On the Synthesis of an Asynchronous Reactive Module

Amir Pnueli and Roni Rosner*
Department of Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel
amir@wisdom.bitnet, roni@wisdom.bitnet

Abstract

We consider the synthesis of a reactive asynchronous module which communicates with its environment via the shared input variable x and the shared output variable y, assuming that the module is specified by the linear temporal formula $\varphi(x, y)$. We derive from $\varphi(x, y)$ another linear formula $\chi(r, w, x, y)$, with the additional *scheduling* variables r, w, and show that there exists a program satisfying φ iff the branching time formula $(\forall r, w, x)(\exists y)A\chi(r, w, x, y)$ is valid over all tree models. For the restricted case that all variables range over finite domains, the validity problem is decidable, and we present an algorithm, of doubly exponential time and space complexity, for constructing a program that implements the specification whenever it is implementable. In addition, we provide some matching lower bounds.

1 Introduction

One of the most central problems in the theory of programming is the systematic construction of a program from its specification, using rules that automatically ensure that the constructed program satisfies the specification. As we all very well know, this problem, in its full generality, can never be algorithmically solved. Consequently, most of the research efforts in this field have concentrated on deriving, on one hand, a formal framework in which humans can exercise their ingenuity in selecting an appropriate derivation sequence, and on the other hand, offer a set of heuristics that will help the developer in his selection. The advantages inherent in the formal framework are that, once the creative part embodied in the selection of the derivation sequence is completed, and each derivation step is formally justified, we are guaranteed to get a program that satisfies the specification. Different derivation paradigms differ from one another by the formal framework they recommend and the set of heuristics they offer.

One of the interesting formal frameworks is that of *Program Synthesis by Constructive Theorem Proving*. According to this paradigm, we construct from the specification an *implementability formula*, which states the existence of an implementation satisfying the specification. Then, from a constructive proof of the implementability formula it is possible to extract an actual program that satisfies the specification. This approach has been successfully applied to sequential programs by Manna and Waldinger in [MW80]. A somewhat different formal approach, but still

*The work of this author was partially supported by the Israel ministry of science and development, the national council for research and development.

basing program synthesis on constructive theorem proving, has been consistently and extensively developed in [Con85].

To illustrate application of this approach to sequential programs, consider a contemplated sequential program module with input x and output y. Let the specification of the module be given by the formula $\varphi(x, y)$. For example, the specification for a root extracting program may be presented by the formula $(x \geq 0) \rightarrow (|x - y^2| < \epsilon)$. Then the implementability formula corresponding to the specification $\varphi(x, y)$ is given by $(\forall x)(\exists y)\varphi(x, y)$.

This approach, which may be called the *AE-paradigm*, or alternately, the *Skolem paradigm*, is based on the observation that the formula $(\forall x)(\exists y)\varphi(x, y)$ is equivalent to the second order formula $(\exists f)(\forall x)\varphi(x, f(x))$, stating the existence of a function f, such that $\varphi(x, f(x))$ holds for every x. If we restrict the proof rules, by which the synthesis formula is to be established, to a particular set of *constructive* rules, then any proof of its validity necessarily identifies a constructive version of the function f, from which a program that satisfies the specification φ can be constructed.

Several interesting efforts have been made to extend this approach to the domain of concurrent and reactive programs. The pioneering works of [MW84] and [EC82], consider a specification $\varphi(x, y)$, given in temporal logic, and base a synthesis algorithm on the *satisfiability* of the specification, showing how to extract a program from the model satisfying φ.

In [PR89], we claimed that the satisfiability-based approach is adequate only for the synthesis of a *closed* system, which maintains no interaction with its environment. However, for a really reactive *open* module, which interacts with its environment, a more complicated implementability formula is called for. This formula characterizes the relation between the module and its environment as a two persons game, in which the module is expected to possess a winning strategy, defining a sequence of y values for each possible sequence of x values, such that together they satisfy $\varphi(x, y)$. In the cited paper (see also [PR88]), we treated the simpler case of *synchronous* systems. These are systems equipped with an external clock, such that on each pulse of the clock, the environment provides an input value in x, and the module responds by placing an output value in y. For such systems, we have shown that the validity of the formula

$$(\forall x)(\exists y)A\varphi(x, y)$$

is a necessary and sufficient condition for implementability. An interesting point emerging from this analysis, is that even if the formula $\varphi(x, y)$ is given in *linear* temporal logic, we need the branching framework to express its implementability, as is evident from the branching operator A, appearing in the above formula, and meaning 'for all paths'. Another part of [PR89] considers the restricted case of finite-state programs. In this case, the specification $\varphi(x, y)$ can be expressed in propositional temporal logic and the validity of the implementability formula becomes decidable. For this case, [PR89] presents a doubly exponential procedure that checks the implementability of $\varphi(x, y)$, and constructs an actual implementation whenever one exists. The particular case of finite state synchronous systems has already been considered in [BL69]. In that paper, Büchi and Landweber provided an algorithm for the synthesis of such a system from a specification, given by a Muller automaton. However, they paid very little attention to the efficiency of the suggested algorithm (no analysis of complexity has been given), and as a result, their algorithm is significantly less efficient than the one presented in [PR89].

While synchronous systems may provide a good model for some classes of hardware circuits (namely, synchronous circuits) — and this was the main motivation for the work of [BL69] — they are not the appropriate model for most of the cases of *reactive programs*. It is rarely the case that a reactive program module is synchronized with its environment by a common clock, which simultaneously controls the rate of progress of both. A more appropriate model

for reactive programs is that of asynchronous systems, where the progress of each component (such as the module or its environment) is essentially independent of the progress of the other components. Of course, some of the main interesting questions about such systems are those of synchronization and coordination, but those are not automatically built into the model, and have to be programmed explicitly. In most such models, we have special synchronization constructs, such as semaphores or synchronous message exchange (hand-shaking), which simplify the task of programming the necessary coordination. With this in mind, we consider the work on synthesis of synchronous systems, reported in [PR89], as a preparatory exercise, paving the way for the main goal of synthesis of asynchronous systems, which are the more realistic model of reactive programs.

In the present paper we take this next step, and consider the problem of synthesizing an asynchronous reactive program from its temporal specification. The framework in which we study this problem is that of processes communicating by shared variables. The simplest case, and this is the main one we study here, is that of a single process (module) interacting with its environment via an input variable x which is writable only by the environment, and an output variable y which is writable only by the module. The extension to many inputs and many outputs is straightforward.

The main results we obtain are the identification of an implementability formula for a given specification $\varphi(x, y)$, which we prove to be a necessary and sufficient condition for the existence of a program satisfying $\varphi(x, y)$. Similarly to the simpler case of synchronous systems, the implementability formula must be stated in a branching framework, to express the fact that the reaction of the module given by the value of y, can only depend on the *previous* values of the input x. The second main result is a complete solution to the asynchronous synthesis problem for the finite state case. This is presented by an algorithm that, given a specification $\varphi(x, y)$, checks whether the implementation formula for this specification is valid, and if it is valid, produces a program satisfying the specification. The time complexity of the algorithm is doubly exponential, and it is based on a new algorithm for checking non-emptiness of a Streett tree automaton.

Implementation of (shared variables) asynchronous programs from their specifications is inherently more difficult than the implementation of synchronous systems. This is because synchronous systems are aware of every change in their inputs, and are allowed an immediate reaction following such a change. The basic model for asynchronous systems, on the other hand, allows them to sample their inputs only at some selected points and to modify their outputs at some other selected points (the points at which their actions are *scheduled*). Consequently, it is possible, in principle, for an asynchronous system to be unaware of some changes in their inputs, and even if aware of such changes, not to be able to correspondly modify their outputs fast enough to maintain the specification. Consider, for example the specification $\Box(x \equiv y)$. In this, and all the other examples presented below, we express specifications in a propositional temporal logic, to be defined formally in a subsequent section. This specification, requiring that the values of x and y be equal at any state, is easily implemented by a synchronous program which consistently copies the current value of x to y. However, this specification cannot be implemented by an asynchronous system, because such a system cannot be guaranteed to be fast enough to react to the changes in x. A more interesting example is one that allows some delay in the response of the system. Consider the specification

$$(\Diamond\Box x) \equiv (\Diamond\Box y).$$

This specification requires that the output y eventually settles at T if and only if the input x eventually settles at T. Note that it allows an arbitrary delay between the settling of x and that of y. Again, this specification is easily implemented by the trivial synchronous program which consistently copies x to y. On the other hand, this specification is unimplementable by an

asynchronous program. The reason this time is not because it cannot respond fast enough, but rather that since it samples the input only at *some* points it is impossible for it to distinguish between the input sequence in which x is true at *all* the states, and the input sequence in which x is true at precisely the sampling points and false elsewhere. The required responses to these two sequences are of course different. For the first sequence, y should eventually settle at $\mathbf{T}$ while, for the second sequence, it should not. To modify this specification into one that is asynchronously implementable we should restrain the environment not to change the input too fast. This is done by adding an appropriately restraining clause as a precondition (antecedent in an implication) of the specification. Two examples of such modifications are:

$$(\Diamond\Box x \vee \Diamond\Box\neg x) \rightarrow (\Diamond\Box x \equiv \Diamond\Box y),$$

or

$$\Box\big(\neg x \rightarrow (\neg x)\,\mathsf{U}(\neg y)\big) \rightarrow (\Diamond\Box x \equiv \Diamond\Box y).$$

The difficulties associated with specifying shared-variables asynchronous systems by temporal logic which refers also to states in which the system does not read or write, were first realized by Lamport (see for example [Lam83]). To overcome some of these difficulties he introduced the notion of *stuttering*. Two computations are considered to be stuttering equivalent if one is obtained from the other by a sequence of operations, each of which is either a finite duplication of a given state, or the deletion of a finite number of contiguous identical states retaining at least one representative. Then he required that all the basic definitions and concepts associated with asynchronous systems should be insensitive to stuttering equivalence. A consequence of this general principle is that the temporal logic to be used should not be able to distinguish between stuttering equivalent sequences. This led Lamport to a variant of temporal logic which does not admit the *next* operator $\bigcirc$. This version of temporal logic has the advantage that a sequence satisfies a formula in this logic iff all stuttering equivalents of this sequence do.

Unfortunately, the property of insensitivity to stuttering breaks down as soon as we allow in the language quantification over dynamic variables. For example, taking the standard definition, by which $(\forall x)\varphi$ holds over a sequence π if φ holds over a sequence π' which differs from π at most by the assignments, the states of π' give to the variable x, it is possible to define $\bigcirc p$ by the formula

$$(\exists y)\Big(y \wedge (\forall x)\big(x \rightarrow x\,\mathcal{U}(p \wedge \neg y)\big)\Big).$$

Since quantification over dynamic variables is an important tool for the representation of hiding and abstraction, and is certainly crucial for our study of implementability, the conflict between the conventional interpretation of quantification and the wish to retain stuttering robustness led to the study of dense time models such as [BKP86]. However, there exists another possibility (inspired by some observations of Lamport [Lam86]), which considers a different interpretation of quantification. This quantification, which we denote by $\exists^{\approx}$, and call *stuttering quantification*, is defined by stating that $(\exists^{\approx} x)\varphi$ holds over a sequence π, if $(\exists x)\varphi$ holds over some sequence π' which is stuttering equivalent to π. Thus, to interpret $(\exists^{\approx} x)\varphi$ we allow first to replace π by any stuttering equivalent π', and then to arbitrarily modify the interpretation of x in π'. This stuttering quantification can no longer express the forbidden property $\bigcirc p$. In the present paper we propose a temporal logic augmented with the stuttering quantification as the appropriate language for specifying and expressing the implementability of asynchronous systems.

In section 2 we present the syntax and semantics of the temporal logic we use. In section 3 we define the notions of an asynchronous program, its behavior, and what does it mean for an asynchronous program to satisfy a specification φ. We show some examples of unimplementable specifications. In section 4 we formulate the main theorem which identifies the implementability

formula corresponding to a specification φ, and gives five equivalent clauses which are sufficient and necessary conditions for the implementability of φ. In section 5 we consider the finite state case and describe the construction of a tree automaton whose non-emptiness guarantees implementability, and such that the algorithm for checking non-emptiness also produces the implementing program if one exists. We also provide some lower bounds on this construction.

2 Temporal Logic

We describe the syntax and semantics of a general branching time temporal language. This language is an extension of CTL* ([CES86,EH86,ES84,HT87]), obtained by admitting variables, terms, and quantification. Its *vocabulary* consists of *variables* and *operators*. For each integer $k \geq 0$, we have a countable set of k-ary variables for each of the following types: *static function variables* — F^k, *static predicate variables* — U^k, *dynamic function variables* — f^k, and *dynamic predicate variables* — u^k. The intended difference between the dynamic and the static entities is that, while the interpretation of a dynamic element in a model may vary from state to state, the interpretation of a static element is uniform over the whole model. For simplicity, we refer to 0-ary function variables simply as (individual) variables, of which we have both the static and the dynamic types. The operators include the classical $\neg$, $\vee$, $\exists$, the *stuttering quantifier* $\exists^{\approx}$, and the temporal $\bigcirc$ (next), $\mathcal{U}$ (until) and E (for some path). A survey of similar approaches in the context of *modal logic* can be found in [Bac80].
Terms: For all $k \geq 0$, if $t_1, \ldots, t_k$ are terms, then so are $F^k(t_1, \ldots, t_k)$ and $f^k(t_1, \ldots, t_k)$.
State formulae are precisely the formulae defined by the following rules:

1. For all $k \geq 0$, if $t_1, \ldots, t_k$, are terms, then $U^k(t_1, \ldots, t_k)$ and $u^k(t_1, \ldots, t_k)$ are (atomic) state formulae.
2. If p and q are state formulae, then so are $\neg p$ and $(p \vee q)$.
3. If p is a state formula and v is any variable, then $(\exists v)p$ and $(\exists^{\approx} v)p$ are state formulae.
4. If p is a path formula then Ep is a state formula.

Path formulae are precisely the formulae defined by the following rules:

1. Every state formula is a path formula.
2. If p is a path formula and v is any variable, then $(\exists v)p$ and $(\exists^{\approx} v)p$ are path formulae.
3. If p and q are path formulae, then so are $\neg p$, $(p \vee q)$, $\bigcirc p$, and $(p \mathcal{U} q)$.

We shall omit the superscripts denoting arities and the parentheses whenever no confusion can occur. We also use the following standard abbreviations: T for $p \vee \neg p$, F for $\neg \mathrm{T}$, $p \wedge q$ for $\neg(\neg p \vee \neg q)$, $p \rightarrow q$ for $\neg p \vee q$, $p \equiv q$ for $(p \rightarrow q) \wedge (q \rightarrow p)$, $(\forall v)p$ for $\neg(\exists v)(\neg p)$, $(\forall^{\approx} v)p$ for $\neg(\exists^{\approx} v)(\neg p)$, $\Diamond p$ (eventually) for $\mathrm{T} \mathcal{U} p$, $\Box p$ (henceforth) for $\neg\Diamond\neg p$, $p \mathrm{U} q$ (unless) for $p \mathcal{U} q \vee \Box q$, $p \Rightarrow q$ (temporal entailment) for $\Box(p \rightarrow q)$, and Ap (for all paths) for $\neg E(\neg p)$. We shall use the letters a, b for static individual variables (*constants*) and x, y for dynamic individual variables, or *propositions*, for the Boolean case.

By restricting this syntax, one gets special cases of the temporal language. In *classical static* logic, there are no dynamic variables nor temporal operators ($\bigcirc, \mathcal{U}, E$). *First-order* logic permits $\exists$ quantification over individual variables only. In *propositional* (resp. *quantified propositional*)

logic, the only variables permitted are propositions, without (resp. *with*) $\exists$ quantification over them. Finally, *linear* temporal logic omits the E operator (i.e. all linear formulae are path formulae).

The semantics of temporal logic is given with respect to *models* of the form $\mathcal{M} = \langle D, M \rangle$, where D is some non-empty data *domain*, and $M = \langle S, R, L \rangle$ is a *structure*. S is a countable set of *states*, $R \subseteq S \times S$ is a binary total *access relation* on S, and L is the *labeling function*, assigning to each state $s \in S$ an *interpretation* $L(s)$ of all variables into appropriate functions and predicates over D. We require all of our models to satisfy the *static consistency condition*, by which for every two states $s, s' \in S$ and each *static* variable a, $L(s, a) = L(s', a)$, i.e., the interpretation of static variables is uniform over all states. Since for most of our discussions we can hold the domain D fixed, we can identify a model $\mathcal{M} = \langle D, M \rangle$ with its structure M.

For a given variable v, we define the structure $M' = \langle S, R, L' \rangle$ to be a *v-variant* of the structure $M = \langle S, R, L \rangle$, **iff** for every state $s \in S$ and variable $w \neq v$, $L'(s, w) = L(s, w)$. Thus, M' agrees with M on all elements, except possibly on the interpretation given to v.

A *path* in M is a sequence $\pi = (s_0, s_1, \ldots)$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. A *fullpath* in M is an infinite path. Denote by $\pi^{(j)} = (s_j, s_{j+1}, \ldots)$ the j^{th} *suffix of* π.

A structure $M = \langle S, R, L \rangle$ is called a *tree-structure*, **iff** the following conditions are satisfied:

1. There exists precisely one state, $r \in S$, called the *root* of M, which has no *parent*, i.e., no state $s \in S$, such that $R(s, r)$.
2. Every other state $t \neq r$, has precisely one parent.
3. For every state $s \in S$, there exists a unique path leading from r to s.

A model $\mathcal{M} = \langle D, M \rangle$, is called a *tree model*, **iff** the structure M is a tree-structure.

We define a relation between structures, as follows. For two structures $M = \langle S, R, L \rangle$ and $M' = \langle S', R', L' \rangle$, we say that M' *stretches* M, and write $M' \gg M$, **iff** there exists a (*narrowing*) function $H : S' \to S$ satisfying the following conditions:

1. For every $s \in S$, $1 \leq |H^{-1}(s)| \leq n_s$, for some integer n_s.
2. R' is the minimal relation on $S' \times S'$, such that for each $s \in S$, the elements of $H^{-1}(s)$ can be arranged in a sequence $s_1, \ldots, s_{n_s}$, where $(s_i, s_{i+1}) \in R'$ for $1 \leq i < n_s$, and $(s, t) \in R \Leftrightarrow (s_{n_s}, t_1) \in R'$.
3. For each $s' \in S'$, $L'(s') = L(H(s'))$.

Thus, H^{-1} stretches each node $s \in S$ into a *linear* sequence $s_1, \ldots, s_n$ in S', such that the labeling of s is identical to those of $s_1, \ldots, s_n$, and the branching structure of s is identical to that of s_n.

We will write $M' \gg_H M$, to emphasize that H is the narrowing function under consideration, in which case we also write $\langle \mathcal{M}', s' \rangle \gg_H \langle \mathcal{M}, H(s') \rangle$ for all $s' \in S'$. For a fullpath $\pi = (s^0, \ldots, s^i, \ldots)$ in M, where $M' \gg_H M$, let π' be the (single) fullpath in M' defined by $\pi' = (s^0_1, \ldots, s^0_{n_0}, \ldots, s^i_1, \ldots, s^i_{n_i}, \ldots)$, where $H^{-1}(s^i) = \{s^i_1, \ldots, s^i_{n_i}\}$ for all $i \geq 0$. We say that π' *stretches* π, and write $\langle \mathcal{M}', \pi' \rangle \gg_H \langle \mathcal{M}, \pi \rangle$.

Using the "$\gg$" relation, we define the *stuttering* equivalence relation "$\approx$", as follows. Let M_1 and M_2 be structures, with fullpaths π_1, π_2 and states s_1, s_2 in M_1 and M_2 respectively. M_1 is said to be *stuttering equivalent* to M_2, written $M_1 \approx M_2$, **iff** there exists a structure M' such that $M' \gg_{H_1} M_1$ and $M' \gg_{H_2} M_2$, for some narrowing functions H_1 and H_2. If there also exists

a fullpath π' in M' which stretches both π_1 and π_2 (under H_1 and H_2 respectively), then we say that π_1 and π_2 are *stuttering equivalent*, and write $\langle M_1, \pi_1\rangle \approx \langle M_2, \pi_2\rangle$. If for some state s' of M', $H_1(s') = s_1$ and $H_2(s') = s_2$ (or equivalently $H_1^{-1}(s_1) \cap H_2^{-1}(s_2) \neq \phi$), we shall write $\langle M_1, s_1\rangle \approx \langle M_2, s_2\rangle$.

In order to interpret applications of static functions to terms at a state s, we use standard definitions over the appropriate interpretation $L(s)$ and the semantics of the given terms.

Satisfiability of a state formula is defined with respect to a model $\mathcal{M}$ and a state $s \in S$, inductively as follows:

S-1 $\langle \mathcal{M}, s\rangle \models u(t_1, \ldots, t_k)$ **iff** $L(s)(u)(d_1, \ldots, d_k) = \mathbf{T}$, where d_i is the semantics of t_i in $\langle \mathcal{M}, s\rangle$, $1 \leq i \leq k$. That is, we apply u, as interpreted in s by L, to the evaluation of $t_1, \ldots, t_k$, as interpreted in s.

S-2 $\langle \mathcal{M}, s\rangle \models \neg p$ **iff** not $\langle \mathcal{M}, s\rangle \models p$.

S-3 $\langle \mathcal{M}, s\rangle \models p \vee q$ **iff** $\langle \mathcal{M}, s\rangle \models p$ or $\langle \mathcal{M}, s\rangle \models q$.

S-4 For a variable v and state formula p, $\langle \mathcal{M}, s\rangle \models (\exists v)p$ **iff** $\langle \mathcal{M}', s\rangle \models p$ for some $\mathcal{M}'$ which is a v-variant of $\mathcal{M}$.

S-5 For a variable v and state formula p, $\langle \mathcal{M}, s\rangle \models (\exists^{\approx} v)p$ **iff** $\langle \mathcal{M}', s'\rangle \models (\exists v)p$ for some $\langle \mathcal{M}', s'\rangle \approx \langle \mathcal{M}, s\rangle$.

S-6 For a path formula p, $\langle \mathcal{M}, s\rangle \models Ep$ **iff** for some fullpath $\pi = (s, \ldots)$ in M, $\langle \mathcal{M}, \pi\rangle \models p$.

Satisfiability of a path formula is defined with respect to a model $\mathcal{M}$ and a fullpath π in M, according to the following:

P-1 For $\pi = (s_0, \ldots)$ and a state formula p, $\langle \mathcal{M}, \pi\rangle \models p$ **iff** $\langle \mathcal{M}, s_0\rangle \models p$.

P-2 $\langle \mathcal{M}, \pi\rangle \models \neg p$ **iff** not $\langle \mathcal{M}, \pi\rangle \models p$.

P-3 $\langle \mathcal{M}, \pi\rangle \models p \vee q$ **iff** $\langle \mathcal{M}, \pi\rangle \models p$ or $\langle \mathcal{M}, \pi\rangle \models q$.

P-4 For a variable v and path formula p, $\langle \mathcal{M}, \pi\rangle \models (\exists v)p$ **iff** $\langle \mathcal{M}', \pi\rangle \models p$ for some $\mathcal{M}'$ which is a v-variant of $\mathcal{M}$.

P-5 For a variable v and path formula p, $\langle \mathcal{M}, \pi\rangle \models (\exists^{\approx} v)p$ **iff** $\langle \mathcal{M}', \pi'\rangle \models (\exists v)p$ for some $\langle \mathcal{M}', \pi'\rangle \approx \langle \mathcal{M}, \pi\rangle$.

P-6 $\langle \mathcal{M}, \pi\rangle \models \bigcirc p$ **iff** $\langle \mathcal{M}, \pi^{(1)}\rangle \models p$.

P-7 $\langle \mathcal{M}, \pi\rangle \models p \mathcal{U} q$ **iff** for some $j \geq 0$, $\langle \mathcal{M}, \pi^{(j)}\rangle \models q$, and for all i, $0 \leq i < j$, $\langle \mathcal{M}, \pi^{(i)}\rangle \models p$.

We say that the model $\mathcal{M}$ satisfies the state formula p, and write $\mathcal{M} \models p$, **iff** $\langle \mathcal{M}, s_0\rangle \models p$, for some state s_0 of $\mathcal{M}$. For the case of a tree model, there is no loss of generality in assuming that s_0 is the root of the model. A state formula p is said to be *satisfiable* **iff** for some model $\mathcal{M}$, $\mathcal{M} \models p$. The formula p is *valid*, denoted by $\models p$, iff it is satisfied by every model.

3 The Implementability problem

We define a (semantic) *program* P from D to D to be a function $f_P : D^+ \to D$, i.e., a function mapping non-empty sequences of D elements into elements of D. The intended meaning of this function is that it represents a program with an input variable x ranging over D, and an output variable y with the same range, such that at each step $i = 0, 1, ...$, the program outputs (assigns to y) the value $f_P(a_0, a_1, \ldots, a_i)$, where $a_0, a_1, \ldots, a_i$ is the sequence of input values observed in x by P over steps $0, 1, \ldots, i$. It is important to note that these values are not necessarily the only values x assumes throughout the computation, but rather the *observable* ones, i.e., the values which the program manages to encounter at the time points when it is *scheduled* to *read*. The difference between synchronous programs and asynchronous ones, lies precisely in the characteristics of the *scheduler* assumed to control its behavior. These characteristics will be now formalized, by defining the set of the synchronous/asynchronous *behaviors* of a program.

Note that our treatment at this point is *semantic*, meaning that we are mainly interested in the existence of such a function, and ignore, for the time being, the question of its expressibility within a given programming language. Obviously, a complete examination of the synthesis problem should consider both the semantic aspect and the *syntactic* aspect.

We define an *asynchronous behavior* of a program P to be an infinite sequence (for simplicity we only consider non-terminating programs) of pairs

$$\sigma \; : \; \langle a_0, b_0 \rangle, \; \langle a_1, b_1 \rangle, \quad \cdots$$

such that $a_i, b_i \in D$ for $i = 0, 1, \ldots$, and there exist two infinite *scheduling sequences* of indices

$$R: \quad r_1 < r_2 < \ldots,$$

$$W: \quad w_1 < w_2 < \ldots,$$

such that

B-1 For every $i = 1, 2, \ldots, \quad r_i \leq w_i < r_{i+1}$.

B-2 For every $i = 1, 2, \ldots$, and every j, $w_i \leq j < w_{i+1}$, it follows that $b_j = f_P(a_{r_1}, \ldots, a_{r_i})$. Also, $b_k = b_0$ for $0 \leq k < w_1$.

The sequence of pairs $\langle a_i, b_i \rangle$, represents the values of the variables x, y, respectively, at state i of the computation. The subsequence $r_1, r_2, \ldots$, identifies the states at which the program reads (observes) the values of x. The subsequence $w_1, w_2, \ldots$, identifies the states at which the program writes (possibly new) values to y. Requirement B-1 stipulates, with no loss of generality, that the program alternates between reading and writing. Requirement B-2 specifies the output value written at step i, $i = 0, 1, \ldots$, to be $f_P(a_{r_1}, \ldots, a_{r_i})$, where $a_{r_1}, \ldots, a_{r_i}$ are the values read by the program in the preceding reading steps. It also requires that the value of y remains constant between subsequent writing operations. Note that no similar constraint is imposed on the input variable x which may change arbitrarily between subsequent reading operations.

The behavior is considered to be *synchronous* if, in addition to B-1 and B-2, it also satisfies

B-3 For every $i = 1, 2, \ldots, \quad r_i = w_i = i$.

Note that according to this definition, synchronous behavior is a special case of an asynchronous behavior.

A behavior is defined to be *strictly asynchronous* (*strict* for short), if in addition to B-1 and B-2, it also satisfies

B-4 For every $i = 1, 2, \ldots, \quad 0 < r_i < w_i < r_{i+1}$.

A program P asynchronously/synchronously/strictly satisfies a linear temporal logic specification $\varphi(x, y)$, written $P \models_{\text{asyn/syn/str}} \varphi(x, y)$, **iff** every asynchronous/synchronous/strict behavior σ of P satisfies $\sigma \models \varphi(x, y)$.

We may now consider the following three problems, which can be presented as a parametric problem with parameter $\alpha \in \{\text{asynchronous, synchronous, strict}\}$.

Problem 3.1 (α-Implementability) *Given a linear time specification $\varphi(x, y)$, does there exist a program P which α-satisfies $\varphi(x, y)$?*

Specifications for which such a program exists are called *α-implementable*, and the program satisfying them is called an *α-implementation* of the corresponding specification.

The main question is to find conditions which are necessary and sufficient for the (semantic) implementability of a given specification. For example, in the *transformational* case, i.e., that of non-reactive terminating sequential programs, the specification is given by a first order (non-temporal) formula $\varphi(x, y)$ where x and y range over some domain D. The basis for synthesis of transformational programs is the theorem that such a specification is implementable iff the *implementability formula* $(\forall x)(\exists y)\varphi(x, y)$ is valid. We are looking for a similar condition for the reactive case.

Clearly, satisfiability of the temporal formula $\varphi(x, y)$ (also guaranteeing its consistency), which is the basis for the synthesis approaches of [EC82] and [MW84], is a necessary but not a sufficient condition for implementability in our sense. To see this, it is sufficient to consider a specification which constrains the sequence of input values provided by the environment, without even saying anything about the sequence of output values. An example of such a specification is given by

$$\varphi \ : \ \Diamond x.$$

This formula states that at least one of the input values is T. In this and all the other examples presented here, we assume the domain D to be the boolean domain, and x,y to be dynamic boolean variables. This specification is unimplementable because, no matter how the function f_P is defined, the program P will always have a behavior of the form

$$\langle \mathrm{F}, b_0 \rangle, \langle \mathrm{F}, b_1 \rangle, \ldots,$$

which violates the above specification. This, of course, is a direct consequence of the fact that in constructing the module P, we cannot control the behavior of the environment, expressed by the sequence of input values it chooses to present to the module.

Our next best attempt has been to mimic the transformational case, and suggest the linear temporal formula $(\forall x)(\exists y)\varphi(x, y)$ as an implementability formula, whose validity is a necessary and sufficient condition for the existence of an implementation. Unfortunately, this does not work either. Consider the specification

$$\varphi(x, y) \ : \ (y \equiv \Diamond x),$$

which requires that the first output value (issued at step 0) is T **iff** some input value, appearing now or later, equals T. It is not difficult to see that the formula $(\forall x)(\exists y)\varphi(x, y)$ is valid. It is sufficient to substitute $\Diamond x$ for y, in order to get the obvious tautology $(\forall x)(\Diamond x \equiv \Diamond x)$. However, this specification is certainly not implementable. To implement it we need a clairvoyant program, which can foresee at the first step whether any of the future inputs will ever equal T. Our definition of programs, which corresponds to the way real programs operate, allows the program to base

its decision of the next output only on the inputs it has seen so far, i.e., only on the past. In [PR89] we have shown that this restriction, requiring y to depend only on the preceding values of x, can be captured by the implementability formula within a branching time structure, yielding the formula

$$(\forall x)(\exists y)A\varphi(x,y).$$

The validity of this formula requires that for every structure whose nodes are labeled by some interpretation for x, it is possible to label the nodes by some interpretation for y, such that $\varphi(x,y)$ holds on all paths (traces) in the structure. If we consider a tree, then the label of y at a given x should be adequate for all possible paths that branch beyond that point. Consequently, the value of y cannot depend on the future, since there are many possible futures, branching from this node, and hence it can only depend on the past.

The consideration of asynchronous behaviors introduces a new constraint. Now the value of y can no longer depend on *all* the values of x in the past, but only on the values of x observable during a read operation. It must be insensitive to changes in the unobserved values of x. Furthermore, it must be insensitive to the precise points at which reading and writing are performed, since these may vary between behaviors.

It follows that asynchronous implementability is harder to satisfy, in the sense that there are many specifications that are synchronously implementable but not asynchronously implementable (obviously, every specification which is asynchronously implementable is also synchronously implementable). Examples of such specifications are

$$\Box(x \equiv y) \qquad \text{and} \qquad \Diamond\Box x \equiv \Diamond\Box y.$$

In the next section we will show how, by using auxiliary variables which explicitly identify the states at which reading and writing is performed, we can express the requirement of independence of unobserved values.

The notions of asynchronous and strictly asynchronous implementations are closely related, as is stated by the following proposition.

Proposition 3.2 (Strictness) *A linear specification φ is asynchronously implementable* **iff** *it is strictly implementable.*

Proof: Assume that P asynchronously implements φ. Since every strict behavior is also an asynchronous behavior, it follows that all strict behaviors of P satisfy φ, and therefore P is also a strict implementation of φ.

In the other direction, let P be a strict implementation of φ. We define the following program Q:

$$\begin{array}{lll} f_Q(a_1) & = b_0 & \text{for every } a_1 \in D \\ f_Q(a_1, a_2, \ldots, a_{2i}) & = & \\ f_Q(a_1, a_2, \ldots, a_{2i+1}) & = f_P(a_1, a_3, \ldots, a_{2i-1}) & \text{for } i \geq 1, \text{ and } a_j \in D,\ j = 1, \ldots, 2i+1. \end{array}$$

Thus, Q lags at least one step behind P, and emits on the $2i$ step the value emitted by P for the first i odd-indexed inputs. To show that Q is an asynchronous implementation of φ, consider any asynchronous behavior σ of Q, with its associated scheduling sequences $R : r_1 < r_2 < \ldots$ and $W : w_1 < w_2 < \ldots$. Define two new sequences, $R' : r'_1 < r'_2 < \ldots$ and $W' : w'_1 < w'_2 < \ldots$, by $r'_i = r_{2i-1}$ and $w'_i = w_{2i}$. It is now straightforward to show that σ with the scheduling sequences R' and W' is a *strict* behavior of P and hence satisfies φ. ∎

Consequently, it is sufficient to consider the strict implementability problem, which is the strategy we follow in the subsequent sections.

4 The Implementability Condition

Assume that we wish to develop a reactive module, communicating with the environment by an input variable x and output variable y. The partition of the variables into input and output means that only the environment can modify x, and only the module can modify y.

Let $\varphi(x,y)$ be a formula in linear temporal logic, specifying the requirements of the module. We assume that x and y are the only free variables in φ. For simplicity, we only consider the case of a single input variable and a single output variable — the extension of the results below, to the case where x and y are vectors of variables, is straightforward. We further assume the validity of $\varphi \rightarrow ((x = a_0) \wedge (y = b_0))$, in order to express our interest in models with roots satisfying a fixed initial condition, namely $x = a_0$ and $y = b_0$.

Let D be some fixed data domain which contains the Boolean values T and F, and the fixed initial values a_0 and b_0. Since D is fixed, we shall identify models $\mathcal{M}$ of the form $\langle D, M\rangle$ with their non-fixed elements, i.e., with the dynamic structure M, and write 'the model M', referring to the whole model $\mathcal{M} = \langle D, M\rangle$.

We formulate an *implementability formula* whose validity over all tree models is a necessary and sufficient condition for strict asynchronous implementability. We begin by defining the *kernel path formula* $\chi(r,w,x,y)$, given by

$$\chi(r,w,x,y): \quad \alpha(r,w) \rightarrow \beta(r,w,x,y).$$

The kernel formula has, in addition to the variables x and y, also the free Boolean variables r and w, which identify (by the event of *becoming* true) the points at which reading and writing, respectively, are performed. The introduction of these *scheduling variables*, is dual to the approach taken in [BKP84], where the choice of scheduling is identified by *transition propositions*. The antecedent $\alpha(r,w)$ ensures that r and w properly encode reading and writing points, and is given by

$$\alpha(r,w): \quad \left(\begin{array}{c} \neg r \wedge (\neg w)\,\mathcal{U}\,r \\ \wedge \\ \Box\neg(r \wedge w) \\ \wedge \\ r \Rightarrow\!\!\succ r\,\mathcal{U}\,(\neg r)\,\mathcal{U}\,w \\ \wedge \\ w \Rightarrow\!\!\succ w\,\mathcal{U}\,(\neg w)\,\mathcal{U}\,r \end{array}\right).$$

Since we want all our constructs to be insensitive to stuttering, we allow continuous intervals of true r and true w, rather than isolated states at which r and w hold. However, the actual operations of reading and writing are considered to occur at the *first* state of the appropriate interval, that is, at the state where the scheduling variable *rises* from F to T. We consider each of the clauses comprising α.

- $\neg r \wedge (\neg w)\,\mathcal{U}\,r$ — This clause requires that both r and w are false on the first state, and that the first interval in the sequence is an r-interval.
- $\Box\neg(r \wedge w)$ — This clause requires that r-intervals and w-intervals never overlap.
- $r \Rightarrow\!\!\succ r\,\mathcal{U}\,(\neg r)\,\mathcal{U}\,w$ — This clause uses a nested until formula to require that every r-state is followed by an r-interval, which is then followed by a (possibly empty) $\neg r$-interval, followed by a w-state. This ensures that every r-interval is followed by a w-interval.

- $w \Rightarrow w\,\mathcal{U}\,(\neg w)\,\mathcal{U}\,r$ — This clause similarly requires that every w-interval is followed by an r-interval. Together, these two clauses ensure that r-intervals and w-intervals alternate, and there are infinitely many of them.

The formula $\beta(r, w, x, y)$, appearing as the consequent of χ, expresses the requirement that y is a good and robust response to x, in order to satisfy $\varphi(x, y)$. It is given by

$$\beta(r, w, x, y): \quad \left(\begin{array}{c} \varphi(x, y) \\ \wedge \\ (\forall a)\Big((y = a) \Rightarrow (y = a)\,\mathsf{U}\,\big(\neg w \wedge (y = a)\,\mathsf{U} w\big)\Big) \\ \wedge \\ (\forall^{\approx} x')\Big(\big(\neg r \Rightarrow (\neg r)\,\mathsf{U}(x = x')\big) \rightarrow \varphi(x', y)\Big) \end{array} \right).$$

Consider each of the clauses appearing in β.

- $\varphi(x, y)$ — This clause obviously requires that the computation satisfies the specification.
- $(\forall a)\Big((y = a) \Rightarrow (y = a)\,\mathsf{U}\,\big(\neg w \wedge (y = a)\,\mathsf{U} w\big)\Big)$ — This clause requires that y may change its value only at states, in which w *rises*, i.e., states at which w has just changed from F to T. Clearly, w rises in a state iff it is the first state in a w-interval. The way this requirement is expressed by the clause is that, if in a certain state $y = a$, then y will remain a at least until we observe $w = \mathrm{F}$, and then will remain a for awhile longer, until we see w rises to T.
- $(\forall^{\approx} x')\Big(\big(\neg r \Rightarrow (\neg r)\,\mathsf{U}(x = x')\big) \rightarrow \varphi(x', y)\Big)$ — This clause requires that the fact that x and y satisfy φ depends only on the values of x at the observed points, i.e., points of rising r. This is expressed by requiring that if we replace x by x', which coincides with x at all states of rising r, but assumes completely arbitrary values at all other states, then still $\varphi(x', y)$ holds.

Note that the quantification over x' is a *stuttering* one. To see why this is necessary, consider the following prefix of a computation (representing the interpretation of each state by a quadruple of Boolean values, interpreting r, w, x and y, respectively):

$$s_0: \langle \mathrm{F}, \mathrm{F}, \mathrm{F}, \mathrm{F}\rangle,\ s_1: \langle \mathrm{T}, \mathrm{F}, \mathrm{T}, \mathrm{F}\rangle,\ s_2: \langle \mathrm{F}, \mathrm{T}, \mathrm{T}, \mathrm{T}\rangle,\ \ldots$$

In this prefix s_1 is a reading state (r rises from F in s_0 to T in s_1) and s_2 is a writing state. We may want to examine the situation for an x' which coincides with x at all the reading states, but changes from F to T at least once between the first reading and the first writing. This cannot be done by modifying x into x' in the precise prefix shown above. This is because at s_1 which is a reading state, we must have $x = x'$, and therefore do not have enough states to cause x' to switch from F to T before the next writing. However, if we perform a stuttering transformation first, we may obtain the prefix

$$s_0: \langle \mathrm{F}, \mathrm{F}, \mathrm{F}, \mathrm{F}\rangle,\ s_1: \langle \mathrm{T}, \mathrm{F}, \mathrm{T}, \mathrm{F}\rangle,\ s_1': \langle \mathrm{T}, \mathrm{F}, \mathrm{T}, \mathrm{F}\rangle,\ s_2: \langle \mathrm{F}, \mathrm{T}, \mathrm{T}, \mathrm{T}\rangle,\ \ldots$$

where s_1' is a duplicate of s_1. On this prefix we can now modify x in s_1' to obtain the prefix (listing now the interpretations for r, w, x' and y):

$$s_0: \langle \mathrm{F}, \mathrm{F}, \mathrm{F}, \mathrm{F}\rangle,\ s_1: \langle \mathrm{T}, \mathrm{F}, \mathrm{T}, \mathrm{F}\rangle,\ s_1': \langle \mathrm{T}, \mathrm{F}, \mathrm{F}, \mathrm{F}\rangle,\ s_2: \langle \mathrm{F}, \mathrm{T}, \mathrm{T}, \mathrm{T}\rangle,\ \ldots$$

Note that x' is still T at s_1, but changes from F to T on going from s_1' to s_2. This shows why, if we wish to represent all possible scenarios of an x behavior, we may have to perform a stuttering extension first and only then to specify the x' interpretation.

Having defined the kernel formula $\chi(r, w, x, y)$, the implementation condition requires that the formula

$$(\forall r, w, x)(\exists y) A\chi(r, w, x, y)$$

holds over all tree models.

Instead of testing $A\chi$ over *all* tree models, it is sufficient to test it on a particular family of tree models that are guaranteed to contain all the interesting input sequences. These tree models should contain all paths of the form (listing the interpretations of r, w and x, respectively):

$$\langle \mathtt{F}, \mathtt{F}, a_0 \rangle,\ \langle \mathtt{T}, \mathtt{F}, a_1 \rangle,\ \langle \mathtt{F}, \mathtt{T}, - \rangle,\ \langle \mathtt{T}, \mathtt{F}, a_2 \rangle,\ \langle \mathtt{F}, \mathtt{T}, - \rangle,\ \langle \mathtt{T}, \mathtt{F}, a_3 \rangle,\ \ldots$$

for all infinite sequences of D values, $(a_1, a_2, a_3, \ldots)$. Note that in this path, all states at *odd* positions (the first position is called 0 and is considered even) are reading states, while all even positions, excluding the first (position 0), are writing states. We have not specified the value of x at writing positions since it is unobservable. We refer to a tree model that contains all these paths as a *canonical model.* It is convenient to specify the value of x at position $2i$, $i > 0$, as a_i, i.e., identical to the value of x in the preceding state. It is also convenient to name a state s in a canonical model by the string of values assumed by x on the path from the root to s, omitting the value at the root. Thus the name of the root is ε — the empty string. Consequently, we define a *canonical model* to be a model $M = (S, R, L)$, where

1. S is the minimal subset of D^* such that for every sequence $(a_1, \ldots, a_i, a_{i+1})$ of elemnts of D, the string $a_1 a_1 \ldots a_i a_i$ of lengt $2i$, and the string $a_1 a_1 \ldots a_i a_i a_{i+1}$ of length $2i+1$, belong to S. In particular, the empty string ε is in S.

2. The relation R holds between two states $z_1, z_2 \in S$, **iff** $z_2 = z_1 a$ for some $a \in D$.

3. The labeling L satisfies, for every $z \in S$ and $a \in D$

   - $L(\varepsilon, x) = a_0$ and $L(\varepsilon, y) = b_0$.
   - $L(za, x) = a$ **iff** $za \in S$.
   - $L(z, r) = \mathtt{T}$ **iff** $|z|$ is odd.
   - $L(z, w) = \mathtt{T}$ **iff** $z \neq \varepsilon$ and $|z|$ is even.

The following lemma states that it is sufficient to test the formula $A\chi(r, w, x, y)$ over canonical models.

Lemma 4.1 (Canonical Model) *The formula* $(\forall r, w, x)(\exists y) A\chi(r, w, x, y)$ *is valid over all tree models* **iff** $A\chi(r, w, x, y)$ *holds over some canonical model.*

Proof: The 'only if' direction follows immediately from the definitions. For the 'if' direction, consider a canonical model $\hat{M} = (\hat{S}, \hat{R}, \hat{L})$ which satisfies $A\chi$, and let $M = (S, R, L)$ be any tree model. Without loss of generality, we may assume that $L(r, x) = a_0$ at the root r of M, and moreover, every maximal fullpath $\pi = (r, \ldots)$ of M satisfies $\alpha(r, w)$. Define the model $M' = (S, R, L')$ to be the y-variant of M (that is, M' differs from M at most on the values assigned by L' to y at some states), with L' defined as follows. With any maximal fullpath $\pi = (r = s_0, s_1, \ldots)$ in M' (and in M), associate a maximal fullpath $\hat{\pi} = (\varepsilon = \hat{z}_0, \hat{z}_1, \ldots)$ in $\hat{M}$, called the *canonical shrinking* of π, to be defined below. Consider first a sequence $z_0, z_1, \ldots$, of states in $\hat{M}$, defined inductively by

$$z_0 = \varepsilon$$

$$z_{i+1} = \begin{cases} z_i a & \text{if } L(s_i, r) = \mathtt{F} \text{ and } L(s_{i+1}, r) = \mathtt{T},\ L(s_{i+1}, x) = a \\ z_i a' & \text{if } L(s_i, w) = \mathtt{F} \text{ and } L(s_{i+1}, w) = \mathtt{T}, \text{ where } a' \text{ is the last element in } z_i \\ z_i & \text{otherwise.} \end{cases}$$

Since π satisfies $\alpha(r,w)$, one can easily verify that the z_i's are well defined, and moreover, all z_i's are states of $\hat{M}$, lying along one maximal fullpath. Now, we can put $L'(s_n,y) = \hat{L}(z_n,y)$ for all states s_n along π (notice that the value of $L'(s_n,y)$ does not depend on the choice of fullpath π of which $(s_0,\ldots,s_n)$ is a prefix), and $\hat{\pi} = (\hat{z}_0,\hat{z}_1,\ldots)$, where $\hat{z}_0,\hat{z}_1,\ldots$, is the infinite subsequence of $z_0,z_1,\ldots$, deleting repetitions. Since $\hat{M} \models A\chi$, also $\langle \hat{M},\hat{\pi}\rangle \models \chi$, and using the third clause of β and the appropriate definitions, it follows that $\langle M',\pi\rangle \models \chi$. But this is true of all maximal fullpaths π in M', so $M' \models A\chi$, and since M' is an y-variant of M, it proves that $M \models (\exists y)A\chi$. Finally, since these arguments apply to every tree model M, we can conclude that $(\forall x)(\exists y)A\chi$ is valid over *all* tree models. ⌟

Given a canonical model M, we can interpret it as a program P^M, represented by the function f_{P^M}, such that

$$f_{P^M}(a_1\ldots a_i) \;=\; L(a_1a_1\ldots a_ia_i,y).$$

Lemma 4.2 $P^M \models \varphi$ iff $M \models A\chi$. ⌟

We conclude the general case by the following theorem.

Theorem 4.3 (Implementability) *The following conditions are equivalent:*

1. *The specification $\varphi(x,y)$ is asynchronously implementable.*
2. *The specification $\varphi(x,y)$ is strictly implementable.*
3. *The implementability formula $(\forall r,w,x)(\exists y)A\chi(r,w,x,y)$ is valid over all tree models.*
4. *The formula $A\chi(r,w,x,y)$ holds over some canonical model.*
5. *The formula $A\chi(r,w,x,y) \wedge A\alpha(r,w) \wedge A\vartheta(r,x)$ is satisfiable, where $\vartheta(r,x) = (\forall a)\Big(\neg r \Rightarrow\!\!\!\!\!- E\Big((\neg r)\,\mathsf{U}\,((x=a)\wedge r)\Big)\Big)$.*

Proof: The equivalence between the first four clauses is established by the previous propositions and lemmata. To relate the last clause, we observe that the conjunct $A\alpha(r,w)$ provides correct scheduling of read and write points, while the conjunct $A\vartheta(r,x)$ ensures that for every node s in the satisfying model, and every $a \in D$, there exists a branch departing from s, such that for the next reading point s_a on this branch, $L(s_a,x) = a$. Thus, a model satisfying this conjunct, has an 'almost' canonical model (up to stuttering equivalence) embedded in it (perhaps in a folded form). ⌟

5 The Finite-State Case

Restricting to Boolean domains, we may clearly treat the logic as purely propositional, by considering all variables to be propositions. Note that a variable ranging over a more general finite domain can always be represented by several Boolean variables.

Our main result, showing how to construct an effective *finite-state* representation of a program which satisfies the *propositional* specification φ, is stated by the following theorem:

Theorem 5.1 (Synthesis Algorithm) *There is a deterministic algorithm such that, given a propositional linear time specification $\varphi(x,y)$ as above, it checks whether the specification is implementable. If the specification is implementable, the algorithm constructs a deterministic finite automaton D, with input x and output y, which represents a finite-state program P^D satisfying $P^D \models \varphi$. The running time of the algorithm and the size of D, are both doubly exponential in the size of φ.*

Proof: Given the specification φ, let $n = |\varphi|$ be the size of φ, and $\chi(r, w, x, y)$ be the appropriate kernel formula defined in a previous section. We proceed in three main steps.

In the first step we construct a non-deterministic Büchi automaton A on ω-strings representing the negated kernel formula $\neg\chi$. The input to the automaton A is an infinite sequence of states, i.e., truth values assignments to the variables r, w, x and y. A sequence is accepted by A iff it satisfies $\neg\chi$. The reason we choose to work with the negation of χ is that all the quantifications in $\neg\chi$, such as the static quantification on a and the stuttering one on x', are now existential, which are easier to represent by *nondeterministic* automata constructions.
It is easy to perform the automata-theoretic operations of intersection, union and projection on a smaller alphabet in the framework of nondeterministic Büchi automata, with no significant increase in the size of the operands. Consequently, we can concentrate on the construction of automata for $\alpha(r, w)$ and for each of the disjuncts of $\neg\beta(r, w, x, y)$, since $\neg\chi = \alpha \wedge \neg\beta$. For a linear time formula without occurrences of the $\exists^{\approx}$ operator, the standard constructions (see [VW86,ES84]) yield, in general, a non-deterministic automaton of size exponential in n. Hence, the only new construction to be considered at this step is that of an automaton representation for $\varrho(r, x, y) = (\exists^{\approx} x')\varrho_0(r, x, x', y)$, where

$$\varrho_0(r, x, x', y) = \Big(\neg r \Rightarrow (\neg r)\,\mathsf{U}\,(x = x')\Big) \wedge \neg\varphi(x', y).$$

Let $A_0 = (S, Q, Q_0, \delta, F)$ be a non-deterministic automaton representation for ϱ_0, with the set of states Q, initial states $Q_0 \subseteq Q$, accepting states $F \subseteq Q$, and transition function $\delta : Q \times S \to 2^Q$, where S is the alphabet consisting of interpretations of the variables r, x, x', y into truth values. The idea is to first modify A_0, so that the modified automaton $A_1 = (S, Q \cup \hat{Q}, Q_0, \delta \cup \hat{\delta}, F \cup \hat{F})$ accepts a sequence iff it is stuttering equivalent to some sequence accepted by A_0. There are two cases to consider:

- Stretching — for $q \in Q$ with $\delta(q, s) \neq \phi$, add a new state q_s to $\hat{Q}$, with $\hat{\delta}(q, s) = q_s$, $\hat{\delta}(q_s, s) = \{q_s\} \cup \delta(q, s)$. That is, q_s is used to simulate any stuttering of the state q under the input s^+. Note that no such q_s is accepting.

- Compressing — for every $q_1, q_2, \ldots, q_k \in Q$ with $q_{i+1} \in \delta(q_i, s)$, $1 \leq i < k$, add the transition $q_k \in \hat{\delta}(q_1, s)$, in order to simulate in one s-transition of A_1 the path of s-transitions allowed by the original automaton A_0 on seeing an input sequence s^+ of length $k > 2$. However, if any of $q_2, \ldots, q_{k-1}$ is in F while both q_1 and q_k are not, then create a new *accepting* state $\hat{q}_k$ and add it to both $\hat{Q}$ and $\hat{F}$, with the transition $\hat{q}_k \in \hat{\delta}(q_1, s)$. After all such compressions are dealt with, add to every $\hat{q} \in \hat{F}$ the transitions of the original $q \in Q - F$, namely, $\hat{\delta}(\hat{q}, s) = \delta(q, s) \cup \hat{\delta}(q, s)$ for all s.

The correct construction performs all the stretching transformations first, and then proceeds to perform the compressing transformations. It is not difficult to verify that A_1 accepts the stuttering equivalence class of A_0. Now, the standard projection of A_1 on the alphabet of interpretations s of r, x, y only (that is, projecting x' out), yields the desired automaton representation for ϱ.

In the second step we construct a tree automaton, representing the implementability formula $(\forall r, w, x)(\exists y)A\chi(r, w, x, y)$, in the sense described below.
From A, it is possible to construct a *complementary deterministic* Rabin automaton B, using the co-determinization procedure of [Saf88] or [EJ89], whose number of states is doubly exponential in n, but its acceptance condition has only an exponential number of pairs. B is deterministic, and accepts the language complementary to the language of A, namely the sequences of states satisfying χ. Since B is deterministic, we can split each state in any accepted path into two components. The input component consists of the values of the universally quantified variables

r, w, x, and the output component consists of the value of the existentially quantified variable y. This way, B can be interpreted as a Rabin automaton $\bar{B}$ on labeled infinite trees, where at each state, the corresponding input component determines the *direction* to be taken in the tree, while the output component determines the label on the appropriate node of the tree.
It is a straightforward exercise to show that $\bar{B}$ accepts some infinite tree t **iff** t represents a tree model for $(\forall r, w, x)(\exists y)A\chi(r, w, x, y)$. If we concentrate on canonical tree models we can easily derive an appropriate implementation for φ from such a tree.

Consequently, the third step consists of applying a non-emptiness algorithm to the tree automaton $\bar{B}$. In [PR89] we have described an efficient algorithm for checking non-emptiness of Rabin tree automata (see also [EJ88]). The algorithm is based on the well known theorem [Rab72,HR72] stating that a tree automaton accepts some tree **iff** it accepts some *regular* tree, using the ideas of [HR72] to actually construct a folded representation of such a regular tree, if it exists. This folded representation, can in turn be re-interpreted as an ω-transducer, whose input is represented by the directions of the tree, and whose output is given by the labelings of the tree. Hence, if the automaton accepts some tree, the algorithm constructs a deterministic ω-transducer D representing an appropriate implementation P^D of the specification φ. The (deterministic) running time of the algorithm, as well as the size of D, are both polynomial in the number of states of the automaton, and exponential in the number of pairs in its acceptance condition, hence, doubly exponential in n. ∎

It should be noted, that permitting a more expressive linear language for φ, such as quantified propositional temporal logic may increase the complexity of the algorithm (and of the synthesized program) significantly. It is shown in [SVW87] that the satisfiability problem for this logic is complete for time non-elementary in the number of alternations of (linear) quantifiers in some normal form representation of the formula.

We supplement the synthesis algorithm with the following optimality results.

Theorem 5.2 (Lower Bound on Synthesis) *The implementability problem for linear propositional temporal specifications is log-space hard for deterministic doubly exponential time.*

Proof: The proof is a slight modification of the proof stating the same lower bound for the logic CTL* in [VS85]. It consists of a construction which, given an alternating Turing machine Z with space bound 2^{cn} for some constant $c > 0$, and an input π to Z, of sufficiently large length n, constructs a specification $\varphi_{Z,\pi}$ which is implementable **iff** Z accepts π. The size of $\varphi_{Z,\pi}$ is polynomial in n, and it can be constructed using logarithmic space, hence the desired lower bound (see [CKS81] for alternating Turing machines and the complexity of their acceptance, and [FL79] for a detailed application of this technique to propositional dynamic logic). In this proof it is sufficient to consider only the synchronous-implementability problem, since the case of strict-implementability (and hence also the general asynchronous-implementability) can be reduced to that of synchronous-implementability by systematically transforming all the occurrences of the $\bigcirc$ operator in $\varphi_{Z,\pi}$, into appropriate occurrences of $\mathcal{U}$.

The formula $\varphi_{Z,\pi}$ describes a path composed of a sequence of intervals of 2^{cn} states each. Such an interval describes, using the output variables $y_1, \ldots, y_{dn}$, for some constant $d > c$, one configuration of Z, so that the whole path describes a path through some *trace* of Z running on π. For an existential configuration of Z, the formula requires that the next configuration in the path is an appropriate successor configuration of Z. Without loss of generality, we may assume that every universal state of Z has precisely 2^k successors (some successor state may be considered more than once). Therefore, we use the input variables $x_1, \ldots, x_k$, to encode the identity index of one of these successors, when we order them in some arbitrary order. At the end of an interval describing an universal configuration, $\varphi_{Z,\pi}$ requires that the next configuration be determined by

the index encoded by the inputs $x_1, \ldots, x_k$. This way, an *accepting trace* of Z is embedded in every *implementation* for $\varphi_{Z,\pi}$. We omit any further details. ⌟

We now turn to the question of a lower bound on the size of the synthesized program. In the fashion of [SVW87], call a quantified propositional temporal formula ψ a Π_1^{SQ} formula iff it has the form $(\forall a_1)\ldots(\forall a_k)\varphi(a_1,\ldots,a_k)$, where $a_1,\ldots,a_k$, $k \geq 1$, are static propositions, and φ is a propositional linear-time formula. We shall abbreviate such formula ψ by $(\forall \bar{a})\varphi(\bar{a})$. Π_1^{SQ} formulae can be translated into pure propositional formulae, but with an exponential (in k) blow up in size. Nevertheless, for this class of formulae, both the lower and upper bounds on the size of implementations match, and are both doubly exponential.

Theorem 5.3 (Lower Bound on Implementation)

(i) Implementability of Π_1^{SQ} specifications is decidable in deterministic doubly-exponential time. If a specification is implementable, then the algorithm constructs an implementation whose size is doubly exponential in the size of the specification.

(ii) For every integer $n > 0$, there is an implementable Π_1^{SQ} specification ψ_n, of size polynomial in n, such that the size of any implementation of ψ_n is at least doubly exponential in n.

Proof: To show *(i)*, consider $\psi(x,y) = (\forall \bar{a})\varphi(x,y,\bar{a})$. Inspecting the proof of theorem 5.1, we observe that it is enough to show how to construct a (non-deterministic) Büchi automaton for $\neg\psi(x,y)$, in time and space exponential in the size of ψ. This will clearly do for the synchronous case too.
For every function $l : \{1,\ldots,k\} \to \{\mathtt{T},\mathtt{F}\}$, we can construct in exponential time and space an automaton A_l for the formula

$$\varphi_l(x,y,\bar{a}): \quad \varphi(x,y,\bar{a}) \wedge \bigwedge_{i=1}^{k} \Box(a_i \equiv l(i)).$$

Now, since $\neg\psi(x,y)$ is equivalent to

$$(\exists \bar{a}) \bigvee_l \varphi_l(x,y,\bar{a}),$$

and there are $2^k < 2^n$ such l's, we can form the nondeterministic union of all the automata A_l, and then project out $\bar{a}$ from this union, in order to achieve the desired automaton.

For the proof of *(ii)*, let $\bar{x} = x_1, \ldots x_n$, and define the formula

$$\psi_n(\bar{x},y): \quad (\forall \bar{a})(\forall b)\Big(\big((\bar{x}=\bar{a}) \wedge (y \equiv b)\big) \Rightarrow \bigcirc\,(\bar{x} \neq \bar{a})\,\mathsf{U}\big((\bar{x}=\bar{a}) \wedge (y \equiv \neg b)\big)\Big),$$

where $\bar{x} = \bar{a}$ abbreviates $\bigwedge_{i=1}^{n}(x_i \equiv a_i)$ and $\bar{x} \neq \bar{a}$ abbreviates $\neg(\bar{x} = \bar{a})$. Clearly $\psi_n(\bar{x},y)$ is implementable by any synchronous program which for any input in $\bar{x}$, outputs in y the value b iff the next time it reads the same input it will output $\neg b$. Such a program must have at least as many states as the number of Boolean functions of n variables, i.e. 2^{2^n}. Small changes in the formula, essentially substituting the $\bigcirc$'s by $\mathcal{U}$'s, and requiring the input not to be too 'wild', are enough for proving the same lower bound for the asynchronous case too. ⌟

6 Related Work and Further Research

The research reported in [Dil88] considers the implementability problem in a somewhat different setup. It uses algebraic trace theory as both the specification and implementation language, but solves the finite-state case in a similar automata-theoretic framework. However, the notion of *receptiveness* introduced in order to handle *speed-independence* between processes, is apparently weaker than our notion of *asynchrony*, since it does not consider the possibility of unnoticed input values. Another treatment of stuttering equivalence in the temporal framework can be found in [Bar87], where a set of temporal operators is introduced in order to handle relevant properties. In [ALW89] the notion of *realizability*, close to our notion of *implementability*, is introduced and investigated. An extension to the approach of [EC82] can be found in [AE89], where an efficient technique is proposed for synthesizing a system composed of many similar sequential processes.

We propose two directions for further research. Currently we are studying possible extensions of the results of this paper to the synthesis of more complex systems, especially systems composed of concurrent components communicating with each other according to a given architecture. Secondly, there is a need for strategies and techniques for the systematic development of *general* modules (specified by first-order formulae).

Acknowledgements

We gratefully acknowledge critical comments and helpful suggestions made by Moshe Vardi.

References

[AE89] P.C. Attie and E.A. Emerson, Synthesis of concurrent systems with many similar sequential processes, *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, 1989, pp. 191–201.

[ALW89] M. Abadi, L. Lamport, and P. Wolper, Realizable and unrealizable concurrent program specifications, *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, 1989. To appear in the LNCS-series, Springer.

[Bac80] J. Bacon, Substance and first-order quantification over individual-concepts, *J. Symb. Logic* **45**, 1980, pp. 193–203.

[Bar87] H. Barringer, The use of temporal logic in the compositional specification of concurrent systems, *Temporal Logics and Their Applications* (A. Galton, ed.), Academic Press, London, 1987, pp. 53–90.

[BKP84] H. Barringer, R. Kuiper, and A Pnueli, Now you may compose temporal logic specifications, *Proc. 16th ACM Symp. Theory of Comp.*, 1984, pp. 51–63.

[BKP86] H. Barringer, R. Kuiper, and A Pnueli, A really abstract concurrent model and its temporal logic, *Proc. 13th ACM Symp. Princ. of Prog. Lang.*, 1986, pp. 173–183.

[BL69] J.R. Büchi and L.H. Landweber, Solving sequential conditions by finite-state strategies, *Trans. Amer. Math. Soc.* **138**, 1969, pp. 295–311.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications, *ACM Trans. Prog. Lang. Sys.* **8**, 1986, pp. 244–263.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer, Alternation, *J. ACM* **28**, 1981, pp. 114–133.

[Con85] R.L. Constable, Constructive mathematics as a programming logic I: Some principles of theory, *Ann. Discrete Math.* **24**, 1985, pp. 21–38.

[Dil88] D.L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, Ph.D. thesis, CMU, 1988. Available as Technical Report CMU-CS-88-119.

[EC82] E.A. Emerson and E.M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, *Sci. Comp. Prog.* **2**, 1982, pp. 241–266.

[EH86] E.A. Emerson and J.Y. Halpern, 'Sometimes' and 'not never' revisited: On branching time versus linear time, *J. ACM* **33**, 1986, pp. 151–178.

[EJ88] E.A. Emerson and C.S. Jutla, The complexity of tree automata and logic of programs, *Proc. 29th IEEE Symp. Found. of Comp. Sci.*, 1988, pp. 328–337.

[EJ89] E.A. Emerson and C.S. Jutla, On simultaneously determinizing and complementing ω-automata, *Proc. 4th IEEE Symp. Logic in Comp. Sci.*, 1989. To appear.

[ES84] E.A. Emerson and A.P. Sistla, Deciding full branching time logic, *Inf. and Cont.* **61**, 1984, pp. 175–201.

[FL79] M.J. Fischer and R.E. Ladner, Propositional dynamic logic of regular programs, *J. Comp. Sys. Sci.* **18**, 1979, pp. 194–211.

[HR72] R. Hossley and C. Rackoff, The emptiness problem for automata on infinite trees, *Proc. 13th IEEE Symp. Switching and Automata Theory*, 1972, pp. 121–124.

[HT87] T. Hafer and W. Thomas, Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree, *Proc. 14th Int. Colloq. Aut. Lang. Prog.*, Lec. Notes in Comp. Sci. 267, Springer, 1987, pp. 269–279.

[Lam83] L. Lamport, What good is temporal logic, *Proc. IFIP Congress* (R.E.A. Mason, ed.), North-Holland, 1983, pp. 657–668.

[Lam86] L. Lamport, Personal Communication, 1986.

[MW80] Z. Manna and R.J. Waldinger, A deductive approach to program synthesis, *ACM Trans. Prog. Lang. Sys.* **2**, 1980, pp. 90–121.

[MW84] Z. Manna and P. Wolper, Synthesis of communicating processes from temporal logic specifications, *ACM Trans. Prog. Lang. Sys.* **6**, 1984, pp. 68–93.

[PR88] A. Pnueli and R. Rosner, A framework for the synthesis of reactive modules, *Proc. Intl. Conf. on Concurrency: Concurrency88* (F.H. Vogt, ed.), Lec. Notes in Comp. Sci. 335, Springer, 1988, pp. 4–17.

[PR89] A. Pnueli and R. Rosner, On the synthesis of a reactive module, *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, 1989, pp. 179–190.

[Rab72] M.O. Rabin, *Automata on Infinite Objects and Churc's Problem*, Volume 13 of *Regional Conference Series in Mathematics*, Amer. Math. Soc., 1972.

[Saf88] S. Safra, On the complexity of ω-automata, *Proc. 29th IEEE Symp. Found. of Comp. Sci.*, 1988, pp. 319–327. An extended version to appear in *J. Comp. Sys. Sci.*

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper, The complementation problem for Büchi automata with application to temporal logic, *Theor. Comp. Sci.* **49**, 1987, pp. 217–237.

[VS85] M.Y. Vardi and L.J. Stockmeyer, Improved upper and lower bounds for modal logics of programs, *Proc. 17th ACM Symp. Theory of Comp.*, 1985, pp. 240–251.

[VW86] M.Y. Vardi and P. Wolper, Automata theoretic techniques for modal logics of programs, *J. Comp. Sys. Sci.* **32**, 1986, pp. 183–221.