illustrates value communication.

If a_1 and a_2 do not synchronize, then transitions t_1 and t_2 are retained in the inferred behavior. This is a significant difference compared to the other option of handling concurrent actions, namely, *nondeterministic interleaving* of the actions a_1 and a_2 in the inferred behavior. The interleaving of actions a_1 and a_2 results in having transitions t_1t_2 and t_2t_1 in the inferred behavior which has the capability of performing a_1 and a_2 in any order. This approach is taken in CSP and CSP based languages. Our approach to handling concurrency results in a very efficient (both in time and space) implementation of *parComp* when compared to the *interleaved* mode. On an average, the number of states in the inferred behavior is a linear function of the number of states in the input HFGs .

Note that, in the above example a_1 and a_2 are primitive actions. The notion of synchronization and value communication can be extended in a similar way to *compound* actions. The details of semantics of *parComp* are presented in [1]. *parComp* has been implemented in Standard ML of New Jersey [5] in the prototype hopCP design environment on a SUN sparcstation. It exhibits acceptable runtimes of the order of seconds on the 8251 USART example.

5.2 Seriality Checking and its Uses

Determining whether two specific actions of an HFG are serial or are potentially concurrent has numerous applications. This check can be used to warn if the asynchronous ports are not being used *safely* i.e. if there are conflicting reads/writes on the shared registers implementing the asynchronous ports. The seriality checking procedure can also be used to establish *determinacy of guards* in some situations and reveal opportunities for *resource sharing*. These optimization hints can be used in the high-level synthesis of VLSI circuits from hopCP specifications.

However, in a distributed environment with several concurrent processes, determining whether two actions are potentially concurrent or not, automatically, is often difficult to formulate and computa-

tionally expensive. There are essentially two problems.

Naive approaches to the detection of seriality can either lead to combinatorial explosion or can miss many opportunities to detect serial usage. Combinatorial explosion can result because many of the techniques to detect seriality are centered around *reachability* analysis paradigm. These problems are tackled in the hopCP framework by restricting the hopCP flow graphs to be one-safe and employing a heuristic-based pruning of the *composite* hopCP flow graphs.

The second, and a more serious problem underlying the feasibility of the above optimizations, is that unless the *context* (environment) of a module is known, it is not possible to tell if two actions within the module definition are serial or not. For this to be done properly, we need a tool to analyze the combined executions of a collection of processes *that constitutes the system description, and that, perhaps, even includes a process to model the abstracted environment.* The algorithm *parComp* outlined in the previous section is appropriate for this task.

Briefly, our seriality-checking procedure involves three phases: First, we invoke *parComp* to infer the composite behavior of the collection of hopCP modules. Then we derive an *abstract HFG* by invoking the pruning heuristic on the inferred behavior with respect to the actions in question. The pruning heuristic *removes* uninteresting states and transitions with respect to the actions in question. The third phase involves computing the set of *reachable configurations* from the initial states and determining if the two actions in questions can be enabled simultaneously or not. All the phases of the seriality-checking procedure have been formalized and implemented in the hopCP design environment. The details are presented in [2].

This procedure was particularly useful on the USART specification because of its complexity. Several errors in the unsafe usage of the asynchronous ports were revealed. In addition we also discovered that the in practice we do not need separate channels for `indata?`, `inctrl?`, `readdata?` and `readstatus?` because they are never used concurrently. So, in an actual circuit implementation one could use a single multiplexed bidirectional channel to implement these four channels. This optimization is almost impossible to detect by manually analyzing the specifications of the XMIT, MAIN and RCVR modules. Typical running times of the seriality-checking procedure are less than ten seconds for most of the subcomponents of the 8251 USART.

We also discovered that a slight modification of the seriality-checking procedure could be used to detect the *liveness* of the specification. Dead-states can be flagged during the generation of reachable configurations.

5.3 CFSIM: A compiled-code concurrent functional simulator

A high-level specification is not very useful unless it is supported by a methodology to *validate* it. There are usually two ways of validating a specification: (i) prove liveness and safety properties of the specification [26]; and (ii) high-level simulation. We follow the second approach. We provide a simulation environment called CFSIM to validate hopCP specifications. In this section we will briefly outline the design of CFSIM and bring out some of its advantages and in the next section we describe how some of the modes of behavior of the USART specification are validated using CFSIM.

CFSIM is compiled-code concurrent functional simulator for hopCP specifications obtained by translating *HFG*s into CML (Concurrent ML) source code. The details of CFSIM are described in [3]. CML is an extension to Standard ML of New Jersey to support first-class synchronous operations [35]. CML being higher-order facilitates building concurrent-programming abstractions and is implemented efficiently capitalizing on the continuation-passing style technology of the SML of New Jersey compiler [6].

The first step in the generation of the simulator in CML is to decompose the hopCP specifications into *sequential HFG*s . A *sequential HFG* is one which all transitions are of the form $(S, a, S^{'})$ where $\mid S \mid = \mid S^{'} \mid = 1$. In otherwords, a sequential HFG is one that is specified by only $[\!]$ and $\rightsquigarrow$ operators in hopCP. Each sequential HFG is modeled as an *independent thread* in CML which can *communicate* with other threads by explicit message-passing through synchronous channels or through *shared-variables*. The constituent transitions of a sequential HFG are modeled as a set of *mutually recursive* function definitions. The control state name of the *precondition* of the transition becomes the name of the function and the corresponding datapath state variables are modeled as the *formal* parameters of the function. The *postcondition* of a transition is modeled as a *function-call*. The *actions* in the transition are translated into CML code fragments such that the execution of the code *simulates* the execution of the hardware module via the action in question. Figure 7, shows the CML translation of the specification shown in figure 1

Synchronous communication actions in hopCP (data queries, data assertions, input and output control actions) are directly implemented by the `send` and `accept` primitives in CML. Assignment action is implemented by a one-place buffer abstraction with non-blocking reads and writes. It is realized by the ML structure `AsyncBarrier` and its associated operations `newPort` and `multicast`.

The '$[\!]$' operator is implemented by `choose` combinator in CML and the '$\rightsquigarrow$' operator is implemented by directly in SML. The crux of the simulator is in implementing *compound actions* and *multiway rendezvous* whose implementation is discussed in [3].

Some of the salient features of CFSIM are:

- 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 .
- Static Checks: Since we translate the HFGs 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.

5.4 Testers and High-Level Debugging of hopCP Specifications

In this section, we will describe how we validate the hopCP specification of the USART through CFSIM. Validation of specifications via CFSIM involves two phases: (i) identification of interesting *modes of behavior* of the system being modeled and (ii) constructing high-level *simulation vectors*, which *enable* the chosen mode of behavior. The simulation vectors are expressed in hopCP itself. A hopCP specification which *enables* a particular mode of behavior is called a *tester module*. Identification of interesting modes of behavior and construction of tester modules are illustrated with respect to the USART specification discussed earlier.

5.4.1 Tester Modules and Modes of Behavior

There is a close analogy between the design validation strategy suggested here and the conventional VLSI test generation. Modes of behavior are like *faults* and tester modules are like physical tester vectors (sequences of binary inputs).

```
fun m2 () =
    let
        val TxRDY = AsyncBarrier.mChannel 0
        val TxRDY__9 = AsyncBarrier.newPort TxRDY
        val c = channel ()
        val b = channel ()
        val a = channel ()
        fun Q x   = (CIO.print( "Waiting for Input on Channel a? \n");
                     let
                         val y =  input_int(sync(CIO.input_line std_in))
                     in
                         CIO.print("Received "^(Integer.makestring y)^" on channel a?\n");
                         s__7 x y
                     end)
        and
            s__7 x y   =
            let
                val _ = 0
            in
                if ((odd y)) then (s__5 x y  ; ())
                else
                    if ((even y)) then (s__3 x y  ; ())
                    else
                        (raise ChoiceError; ())
            end
        and
            s__3 x y  =
            let
                val s__31_chan = channel ()
                fun s__31_fun x y   =
                    ( (CIO.print (" Output on Channel b!"^Integer.makestring(((f x) y))^"\n");
                       send (s__31_chan,0)))
                     val s__32_chan = channel ()
                     fun s__32_fun x y   =
                         ((AsyncBarrier.multicast(TxRDY,1 ) ;  send (s__32_chan,0)))
            in
                (spawn  ( fn () => s__32_fun x y  );
                 spawn  ( fn () => s__31_fun x y  );
                 let
                     val _ = accept s__31_chan
                     val _ = accept s__32_chan
                 in
                     Q (y + 1)   end )
            end
        and
            s__5 x y   = ( (CIO.print (" Output on Channel c!"^Integer.makestring( (subvector(y,0,4)) )^"\n");
                           Q y ))
    in
        spawn (fn () => Q 5  );
        ()
    end;
```

Figure 7: Simulator for Example in Figure 2 obtained by CFSIM

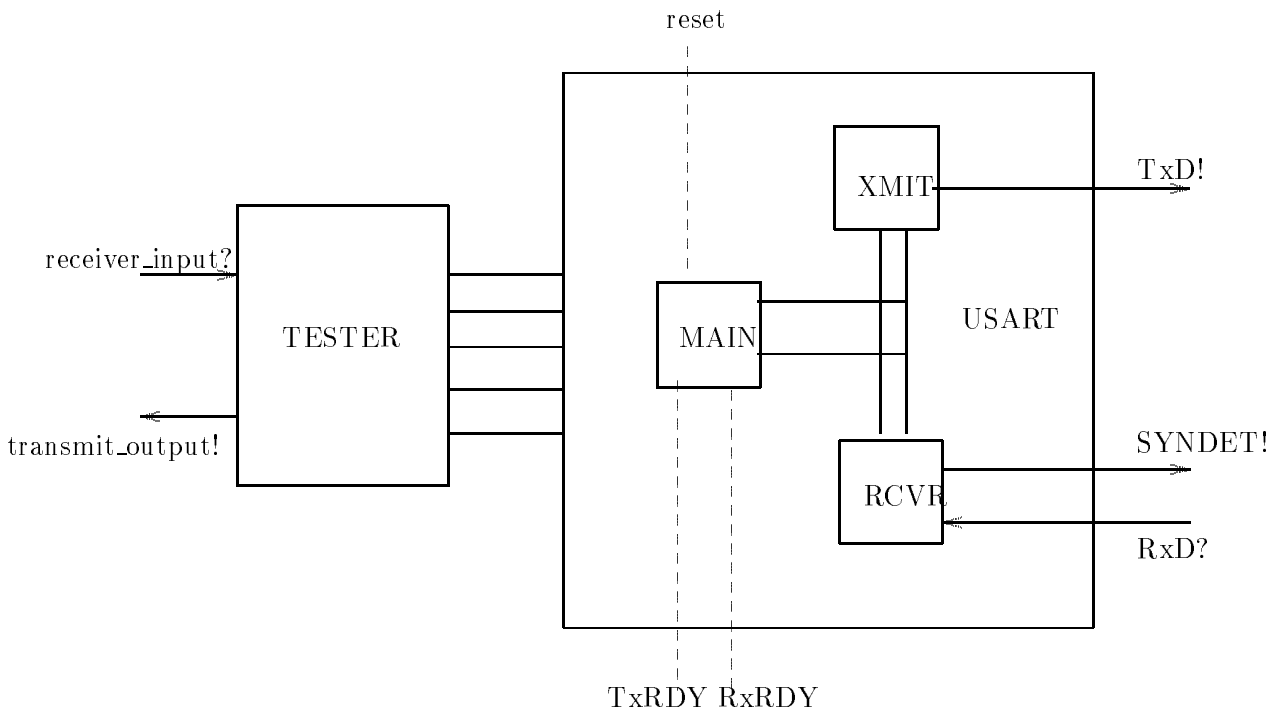

Figure 8: Illustrating Tester Modules in hopCP

A *mode of behavior* of a system can be defined as a *finite* execution trace of the system involving at least one input communication action and one output communication action. The input and output communication actions in the execution trace give us the ability to *control* the system execution and *observe* the response of the system, respectively. A execution trace consisting of only internal actions is not a useful mode of behavior. Some of the useful modes of behavior in the USART example are: receiving a character from the CPU and transmitting it on the serial output `TxD` in the synchronous mode, receiving a character on the serial input `RxD` and transmitting it in parallel to the CPU, hunting for the synchronizing characters in the synchronous mode receive etc.

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 as shown in figure 8. 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.

5.4.2 Illustrating Tester Modules

Let us consider validating the *asynchronous mode of behavior* of the USART specification. To recapitulate (from section 2), transmission in the asynchronous mode involves receiving a character from the CPU and transmitting it serially on the `TxD` pin at a specified baud rate after padding it with a start bit and some specified number of stop bits and an optional parity bit. The baud rate, number of stop bits and kind of parity to be checked is specified by the MODE word. Reception in the asynchronous mode involves receiving a serial stream of data from the `RxD` input, and checking for parity and framing errors and transmitting the character to the CPU.

```
Module TEST
Type
        byte: vector 8 of bit;
SyncPort
        input_mode?,input_command?, input_data?,indata!, inctrl! : byte;
        receiver_input?, transmit_output! : byte;
        txd?, RxD! : bit;
AsyncPort
        reset! : bit;
Function

Behavior
(TEST_RESET [] <=   (reset := 1) -> TEST_START [];
 TEST_START [] <=   input_mode?m -> indata!m -> input_command?c ->
                                    inctrl!c -> TEST_LOOP [];
 TEST_LOOP []  <= input_data?d -> indata!d   -> TEST_LOOP []
)
 ||
(RECEIVE []    <= receiver_input?din -> RxD!din   -> RECEIVE [])
 ||
(TRANSMIT []   <= txd?din -> transmit_output!din  -> TRANSMIT [])
End
```

Figure 9: Tester Module for Testing Asynchronous Mode of Operation

A tester module called TEST which validates the *asynchronous mode of behavior* of the hopCP USART specification is shown in figure 9. It communicates with the user via synchronous ports `input_mode?, input_command?, input_data?, receiver_input?,` and `transmit_output!` and communicates with the USART with the rest of the ports. TEST basically encapsulates three *concurrent* processes which represent the CPU, and a serial transmitter and a serial receiver (denoting a Modem for example). `TEST_RESET` process, *resets* the USART via an assignment action on the `reset` port, and then loads the MODE and COMMAND word. It then enters a state called `TEST_LOOP` where it continuously transmits characters to the USART. The `RECEIVE` process receives serial data from the `receiver_input?` and transmit it on the `RxD!` port while the tt TRANSMIT process receives serial data from the USART and outputs it on the `transmit_output!`.

The TEST module is compiled into a HFG along with XMIT, MAIN and RCVR modules and the resultant HFGs are compiled into CML code using CFSIM. Figure 10 illustrates the compilation and simulation of the asynchronous mode of behavior in CML environment.

`simulate_processes` on line 1 is the top-level function which compiles a list of hopCP modules into HFGs and generates two files: a file containing the user-defined functions and the simulator. The files are loaded in sequence as shown in lines 2 and 3. `TEST_MAIN_XMIT_RCVR_N` is the top-level function which simulates the USART and the TEST module and is invoked on line 4 by a CML function called `doit`. CML uses *pre-emptive* scheduling. The desired *time-slice* can be presented as a parameter to `doit`. (In our example the time-slice is 10 milliseconds). On line 5 we present the MODE word, 82, which enables the asynchronous mode operation, enables odd parity, sets the baud rate to 1x, sets number of stop bits to 1 and number of bits per character to 5. The COMMAND word is presented on line 6, which enables the transmitter and receiver. On Line 7, the tester module prompts the user for the input data which is provided on line 8. The next few lines shows the

```
- simulate_processes ["test.h", "main.h", "xmit.h", "rcvr.h"];  (* Line 1 *)
val it = () : unit

- use "TEST_MAIN_XMIT_RCVR_N_fun.sml";                          (* Line 2 *)
[opening TEST_MAIN_XMIT_RCVR_N_fun.sml]
....
....
[closing TEST_MAIN_XMIT_RCVR_N_fun.sml]

- use "TEST_MAIN_XMIT_RCVR_N_sim.sml";                          (* Line 3 *)
[opening TEST_MAIN_XMIT_RCVR_N_sim.sml]
val TEST_MAIN_XMIT_RCVR_N = fn : unit -> unit
[closing TEST_MAIN_XMIT_RCVR_N_sim.sml]

- doit(TEST_MAIN_XMIT_RCVR_N, SOME 10);                         (* Line 4 *)
Waiting for Input on Channel receiver_input?
Waiting for Input on Channel input_mode?
0
82                                                              (* Line 5 *)
Waiting for Input on Channel input_command?
5                                                               (* Line 6 *)
Waiting for Input on Channel input_data?                        (* Line 7 *)
Waiting for Input on Channel receiver_input?
15                                                              (* Line 8 *)
 Output on Channel TxRDY0
 Output on Channel TxEMPTY0
 Output on Channel TxRDY1
 Output on Channel transmit_output!0
 Output on Channel transmit_output!1
 Output on Channel transmit_output!1
 Output on Channel transmit_output!1
 Output on Channel transmit_output!1
 Output on Channel transmit_output!0
 Output on Channel transmit_output!0
 Output on Channel transmit_output!1
 Output on Channel TxEMPTY = 1                                   (* Line 9 *)
1
Waiting for Input on Channel receiver_input?
1
Waiting for Input on Channel receiver_input?
0
Waiting for Input on Channel receiver_input?
1
Waiting for Input on Channel receiver_input?
1
Waiting for Input on Channel receiver_input?
0
Waiting for Input on Channel receiver_input?
 Output on Channel RxRDY = 1                                    (* Line 10 *)
1
..
..
CML: Interrupt                                                  (* Line 11 *)
val it = () : unit
-
```

Figure 10: Illustrating the Simulation of hopCP Specifications

functionality of the transmitter. Note that the bits are shifted out according to the asynchronous transmit protocol with the least significant bit first. Successful transmission is indicated on line 9 when `TxEMPTY` status flag is set. In parallel, receive operation is going on, whose termination is detected by the status flag `RxRDY` on line 10. Since, the tester module was constructed to be an infinite process, we terminate the simulation session by an interrupt shown on line 11.

5.4.3 Discussion

hopCP specification of the USART was checked for different modes of behavior like internal and external reset operation, synchronous receive and transmit, setting of various error flags (overrun, parity, framing) and status flags `TxRDY, RxRDY, TxEMPTY, SYNDET`, external character synchronization, by constructing appropriate TESTER module. Actually something similar to conventional *fault simulation* was employed, in the sense that a single tester module was programmed with appropriate MODE words to check for more than one mode of behavior. Several errors were detected by this validation process which were subsequently fixed.

The attractive features of this style of validation process 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.

- We feel that *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 as shown in figure 8.

- By expressing the *tester module* in hopCP and simulating it via CFSIM we are actually *validating* our system in *truly concurrent* environment. This is illustrated in the example of the tester module above (which validates the asynchronous mode of behavior). Note that the tester itself has three concurrent processes. So, the validation of the USART in the CFSIM environment has the effect of connecting an 8251 to a modem and a microprocessor and switching on the power quite realistically.

It is interesting to compare the style of validation we have presented against that offered in verification tools such as SMV [30] and COSPAN [21]. SMV is a system that allows concurrent finite-state systems to be described in a language reminiscent of data-flow languages as well as guarded-command languages. The transition relation underlying finite state systems modeled in SMV are represented using binary decision diagrams (BDDs) which are often efficient representations for relations. SMV accepts queries about the finite-state system from the user in the language of computational tree logic (CTL) and answers these queries through symbolic model checking. In our experience, a system such as SMV will be valuable to interface to the hopCP system. Control aspects of HFGs can be compiled into SMV in a straightforward manner. COSPAN is again a formalism for specifying and verifying concurrent automata where validation is achieved by exhibiting homomorphisms between the specification and the implementation automata. Therefore, existing validation tools such as COSPAN and SMV can be interfaced with HFGs to derive those advantages offered by these tools that are not offered by CFSIM. However, validation tools such as SMV and COSPAN are not well suited for modeling data aspects of systems. Since a system such as the 8251 USART involves control aspects that are data independent, data aspects, as well as control aspects that are data dependent,

it becomes necessary to apply a variety of techniques and tools to validate systems such as the 8251. Hence, tools such as CFSIM can complement other finite state validation tools.

Other formal verification systems such as HOL [19] and Nqthm [9] are considerably more powerful and can, in principle, deal with systems with potentially unbounded amounts of state. However, these tools have not been widely used for modeling or verifying control dominated ICs such as the 8251, to the best of our knowledge.

6 Concluding Remarks

The major contributions of this work are: (i) Introduction of a simple HDL called hopCP to describe concurrent state-transition systems, and the illustrattion of the expressive power of hopCP by capturing the behavior of a fairly complex chip namely Intel 8251 at a high level; (ii) A behavioral inference tool called *parComp* and its applications; (iii) High-level validation of hopCP specifications via a compiled-code concurrent functional simulator (CFSIM) and a notion of *tester modules*.

In a nutshell the main contribution of this work is the demonstration of the applicability of formal methods in the specification and validation of a *realistic* and existing hardware design.

Currently we are engaged in systematically transforming hopCP specifications into VLSI circuits with as little manual intervention as possible. We are also exploring the possibilities of augmenting hopCP with mechanisms to handle timing constraints and using CFSIM and its variants to perform basic timing simulation.

References

1. 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.
2. Akella, V., and Gopalakrishnan, G. Static Analysis Techniques for the Synthesis of Efficient Asynchronous Circuits. Tech. Rep. UUCS-91-018, Department of Computer Science, University of Utah, Oct. 1991.
3. Akella, V., and Gopalakrishnan, G. CFSIM: A Compiled-Code Concurrent Functional Simulator for VLSI Systems. Tech. Rep. UUCS-TR-92-002, Department of Computer Science, University of Utah, Jan. 1992. To appear in International Journal in Computer Simulation.
4. Akella, V., and Gopalakrishnan, G. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-aided Design, ICCAD 92* (Nov. 1992), pp. 587–591.
5. Appel, A., and MacQueen, D. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture* (Sept. 1987).
6. Appel, A. W. *Compiling with Continuations.* Cambridge Univ. Press, 1992. ISBN 0-521-41695-7.
7. Apt, K. R., and Olderog, E.-R. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991. *ISBN 0-387-97532-2.*
8. Barbacci, M. R. Instruction Set Processor Specifications (ISPS): The Notation and Its Applications. *IEEE Transactions on Computers C-30*, 1 (Jan. 1981), 24–40.
9. Boyer, and Moore. *A Computational Logic.* Academic Press, 1979.
10. Brunvand, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, Nov. 1991.
11. Charlesworth, A. The Multiway Rendezvous. *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 350–366.

12. Cleveland, R., Parrow, J., and Steffen, B. The concurrency workbench: A semantics based tood for the verification of concurrent systems. Tech. Rep. ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, Univ of Edinburgh, Aug. 1989.

13. Dill, D. L. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

14. Dill, D. L., Nowick, S. M., and Sproull, R. F. Specification and automatic verification of self-timed queues. *Formal Methods in System Design 1*, 1 (July 1992), 29–62.

15. Ebergen, J. C. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56.*

16. Gopalakrishnan, G., and Fujimoto, R. Design and verification of the rollback chip using hop: A case study of formal methods applied to hardware design. Tech. Rep. UU-CS-TR-91-015, University of Utah, Department of Computer Science, 1991.

17. Gopalakrishnan, G., and Josephson, L. Towards amalgamating the synchronous and asynchronous styles. In *TAU 93: Timing Aspects of VLSI, Malente, Germany* (Sept. 1993), ACM.

18. Gopalakrishnan, G., Michell, N., Brunvand, E., and Nowick, S. M. A correctness criterion for asynchronous circuit verification and optimization. *IEEE Transactions on Computer-Aided Design* (1992). *Accepted for Publication.*

19. Gordon, M. J. Mechanizing programming logics in higher order logic. In *1988 Banff Hardware Verification Workshop, Banff, June 1988* (1988), G.Birtwistle and P.A.Subrahmanyam, Eds. *Invited Paper, to appear as a chapter in a forthcoming Springer-Verlag book.*

20. Hall, A. Seven myths of formal methods. *IEEE Software*, 9 (Sept. 1990).

21. Har'El, Z., and Kurshan, R. P. Software for analytical development of communication protocols. *AT&T Technical Journal* (Jan. 1990). *To appear.*

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

23. Intel. 8251A Programmable Communication Interface. In *Intel Microprocessor and Peripheral Handbook* (1986), pp. 6.122–6.138.

24. Johnson, S. D. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

25. Ku, D., and Micheli, G. D. HardwareC - A Language for Hardware Design, Version 2.0. Tech. Rep. CSL-TR-90-419, Computer Science Laboratory, Stanford University, April 1990.

26. Lamport, L. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM 32*, 1 (Jan. 1989), 32–45.

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

28. Page, Ian and Luk, Wayne. Compiling Occam into Field-Programmable Gate Arrays. International Workshop on Field Programmable Logic and Applications, September, 1991, Oxford University, UK.

29. Martin, A. J., Burns, S., T.K.Lee, D.Borkovic, and P.J.Hazewindus. The design of an asynchronous microprocessor. In *Proc. Decennial Caltech Conference on VLSI* (1989), C.L.Seitz, Ed., MIT Press.

30. McMillan, K. L. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

31. McFarland, M. C., Parker, A. C., and Camposano, R. The high-level synthesis of digital systems. *Proceedings of the IEEE 78*, 2 (Feb. 1990), 301–318.

32. May David *Compiling Occam into Silicon.* Developments in Concurrency and Communication Addison-Wesley, 1990.

33. Nalumasu, R., and Gopalakrishnan, G. Verifying an asynchronous wavefront crossbar arbiter using smv, 1993. Class Project Notes, CS 611, Available upon request from ratan@cs.utah.edu.

34. Occam programming manual, 1983.

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

36. Robin Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.

37. Rosenberger, F. U., Molnar, C. E., Chaney, T. J., and Ting-Pein Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers 37*, 9 (Sept. 1988), 1005–1018.

38. Sheeran, M. Design of regular hardware structures using higher order functions. In *Proceedings of the Functional Programming and Computer Architecture Conference* (Sept. 1985), Springer-Verlag, LNCS 201. Nancy, France.

39. Srivas, M., and Bickford, M. Formal verification of a pipelined microprocessor. *IEEE Software*, 9 (Sept. 1990).

40. Sternheim, E., Singh, R., and Trivedi, Y. *Digital Design with Verilog HDL.* Automata Publishing Company, Cupertino, CA, 95014, 1990. ISBN 0-9627488-0-3.

41. Sutherland, I. Micropipelines. *Communications of the ACM* (June 1989). *The 1988 ACM Turing Award Lecture.*

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

43. Thomas, D. E., and Moorby, P. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9126-8.

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

7 Appendix

```
(* Specification of the Transmitter module of the Intel 8251 Usart *)

Module XMIT

Type
        byte: vector 8 of bit;
        Bit : vector 1 of bit

SyncPort
        broadcastctrl?, broadcastdata?, xmitmain? : byte;
        txd!, intReset? : Bit

AsyncPort
        XmitBufferEmpty?, extReset? : Bit;
        TxEMPTY! : Bit

Function

fun TransmitEnable ctrl = if (subvector(ctrl,0,0)=1) then true else false endif;

fun DataSize mode =  (subvector (mode, 2,3) + 6 + subvector(mode,4,4)
                      + subvector(mode,6,7));

fun AssembleAsyncData mode data = if (subvector(mode,4,4) = 1) then
              orb(lshift(orb (0,data),1),
                 lshift(update(7,0,(parity data)),(subvector(mode,2,3) + 6)))
             else
              orb(lshift(orb (0,data),1),
                 lshift(7,(subvector(mode,2,3) + 6)))
             endif ;

fun ExtractBaudRate mode = if (subvector(mode,0,1) = 1) then 1
                           else
                                   if (subvector(mode,0,1) = 2) then 16
                                   else 64
                                   endif
                           endif;

fun Lsb data = index(data,0);

fun Decrement x = x - 1
```

```
Behavior
XMIT_START [] <= ((IsTrue extReset) -> XMIT_INITIATE [])
                |((not(IsTrue extReset)) -> XMIT_START []);

XMIT_INITIATE [] <= broadcastdata?m ->(((SyncMode m) -> broadcastdata?s1 ->
                                                  XMIT_READ_SYNC2 [m,s1])
                                  |((not(SyncMode m))-> XMIT_EXECUTE [m,0,0]));

XMIT_READSYNC2 [m,s1] <= ((ReadSync2 m) -> broadcastdata?s2 -> XMIT_EXECUTE [m,s1,s2])
                       | ((not (ReadSync2 m)) -> XMIT_EXECUTE [m,s1,0]);

XMIT_EXECUTE [m, s1, s2] <=  broadcastctrl?ctrl -> XMIT_SEND_DATA [m,s1,s2,ctrl];

XMIT_SEND_DATA [m, s1, s2, ctrl ] <= ((TransmitEnable ctrl) -> ((SyncMode m) -> XMIT_SYNC [m,s1,s2])
                                                             |((not (SyncMode m)) -> XMIT_ASYNC[m,s1,s2]))
                                  |((not (TransmitEnable ctrl)) -> XMIT_EXECUTE [m,s1,s2]);

XMIT_ASYNC [m, s1, s2 ] <=  ((IsFalse extReset) -> XMIT_START [])
                          |(intReset? -> XMIT_INITIATE [])
                          |(xmitmain?x -> TxEMPTY := 0 -> XMIT_ASYNC_SEND [m, s1, s2,(AssembleAsyncData m x),
                                                                  (DataSize m),(ExtractBaudRate m)])
                          |(broadcastctrl?ctrl -> XMIT_SEND_DATA [m,s1,s2,ctrl]);


XMIT_ASYNC_SEND [m,s1,s2,data,size,brate] <= ((not(size=0))-> txd!(Lsb data) ->
                                                XMIT_ASYNC_SEND [m,s1,s2,rshift(data,1),(Decrement size)])
                                          |((size=0) -> TxEMPTY := 1-> XMIT_ASYNC [m, s1, s2]);

XMIT_SEND_SYNC_DATA [ m, s1, s2, data, size]  <=  ((not (size=0)) -> txd!(Lsb data) ->
                                 XMIT_SEND_SYNC_DATA[m, s1, s2, rshift(data,1), (Decrement size)])
                                             |((size=0) -> XMITSYNC [m,s1,s2]);

XMIT_SYNC [m,s1,s2 ] <=  ((xmitmain?x -> TxEMPTY:=0 -> XMIT_SEND_SYNC_DATA [m,s1,s2,x,(DataSize m)]))
                        |(broadcastctrl?ctrl -> XMIT_SEND_DATA[m,s1,s2,ctrl])
                        |((IsFalse extReset) -> XMIT_START [])
                        |(intReset? -> XMIT_INITIATE [])
                        |((IsTrue XmitBufferEmpty)-> TxEMPTY:=1 -> XMIT_SYNC_CHAR [m,s1,s2,s1,8]);


XMIT_SYNC_CHAR [m, s1, s2, data, size ] <=  ((not(size=0)) -> txd!(Lsb data) ->
                                              XMIT_SYNC_CHAR [m,s1,s2,rshift(data,1),(Decrement size)])
                                        |((size=0) ->  ((ReadSync2 m) ->
                                                          XMIT_SECOND_SYNC_CHAR [m,s1,s2,s2,8])
                                                     |((not (ReadSync2 m))-> XMIT_SYNC[m,s1,s2]));

XMIT_SECOND_SYNC_CHAR [ m, s1, s2, data, size]  <=
               ((not (size=0)) -> txd!(Lsb data) -> XMIT_SECOND_SYNC_CHAR [m, s1, s2,
                                                          rshift(data,1), (Decrement size)])
              |((size=0) -> XMIT_SYNC [m,s1,s2])
End
```