Synthesis of Delay-Insensitive Circuits by Refinement into Atomic Threads

H.F. Li  S.C. Leung  P.N. Lam

Department of Computer Science, Concordia University
1455 de Maisonneuve Blvd. W., Montreal, Quebec, Canada H3G 1M8

Abstract

In [7], an optimization strategy based on time-sharing was proposed. A process is decomposed into threads and the technique of time-sharing is applied in the synthesis of each thread and the synchronization requirements among threads. This paper is an extension of the methodology to nondeterminate processes. The synchronization requirements among threads include the passing of choice information as well as the causal relations between events from different threads. Substantial reduction in circuitry for realizing the synchronization requirements is observed if the choice states of threads are atomic (necessarily reached) and all the occurrences of an action belong to a thread. A theory on extracting threads that have the above properties and on synthesizing the synchronization requirements among threads is outlined.

1 Introduction

A delay-insensitive (DI) circuit is a form of asynchronous sequential circuit in which its functional correctness is unaffected by the delays in its components and its interconnecting wires. It is the most robust form of self-timed circuits [13]. Other forms allow more relaxed delay-insensitivity requirements, such as the use of equipotential regions, local clocks, or relative wire delay assumption, e.g. isochronic forks in speed-independent circuits. Research into this area has been motivated by the problems encountered in large scale synchronous design, such as the problem of distributing a global clock signal correctly as a result of clock skew. Work on asynchronous circuit design/synthesis has appeared in [1-10,12,14].

2 The Pomset Behavior Machine Model

Pomsets (partially ordered multisets) [11] are used to model the computational behaviors of hardware processes. A pomset is a set of partially ordered events labeled with process actions. An action of a hardware process models a signal transition (from Gnd to Vdd or from Vdd to Gnd) at an input/output port. An event is an instance of an action. The partial order on the events represents the necessary causalities between the events.

Fig. 1 is the behavior of a C-element expressed in a pomset. Output events are underlined, while input events are not. A state of the circuit can be represented by a cut of the pomset. At the initial state of the C-element, the cut is to the left of the whole pomset, and the environment (the user of the C-element) is allowed, but not required, to send in signal transitions at input ports a and b. Sending in the signal transitions will advance the cut beyond the first pair of a and b. At that state, the circuit is allowed and required to produce a signal transition at output port c. Producing a signal transition at c will advance the cut beyond the first c. Thus, an execution sequence of events can be visualized as successive advancements of the cut.

The specification of a reactive hardware process consists of a set of infinite complete behaviors: no behavior in the set is a part of another behavior in the set. Processes that have only one (more than one) computational behavior are said to be determinate (nondeterminate).

A computational behavior of a DI process can be characterized as follows: (1) there is no autoconcurrency, i.e. no two events of the same action label are concurrent, (2) there is no enabling between events of the same type, i.e. input (output) events do not enable input (output) events, and (3) there is no disabling between events of different types, i.e. input (output) events do not disable output (input) events. Additional rules and explanations for the model can be found in [11]. [17] gave a formal characterization of delay-insensitivity in trace theory.

Finite representation of a process is by a behavior machine [11]. It has several named states, one of which is the initial state, and transitions (called rules) between named states, which are pomsets. A state of the process is named if it is a choice state (see the next section) or one of the recurrent states of a determinate behavior. Fig. 2 is an example specification which has a named state 0, which is also the initial state of the process. There are four possible transitions from named state 0 back to itself.
Fig. 3 is another example. It is a circuit element used in the synthesis procedure. Other elements include toggle, XOR, inverter, C-element, and fork which are similar to those in [15]. On receiving input events, a toggle produces events at its output successively, starting from the output marked by a dot. An XOR produces an output transition whenever it receives a transition at any one of its inputs. The specification of the demultiplexer consists of 6 rules, \( r_1 \) to \( r_6 \), and \( \{(0)\} \) is the initial state. The behavior can be reasoned as follows. A transition at a results in a transition at \( x \) or \( y \). Selection of \( x \) \( (y) \) is done by sending in \( s \) \( t \) and waiting for the acknowledge at \( s \) \( t \). Initially, the output marked with a dot is selected.

Infinite behaviors of a process are generated by concatenating pomsets in the rules, starting from a rule following the initial state. The pomsets in two rules can be concatenated if the named state following one rule is the same as the named state preceding the other rule. Each named state consists of a set of slots for proper matching in concatenation, e.g. \( \{(0),(1)\} \) for named state \( 0 \) in fig. 2. Slots are introduced as a structural construct for expressing causalities between events across a named state. An event \( e \) in rule \( r_1 \) precedes an event \( f \) in rule \( r_2 \) after the concatenation of \( r_1 \) and \( r_2 \) only if \( e \) precedes a slot \( L \) in \( r_1 \) and \( L \) precedes \( f \) in \( r_2 \).

3 Characterization of Choice

A free choice of a process is the selection of one from several sets of actions. A choice state of a process is a state at which the process cannot proceed further without invoking free choices. A process is said to be well-behaved if the outcomes of concurrently enabled choices are independent of one another.

For nondeterminate processes, each free choice is enabled by a slot. A slot that enables a free choice is called a choice slot. The events right after a choice state are called choice events. For the specification in fig. 2, at named state \( 0 \) (a choice state), slot \( 0 \) enables a choice which allows the environment to give \( a \) or \( c \), while slot \( 1 \) enables the environment to select \( b \) or \( d \). Well-behavedness requires that every one of the four combinations, \( \{a, b\}, \{a, d\}, \{b, c\} \) and \( \{c, d\} \), is allowed when the concurrent free choices are enabled.

4 Correctness of Implementation

Given below is an informal description of the correctness definition in [11].

A circuit is said to implement a specification if the correctness properties hold for the closed system formed by the mirror of the specification and the circuit. The mirror of the specification is obtained by converting all input (output) events in the original specification into output (input) events. Intuitively, the mirror is the most liberal environment in which the circuit will operate. A closed system is formed if every input (output) of the mirror is connected to an output (input) of the circuit of the same action label, and vice versa. The correctness properties to be checked are: in every run of the closed system, (1) there is no input safety violation in any component, i.e. the receiver component is ready to receive, and (2) the mirror progresses maximally, i.e. projecting every infinite run onto the action set of the mirror is a complete behavior.

5 Motivation of the Work

In [6], a syntax-directed translation (SDT) method was proposed for the behavior machine. The specifications produced are of large areas. A close examination of the SDT circuits reveals that because of improper decomposition, substantial amount of circuitry is for passing choice information among sub-circuits and for resetting the circuit to be ready to receive input actions that may be given by the environment in the next named state.

Much less choice information passing and fewer circuit elements for resetting will be necessary if the decomposed threads have atomic (necessarily reached) choice states and all occurrences of an input action are localized to a thread. Fig. 2 shows such a decomposition of \( P \). The specifications of \( U_1 \) and \( U_2 \) are in fig. 4(a) and 4(b), and the SDT circuits are in fig. 4(c) and 4(d).

The operation of the SDT circuit for \( U_1 \) as follows. Each rule is implemented by a circuit block. Only the one for rule 1 is shown because others are just wires. When \( c \) is received at the initial state, it is used to switch the demultiplexers for \( a \) and \( c \) to the lower channel, denoting state 1. Prior to generating \( y \) when \( a \) or \( c \) is received at state 1, the demultiplexer is reset to the upper channel.

Should \( P \) be translated directly by the SDT, a much larger circuit, fig. 4(f), will be obtained. According to the choice slots of named state \( 0 \), the SDT decomposes \( P \) into \( T_1, T_2, T_3 \), fig. 4(e), which are respectively preceded by slot \( 0 \) only, slot \( 1 \) only, and both slots. Large number of circuit elements is used for passing choice information (i.e. the information on the outcomes of the free choices enabled by slots \( 0 \) and \( 1 \)) between sub-circuits for \( T_1, T_2 \) and \( T_3 \) (which are pairs of wires and are not explicitly shown), and for resetting the circuit.

Some observations: (1) The behavior of \( U_1 (U_2) \) after synchronization with \( U_2 (U_1) \) does not depend on the outcome of the free choice at \( U_2 (U_1) \), and (2) The input action received by \( U_1, a \) or \( c \), after synchronization with \( U_2 \) is sufficient to indicate whether \( b \) or \( d \) was the outcome of the free choice at \( U_2 \). Consequently, there is no need to pass choice information between \( U_1 \) and \( U_2 \).
The amount of circuitry for resetting the circuit depends on how many possible ways the process may determinately re-enter named state 0 from a different state. For the SDT circuit, there are four possible ways that control is passed back from T3 to T1 and T2 at named state 0, so the SDT circuit has some 4-way demultiplexers for resetting. For thread U1, there are two ways that U1 can re-enter named state 0 from named state 1. Hence, only a 2-way demultiplexer is needed. This is similar for U2.

Because there are less unnecessary choice information passing among threads that have atomic choice states and localized input events, the number of possible ways that a thread will enter a named state from a different state tends to be reduced. Thus, a reduction in the amount of circuitry for resetting is possible.

6 Decomposing Process into Threads

This section sketches the strategy for extracting threads that have atomic choice states and the localization property. Then presented are the important issues on using graph refinement to transform a specification (blackbox) into a system (whitebox) by introducing internal events and causalities, while preserving the externally observable behaviors. The use of graph refinement is in spirit similar to marked graph synthesis / Petri net refinement [16]. The purpose is to transform the specification such that each of the threads can be synthesized independently.

Some definitions: A thread is a behavior machine in which every choice state is atomic. A synchronization thread of a process is a behavior machine which has internal events only. A thread T of process P is localized if no input action of T has occurrences in another thread of P.

6.1 Thread Extraction

The example in section 5 demonstrates the appeal of localization. Due to the absence of autocorcurrency in DI systems, such localization is always possible:

Theorem 1. (Localization Theorem) It is always possible to extract localized threads of a process. The thread extraction problem is to find a set of localized threads that have atomic choice states after refinement of interthread causalities. The difficulty lies in the fact that it is not easy to tell where the choice states of the threads will be before thread extraction is performed.

The strategy is to find out a set of input events, called pseudo choice events, from each rule. The significance is that when they are received, the threads involved may need to obtain choice information before they can produce the appropriate output events. Hence, the states at which the threads receiving these events may become choice states of the threads. By identifying these potential choice states of threads (which have still to be extracted), a set of constraints is established which restricts how input actions can be clustered into threads.

Fig. 5 illustrates the strategy. Pseudo choice events are squared. Suppose a is the only input action in a thread T. Consider rule 1 and rule 2, when T receives the pseudo choice event of a, it should produce x or y, depending on what input action, c or d, is received from the environment when the other free choice is enabled by slot (1). Because T obtains choice information when that pseudo choice event of a is received, no other event of T is allowed to be concurrent with that event of a so that after refinement, the choice state of T is atomic. Thus, actions d, w and y cannot be clustered with a into a thread.

These events are called pseudo choice events because the receipts of those events by the threads do not necessarily lead to choice states of the threads after refinement. Consider again the above example. If a and e are clustered to a thread, then only the state at which the thread receives the pseudo choice event of e will become a choice state of the thread, but not when the pseudo choice event of a is received. This is because if the thread knows what it should produce ((x) or (y)) when it receives the pseudo choice event of e, it should also know what to be produced ((y) or (x)), respectively when it receives the pseudo choice event of a. So, the thread only needs to get choice information from other threads when e is received.

The thread extraction problem is formulated as: find the minimum number of clusters of input actions (each identifies a thread) such that, in every rule, for every cluster of input actions, there exists a unique greatest lower bound among the pseudo choice events of the input actions in the cluster (i.e. among all those pseudo choice events, there is one and only one of them from which the other pseudo choice events in the rule can be reached).

The uniqueness of the greatest lower bounds in different rules guarantees the atomicity of the choice states of threads after refinement. The minimum number of threads
is found because it minimizes the amount of circuitry involved in passing choice information.

Events of output actions needed not be localized. Different occurrences of an output action can be assigned to different threads, provided that, in each thread, there is no event concurrent with the greatest lower bounds.

More details can be found in [8] which includes a systematic way for identifying pseudo choice events.

6.2 Refinement of Interthread Causalities

The extracted threads cannot be implemented independently yet. First, process controlled causalities (solid arrows) from one thread to another are to be enforced. Second, circuit elements are required to store choice information from threads so that whenever a thread needs such information, it can retrieve it from the circuit elements. The problem of refinement of interthread causalities is to take care of the above so that the extracted threads can be synthesized independently.

Some issues of interest: (1) When choice information passing is necessary? (2) When choice information passing, i.e., at which synchronization points between the threads concerned, should be done? (3) How is choice information stored/retrieved in order to preserve the externally observable behaviors of the process? (4) How are process-controlled causalities enforced? [8] details answers to these questions.

Fig. 6(a) show a step-by-step refinement of rule 1 in fig. 5. Arrows labeled \( \rightarrow \) are synchronization points chosen to pass choice information between \( T_1 \) and \( T_2 \) via \( D_1 \) and \( D_2 \). It can easily be checked that in executing rules 1 and 3, \( T_1 \) and \( T_2 \) do not communicate directly and the externally observable behaviors of the process remain unchanged, i.e., no safety/progress violations.

Comparing the refined rules, one may ask: (1) When two internal events from different rules, e.g., those labeled \( i_k \), \( i_0 \), etc., should be given the same action labels? What are the constraints on assigning action labels to internal events? Why? (2) Why are some internal events and causalities appearing in different rules are clustered into a synchronization thread, e.g., \( D_1 \) and \( D_2 \)? Again, when these clustering should be done, and what are the constraints to be satisfied? [8] offers more complete details.

Since a process implements itself, the correctness properties hold for the closed system formed by the process and its mirror. As each of the refinement steps preserves the correctness properties, at the end of refinement, the network of threads implements the original process. The result is summarized in the following theorem.

**Theorem 2. (Decomposition Theorem)** A process can be decomposed into threads which can be synthesized independently.

6.3 Synthesis of Threads

After refinement, the threads can be synthesized independently, e.g., by SDT. More area-efficient circuits will be obtained if the technique of time-sharing [7] is applied.

7 Time-Sharing

The technique of time-sharing is applied in the synthesis method. Different joins/fork time-share one circuit element instead of one circuit element for each join/fork. Details of applying the technique can be found in [7].

8 Conclusion

Proper decomposition of a process into threads is demonstrated to be essential to the overall implementation cost. Atomicity of choice states and localization of events of input actions to threads are important for reducing the passing of choice information among threads. A theory is sketched to achieve such decompositions. Formal results include a decomposition theorem and a localization theorem. Future work covers other uses of threads, thread extraction algorithms, and further optimization of threads.

**References**


![Fig. 1 The complete computational behavior of a C-element](image)

![Fig. 2 A Specification P](image)

![Fig. 3 A 2-way demultiplexer](image)
(a) Specification of $U_1$

(b) Specification of $U_2$

(c) SDT circuit for $U_1$

(d) SDT circuit for $U_2$

(e) A decomposition of $P$

(f) SDT circuit for $P$