Hierarchical Optimization of Asynchronous Circuits*

Bill Lin Gjalt de Jong Tilman Kolks

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract — Many asynchronous designs are naturally specified and implemented hierarchically as an interconnection of separate asynchronous modules that operate concurrently and communicate with each other. This paper is concerned with the problem of synthesizing such hierarchically defined systems. When the individual components are synthesized and implemented separately, it is desirable to take into account the degrees of freedom that arise from the interactions with the other components and from the specification. Specifically, we consider how one can find the set of implementations that can be "correctly substituted" for a component in the system while preserving the behavior of the total system. The notion of correct substitution is formally defined for a hierarchical network of possibly non-deterministic modules and a new solution framework based on trace theory is presented to compute and represent this complete set of correct substitutions. We show that the complete set can be captured by a single trace structure using the notion of a "maximal trace structure". We indicate how asynchronous synthesis methods may be applied to explore the solution space e.g. to generate a delay-insensitive implementation.

I Introduction

The role of asynchronous design is gaining importance due to increasing difficulties with problems like clock skew and power dissipation. In many design scenarios, the design is most naturally described and implemented hierarchically as a network of separate asynchronous modules that operate concurrently and communicate with each other. Hierarchical representations are especially important when the design is compiled from a high-level language. Also, implementation constraints may force their physical implementations to be separated. Indeed, real-life designs show that many asynchronous applications are inherently distributive and highly interactive, thus requiring methods to handle them. When optimizing each individual component separately, it is desirable to take into account the degrees of freedom that arise from the interactions with the other components and from the specification. This is analogous to the problem of optimizing hierarchically combinational and synchronous circuits, which has been studied extensively.

For the design of asynchronous control circuits, the existing literature describes a number of automated algorithmic approaches based on Petri net and state machine models [2, 4, 10, 11, 15, 20, 21, 22, 23]. These works have focussed mainly on the synthesis and optimization of individual asynchronous modules without considering the optimization freedom available in optimizing concurrent behavior. Methods based on models of communicating processes have also been extensively studied in the literature [1, 3, 13, 18]. These methods are mainly based on syntax-directed translation techniques. Though these methods can effectively model concurrent behavior, they also have not considered the degrees of freedom available for hierarchical optimization. Recently, an approach based on a communicating Petri net model was proposed for hierarchically optimizing concurrent asynchronous modules [5]. The approach works at the Petri net level. For each module in the hierarchical description, the approach aims at computing a "reduced" Petri-net model for it by considering the restrictions imposed by the interactions with other modules. However, not all degrees of freedom were captured.

In this paper, we present a new approach to the hierarchical optimization problem based on trace theory. Our work is mainly focussed at the signal-level, comparable to the level being explored by current Signal transition graph and State graph based methods [2, 4, 10, 11, 20, 21, 22]. Trace theory has been developed by Rem, van de Snepscheut, and Udding [16, 19], and by Dill [6] for the analysis and verification of speed-independent circuits. In contrast to these works, our work here is focussed on *automated synthesis* and *hierarchical optimization*. We use a version of prefix-closed trace structures as described by Dill [6], which are closely related to the Muller state graph model [14] widely used for synthesis, to model both the global specification and the individual components. The global specification represents the overall desired behavior, which may be mechanically compiled from different high-level formalisms [4, 13, 18, 20]. The individual components that represent the partial implementation may either be described by a high-level specification, if it has still to be synthesized, or by a circuit-level description, if it has already been implemented. In both cases, a trace structure representation can be derived for each individual component.

In Figure 1, the hierarchical optimization problem is depicted. In this figure, T_1 is the component being optimized, T_2 is the partial implementation, which may be composed from several smaller components, and T is the overall desired specification that we wish to implement. When optimizing T_1, it is possible to exploit the degrees of freedom that arise from the interactions with the other components, which in this case are T_2 and the environment. The behavior of the environment is captured in T. For example, certain "sequences of events" are never generated by T_2 or the environment, representing "don't care" conditions that can be exploited in optimization. Similarly, there are possibly alternative sequences of events that can be generated by T_1 that do not change the overall correctness of the composed system. In general, the problem is to find all the behaviors that can be "correctly substituted" for T_1. This general problem subsumes notions of "don't care sequences" as there are other degrees of freedom that can be exploited in optimization. We say an implementation is a *correct substitution* at T_1 if the result of the entire system after composing with T_2 meets the specification T. When the partial implementation T_2 *has not been fully synthesized*, meaning that some of the components are still at the specification level, T_1 must be optimized in such way that it leads to a correct implementation of T *independent of the eventual implementation of* T_2 as long as this eventual implementation meets the partial specification of T_2. This general situation is realistic in practice and is properly handled in our framework.

To formally address the correct substitution problem, an implementation preorder is needed that states when a circuit may

*This research was sponsored in part by the European Commission under the ESPRIT (6143) project "EXACT" and HCM contract ERBCHBGCT920056.

32nd ACM/IEEE Design Automation Conference ®

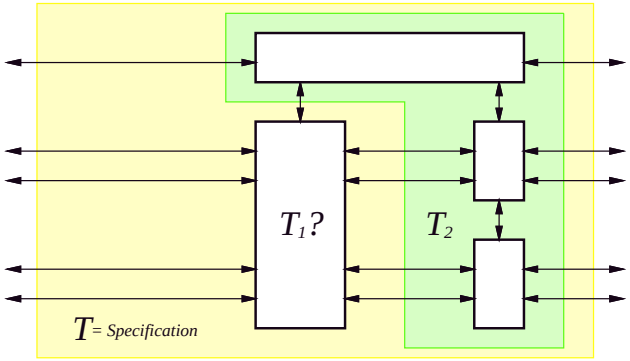

Figure 1: Specification T of a hierarchical design; T_1 is the component to be optimized; T_2 represents the partial implementation.

be "safely" substituted by another one, irrespective of its environment. In e.g. [6, 7], such necessary conditions are given. The insufficiency of these conditions, as they may lead to "incorrect circuits", has also been observed in [8], where constraints are defined to remedy this. However, these constraints restrict the design space too severely by stating that *all* specified behavior of the specification must be made. This is not necessary in the case of non-deterministic output choices. In this paper we extend the implementation pre-order of conformance to remedy its deficiency for hierarchical synthesis and optimization of incompletely specified and non-determistic specifications. In [12], a similar approach is used for synchronous systems. Non-determinism is usually the key to succinct (hierarchical) specifications as it allows to describe a set of possible behaviors in a single specification. It provides synthesis methods with all the possible freedom for optimization.

We show that *all possible correct substitutions* for a hierarchically defined component can be captured by a *single* trace structure using a notion of a *maximal trace structure*, and we present a general solution framework for computing this maximal trace structure. The computed trace structure is maximal in the sense that all trace structures that correspond to "correct substitutions" can be derived from it via some combination of "concurrency reduction" and "state assignment". From a synthesis point of view, this maximal trace structure can be viewed as a state graph where state graph based concurrency reduction, state assignment, and gate-level hazard-free synthesis methods can be applied to explore the solution space [2, 4, 10, 11, 15, 20, 21, 22]. Note that an initial behavior of T_1 may not be given. This situation arises in the "rectification" problem where we are trying to modify an "existing" design (the partial implementation) by attaching additional circuitry to it (the component under consideration).

The outline of the paper is as follows. Section II provides notation and terminology used in this paper. Section III develops notions of *progress* and *simulation* that extend the basic notion of conformance used in verification in order to characterize the correct substitution problem and the issue of implementability. Section IV presents the framework for characterizing and computing the set of all correct substitutions. Section V addresses the problem of synthesis and optimization from the maximal set of correct substitutions. We have implemented our solution framework and have applied it to a number of examples. Section VI summarizes these experiments.

II Notations and Terminology

In this section we will introduce the mathematical notions used in this paper. First we will present state graphs as the specification formalism for an asynchronous circuit to be synthesized.

A State Graphs

State graphs are a commonly used entry point for the automatic synthesis of asynchronous circuits [21, 22]. State graphs can be derived procedurally from higher-level specifications like bounded Petri-nets (i.e. Signal Transition Graphs [4]) and can thereafter be manipulated to yield, for example, a hazard-free circuit.

Formally, a state graph is a finite state machine represented by a five-tuple $G = (I, O, S, \delta, s_o)$. I is the set of *input* wires, and O is the set of *output* wires, ($I \cap O = \emptyset$). S is a finite set of states and $s_0 \in S$ is the *start state*. Let the alphabet A be defined by $A = I \cup O$. Then $T = A \times \{+, -\}$ is the set of *signal transitions*, where each transition is represented as $a+$ or $a-$ for a rising or a falling transition on a wire $a \in A$ respectively. Then $\delta : S \times T \mapsto 2^S$ is a mapping representing the *transition relation* such that if $\delta(s, t) = s'$ is defined, t is said to be *enabled* in state s and the *firing* of t takes the system from state s to state s'. This is also denoted as $s \xrightarrow{t} s'$.

Each state in the state graph is labeled with a *binary vector* $\langle s(1), s(2), \ldots, s(n) \rangle$ according to the wires $A = \{a_1, a_2, \ldots, a_n\}$ of the system. The labeling is given by a *state assignment function* $\lambda_A : S \times A \mapsto \{0, 1\}$ where for a given state $s \in S$, $s(i)$ denotes the i-th component of s corresponding to the value of wire $a_i \in A$. The state assignment function is defined as follows: (1) if $s(i) = 0 \wedge t = a_i+$ then $s'(i) = 1$; (2) else if $s(i) = 1 \wedge t = a_i-$ then $s'(i) = 0$; (3) otherwise, $s'(i) = s(i)$.

A state in the state graph captures the state of all wires in a circuit, while a state transition is a transition of exactly one wire. There may be many wires enabled in a state, but exactly one signal transition is fired at a time implying it to be an *interleaved concurrency* model.

B Trace Structures

In the previous section, we have defined state graphs as a specification model of asynchronous circuits. In [6, 16, 19], trace structures are defined to formally reason about asynchronous circuits. Trace structures are an extension of state graphs by having an additional notion of failure such that a refinement relation can be introduced for correctness implementability. For this paper to be self-contained, we will in this section review trace theory as can for instance be found in [6].

A trace is a sequence of input and output transitions defined over the input signals I and output signals O of a circuit. A *trace structure* then defines the set of success, failure and impossible traces of an asynchronous circuit. Formally and most generally, a *trace structure* T is a quadruple (I, O, S, F), where I is a finite set of *input signals* and O is a finite set of *output signals* ($I \cap O = \emptyset$). Let $A = I \cup O$. Then $S \subseteq A^\star$ is the set of *success traces* and $F \subseteq A^\star$ is the set of *failure traces*. The set P of *possible traces* is defined to be $P = S \cup F$ and the set $X = A^\star - P$ forms the *impossible traces*.

For asynchronous circuits, we only need to consider *canonical prefix-closed trace structures* in which S is prefix-closed, $S \cap F = \emptyset$ and $F = ((S \cup \{\epsilon\})I - S)A^\star$. It follows that for canonical trace structures the sets F and X are suffix closed and are completely determined by the success language S of the trace structure. Semantically, for a canonical trace structure representing a circuit, F captures the input events under which the behavior of the circuit is *not defined*. X represents traces with outputs that are *not generated* by the circuit. Although we are only interested canonical trace structures, some of the intermediate steps below may yield non-canonical trace structures.

To define operations on trace structures, we first introduce two auxiliary functions. Let $\phi(D)(x) : A^\star \mapsto (A - D)^\star$ be a function that maps a string x, defined over alphabet A, to a string x'

defined over $A - D$ such that any occurrence of a symbol in x that is in D is replaced by the empty string ϵ. $\phi(D)(x)$ on a string x is extended in the natural way to a function $\phi(D)(X)$ over a set of strings X. Let $\phi^{-1}(D)(x') = \{x \in A^\star \mid \phi(D)(x) = x'\}$ be the *inverse homomorphic image* function of ϕ.

The **hide** operator is used to define a trace structure in only a partial alphabet. Formally, let D be a set of signals to hide. Then $T' = \mathbf{hide}(D)(T)$ is the trace structure $T' = (I - D, O - D, \phi(D)(S), \phi(D)(F))$. Normally, hiding is only applied to a set of *output* signals. The *projection* onto a set of signals is the dual of the hide operator: $T' = \mathbf{project}(D)(T) = \mathbf{hide}((I \cup O) - D)(T)$. As a result of applying **hide** the trace structure may be non-canonical since the resulting success and failure language may overlap. When a system is supposed to fail whenever it may *possibly* fail, the resulting trace structure can be made canonical by removing the overlapping failure traces from S, as in [6].

The essential operation on circuits is the construction of a network of circuits by interconnecting them through common wires. This composition operator determines the *common behavior* of the two trace structures. To allow this, the languages have to be defined over a common alphabet. Let $T = (I, O, S, F)$ and $T' = (I', O', S', F')$, with $O \cap O' = \emptyset$, be two trace structures defined over the same alphabet. Formally, the *composition* of T and T', denoted as $T \parallel T'$, is defined as the trace structure $T'' = ((I \cup I') - (O \cup O'), O \cup O', S'', F'')$ with $S'' = S \cap S'$, $F'' = (F \cap P') \cup (P \cap F')$ and $X'' = X \cup X'$. For trace structures T and T' that are not defined over the same alphabet, the composition is defined as $T \parallel T' = \phi^{-1}(A' - A)(T) \parallel \phi^{-1}(A - A')(T')$ by first expanding them to a common alphabet.

C Conformance

Given a specification T_S, and an implementation T_I, T_I is said to be a "safe substitution" when it does not show more behavior than given by the specification. This safe substitution is also called *conformance* and can be formally defined in terms of languages as:

Definition 1 *Let $T = (I, O, S, F)$ and $T' = (I', O', S', F')$ be two canonical, non-empty trace structures. T* conforms *to T', denoted $T \preceq T'$, iff $I = I'$, $O = O'$ and $P \subseteq P'$ and $F \subseteq F'$ (or equivalently: $X \supseteq X'$ and $F \subseteq F'$).*

Conformance is a necessary condition for a circuit to correctly implement a specification and the above definition directly provides a procedure for checking it based on language containment.

A trace structure $T = (I, O, S, F)$ is called *failure free* if its failure set F is empty. The *mirror* or *reflection* of a trace structure $T = (I, O, S, F)$, denoted by $\mathbf{mirror}(T)$, defines a unique maximum environment T^M such that T composed with that environment T^M yields a failure-free trace structure [6, 7]. For a canonical trace structure $T = (I, O, S, F)$, $\mathbf{mirror}(T)$ is defined as (O, I, S, X). Conformance can now equivalently be defined using the mirror operation:

Lemma 1 *Let T_I and T_S be two canonical, non-empty trace structures. Then $T_I \preceq T_S$* iff *$T_I \parallel \mathbf{mirror}(T_S)$* is failure-free.

While state graphs are used as a starting point for the synthesis of a circuit implementing it, they can be translated very easily into trace structures so that it is possible to formally reason with them. As we consider only canonical trace structures $T = (I, O, S, F)$ in which the failure set and the impossible trace set are determined uniquely by the success language, such trace structures are fully characterized by a triple (I, O, S). In Section V we show that all operations on trace structures can be implemented by operations on (deterministic) finite automata [9]. It is then straightforward to see that state graphs are canonical trace structures in which the state graph itself represents exactly the success language.

Note that state graphs have a 4-phase interpretation of signal transitions by using explicit rise and fall transitions in the description, whereas the trace structures defined in [6] use a 2-phase interpretation of such events. However, this difference in interpretation only shows up in the computation of the failure (and impossible) trace sets for a canonical trace structure. Dill's trace theory can then be applied without further change.

III Progress and Simulation

In this section, we develop notions of *progress* and *simulation* that extend the basic notion of conformance used in verification in order to characterize the correct substitution problem. Conformance is a necessary but not sufficient condition for a trace structure T_I to actually implement a trace structure T_S. It guarantees that nothing "bad" happens, but it does not require for the actual implementation to produce sufficiently enough success traces. As an example, the empty trace set is always conforming.

To remedy this, we define a notion of *progress* of a trace structure T_I with respect to a trace structure T_S with the same input and output signals. Intuitively, progress states that if a success trace is in both trace structures T_S and T_I, and that trace is followed in the specification T_S by events on output signals only, then T_I generates at least one of those output events. This property is not guaranteed by conformance alone. Conformance would actually allow that success trace to become an impossible trace on those outputs in T_I, resulting in a practically useless circuit. Formally, progress is defined as follows.

Definition 2 *Let $T_I = (I, O, S, F)$ and $T_S = (I', O', S', F')$ be two trace structures with $I = I', O = O'$. The set $V(x) = \{a | a \in A, xa$ in $S'\}$ is the set of symbols that are accepting suffixes of a string x in S'. Then T_I is said to make* progress *w.r.t. T_S iff*

$$x \in S \wedge V(x) \subseteq O \Rightarrow \exists p \in V(x) : xp \in S$$

I.e. when a success trace can only be extended by some outputs in the specification, the set of success traces of T_I must include at least one of these outputs.

Now, we can define the necessary and sufficient conditions of a trace structure to correctly implement a specification, expressed by the term simulation:

Definition 3 *Let $T_I = (I, O, S, F)$ and $T_S = (I', O', S', F')$ be two canonical, non-empty trace structures with $I = I', O = O'$. T_I* simulates *T_S, denoted as $T_I \sim T_S$, iff $T_I \preceq T_S$* and *T_I* makes progress w.r.t. *T_S*

When we say that T simulates T', we imply several properties. First, all input choices in T' are maintained in T and in addition no outputs may be generated by T that were not specified in T'. These properties are expressed by the conformance constraint. Furthermore, the progress constraint expresses that at least one output has to be generated by T when there are only output choices for a trace in T'. As an example, let $T' = (\{a\}, \{b, c, d\}, (a(b + c + d))^\star, A^\star a a A^\star)$. Then $T = (\{a\}, \{b, c, d\}, (a(b + c)a(c + d)a(b + d))^\star, A^\star a a A^\star)$ makes progress w.r.t. T' but $T'' = (\{a\}, \{b, c, d\}, a(d)a, A^\star a a A^\star)$ does not.

Note that in a situation where for a success trace there are input choices as well as output choices, simulation implies that all inputs choices must be present in T but an arbitrary (and possibly empty) subset of the output choices may be present in T. This reduction of output choices allowed by the simulation property, is called *output concurrency reduction*, and it is valid from a *behavioral* point of view, i.e. in terms of success traces. However, it is

possible that synthesis techniques do not allow *arbitrary* output concurrency reduction as the assumed underlying architecture of the circuit may require certain properties about the structure of the state graph and therefore of the set of success traces.

IV Finding the Maximum Set of Correct Substitutions

A known and important problem in the context of interacting hierarchical asynchronous circuits, is the *correct substitution problem*. This problem was already motivated and intuitively depicted in Section I. In this section, the theoretical framework for this problem is stated in terms of trace structures using the notions of progress and simulation defined in the previous section. This will provide us with a *general* solution to the problem by the derivation of a *maximal trace structure*.

Due to the interaction of the replaceable circuit with its surrounding network, of which the behavior is (partially) specified, the environment of the circuit to be replaced may be more restricted, e.g. by generating less inputs to this circuit, then given by its initial specification. Therefore, in general more than a single solution exists to the correct substitution problem.

Besides this cause of possibly having more than one correct substitution, the choice of network topology also affects the solution space. A *network topology* defines the actual interconnection of the substituted module with the rest of the system. In Figure 1, the replaceable module T_1 is shown to have a particular set of input and output signals. It can, however, be the case that a solution for T_1 with also using some internal signal of T_2, or primary input or output of T which was not specified as input or output for T_1, may result in a more efficient solution replacement for T_1. Those signals reflect behavior already implemented by another module which may be used in the implementation of T_1.

To represent all the possible solutions to the correct substitution problem, we will exactly define the maximal behavior to which the solution must conform. It will be proven that this maximal behavior can be represented by a *single* canonical trace structure. By the use of trace structures, which describe the behavior of hierarchical systems in terms of languages, a full general model is provided. In addition, it can be made independent of the specific network topology of (local) input and (local) output signals. This allows to describe solutions ranging over an *arbitrary topology*. Therefore, we formulate the solution of the correct substitution problem in two ways. One describes the circuit to be substituted depending on the global space of all possible inputs, while the other describes it in terms of a particular topology. In the following, let the specification of the overall circuit be given by a trace structure $T = (I, O, S, F)$, and the module that is to be replaced be given by $T_1 = (I_1, O_1, S_1, F_1)$. The remaining and (possibly only partially) specified partial implementation is represented by a single canonical trace structure $T_2 = (I_2, O_2, S_2, F_2)$. T_2 can be derived by the parallel composition of all the trace structures of its modules.

In the following, we will show that a maximal trace structure can be derived for the correct substitution problem. This circuit is maximal in the sense that there exists no other solution that has more concurrency. It is the solution that does not restrict the freedom of synthesis tools to generate an implementation with respect to their own implementation constraints and optimization strategies. Two main techniques are used in automatic asynchronous circuit design: (1) reduction of output concurrency, and (2) addition of state signals. State signals may be added to get a state assignment with which, for example, a speed-independent circuit can be generated. We show that any implementation generated in this way from the maximal solution is also a solution to the correct substitution problem. The correct substitution problem is formally defined as follows.

Problem 1 *Let T_2 and T be canonical trace structures. T represents the specification and T_2 represents a partial implementation. The correct substitution problem is defined to be the problem of finding T_1 such that $T_1 \parallel T_2 \sim T$*

To provide a solution to this problem we make use of the following intermediate results.

Theorem 1 (Global Space) *Let T_1, T_2 (implementations) and T (specification) be canonical trace structures with alphabets A_1, A_2 and A, respectively. Assume $O_1 \cap O_2 = \emptyset$ and that $A = A_1 \cup A_2$ and $A_1 = A \cup A_2$. Then $T_1 \parallel T_2 \preceq T$ if and only if $T_1 \preceq$ **mirror**$(T_2 \parallel$ **mirror**$(T))$.*

Proof: Let $T_I = T_1 \parallel T_2$, which implies $T_I \preceq T$. By the definitions of conformance and the mirror operator, we know that $T_I \preceq T$ if and only if $T_I \parallel$ **mirror**(T) is failure free. Expanding on T_I, it follows that $T_1 \parallel T_2 \preceq T$ if and only if $(T_1 \parallel T_2) \parallel$ **mirror**(T) is failure free. Because the compose operator is associative, it follows that $T_1 \parallel (T_2 \parallel$ **mirror**$(T))$ must also failure free. This is the case if and only if $T_1 \preceq$ **mirror**$(T_2 \parallel$ **mirror**$(T))$. □

Corollary 1 *If $T_1 \preceq$ **mirror** $(T_2 \parallel$ **mirror**$(T))$, then $\forall\ T_2' \preceq T_2 : T_1 \parallel T_2' \preceq T$.*

Let $T_{max} =$ **mirror**$(T_2 \parallel$ **mirror**$(T))$ be the maximal trace structure. Theorem 1 shows that all $T_1 \preceq T_{max}$ composed with T_2 conform to T. However, as the corollary shows, this is also valid when T_1 is composed with an arbitrary implementation T_2' of T_2. This shows that T_{max} is maximal wrt to the conformance ordering $\preceq$.

Theorem 1 defines the trace structure T_1 in terms of the *global space* of *all possible input signals*. However, in reality, a component in a network of asynchronous circuits may only be defined in terms of a subset of all possible input signals. This can be required for reasons of optimization, to reduce the number of dependencies, or simply because in the actual implementation of the network, T_1 does *not* have all possible input signals at its disposal. Then, T_1 in Theorem 1 should be made independent of certain inputs. Equivalently, this can be expressed as the hiding of *output* signals on the *mirror* of T_1.

Now, we can express the conformance constraint to find the maximal trace structure in terms of a local space for T_1'.

Theorem 2 (Local Space) *Let T_1, T_2 (implementations) and T (specification) be trace structures with alphabets A_1, A_2 and A, respectively. Assume that T_1 and T_2 do not share any outputs (so they can be composed), and that $A = A_1 \cup A_2$. Also assume that $A_1 \subseteq A \cup A_2$. Then* **project**$(A)(T_1 \| T_2) \preceq T$ *if and only if $T_1 \preceq$* **mirror**$($**project**$(A_1)(T_2 \|$**mirror**$(T)))$.

A similar corollary as above can be stated.

As T_{max} is in general not independent on all input signals, it is evident that signals may not be arbitrarily hidden. For those network topologies, an implementable solution does not exist to the correct substitution problem. It will be shown later how this can be checked according to the progress notion.

Theorem 3 *Let $T_{max} =$* **mirror**$(T_2 \parallel$ **mirror**$(T))$ *in the case of the global space situation, and let $T_{max} =$* **mirror**$($**project**(A_1) $(T_2 \parallel$ **mirror**$(T)))$ *in the case of the local space situation. Then $(T_1 \parallel T_2) \sim T$ if and only if $T_1 \preceq T_{max}$ and $(I, O, S_1, (S_1.I - S_1).A^\star) \sim T$.*

Note that the latter condition equals to $T_1 \sim T$ in case T_1 has the same set of inputs and outputs as T (e.g. in the global space situation).

Proof: By definition of simulation, $(T_1 \| T_2) \sim T$ holds iff $(T_1 \| T_2) \preceq T$ and $(T_1 \| T_2)$ makes progress w.r.t. T. This first condition holds iff $T_1 \preceq T_{max}$, by Theorem 1.

Regarding the second necessary condition that $(T_1 \| T_2)$ must make progress w.r.t. T, note that the *success* language generated by $(T_1 \| T_2)$ is at most $S_{max} \cap S_2 = (S \cap S_2) \cap S_2 = (S \cap S_2) = S_{max}$. Therefore, $(T_1 \| T_2)$ makes progress w.r.t. T iff the *success* language of $T_1 \preceq T_{max}$ actually makes progress w.r.t. T. Therefore, since strict formally we can not compare T_1 and T directly, since their input and output signals are not compatible, we use $(I, O, S_1, (S_1.I - S_1).A^\star) \sim T$. □

Instead of checking for all possible $T_1 \preceq T_{max}$ if it simulates T to see if *a* solution exists to the correct substitution problem, we actually compute T_{max} first and remove all traces from it that do not make progress w.r.t. T. The resulting trace structure T'_{max} contains all possible solutions. If T'_{max}, after composition with $(I_2, O_2, A_2^*, \emptyset)$, does not simulate T, then no solution exists, due to the restricted behavior of T_2.

V Synthesis Procedure

The theorems stated in the previous sections provide us a with a solution framework to optimize or synthesize a hierarchical network of asynchronous circuits.

Starting from a high-level specification, for example in the form of a signal transition graph, of the specification P and a partial implementation P_2, we first convert both specifications into state graphs G and G_2, respectively. As described in Section II, the state graphs can be converted to trace structures T and T_2, respectively, in a straightforward way. To find all possible ways to optimize T_1, or just to find a legal T_1 in the case of the rectification problem, such that T_1 composed with T_2 simulates T, we apply Theorem 3 and compute T_{max} by

$$T_{max} = \textbf{mirror}(\textbf{project}(A_1)(T_2 \| \textbf{mirror}(T))),$$

where A_1 may be a subset of all signals present in the network.

Then, we check if there is a solution at all in transforming T_{max} into a T'_{max} by removing all non-simulating traces recursively. To satisfy the conditions of Theorem 3, it is checked if, this trace structure properly simulates the specification.

Before converting the maximal trace structure T_{max} back to a state graph, we can first apply classical state minimization [9] as T_{max} is now canonical and can be represented by a deterministic finite automaton G_{max}, which is the maximal state graph. Then we apply an asynchronous circuit synthesizer on G_{max}, e.g. [21, 22] to produce a speed-independent implementation for it.

A Implementation details

For finite state systems, all canonical prefix-closed trace structures can be modeled by classical finite automata[6, 12], In these automata, every state is accepting. The set of failure traces and the set of impossible traces can be modeled by adding two states, the failure state F and the impossible state X to which the undefined inputs and outputs are directed. All trace structure operations can be implemented as operations on these automata [9]. For the global space problem, T_{max} can then be computed in polynomial time and space. For the local space problem, a determinization procedure may be needed, which is exponential in the worst case.

T_{max} is the solution to the correct substitution problem when we only consider conformance. In that case, any conforming $T_1 \preceq T_{max}$ is a solution. As all success trace sets $\subseteq S_{max} \cup X_{max}$ are conforming, any subautomaton of the (unfolded) T_{max} is a solution. For implementability, we need to check also the progress condition. This check and the removal of the non-simulatable traces in T_{max} can be implemented (in a similar way as in [12]) as additional language containment checks to generate a T'_{max} in which all subautomata (possibly after unfolding) are simulating solutions to the correct substitution problem. For the case of a non-deterministic T_2, it may not be clear how to construct a satisfying $T_1 \sim T_{max}$ as the choices made when implementing T_2 may influence the correct choice of T_1. However, it is possible to derive conditions such that any $T_1 \sim T_{max}$ is a correct solution for any choice of $T'_2 \sim T_2$. But in many cases, such a solution does not exist. But the above method can be used if the partial implementation T_2 is completely given.

B Synthesis techniques

The above T_{max} is identical, before or after classical state minimization[9], to the maximal state graph G_{max}. We will now show that all kinds of optimization and implementation strategies as used by automatic synthesis tools for asynchronous circuits can be applied to this G_{max} to yield an implementable solution to the correct substitution problem. One of the first steps in synthesis is state assignment. Different methods of state assignment exist that operates at the state graph level, e.g. [11, 21]. However, this is not the concern of this paper. The goal for state assignment is to transform the state graph to make it satify certain properties, like distributivity, in order to synthesize a hazard-free implementation. It also depends on the implementation architecture, e.g. by using bounded delay assumptions and delay information [11], speed-independent circuits with complex gates [4, 13, 20] or other sum-of-product architectures [2], or only external hazard-free architectures [17].

State assignment adds new state signals Q and transitions $Q \times \{+, -\}$ to G to produce $G' = (I, O \cup Q, S')$. The associated trace structures to these state graphs are $T = (I, O, S, F)$ and $T' = (I, O \cup Q, S', F')$, as adding state signals changes the network topology by adding outputs. But because for all these state assignments

$$\textbf{hide}(Q)(T') \equiv T,$$

it is clear that such state assignments will only yield trace structures that form solutions to the correct substitution problem, i.e. they always conform to T_{max}.

Another synthesis technique is "concurrency reduction" of outputs, e.g. [22]. In the most general case, this means that when more than one output choice exists at a state in the specification, only a subset of those "excited" outputs is generated in the implementation. As input choices are not reduced, the implementation will be conforming to the specification, as it will also not generate more outputs than specified. In the case of having only output choices at a state in the state graph, at least a non-empty subset of these outputs should be generated to satisfy the progress property. Together, this yields that all types of output concurrency reduction are allowed. The synthesis tool may use more restricted forms of concurrency reduction to satisfy other constraints, as for instance induced by the type of architecture for an implementation.

Thus we have shown that T'_{max} contains all correct and safe substitutable solutions in the sense that any $T'_1 \sim T'_{max}$ is possible. T'_1 corresponds to some output concurrency reduction of T'_{max} in addition to some new signal insertions.

VI Implementation and Experiments

We have implemented the solution framework described in this paper in C. Currently, very few examples are available for use as benchmark circuits to evaluate the described method. We have examined an interface protocol conversion module, for which the block diagram is shown in Figure 2. This example consists of two modules: a VME bus controller module **vmectrl** and a local conversion module **pcm**. Both modules were described using the Signal Transition Graph model [4] and were mechanically translated to state graphs. The processor can read and write data from and to a memory.

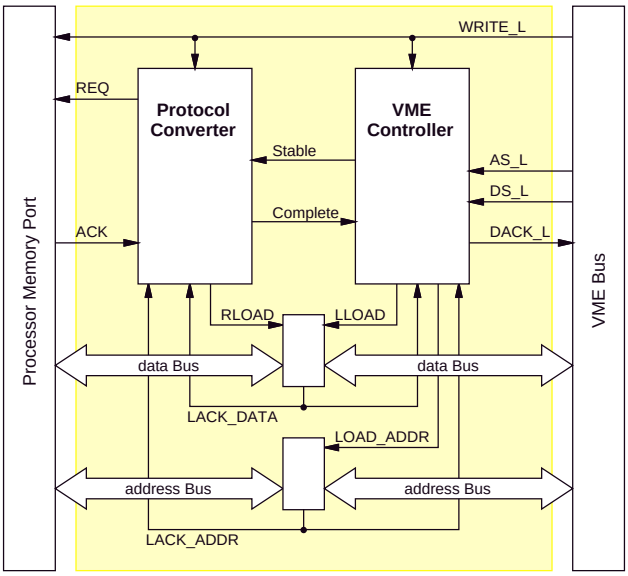

Figure 2: Block diagram of protocol converter.

The overall design can operate in two modes: a read and a write mode. We took as global specifications two versions of the problem where the environment restricts the operations to only read or only write transfers. This gives a global specification for the read-only mode and the write-only mode respectively. For each of these two specifications, we computed the maximal trace structure for each of the two components by regarding the other component as the partial implementation. Thus, in one experiment we regarded **vmectrl** as partial implementation and tried to find the maximal replacement for **pcm**, and vice-versa for the second experiment. The suffixes **-ro** and **-wo** indicate that the read-only or write-only mode global specification was used.

The results of these experiments are shown in Table 1. The column labeled "I / O wires" gives the number of inputs and outputs of the corresponding module. The column labeled "CPU (sec)" shows the CPU time (on an HP9000 / 715 workstation) required to compute the maximum set of correct substitutions for that module as performed by an automaton manipulation package which implements all types of operations on finite automata, e.g. language intersection and automaton minimization, by means of fully explicit enumeration methods. The four columns labeled under "without optimization" show the results of synthesis of the original state graph. The columns labeled under "with optimization" show the synthesis results of the state graph obtained from the maximum correct substitution procedure. The number of states of the corresponding state graphs are shown under the columns labeled "#states". Each of the state graphs were synthesized using the state assignment technique described in [21]. The number of new state signals inserted by this technique for each module is shown under the columns labeled "#signals". The SYN tool of [2] was used to produce a gate-level hazard-free circuit for each module and it reported the area and delay figures under the columns "area" and "delay". These are still preliminary experiments, but they indicate that optimization is possible.

Table 1: Experimental results.

name	I/O wires	without optimization		with optimization		CPU (sec)
		Q / s	A / D	Q / s	A / D	
pcm-ro	5 / 3	68 / 2	1322 / 4.8	34 / 0	300 / 2.4	1.8
vmectrl-ro	6 / 4	192 / 2	1724 / 6.0	62 / 1	670 / 3.6	2.1
pcm-wo	5 / 3	68 / 2	1322 / 4.8	32 / 1	316 / 2.4	2.6
vmectrl-wo	6 / 4	192 / 2	1724 / 6.0	84 / 1	1032 / 4.8	3.6

Q: number of states A: area
s: number of state signals D: delay

VII Conclusions

In this paper, we addressed the correct substitution problem for a design which is implemented hierarchically as an interconnection of separate asynchronous modules. Specifically, we have presented a new solution framework based on trace theory for computing and representing the complete set of correct substitutions (permissible behaviors) for an asynchronous module embedded in such a hierarchically defined network. With this theory, we have derived a single maximal trace structure which describes this set of solutions. This maximal trace structure takes into account all the degrees of freedom that arise from the interaction with the other components and from the overall specification. We also have shown that the maximal trace structure does not restrict the freedom of available asynchronous synthesis tools to generate hazard-free circuits.

References

[1] V. Akella and G. Gopalakrishnan. Shilpa: A High-Level Synthesis System for Self-Timed Circuits. In *Proc. Int. Conf. on CAD*, Nov. 1992.

[2] P.A. Beerel and T. Meng. Automatic Gate-Level Synthesis of Speed-Independent Circuits. In *Proc. Int. Conf. on CAD*, Nov. 1992.

[3] E. Brunvand and R.F. Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *Proc. Int. Conf. on CAD*, Nov. 1989.

[4] T.-A. Chu. Synthesis of Self-Timed VLSI circuits from Graph-Theoretic Specifications. Technical Report MIT-LCS-TR-393, 1987.

[5] G. de Jong and B. Lin. A Communicating Petri Net Model for the Design of Concurrent Asynchronous Modules. In *Proc. Design Aut. Conf.*, pages 49–55, June 1994.

[6] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[7] J.C. Ebergen *Translation Programs into Delay-Insensitive Circuits.* Ph.D. Thesis, Eindhoven University of Technology, 1987.

[8] G. Gopalakrishnan, E. Brunvand, N. Michell and S.M. Nowick. A Correctness Criterion for Asynchronous Circuit Validation and Optimization. In *IEEE Trans. of CAD*, 13(11):1309-1318, Nov. 1994.

[9] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[10] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic Gate Implementation of Speed-Independendent Circuits. In *Proc. Design Aut. Conf.*, pages 56–62, June 1994.

[11] L. Lavagno, C.W. Moon, R.K. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignment Problem for Signal Transition Graphs. In *Proc. Design Automation Conf.*, pages 568–572. 1992.

[12] B. Lin, G. de Jong and T. Kolks Modeling and Optimization of Hierarchical Synchronous Circuits In *Proc. Eur. Design and Test Conf.*, March 1995.

[13] A.J. Martin. Programming in VLSI: From Communicating Processes to Self-Timed VLSI Circuit Synthesis. In C.A.R. Hoare, editor, *Concurrent Programming*, pages 1–64. Addison-Wesley, 1989.

[14] D.E. Muller. Lecture Notes on Asynchronous Circuits Theory. Urbana: Univ. of Illinois, 1961.

[15] C. Myers and T. Meng. Synthesis of Timed Asynchronous Circuits. In *Proc. Int. Conf. Computer Design*, Oct. 1992.

[16] M. Rem, J.L.A. van de Snepscheut, and J.T. Udding. Trace Theory and the Definition of Hierarchical Components. In *Third Cal-Tech Conference on Very Large Scale Integration*, pages 225–239, 1983.

[17] M. Sawasaki, Ch. Ykman-Couvreur, and B. Lin. Externally Hazard-Free Implementations of Asynchronous Circuits In this proceedings.

[18] K. van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* Ph.D. thesis, Eindhoven University of Technology, 1992.

[19] J.L.A. van de Snepscheut. *Trace Theory and VLSI design.* Ph.D. Thesis, Eindhoven University of Technology, Oct. 1983.

[20] V.I. Varshavsky, Editor. *Self-Timed Control of Concurrent Processes.* Kluwer Academic Publishers, 1990.

[21] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proc. Int. Conf. on CAD*, Nov. 1992.

[22] Ch. Ykman-Couvreur, P. Vanbekbergen, and B. Lin. Concurrency Reduction Transformations on State Graphs for Asynchronous Circuit Synthesis. In *Proc. Int. Workshop on Logic Synthesis*, May 1993.

[23] K.Y. Yun, D.L. Dill, and S.M. Nowick. Automatic synthesis of 3d asynchronous state machines. In *Proc. Int. Conf. Computer Design,*, Oct. 1992.