Static Analysis Techniques for the Synthesis of Efficient Asynchronous Circuits [1]

VENKATESH AKELLA[2]
GANESH GOPALAKRISHNAN[3]

UUCS-91-018

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

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

October 7, 1991

Abstract

In the context of deriving asynchronous circuits from high-level descriptions, determining whether two actions are potentially concurrent (overlapped execution) or serial (non-overlapped execution) has several advantages. This knowledge can be utilized to efficiently implement shared variables, support speculative guard evaluation, and optimize resources (circuitry) by sharing. In a distributed environment with several concurrent processes, determining whether two actions are potentially concurrent or not, automatically, is often difficult to formulate and computationally expensive. In this paper, we present techniques to overcome these problems. First, we present a tool called **parComp** *which infers the composite behavior of a collection of modules, and then we present an algorithm called* **conCur** *to analyze the* inferred behavior *to detect the seriality of two actions. Simple heuristics are presented for the* abstraction *of the inferred behavioral descriptions and improving the efficiency of* **conCur**. *The algorithms* **parComp** *and* **conCur** *are illustrated in the hopCP framework and implemented in Standard ML of New Jersey. Execution times of the algorithms are reported on a variety of examples. The results are quite encouraging.*

---

[1]Keywords: Performance-Directed Synthesis, Asynchronous Circuits, Static Analysis, Parallel Composition, Petri-Nets

[2]Supported in part by a University of Utah Graduate Research Fellowship

[3]Supported in part by NSF Award MIP-8902558

1 Introduction

Considerable work has been done in performance-directed synthesis of synchronous circuits from high-level descriptions [CW91]. Flow analysis techniques have been studied in the context of synchronous circuits for area-time optimizations in resource-allocation and scheduling. Recently, there has been a growing interest in the synthesis of asynchronous circuits [Mar89, BS89, Chu87, Ebe89]. Many of the popular asynchronous compilation approaches are based on a *communicating sequential process paradigm.* Flow Analysis based optimization is conspicuously absent in these approaches. In this paper, we identify one simple criterion, namely,

> detection of whether two *actions* a and b are serial or concurrent in the execution of a hardware module M which consists of sub-modules $M_1 \ldots M_n$.

Actions a and b could occur in the *sorts* of any submodule $M_1 \ldots M_n$. Actions could denote communication actions as in CSP like $p?x$, $q!exp$, and computational aspects like $x + y$, $(effective\_addr\ op1\ op2)$ or low-level register-transfer operations like ld_reg_Y, ALU_1_add etc. We investigate the significance of this criterion in the performance optimization of asynchronous circuits. Then we present algorithms to *statically* detect this criterion in the context of a communicating sequential process paradigm used in the specification of asynchronous circuits.

We illustrate our ideas within the hopCP design environment which is being currently developed [AG91b]. hopCP is a simple language for the specification, simulation and synthesis of synchronous and asynchronous circuits. We restrict ourselves to asynchronous circuits in this paper. hopCP can be best viewed as a first-order functional language augmented with features to support synchronous and asynchronous styles of value communication explicitly. A hopCP behavioral specification describes a concurrent state-transition system called HFG or hopCP Flow Graph. The actions in the HFG denote communication (e.g. $p?x, q!e$) and computation (e.g. $(x+y)$) aspects of behavior. The control aspects are captured by *sequencing* (analogous to ";" of CSP) and *choice* (analogous to the *alternate* command in CSP) operators. In addition we have a *parallel* operator which *derives* a composite HFG for a hardware module M from the HFGs for its submodules $M_1 \ldots M_n$.

A distinctive feature of hopCP is the support for asynchronous communication through a restricted form of shared variables called *asynchronous ports.* In general, asynchronous

communication is considered hazardous in a distributed environment due to its propensity to cause deadlocks and metastable behavior. The static analysis scheme proposed in this paper can issue *a priori* warnings for the *unsafe* usage of asynchronous ports and also help implement them *efficiently* if the accesses to the asynchronous ports are indeed *serial*.

Determining whether two actions are serial or concurrent leads to the following optimizations in area and time of the resultant circuits: (i) mapping logical channels to physical channels, (ii) concurrent and speculative guard evaluation in the *alternate* command, (iii) detecting sharing to optimize resources, and (iv) efficient implementation of *shared* variables used to support asynchronous communication.

In general, it is difficult to determine *statically* whether two actions are serial or concurrent because all possible interactions of the submodules have to be taken into account. We achieve this through *behavioral inference*, and present a tool called *parComp* which *infers* the *composite* behavior of a collection of hopCP modules. A formal definition of *parComp* and strategies to contain its complexity are discussed in this paper.

Reachability analysis of the *inferred behavior* with respect to the pair of actions in question reveals whether they are serial or concurrent. Naive reachability analysis can lead to two undesirable scenarios: (i) Combinatorial explosion in the size of the reachability graph, and (ii) In general one may have to deal with potentially *infinite* states in the reachability tree if the input flow graphs are not *bounded* (i.e. k-safe for some fixed k).

In this paper we present techniques to address both the above problems. First, we prove that the hopCP flow graphs are 1-safe. This results in a reachability graph with *finite* states, (actually *finitely represented symbolic states*). Then we present an abstraction mechanism to *eliminate* unnecessary states in the *inferred behavior*. This results in *contraction* (abstraction through the replacement of subgraphs containing more than one transition with a single transition) of the reachability graph. Thereafter, algorithm *conCur* performs *efficient* reachability analysis of the contracted inferred behavior and detects if two actions are serial or concurrent.

Organization of the Paper

In the next section we present scenarios in the asynchronous circuit compilation where the proposed strategy can be used to optimize area or speed of the resultant circuits. Then, we briefly introduce the language hopCP and the HFG notation. This is followed by a formal

description of *parComp* and *conCur* algorithms. The implementation of the algorithms and their execution times on a variety of examples is presented next. Finally, we conclude with a summary of the significant contributions of the proposed work and directions for future work.

2 Application Scenarios

We identify the following optimizations scenarios that can result in efficient asynchronous circuits, and present a unified suite of static analysis algorithms for addressing these optimizations.

Mapping From Logical to Physical Channels

In a CSP based program, a pair of channels $(p!, p?)$ are used for transmitting data between processes based on the rendezvous paradigm. A channel is typically implemented by a set of control wires to carry the *signaling* information and a set of data wires to carry the data values encodings. In this framework, two pairs of channels $(p!, p?)$ and $(q!, q?)$ are semantically unrelated: a communication on $(p!, p?)$ does not affect one on $(q!, q?)$, and vice versa. Most existing approaches to hardware compilation compile two distinct channel implementations in hardware to support $(p!, p?)$ and $(q!, q?)$. However this may not be always necessary. Suppose, in a certain context, it is guaranteed that communications through the p and q channels are serial; then, hardware resources to support these channels can be shared. The question now is how to say if p and q are used serially.

In hopCP, we address this problem by assuming that all the channel names in the initial hopCP specification denote *logical* channels. A logical channel may or may not have a direct manifestation in the final circuit. The channels in the final circuit are called *physical* channels. More than one logical channel can be mapped into a single physical channel if it is guaranteed that the corresponding actions are *serial*. This optimization has two significant consequences: firstly, it makes the specifications more *abstract* and secondly, it facilitates *sharing* of buses as in standard synchronous circuit synthesis (albeit, it needs extra multiplexors and control logic).

Concurrent, and Speculative Guard Evaluation

Consider an instance of the *alternate* command in the description of a module M in a CSP-like language,

```
( ...  p?x -> Q [] q?y -> R )
```

In the above expression it not apparent whether the context (environment) of M would generate $p!$ and $q!$ concurrently or not. Existing asynchronous compilation methodologies (e.g. [BS89]) typically synthesize a circuit that checks for the *arrival* of the guard communications in a *round-robin* fashion and then pick the first one that succeeds. (For fairness, one could remember the last guard which succeeded and start the round-robin search from there.) Another technique typically employed involves arbiters. However, arbiters are an overkill if the guards are indeed *serial*; if they are not serial, of course an arbiter or a round-robin based mechanism is essential. Sequential guard evaluation such as above has a potential disadvantage. It could incur a penalty that is linear in the number of arms of the alternate command. An alternate approach is to try all the guards *concurrently* and proceed with the guard which succeeds. But, concurrent guard evaluation has a caveat; one should be able to determine *statically* that the actions in question $p!$ and $q!$ are serial and one should have a mechanism in hardware to expunge the partial evaluation of the unsuccessful guards. One could adopt the strategy suggested in [Ebe89] where a CAL component is used to concurrently wait for the communication actions. (A CAL component is also known as a decision-wait element in literature and is discussed by Molnar et. al in [CEM85]) Concurrent guard evaluation could avoid the cost of round-robin checking.

Consider a more complex alternate command which has a mixture of boolean expressions and communication actions for guards.

```
( E1,p?x -> P [] E2,q?y -> Q )
```

Let the evaluation of expression E1 require a more complex computation (takes more time) than that for E2. Further, assume that in one scenario $q?$ is going to synchronize. In that scenario, the evaluation of E1 followed by the checking of the arrival of $p?$ (which is not going to succeed anyway in this scenario) is wasted work. A better approach would have been to concurrently start the evaluation of E1 and E2, and when the respective evaluations finish, to wait for the respective communication actions, and then when they occur (exactly one is guaranteed to occur by the seriality check) to *fire* the CAL component at its appropriate input.

This would allow P or Q (which matters in the given scenario) to be started the earliest.

In the example scenario presented above, the evaluation of E1 was wasted work because only $q!$ was guaranteed to occur. This did not slow-down the evaluation of E2 because it is assumed that dedicated hardware exists for the computation of E1 and E2. In functional programming literature, this kind of evaluation is referred to as *speculative* evaluation. Speculative evaluation is not a novel idea in hardware design either. For example, the familiar carry-select adder is based on it. What we propose in this paper is a mechanism to support speculative evaluation in a asynchronous compilation framework via static analysis of the behavioral descriptions.

Sharing Resources

Sharing resources by detecting seriality constraints is a well-known idea in high-level synthesis of synchronous circuits. We propose to integrate it in a asynchronous compilation methodology by appealing to the same static analysis tools as for the other optimizations discussed in this section. Briefly, sharing of resources by detecting seriality constraints can be explained as follows: Consider two invocations of the "+" operation in a flowgraph representation used in high-level synthesis. With respect to this flowgraph these two invocation can be supported by one physical adder if it can be guaranteed that these invocations are *serial* in all possible circumstances (for all input data values, system states and system environments).

Efficient Implementation of Shared Variables

In hopCP we support a restricted form of asynchronous communication by using shared port variables called *asynchronous ports*. In general, CSP-like languages do not allow shared variables. Occam allows shared variables but does so in a limited way: a variable can be written into only in serial threads; then, when the serial thread splits into concurrent threads, the variable may only be read in the concurrent threads. The usage of shared variables allowed by hopCP is more general, with the proviso that exactly one hopCP module owns an asynchronous port A (i.e. it can write into it), and, all other modules use A in a read-only manner. Asynchronous communication via shared variables enhances the expressive power of hopCP. It enables us to model common hardware scenarios like *busy-waiting* (which involves *polling* for a certain condition) and *status signals* (which are written once and could be read several

times by different processes) very elegantly.

Reliable communication through asynchronous ports can be guaranteed only in one of two ways:

- guarantee that the *writes* occur before any *reads*
- Use an arbiter-based circuit such as an ATS module described in [Kel74]. If the inputs to the ATS module are signaled simultaneously, the module acts as if one input, then the other, occured. In other words, it *serializes* the inputs in hardware, at run-time.

Of the two alternatives, the serialization alternative is cheaper as it does not involve the use of an arbiter. Hence, the check for seriality is once again central to the efficient implementation of asynchronous ports in hopCP.

Section Summary

The common basis for all the four optimizations suggested in this section is the ability to detect *statically* from the behavioral description of the module, whether two given actions are *serial* or *concurrent*. 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. This is 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 details are presented later in this paper. 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.* We have developed and implemented such a tool called *parComp*. Briefly, *parComp* deduces the composite flow graph of a parallel composition of several hopCP modules. It is analogous in effect to the expansion rule in CCS which statically composes a set of CCS agents. Our algorithm differs from the expansion rule in that it handles compound actions, value communication, multiway rendezvous and asynchronous communication, which are salient features of hopCP. Details of *parComp* are

presented in section 4. In the rest of the paper we present the definition, implementation, and performance results of the static analysis tools in the hopCP framework.

3 Introducing hopCP

hopCP is a notation to describe a concurrent state-transition system called hopCP Flow Graphs, augmented with features to model computation in a purely functional style, and mechanisms to support synchronous and asynchronous communication. In hopCP, hardware is modeled by a structural entity called a *module* which contains two parts: (i) a *behavioral* entity which captures the state-transition system describing the hardware in question, and (ii) a set of *communication* ports with which the hardware *interacts* with its environment. A hopCP specification has six sections: The MODULE section introduces the name of the module being described, the TYPES section introduces the data types of the communication ports used, and the SYNCPORT section declares all the *synchronous* communication ports used in the specification. A *synchronous* port allows *rendezvous* style communication as in CSP. The ASYNCPORT section declares all the *asynchronous* communication ports used. An *asynchronous* port allows value communication communication between two modules without explicit synchronization (note that synchronization with the resources implementing the communication may still be necessary). The FUNCTION section contains the *user-defined* functions used in the specification. The functions are written in a *first-order* functional language. The syntax of Standard ML of New Jersey is used. The BEHAVIOR section describes the state-transition system which captures the behavior of the hardware system being specified. The state-transition system being described is called HFG and is discussed in detail in the following section.

Informal Semantics of HFG

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

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

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

```
MODULE ex1

TYPES
        byte : vector 8 of bit

SYNCPORT
        a?,b! : byte

FUNCTION
(* index(a,0) extract the bit at position 0 in the word a,
   update(b,2,0) sticks in a 0 at bit position 2 in the word b
*)
        fun f a b =  if (index(a,0)=1) then update(b,2,0) else b;

BEHAVIOR

        P [x] <= a?y ->  b!(f x y) -> P [y])
END
```

Figure 1: hopCP Specification of a Simple Pipeline Stage

where $Guard, CompoundAction$ are syntactic objects, and $\mathcal{P}$ *and* $\oplus$ denote *power set* and *disjoint sum operations on sets*, respectively. *Guards* in hopCP include input communication actions and Boolean expressions. *istate* denotes the set of *initial* states of the specification and *trel* denotes the set of *transitions* in the HFG . A *transition* $tr \in Transition$ is a triple $(pre(tr), act(tr), post(tr))$ where $pre(tr)$ denotes a set of states called the *precondition* of the transtion, $post(tr)$ denotes a set of states called the *postcondition* of the transtion, 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 illustrate the features of hopCP using the example of a pipeline stage. Figure 1 shows a

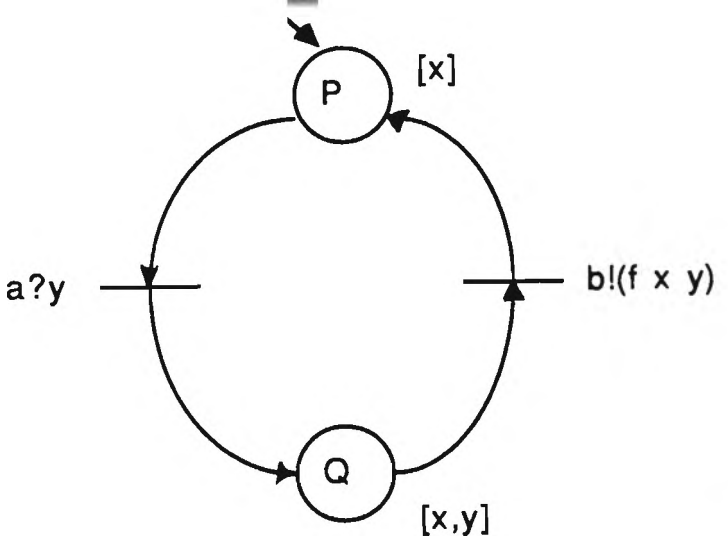

Figure 2: hopCP Flow Graph (HFG) of hopCP Specification in Figure 1

complete hopCP specification of the pipeline stage. It does not have a ASYNCPORT section. It declares an input synchronous channel a and and an output synchronous channel b of type *byte*. Figure 2 denotes the HFG corresponding to the hopCP specification shown in Figure 1 and is textually described as follows:

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

It is more convenient to draw pictures to denote HFGs where circles denote the control state names ($ProcName$) and "bars" denote the actions. The HFG is interpreted (read) as follows: Module $ex1$ is *initially* in a state $(P,[x])$ where P is the control state (known as $ProcName$) which is analogous to *program counter* in a conventional computer architecture terminology while x is the *datapath state* (also known as $LocalStore$) which is a *snapshot* of its relevant *internal* state. In the state $(P,[x])$ the system can engage in a communication action $a?y$ which will be henceforth referred to as a *data query* and go to a state denoted by $(Q,[x,y])$. Data query $a?y$ denotes a *synchronous* communication action: read from *synchronous* input channel a. Note that by performing the action $a?y$ the internal state of the system is modified to include the value received on channel a which is reflected by the presence of the variable y in the state $(Q,[x,y])$. We could have a synchronous communication action without value communication, i.e. merely $a?$ which is referred to as *input control action*. We have just described the execution of the system via the transition $((P,[x]), a?y, (Q,[x,y]))$. As a consequence of this execution we find that the transtion $((Q,[x,y]), b!(f\ x\ y), (P,[y]))$ is *enabled*. In the state $(Q,[x,y])$, the module can perform the communication action $b!(f\ x\ y)$ and proceed to the state denoted by $(P,[y])$. $b!expr$, where $expr \in EXPR$,(domain of expressions allowed by

hopCP syntax) is said to be a *data assertion* and represents the *synchronous* communication action of *outputting* the value denoted by the expression *expr* on the channel b. A data assertion without value communication, for example just $b!$, is called an *output control action*. In the example, *expr* is the application of user-defined function f on arguments x (original internal state) and y (received as a consequence of the action $a?y$). The function f could involve arbitrary computation and is expressed in a purely first-order functional language. hopCP has a wide repertoire of bit-level manipulation routines commonly used in hardware systems like *lshift*, *rshift*, *exor*, *subvector*, *index-vector*, *update-vector*, *parity* etc. $(P, [y])$ denotes the fact that the system goes back to the same control state P (as the initial state) but the datapath state is now y instead of x. In a programming language sense, this could be viewed as invoking a function P with an *actual* parameter y for the *formal* parameter x.

Salient Features of hopCP

The other significant features of hopCP which could not be illustrated in the above example are:

Compound Actions: A compound action *ca* in hopCP is a set of primitive actions $a_1, a_2, \ldots, a_m$ with the restriction that all $a_i, a_j \in ca$ should be *non-interfering i.e.* no two a_i and a_j should use the same channel or try to update the same datapath variable. The underlying semantics of a compound action is that of a fork-join construct.

Multiway Rendezvous: Multiway rendezvous is said to occur when more than two processes (modules) wait for each other (synchronize) and communicate. Multiway rendezvous is a powerful notion which facilitates the specification of a wide variety of concurrent algorithms very naturally [Cha87]. It subsumes *broadcast* style of communication (point to multipoint communication) which is very natural in hardware.

Asynchronous Communication: This is facilitated by special channels called *asynchronous ports*. The details of its scope and implementation were discussed in the previous section.

Functional Sublanguage: A significant feature in hopCP is the facility to specify computational aspects of hardware behavior in a *functional* language. This allows us to get *maximally* parallel implementations for datapaths, and also facilitates formal reasoning about hopCP specifications.

4 Parallel Composition

The "||" operator specifies concurrent behavior in hopCP. It defines the interaction of independently specified hopCP specifications. In this section we will define interaction of hopCP specifications by describing a tool called *parComp*. hopCP modules interact via communication actions. The interaction could be synchronous (via handshake or rendezvous) or asynchronous (via global store). Synchronous interaction is possible when modules are willing to perform complementary actions (query/assertion) on the same channel. When the number of modules willing to perform a query is equal to one for a given assertion we have a *two-way rendezvous* (similar to CCS for example), when the number of modules willing to perform a query is more than one for a given assertion we have a *multiway rendezvous*. Multiway rendezvous is a powerful notion to express several hardware-oriented algorithms very naturally. In the next section we will provide the formal definition of *parComp* and illustrate it with an example.

Formal Definition of parComp

First we define an auxillary function *conjugate* which checks if two primitive action can synchronize or not. Two primitive actions synchronize when they are complementary and both use the same channel. For example, $conjugate(a?x, a!(p+1))$ and $conjugate(a?, a!)$ yields *true* while $conjugate(a?x, b!(p+1))$ or $conjugate(a?x, a?y)$ or $conjugate(a!x, a!(p+1))$ yields $false$. A formal definition of *conjugate* is omitted to conserve space. In hopCP, parallel composition is complicated by the presence of compound actions. This is handled by the following definitions which determine whether a pair of compound actions a and b are synchronous or asynchronous.

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

and

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

parComp is a function which composes two concurrent state-transition systems ($HFGs$) It uses auxillary functions $Valuecomm$ to perform value communication and $RetainAsOutput$ to facilitate multiway rendezvous. The auxillary functions are defined as follows where $(isDquery\ x)$ and $(isDassert\ x)$ are predicates which check if the action x is a data query or a data assertion respectively.

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

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

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

let

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

in

if $((a_i \in a) \wedge (isDquery\ a_i) \wedge (\exists b_i \in b.conjugate(a_i, b_i)) \wedge (FD_2, \sigma_2, \sigma_{g_2}, e_i) \Longrightarrow_e v_i)$

then $ValueComm\ (s_1, s_2, a \setminus a_i, b \setminus b_i, (P_1', \sigma_1'[v_i/x_i]), s_2')$

else if $((a_j \in a) \wedge (isDassert\ a_j) \wedge (\exists b_j \in b.conjugate(a_j, b_j)) \wedge (FD_1, \sigma_1, \sigma_{g_1}, e_j) \Longrightarrow_e v_j)$

then $ValueComm\ (s_1, s_2, a \setminus a_j, b \setminus b_j, s_1', (P_2', \sigma_2'[v_j/x_j]))$

else (s_1', s_2')

end if

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

$$hfg_1 = \{istate = is_1, trel = tr_1\}$$

$$hfg_2 = \{istate = is_2, trel = tr_2\}$$

$$hfg_3 = \{istate = is_1 \cup is_2, trel = tr_3\}$$

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

(i) : One rule for building tr_1'

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

(ii) : Another rule for building tr_2'

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

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

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

This rule is applied when the compound actions a and b synchronize completely. $ValueComm$ performs the value communication across the HFGs while $RetainAsOutput$ retains the output counterparts of the synchronized actions to facilitate multiway rendezvous.

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

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

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

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

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

Then,

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

where

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

This rule handles the remaining case which is not handled by (iii) and (iv) i.e. when compound actions a and b synchronize partially. The definition is based on the interpretation of compound actions as *sets* of primitive actions and pattern-matching on the structure of the compound actions. We partition a and b to extract the components (which are themselves compound actions) which synchronize and which do not synchronize and recursively invoke rules (i) and (ii). The partitioning is done by appealing to the *refinement* of a compound action into its corresponding HFG (describing its fork-join structure). The states $(s^i_{a_1}, s^c_{a_1}, s^i_{a_1}, s^c_{a_1})$ and actions (a^i, a^c) introduced in the $prefine$ definition have significance in the synthesis of circuits from hopCP specifications [AG91a].

The following example illustrates *parComp*. Consider, the specification of two-stage pipeline obtained by composing two copies of one-stage pipeline illustrated in figures 1, 2. The composite HFG of the two-stage pipeline is obtained by applying *parComp* on the individual HFGs describing the single stages which have two transitions each. The resultant HFG is shown in figure 4. Transitions involving $a?y1$ and $c!(g\ x2\ y2)$ are retained as they are because they do not interact (synchronize) with any other actions. This is an application of rule 1 in the definition of *parComp*. The actions $b?y2$ and $b!(f\ x1\ y1)$ *do* interact, so they are merged by into a single transition involving synchronization and value communication. This illustrates the application of rule 2 in the definition of *parComp*.

```
MODULE ex3

SYNCPORTS a?,b!,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 3: hopCP Specification Illustrating Parallel Behavior

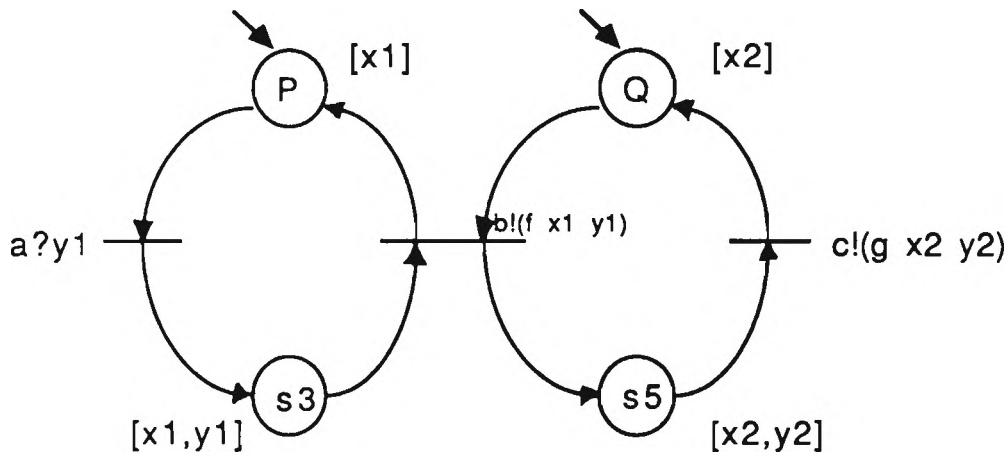

Figure 4: Inferred Behavior of 2-Stage Pipeline Unit

5 Detection of Seriality

Given two actions a and b and the composite HFG h obtained through $parComp$ (which denote the *inferred behavior*) algorithm $conCur$ determines if a and b are serial or concurrent. $conCur$ proceeds in two phases. In the first phase, all the *reachable configurations* of h are deduced by the procedure RC and in the second phase, we check if there is any configuration in the set of reachable configurations of h such that actions a and b are simultaneously enabled. This is accomplished by procedure $Concurrent$. Before we present the formal definition of RC and $Concurrent$ based on HFGs , we digress a little bit and show that HFGs generated by hopCP syntax are *one-safe*.

Relationship with Petri Nets and One-Safety

There is a close similarity between Petri-nets and the hopCP Flow graphs. The basic hopCP flowgraphs defined without using the || operator correspond to *finite-state* machines in the Petri-Net terminology [Pet81]. The only difference is the presence of *compound actions* which introduce local fork-join structures and support restricted form of non-interfering parallelism. With the || operator, one could specify HFGs which are more general than finite-state machines because of the ability to represent synchronization, value communication, and concurrency.

By one-safeness we imply the standard definition from Petri-net theory applied to HFGs interpreted as Petri-nets. This involves interpreting the control states as places and the actions as transitions. To show that HFGs are one-safe, one can appeal to the Petri-net analogy hinted above. The basic HFGs (without || operator) are one-safe because they are *finite-state* machines. The local fork-join structures introduced by compound actions still preserves one-safeness. $parComp$ basically retains unsynchronized transitions, or *merges* transitions which synchronize, both of which preserve one-safeness. Hence, the HFGs corresponding to the inferred behavior which will be presented to the $conCur$ tool are guaranteed to be one-safe.

Reachable Configurations

A *configuration* is defined as a set of control states of a HFG . To keep the analysis simple, we do not consider data-dependent behavior. We assume that all the data related guards could

evaluate to true rendering our analysis slightly pessimistic. Let $h \in HFG$, $h.istate$ and $h.trel$ denote the set of initial states and the set of transitions in h, and RC_h denote the set of all reachable configurations from $h.istate$. RC_h is defined inductively as follows:

$$(i) \quad \frac{true}{h.istate \in RC_h} \qquad (ii) \quad \frac{(s_1, a, s_1') \in h.trel, (\exists y \in RC_h . s_1 \subseteq y)}{s_1' \cup (y \setminus s_1) \in RC_h}$$

The first rule is the basis case, while the second rule computes the set of reachable configurations recursively by checking for all possible *enabled* transitions in a given configuration and incorporating them in the set of reachable configurations. One-safeness of the HFGs corresponding to the inferred behavior ensures that the set of reachable configurations is *finite*.

Using, the definition for RC_h, we define a predicate *Concurrent* which takes two actions a and b and a HFG h and checks if a and b are concurrent in h or not.

$$\begin{aligned} Concurrent\ a\ b\ h \quad \triangleq \quad & (\{(s_1, a, s_1'), (s_2, b, s_2')\} \subseteq h.trel) \\ & \wedge\ (s_1 \cap s_2 = \emptyset) \\ & \wedge\ (\exists y \in RC_h . (s_1 \cup s_2) \subseteq y) \end{aligned}$$

The predicate *Concurrent* first checks if the two actions a and b belong to the sort of the module in question and that the actions are not mutually exclusive, then it scans the set of reachable configurations in h to see if there is a configuration in which a and b can be *simultaneously* enabled.

The major bottleneck in the strategy suggested thus far, is the explicit generation of the set of reachable configurations. For a realistic circuit with over hundred states in the inferred behavior the computation of RC_h could be expensive in time and space. To circumvent this problem we suggest a heuristic next

Heuristics for Pruning the Inferred Behavior HFG

Motivation

Consider the HFG shown in figure 5a. It denotes the inferred behavior of hopCP modules M_1 and M_2 whose behavior is given by `A [] <= (a? -> b? -> A [] | (c? -> (d!,e?) -> A [])` and `B [] <= f! -> (g? -> B []) | (d? -> h! -> B [])` Let us assume we are interested in finding out whether `e?` and `f!` are serial or not. Note that

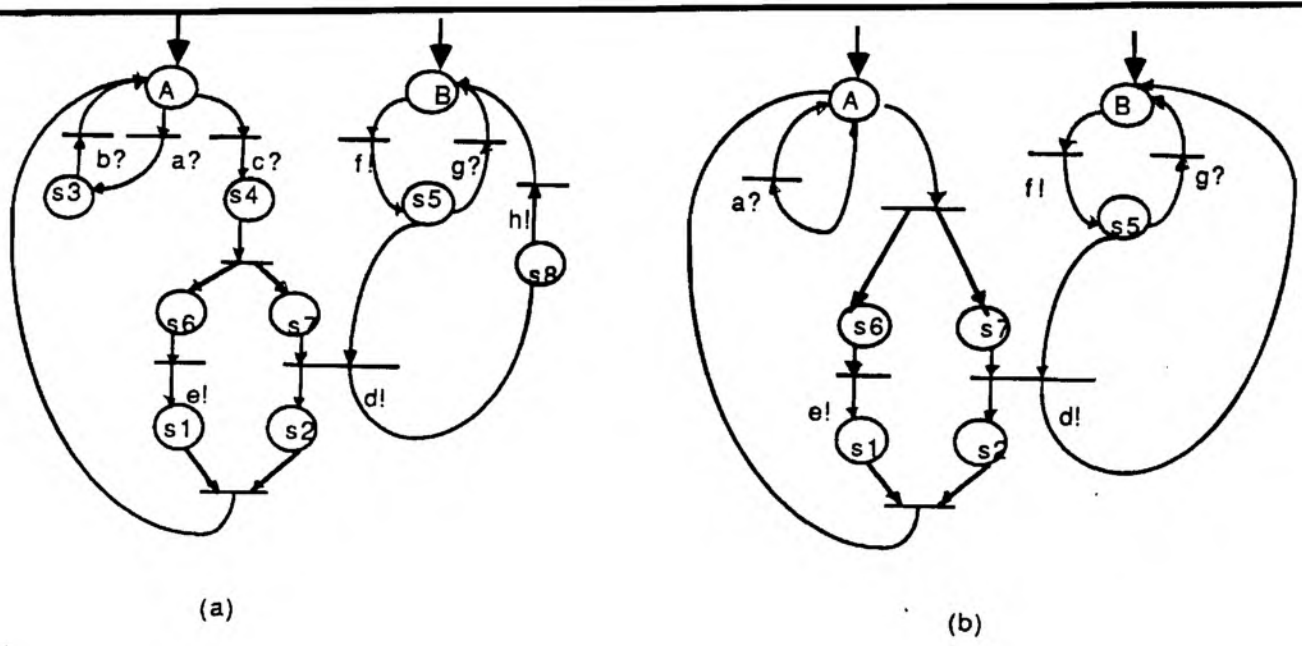

Figure 5: Illustrating Pruning of the hopCP Flow Graphs

the inferred behavior has 10 states (denoted by circles). To determine if `e?` and `f!` are serial or not, it is not necessary to consider the possible execution of the system through *sequential* states such as s_3. The only *interesting* states that can influence the causality of actions are those that involve *synchronization* or *choice.* The *pruning* heuristic is based on the above observation. It basically involves, eliminating (or abstracting) states that are not *relevant* with respect to the actions in question. For example, employing our heuristic (which will be described shortly) one can get an *contracted HFG* shown in figure 5b. Note that we could eliminate 3 states. (Note: we have so far analysed and implemented only a subset of the possible heuristics to optimize the inferred behaviors)

Algorithm for Pruning

The transformation implemented by the pruning heuristic on a inferred behavior $h \in HFG$ with respect to actions $a, b \in sort(h)$, can be formally defined by the relation $\longrightarrow_t$ as shown below

(i)

$$\frac{true}{h.trel \longrightarrow_t h.trel}$$

$$\frac{h.t_1 \longrightarrow_t h.t_2, h.t_2 \longrightarrow_t h.t_3}{h.t_1 \longrightarrow_t h.t_3}$$

This rule says that $\longrightarrow_t$ is reflexive and transitive

(ii)

$$\frac{(s_1, a_1, s_1') \in h.trel, (s, b_1, s_2) \in h.trel, (|\ s\ | = 1), (s \subseteq s_1'), (b_1 \neq a \neq b), (h.istate \cap s = \emptyset)}{h.trel \longrightarrow_t (h.trel \setminus \{(s_1, a_1, s_1') \in h.trel, (s, b_1, s_2)\}) \cup \{(s_1, a_1, (s_1' \setminus s \cup s_2))\}}$$

This rule captures the elimination of s in $HFG\ h$. We first check if the transitions (s_1, a_1, s_1') and (s, b_1, s_2) are *sequential*, and then make sure that s does not *enable* either action a or b. If both the conditions are satisfied we replace the sequential transition pair by a single transition after suitably modifying its postcondition.

Algorithm *conCur*

In this section we will summarize the overall scheme to detect if two actions a and b are serial or not in the executions of a hopCP module M which contains submodules $M_1, M_2, \ldots M_n$. Algorithm *conCur* has four major steps:

STEP 1 Derive the composite $HFG\ h$ by applying *parComp* to the collection of HFGs $h_1, h_2 \ldots h_n$, i.e. $h = h_1 \parallel h_2 \parallel \ldots h_n$

STEP 2 Apply the pruning heurisitic to h with respect to actions a and b to derive h_p, i.e.$h \longrightarrow_t^* h_p$ (h_p is the normal form for the transformation process via $\longrightarrow_t$)

STEP 3 Apply algorithm RC_h to generate the set of reachable configurations in h_p

STEP 4 Invoke the procedure ($Concurrent\ a\ b\ h_p$)

6 Implementation, Results and Discussion

All the algorithms discussed in this paper have been implemented and tested on a wide suite of examples, in the hopCP design environment. The implementation is in Standard ML of New Jersey (version 0.66). The hopCP descriptions are parsed using SML-Lex/Yacc and converted into HFGs following the operational semantics discussed in [AG91b]. A compiled-code concurrent functional simulator called $CFSIM$, which facilitates functional simulation of hopCP specifications exists. Once the specifications are simulated satisfactorily, they are ready to be compiled into asynchronous circuits using a technique called *action-refinement*. Details of action-refinement are reported in [AG91a]. The techniques discussed in this paper are designed to make *action-refinement* more efficient.

Figures 6, 7, 8 show the performance of the algorithms developed in this paper on a few examples. *2-Stage Pipeline Unit* is the specification discussed in section 4, *Mutex* is the specification of a simple mutual-exclusion protocol in hopCP, *UsartMain, UsartRcvr, UsartXmit*

	Circuit	*# States*	*# Transitions*	*Time (secs)*
1	2-Stage Pipeline Unit	4	3	0.05
2	Mutex	9	6	0.19
3	UsartMain	33	34	
4	UsartXmit	43	52	
5	UsartRcvr	41	47	
6	UsartMain || UsartXmit || UsartRcvr	136	136	79.38
7	CpuMain || ExtRcvr || ExtXmit	15	15	0.33
8	UsartMain || UsartXmit || UsartRcvr || CpuMain || ExtRcvr || ExtXmit	166	154	141.42

Figure 6: Performance of Algorithm *parComp*

	Circuit	*Transitions Before Pruning*	*Transitions After Pruning*	*Time for Pruning (secs)*
1	2-Stage Pipeline Unit	3	3	0.0001
2	Mutex	6	6	0.01
3	UsartMain	34	17	0.13
4	UsartXmit	52	38	0.23
5	UsartRcvr	47	38	0.19
6	UsartMain || UsartXmit || UsartRcvr	136	107	4.19
7	CpuMain || ExtRcvr || ExtXmit	15	4	0.02
8	UsartMain || UsartXmit || UsartRcvr || CpuMain || ExtRcvr || ExtXmit	154	133	7.29

Figure 7: Illustration of *Maximal* Pruning

	Circuit	Typical Running Time (secs) with optimization for a pair of *randomly* chosen actions
1	2-Stage Pipeline Unit	0.02
2	Mutex	0.05
3	UsartMain	0.18
4	UsartXmit	0.43
5	UsartRcvr	0.40
6	UsartMain || UsartXmit || UsartRcvr	12.09
7	CpuMain || ExtRcvr || ExtXmit	0.04

Figure 8: Typical Performance of the Seriality Detection Procedure

are the hopCP specifications of the three main components of the Intel 8251 USART, a commercial VLSI chip. Details are presented in [AG91c]. *CpuMain* is the CPU interface to the Usart, *ExtRcvr* and *ExtXmit* are the external serial devices communicating with the Usart. Figure 6 shows the performance of algorithm *parComp*. Note that the timings for *UsartXmit*, *UsartRcvr* and *UsartMain* are omitted because they do not involve the "||" operator. Also, note the *absence* combinatorial explosion of states/transitions in the composite behaviors 6 and 8. This is a consequence of the *non-interleaving* semantics of || operator in hopCP. Figure 7 shows the effect of the *pruning* heuristic. The results reflect *maximal* pruning. Maximal Pruning gives us an upper-bound on the number of transitions that can be eliminated by our heuristic. Note that the pruning algorithm has no effect on circuits 1 and 2; this is because there are no nodes which satisfy the conditions outlined for pruning. There is a considerable saving in the number of transitions for the other circuits. Figure 8 shows the typical performance of the overall algorithm for a pair of randomly chosen actions. Note that even for the USART example (which is fairly large), the time consumed is only in tens of seconds. Timing measurements were conducted on a implementation in SML (Version 0.66) running on a Sparc-IPC.

7 Conclusion and Future Work

We observe that *statically* determining whether two high-level actions are potentially concurrent or not, could lead to a variety of area-time optimizations in the high-level synthesis of

asynchronous circuits. We presented a formal technique to conduct this analysis within the hopCP framework. We also discussed the implementation of the algorithms and their performance on a few realistic examples. The techniques developed in this paper are fairly general and could be applied to the circuits generated by other asynchronous compilation approaches such as [BS89]. These techniques were employed successfully to discover many useful facts about the USART specification. These include the ability to map several logical channels to a single physical channel, concurrent accesses (unsafe) to asynchronous ports, and determinacy of guards. This information was exploited to improve the specification and will be used in the action-refinement based synthesis scheme. Currently we are engaged in incorporating these algorithms into the *action-refinement* compilation strategy centered around hopCP.

References

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

[AG91b] Venkatesh Akella and Ganesh Gopalakrisnan. hopCP : A Concurrent Hardware Description Language. Technical report, Department of Computer Science, University of Utah, 1991. Under Revision; Latest Version available upon request from the authors.

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

[BS89] Erik Brunvand and Robert F. Sproull. Translating Concurrent Communicating Programs into Delay-Insensitive Circuits. In *International Conference on Computer-aided Design, ICCAD 89*, April 1989.

[CEM85] F.U. Rosenberger C. E. Molnar, T.P. Fang. Synthesis of Delay-Insensitive Modules. In *1985 Chapel Hill Conference on Very Large Scale Integration*, 1985.

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

[Chu87] Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

[CW91] Raul Camposano and Wayne Wolf. *High-Level VLSI Synthesis.* Kluwer Academic Publishers, 1991. ISBN-0=7923-9159-4.

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

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

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

[Pet81] James L. Peterson. *Petri Net Theory and The Modeling Of Systems.* Prentice-Hall, 1981.