On the Synthesis of a 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 module with input x and output y, which is specified by the linear temporal formula $\varphi(x, y)$. We show that there exists a program satisfying φ iff the branching time formula $(\forall x)(\exists y)A\varphi(x, y)$ 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 for constructing the program whenever it exists. The algorithm is based on a new procedure for checking the emptiness of Rabin automata on infinite trees in time exponential in the number of pairs, but only polynomial in the number of states. This leads to a synthesis algorithm whose complexity is double exponential in the length of the given specification.

1 Introduction

An interesting and fruitful approach to the systematic construction of a program from its formal specification is based on the idea of *program synthesis* as a theorem proving activity. In this approach, a program with input x and output y, specified by the formula $\varphi(x, y)$, is constructed as a by-product of proving the theorem $(\forall x)(\exists y)\varphi(x, y)$. The specification $\varphi(x, y)$ characterizes the expected relation between the input x presented to the program and the output y computed by the program. For example, the specification for a root extracting program may be presented by the formula $|x - y^2| < \epsilon$.

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.

The AE-paradigm for the synthesis of sequential programs has been introduced in [WL69] (but see also [Elg61]), and has been the subject of extensive research [MW80,Con85] directed at extending the class of programs that can be synthesized, and the theories that may be used for the proofs, as well as strengthening the proof rules and the mechanisms for extracting the program from the proof.

The success of this approach to sequential programming should not be judged only by the number and complexity of programs that can be fully automatically derived, even though serious efforts are continuously invested in extending the range and capabilities of fully automatic synthesizers, in much the same way we keep improving the power of automatic theorem provers. The more important contribution of this ap-

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1989 ACM 0-89791-294-2/89/0001/0179 $1.50

proach is in providing a conceptual framework for the rigorous derivation of a program from its specification. Once this scheme is accepted, it can, in principle, be followed in a completely manual fashion, but encourages, on the other hand, the open ended development of a support system that will offer automatic support to increasingly larger parts of the procedure. Equally important is the realization of the identity between the processes of theorem proving and program construction. It has been recognized very early that every system for the formal development of programs must contain at least a powerful theorem prover as an important component. The approach of synthesis by theorem proving tells us that such a system need not contain much more than a theorem prover.

In view of the success of this approach for sequential programs, there is no wonder that several attempts have been made to extend it to concurrent programs. These attempts were held back for awhile by the question of what was the appropriate language to use for expressing the specification formula φ. While, for sequential programs, it is obvious that a properly enriched first order language is adequate, it took time to propose a similarly adequate specification language for concurrent programs. One of the more stable proposals is that of *temporal logic* ([Pnu77,GPSS80,MP82,SC85,Pnu86]). The basic supposition underlying temporal logic is that concurrent programs often implement *reactive systems* (see [HP85,Pnu85]) whose role is not to produce an output on termination, but rather to maintain an ongoing interaction with their environment. Therefore, the specification should describe the expected *behavior* of the system throughout its activity.

Indeed, the two main works on the synthesis of concurrent programs, which are reported in [CE81] and [MW84], consider a temporal specification φ, and show that if it is satisfiable, we can use the model that satisfies φ to construct a program that implements φ.

There are, however, some limitations of the approach, as represented in these two pioneering contributions, due to the fact that the approach is based on *satisfiability* of the formula expressing the specification $\varphi(x, y)$. The implied limitations are that the approach can in principle synthesize only *entire* or *closed* systems.

To see that, assume that the system to be constructed has two components, C_1 and C_2. Assume that only C_1 can modify x (a shared variable used for communication) and only C_2 can modify y. The fact that $\varphi(x, y)$ is satisfiable means that there exists at least one behavior, listing the running values of x and y, which satisfies $\varphi(x, y)$. This shows that there is a way for C_1 and C_2 to cooperate in order to achieve φ. The hidden assumption is that we have the power to construct both C_1 and C_2 in a way that will ensure the needed cooperation. If indeed we are constructing a closed system consisting solely of C_1 and C_2 and having no additional external interaction, this is quite satisfactory.

On the other hand, in a situation typical to an *open* system, C_1 represents the *environment* over which the implementor has no control, while C_2 is the body of the system itself, to which we may refer as a *reactive module*. Now the situation is no longer that of peaceful cooperation. Rather, the situation very much resembles a two-person game. The module C_2 does its best, by manipulating y, to maintain $\varphi(x, y)$, despite all the possible x values the environment keeps feeding it. The environment, represented by C_1, does its worst to foil the attempts of C_2. Of course, this anthropomorphism should not be taken too literally. The main point is that we have to show that C_2 has a winning strategy for y against all possible x scenarios the environment may present to it.

It seems that the natural way to express the existence of a winning strategy for C_2, is again expressed by the AE-formula $(\forall x)(\exists y)\varphi(x, y)$. The only difference is that now we should interpret it over temporal logic, where x and y are no longer simple variables, but rather sequences of values assumed by the variables x and y over the computation. In contrast, we may describe the approach presented in [MW84] and [CE81] as based on the formula $(\exists x)(\exists y)\varphi(x, y)$.

This is indeed the main claim of this paper. Namely, that the theorem proving approach to the synthesis of a reactive module should be based on proving the validity of an AE-formula. As we will show below, the precise form of the formula claiming the existence of a program satisfying the linear time temporal formula $\varphi(x, y)$, is $(\forall x)(\exists y)A\varphi(x, y)$, where A is the "for all paths" quantifier of branching time logic. Thus, even though the specification $\varphi(x, y)$ is given in linear logic, which is generally considered adequate for reactive specifications, the synthesis problem has to be solved in a branching framework. This conclusion applies to the synthesis of both synchronous and asynchronous programs, yet for simplicity of presentation, we prefer to restrict the exposition in this paper to the synthesis of synchronous programs. The application of our approach to the synthesis of asynchronous programs will be presented in a subsequent paper.

An interesting observation is that the explicit quantification over the *dynamic* (i.e., variables that may change their values over the computation) interface variables, x and y, is not absolutely necessary. As

we will show in the paper, there exists an equivalent branching time formula, which quantifies only over *static* variables (i.e., variables which remain constant over the computation), whose *satisfiability* guarantees the existence of a program for $\varphi(x, y)$. For the case of finite state programs, this other formula becomes purely propositional, and its satisfiability, therefore, can be resolved by known decision methods for satisfiability of propositional branching time formulae. However, for the more general case that deductive techniques have to be applied, we prefer to establish validity, rather than satisfiability, in particular since the explicitly quantified version emphasizes the asymmetry between the roles of the variables x and y in the program.

We justify our main claim on two levels. First we consider the general case and show that the synthesis formula is valid iff there exists a *strategy tree* for the process controlling y. We then argue that such a strategy tree represents a program by specifying an appropriate y for each history of x values. On this level we pay no attention to the question of how effective this representation of a program is, which becomes relevant when we wish to obtain a program represented in a conventional programming language.

Hence, in a following section we consider the more restricted case in which the specification refers only to *Boolean variables*. In this case the validity of the synthesis formula is *decidable*, and we present an algorithm for checking its validity and extracting a finite-state program out of a valid synthesis formula.

A related investigation of synthesis for the finite state case, based on a similar approach, has been carried out in [BL69]. The question, formulated for the first time in [Chu63], has been asked in an automata-theoretic framework, where the specification $\varphi(x, y)$ is given by an ω-automaton accepting a combined x,y-behavior, and the extracted automaton is an ω-transducer. The solution presented in [BL69] uses game-theoretic ideas, and it is of very high computational complexity. Later, [HR72] and [Rab72] have observed, similar to us, that even though the specification is expressed by automata on strings (corresponding to linear temporal logic), its synthesis must be carried out by using tree automata. In our own approach we had to use a similar algorithm for checking emptiness of ω-tree automata. The previously best known complexity of this problem has been nondeterministic polynomial time in the overall size of the automata ([VS85,Eme85]). Another important result of our paper is a derivation of a better emptiness checking algorithm, whose complexity is deterministic polynomial time in the number of states and exponential in the number of pairs in the acceptance condition of the automata (a different algorithm yielding similar complexity has been recently reported in [EJ88]). Using this improved algorithm, the complete synthesis process can be performed in deterministic time which is doubly exponential in the size of the specification.

The papaer is organized as follows. The second section introduces a general temporal logic. The third section presents the implementability problem, while the fourth and the fifth sections suggest a temporal framework for the development of reactive modules, for the general (first-order) and for the finite-state cases, respectively. Examples of the development of finite-state modules are exhibited in the sixth section. The seventh and the eighth sections are concerned with automata and formal languages in general, and with the emptiness problem of automata on infinite-trees in particular.

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 $\neg$, $\vee$, $\exists$ 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 every $k \geq 0$, 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 the (only) 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 $\neg p$, $(p \vee q)$ and $(\exists \alpha)p$ where α is any variable.
3. If p is a path formula then Ep is a state formula.

Path formulae are the (only) formulae defined by the following rules:

1. Every state formula is a path formula.
2. 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 $\neg$T, $p \wedge q$ for $\neg(\neg p \vee \neg q)$, $p \rightarrow q$ for $\neg p \vee q$, $p \equiv q$ for $(p \rightarrow q) \wedge (q \rightarrow p)$, $(\forall \alpha)p$ for $\neg(\exists \alpha)(\neg p)$, $\Diamond p$ for $\mathrm{T} \mathcal{U} p$, $\Box p$ for $\neg \Diamond \neg p$, $p \mathcal{U} q$ for $p \mathcal{U} q \vee \Box q$ and Ap 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 $\mathcal{M} = \langle D, I, M \rangle$. D is some non-empty data *domain*, I is a *(static) interpretation* of all static variables into appropriate functions and predicates over D, 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. L is the *labeling function*, assigning to each state $s \in S$ a *(dynamic) interpretation* $L(s)$ of all dynamic variables into appropriate functions and predicates over D. 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 jth *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 a 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 $\mathcal{M} = \langle D, I, M \rangle$, is called a *tree-model*, iff the structure M is a tree-structure.

In order to interpret applications of static functions and predicates to terms, and applications of existential quantifiers over static variables to formulae, we use standard definitions, over the static interpretation I and the semantics of the given terms or formulae. Analogously, applications of dynamic functions are interpreted with respect to dynamic interpretations.

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

1. $\langle \mathcal{M}, s \rangle \models u(t_1, \ldots, t_k)$ iff $L(s)(u)(d_1, \ldots, d_k) = true$, 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.
2. $\langle \mathcal{M}, s \rangle \models \neg p$ iff not $\langle \mathcal{M}, s \rangle \models p$.
3. $\langle \mathcal{M}, s \rangle \models p \vee q$ iff $\langle \mathcal{M}, s \rangle \models p$ or $\langle \mathcal{M}, s \rangle \models q$.
4. For a dynamic variable α, $\langle \mathcal{M}, s \rangle \models (\exists \alpha)p$ iff $\langle \mathcal{M}', s \rangle \models p$ for some $\mathcal{M}'$ which equals $\mathcal{M}$ except that for any $t \in S$, $L'(t)(\alpha)$ may be different than $L(t)(\alpha)$.
5. 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:

1. For $\pi = (s_0, \ldots)$ and a state formula p, $\langle \mathcal{M}, \pi \rangle \models p$ iff $\langle \mathcal{M}, s_0 \rangle \models p$.
2. $\langle \mathcal{M}, \pi \rangle \models \neg p$ iff not $\langle \mathcal{M}, \pi \rangle \models p$.
3. $\langle \mathcal{M}, \pi \rangle \models p \vee q$ iff $\langle \mathcal{M}, \pi \rangle \models p$ or $\langle \mathcal{M}, \pi \rangle \models q$.
4. $\langle \mathcal{M}, \pi \rangle \models \bigcirc p$ iff $\langle \mathcal{M}, \pi^{(1)} \rangle \models p$.
5. $\langle \mathcal{M}, \pi \rangle \models p \mathcal{U} q$ 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) *synchronous program* P from D to D to be a function $f_P : D^+ \rightarrow D$, 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 assumed by x over steps $0, 1, \ldots, i$.

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

$$\sigma : \langle a_0, b_0 \rangle, \langle a_1, b_1 \rangle, \ldots$$

such that, for every $i \geq 0$, $a_i, b_i \in D$, and $b_i = f_P(a_0, a_1, \ldots, a_i)$. A program P *satisfies* a linear temporal logic specification $\varphi(x, y)$, written $P \models \varphi(x, y)$, iff every behavior σ of P satisfies $\sigma \models \varphi(x, y)$. We may now consider the following problem:

Problem 1 (Implementability) *Given a linear specification $\varphi(x, y)$, does there exists a program P which satisfies $\varphi(x, y)$?*

Specifications for which such a program exists are called *implementable*, and the program satisfying them is called an *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 [CE81] 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

$$\varphi : \bigcirc x.$$

This formula states that the next input value (the one presented at step 1), 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

$$\langle a_0, b_0 \rangle, \langle \mathrm{F}, b_1 \rangle, \ldots$$

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. Somehow, we have to introduce this restriction of the past-dependency of y on x, into the implementability formula. As we will show next, this can be done if we extend our logic framework to *branching time* temporal logic.

4 Development of Reactive Modules

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. By our restriction to synchronous systems, each step in a computation starts by the environment setting a value to x, and the module reading the value and responding by setting a value to y. Thus, a typical behavior of such a system is a sequence $\langle a_0, b_0\rangle, \langle a_1, b_1\rangle, \ldots$ where $a_1, a_2, \ldots$, are the values given to x by the environment, and $b_1, b_2, \ldots$, are the responses of the module. Without loss of generality, we may assume that a_0, b_0 are fixed, and for every $i > 0$, b_i may depend on $a_1, \ldots, a_{i-1}, a_i$.

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. We consider x and y to be just single variables for simplicity — 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 our fixed initial condition, namely $x = a_0$ and $y = b_0$.

Let D be some fixed data domain and I some fixed interpretation over D. Since D and I are fixed, we shall identify models $\mathcal{M}$ of the form (D, I, M) with their non-fixed elements, i.e., with the dynamic structure M, and write 'the model M', referring to the whole model $\mathcal{M} = (D, I, M)$. For definitions and notations for some of the constructs used below, see the section on formal languages. A *full-x-tree* is a model $M = (S, R, L)$, where $S = D^*$, $(z_1, z_2) \in R \iff z_2 = z_1 a$ for some $a \in D$, and $L : S \times \{x, y\} \rightarrow D$ satisfies $\langle L(\varepsilon, x), L(\varepsilon, y)\rangle = \langle a_0, b_0\rangle$, and $L(za, x) = a$. Thus, a full-x-tree is based on a structure whose states are named after *strings* of the elements of D. For example, if $D = \Sigma_2 = \{0, 1\}$ (a Boolean domain), then the states are strings over Σ_2, i.e., elements of $(0 + 1)^*$. In general, a state $z = a_1, \ldots, a_k$, for $a_i \in D$, $1 \leq i \leq k$, has succesors zb for each $b \in D$ ($z0$ and $z1$ for the Boolean domain). We further require that L assigns the fixed values a_0 to x and b_0 to y at the root, and the value $a_k \in D$ to x in the state $z = a_1, \ldots, a_k$. The intuition behind a full-x-tree is that it should contain all the possible sequences of x values.

Lemma 1 *The formula $\psi = (\forall x)(\exists y)A\varphi$ is valid over all tree-models* **iff** *$A\varphi$ holds over some full-x-tree M.*

Proof: The 'only if' direction is obvious from the definition of validity. For the 'if' direction, assume $M = (D^*, R, L) \models A\varphi$, and let $\hat{M} = (\hat{S}, \hat{R}, \hat{L})$ be any other tree-model. Define $M' = (\hat{S}, \hat{R}, L')$, with $L'(s, z) = \hat{L}(s, z)$ for every dynamic variables $z \neq y$. Let $\hat{L}(\varepsilon, x) = \hat{a}_0$. For every $s \in \hat{S}$, let $(r, s_1, \ldots, s_k = s)$ be the unique path in $\hat{M}$ (and in M') leading from the root r to s, and let $\hat{a}_0, a_1, \ldots, a_k \in D^+$ be the sequence of values assigned to x on this path by $\hat{L}$. Consider the state $z = a_1, \ldots, a_k \in D^*$ in the structure M, and define $L'(s, y) = L(z, y)$. Since $M \models A\varphi$, it follows that $M' \models A\varphi$, and hence $\hat{M} \models (\exists y)A\varphi$. But this applies to every such $\hat{M}$, hence ψ is valid over all tree-models. ∎

Given a full-x-tree M, we can interpret it as a program P^M, represented by the function f_{P^M}, such that

$$f_{P^M}(a_0, a_1, \ldots, a_i) = L(a_0, a_1, \ldots, a_i, y).$$

The proof of the following lemma is immediate from the definitions and from lemma 1.

Lemma 2 $P^M \models \varphi$ **iff** $M \models A\varphi$. ∎

We conclude the general case by the following theorem.

Theorem 1 (Implementability) *The following conditions are equivalent:*

1. *The specification $\varphi(x, y)$ is implementable.*
2. *The formula $(\forall x)(\exists y)A\varphi(x, y)$ is valid over all tree-models.*
3. *The formula $A\varphi(x, y)$ holds over some full-x-tree.*
4. *The formula $A\varphi(x, y) \wedge A\Box(\forall a)E\bigcirc(x = a)$ is satisfiable.*

Proof: The equivalence of 1–3 is straightforward. To relate clause 4, we observe that the conjunct $A\Box(\forall a)E\bigcirc(x = a)$ ensures that every node n in the constructed model, has descendants n_a with $L(n_a, x) = a$ for every $a \in D$. Thus, a model satisfying this conjunct, has a full-x-tree 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. Full-x-trees become full binary trees, over the set of states $S = \Sigma_2^*$, where $\Sigma_2 = \{0, 1\}$. Note that a more general finite domain can always be reduced to several Boolean domains.

Therefore, what we are looking for is a full binary tree with assignments $L(s, x)$ and $L(s, y)$ to each state s, such that the formula $\varphi(x, y)$ holds on each infinite rooted path. Since the assignment of x is already determined in a full-x-tree (left son always gets $x = \mathrm{F}$,

and right son gets $x = \mathrm{T}$), the only element left open is the assignment $L(s,y)$.

We approach this problem by constructing a finite state tree automaton A over infinite binary trees, whose nodes are labeled by the assignment $L(s,y)$. The automaton A will accept a labeled tree iff the y-assignment is such that $\varphi(x,y)$ holds on all rooted paths in the tree, when we take for $L(s,x)$ the assignment implied by the directions in the tree. The construction uses techniques similar to those used in [ES84,VW86]. The validity of ψ over tree-models is easily reduced to non-emptiness of $T_\omega(A)$, the set of trees accepted by A. We remind the reader that what the automaton accepts is a set of *labelings* for the nodes of the tree. We then use the technique described in the proof of proposition 3 to check the emptiness of $T_\omega(A)$

As is well known, [HR72,Rab72], and will also be reestablished in our proof of proposition 2, a tree automaton A accepts some infinite labeling iff it accepts some *regular* labeling. A regular labeling is one that can be generated by a deterministic (string) ω-*transducer*. The transducer is fed a sequence of directions, encoded by 0, 1, defining a path in the tree, and outputs at each visited node the corresponding label (from Σ_2 in our case). Given such a transducer C, we can immediately interpret it as a program $P = P^C$. The program reads an input history $a_1, \ldots, a_k$, from the variable x, and sets the variable y, at each step, to the value emitted by the transducer at this step.

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 2 (Synthesis) *There is a deterministic algorithm, such that given a propositional formula $\psi = (\forall x)(\exists y)A\varphi$ as above, checks whether the specification φ is implementable. If it is implementable, the algorithm constructs a deterministic transducer C, such that $P^C \models \varphi$. The running time of the algorithm and the size of the transducer C, are both at most double-exponential in the length of φ.*

Proof: Given a linear time formula φ as above, with $|\varphi| = n$, we first construct a Büchi automaton A on infinite strings over $\Sigma_2 \times \Sigma_2$, such that $\sigma = \langle a_i, b_i\rangle_{1 \le i < \omega} \in L_\omega(A)$ iff the linear model $\mathcal{M} = \langle S, L\rangle$, where $S = (s_0, s_1, \ldots)$, $L(s_i, x) = a_i$ and $L(s_i, y) = b_i$ for $0 \le i < \omega$ (recall that a_0 and b_0 are fixed), satisfies φ. This standard construction (see [VW86,ES84]) yields in general, a non-deterministic automaton of size $|A| = 2^{c_0 n}$, for some constant c_0. Secondly, we use the determinization procedure of [Saf88], to construct a deterministic Rabin automaton B equivalent to A, with $2^{2^{c_1 n}}$ states and $2^{c_0 n}$ pairs, for some constant $c_1 > c_0$. Both A and B have the property that for each state p, there exists an unique letter b_p, such that for every transition $p \in \delta(q, \langle a, b\rangle)$ leading to p, necessarily $b = b_p$. For the single initial state q_0, put $b_{q_0} = b_0$. This property enables us to interpret B as a tree-automaton B' on infinite trees over Σ_2, where a transition $p \in \delta(q, \langle a, b_p\rangle)$ of B corresponds to a transition $p \in \delta'(q, b_q) \Downarrow a$ of B'. Obviously, all transitions leaving the single initial state q_0, generate the constant letter b_0. The third step consists of applying the emptiness checking algorithm of proposition 3 to B'.

Theorem 1 supports our main claim, which states that φ is implementable iff $T_\omega(B') \neq \emptyset$, and if implementable, the algorithm of proposition 3 also constructs the required deterministic-transducer C. Time complexity of the whole synthesis procedure and the size of C, both equal $O(2^{2^{cn}})$, for some constant $c > c_0 + c_1$. ∎

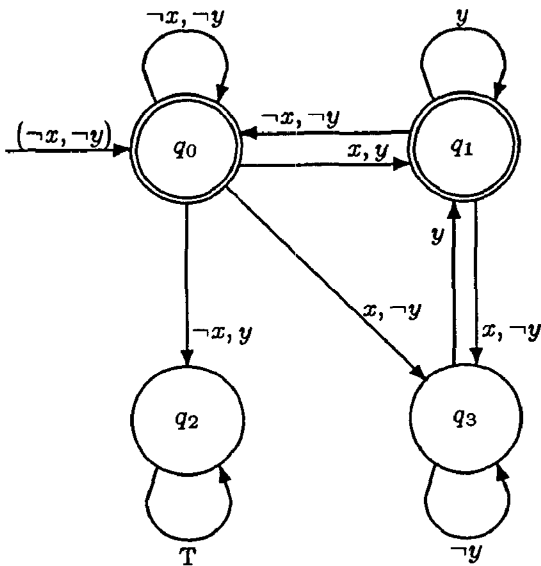
6 Examples of Modules Development

We consider two examples which demonstrate the development of reactive, finite-state modules. For each example, we present a formal specification, and several sketches of appropriate automata, representing significant steps along the synthesis process.

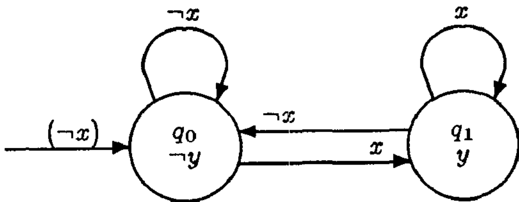
Example 1 (Responder) *Synthesis of a consistent responder, specified by $(\forall x)(\exists y)A\varphi_1$, where*

$$\varphi_1 : \quad \neg x \rightarrow \begin{pmatrix} \neg y \\ \wedge \\ \Box(x \rightarrow \Diamond y) \\ \wedge \\ \Box(\neg y \rightarrow (\neg y) U x) \end{pmatrix}.$$

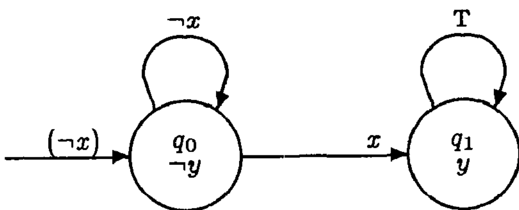
A deterministic and complete Büchi ω-automaton for φ_1:



We present two solutions. The first ω-transducer corresponds to the strategy: *Keep y equal to x forever.*



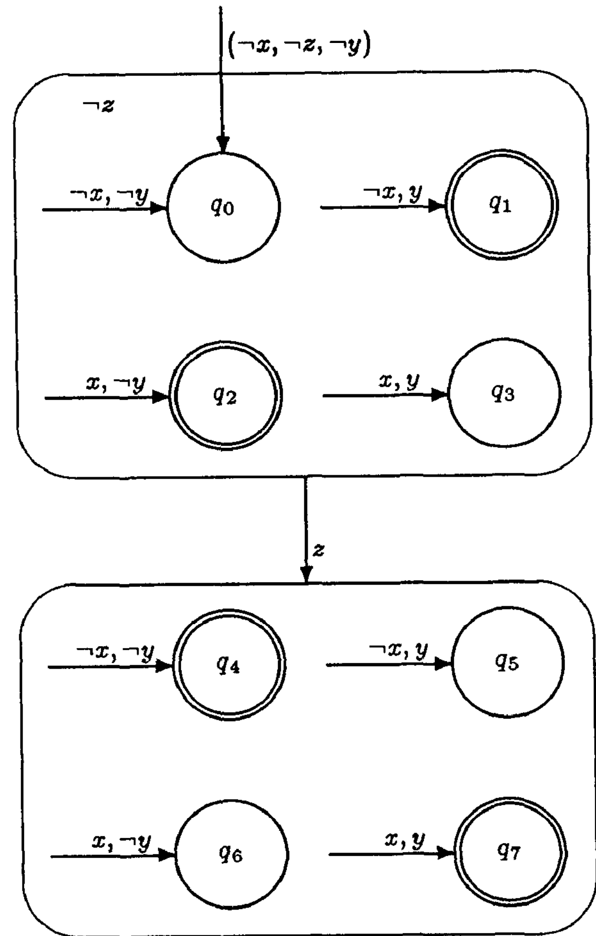
The second ω-transducer corresponds to the strategy: *Once x gets true — keep y true forever.*



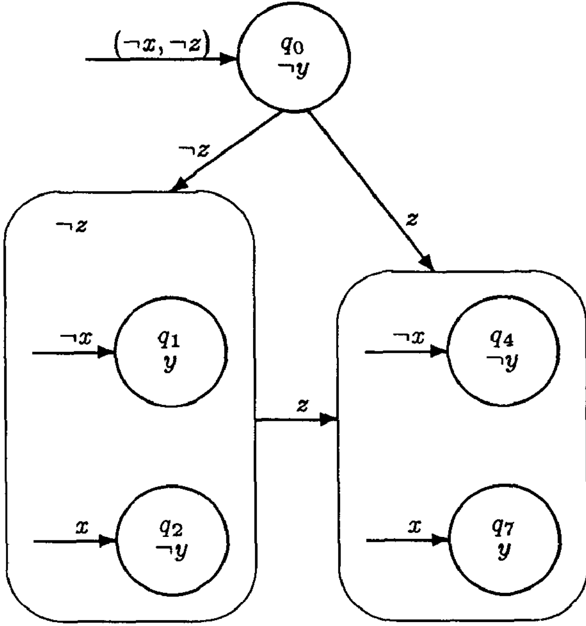
Example 2 (Yet Another Responder)
Synthesis of a more complicated responder, specified by $(\forall x, z)(\exists y)A\varphi_2$, where

$$\varphi_2 : \quad (\neg x \wedge \neg z) \rightarrow \left(\begin{array}{c} \neg y \\ \wedge \\ \Diamond z \rightarrow \Diamond\Box(x \equiv y) \\ \wedge \\ \Box(\neg z) \rightarrow \Diamond\Box(x \not\equiv y) \end{array} \right).$$

The automata in this example are given in a compact, visual presentation, similar to the *statecharts* of [Har87]. Following, is a deterministic and complete Rabin ω-automaton for φ_2, with the acceptance condition $\Omega = \{\langle L_1, U_1\rangle\}$, $L_1 = \{q_0, q_3, q_5, q_6\}$ and $U_1 = \{q_1, q_2, q_4, q_7\}$.



The strategy corresponding to the following ω-transducer for φ_2 says: *Keep y different than x as long as z is false. Once z gets true - keep y equal to x forever.*



7 Formal Languages and Finite Automata

An *alphabet* Σ is a non-empty set of *letters*. In particular, if $n > 0$, $\Sigma_n = \{0, \ldots, n-1\}$. As usual, ε denotes the *empty string*, Σ^* denotes the set of *finite strings over* Σ, $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ and Σ^ω is the set of *infinite (omega) strings over* Σ. For all strings $x, y \in \Sigma^\dagger = \Sigma^* \cup \Sigma^\omega$, $0 \leq |x| \leq \omega$ denotes the *length of* x, for $0 \leq j \leq k < |x|$, $x[k] \in \Sigma$ denotes the kth letter of x, $x[j,k] = \{x[j], \ldots, x[k]\}$ and xy is the *concatenation* of x and y (if $|x| = \omega$ then $xy = x$). If $x \in \Sigma^\omega$ we denote by $Inf(x)$ (the *infinity set of* x), the set of all letters $a \in \Sigma$ such that $x[i] = a$ for infinitely many i's. We define two partial orderings $\leq$ and $\preceq$ on $\Sigma^\dagger$ as follows: for $x, y \in \Sigma^\dagger$, $x \leq y$ (x is a *prefix of* y) iff $y = xz$ for some $z \in \Sigma^\dagger$ and $x \preceq y$ (x is a *suffix of* y) iff $y = zx$ for some $z \in \Sigma^*$.

Languages (resp. *finitary languages*, ω*-languages*) over Σ are just subsets of $\Sigma^\dagger$ (resp. Σ^*, Σ^ω). For a tuple $a = \langle a_0, \ldots, a_{k-1}\rangle$ and integer l, $0 \leq l < k$, define the *l-projection of* a to be $a \Downarrow l = a_l$. The usual operations on languages include the Boolean operations, the Cartesian product '$\times$', and the natural extensions over sets of projection '$\Downarrow$', concatenation, and different types of iteration: '^{n}', '*', '$^+$', '$^\omega$' and '†'.

A *finite automaton* is a 5-tuple $A = \langle \Sigma, Q, \delta, Q_0, F\rangle$ where Σ is a finite alphabet, Q is a finite, nonempty set of *states*, $\delta : Q \times \Sigma \to 2^Q$ is the *transition function*, $Q_0 \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. A is said to be *deterministic* iff $|Q_0| = 1$ and for all $q \in Q$ and $a \in \Sigma$, $|\delta(q,a)| \leq 1$. A *run* of A on a string $x \in \Sigma^\dagger$ is a string $r \in Q^{1+|x|}$ such that $r[0] \in Q_0$ and for all i, $0 \leq i < |x|$, $r[i+1] \in \delta(r[i], x[i])$. A run r on x is *accepting* iff either x is finite and $r[|x|] \in F$ or x is infinite and $Inf(r) \cap F \neq \emptyset$ (*Büchi acceptance*). A string x is accepted by A iff there is some accepting run of A on x, and the set of all finite (resp. ω) strings accepted by A is called the *finitary* (resp. ω) *language* of A, and denoted by $L_f(A)$ (resp. $L_\omega(A)$).

A *k-ary tree*, is a subset T of $\Sigma_k^\dagger$ which is either infinite and equals Σ_k^*, or is finite and satisfies

1. T is *prefix-closed*: $x \in T$ and $y \leq x$ imply $y \in T$.
2. T is *frontiered*: $xl \in T$ for some $l \in \Sigma_k$ implies $xl \in T$ for all $l \in \Sigma_k$ (the set of *leaves*, i.e. maximal elements of T, is called the *frontier of* T and denoted by $Ft(T)$).

If $x \in T$ then the *subtree of* T *rooted at* x is the set $T_x = \{y \in T \mid x \leq y\}$. In particular $T_\varepsilon = T$. A *path* π *in* T_x is a subset of T_x such that $x \in \pi$ and for all $y \in \pi - Ft(T_x)$, $yl \in \pi$ for precisely one $l \in \Sigma_k$. A Σ*-tree* is a pair (v, T) where T is a tree and $v : T - Ft(T) \to \Sigma$ assigns to each *node* of T a *value* from Σ. *Tree languages* and Σ*-tree languages* are defined analogously to string languages. In the following, we will mainly speak of binary Σ-trees, where the alphabet Σ is clear from the context (or simply any fixed alphabet), in which case we omit explicit indications to the type of trees.

A *finite tree-automaton* (*t.a.* for short) is a 5-tuple $A = \langle \Sigma, Q, \delta, Q_0, \Omega\rangle$ where Σ is a finite alphabet, Q is a finite, nonempty set of *states*, and $Q_0 \subseteq Q$ is the set of *initial states*. $\delta : Q \times \Sigma \to 2^Q \times 2^Q$ is the *transition function* assigning to each state q and letter $a \in \Sigma$ a pair of sets of states S_0 and S_1 called the sets of 0 and 1 (resp.) *successors* of q (upon generating a). Ω is the *acceptance condition*, where for an automaton on finite trees $\Omega \subseteq Q$, while for an automaton on infinite trees $\Omega \subseteq 2^{Q \times Q}$ (*Rabin's acceptance condition*). One may observe that the original formulation of t.a. (on infinite trees) in [Rab69] defines the transition function as $\delta : Q \times \Sigma \to 2^{Q \times Q}$. However, our definition seems to be more convenient and is polynomially equivalent to the original one (see [GH82]). A *run* of A on a Σ-tree (v, T) is a Q-tree $r = (w, T \cup (T\Sigma_2))$ such that $w(\varepsilon) \in Q_0$ and for all $x \in T$ and $a \in \Sigma_2$, $w(xa) \in \delta(w(x), v(x)) \Downarrow a$. A run $r = (w, T')$ on (v, T) is *accepting* iff either T is finite (corresponding to $\Omega \subseteq Q$) and for each leaf $x \in Ft(T')$, $w(x) \in \Omega$, or $T = \Sigma_2^*$ is infinite (corresponding to $\Omega \subseteq 2^{Q \times Q}$)

and for each path $\pi \in T'$, there is a pair $(L, U) \in \Omega$ such that $Inf(w(\pi)) \cap U \neq \emptyset$ and $Inf(w(\pi)) \cap L = \emptyset$ (*Rabin acceptance*). A tree (v, T) is accepted by A iff there is some accepting run of A on (v, T), and the set of all finite (resp. infinite) trees accepted by A is called the *finitary* (resp. *infinite*) *tree language* of A, and denoted by $T_f(A)$ (resp. $T_\omega(A)$).

A t.a. $A = \langle \Sigma, Q, \delta, Q_0, \Omega \rangle$ is said to be *deterministic* iff $|Q_0| = 1$ and for all $q \in Q$, $a \in \Sigma$ and $b \in \Sigma_2$, $|\delta(q, a) \Downarrow b| \leq 1$. A is called a *state-machine* (or alternatively an *ω-transducer*) iff its acceptance condition is degenerated, i.e. $\Omega = Q$ for automata on finite trees, or $\Omega = \{(Q, \emptyset)\}$ for automata on infinite trees. A transducer $B = \langle \Sigma Q, \delta, \{q_0\} \rangle$ is said to be deterministic iff for each state $q \in Q$, there is a unique $a \in \Sigma$ such that $|\delta(q, a) \Downarrow 0| = |\delta(q, a) \Downarrow 1| = 1$, but $|\delta(q, b) \Downarrow 0| = |\delta(q, b) \Downarrow 1| = 0$ for all $b \neq a$. This definition implies that $|T_\omega(B)| = 1$, which clearly characterizes any deterministic transducer A with $T_\omega(A) = \{t\}$ as a finite and effective representation of the single tree t.

8 Emptiness of Automata on Infinite Tress

The following proposition is a collection of well known results.

Proposition 1 *Given a finite automaton A and a t.a. B on finite trees, the following emptiness problems are decidable:*

1. $L_f(A) \stackrel{?}{=} \emptyset$ *(in logarithmic space).*
2. $L_\omega(A) \stackrel{?}{=} \emptyset$ *(in logarithmic space).*
3. $T_f(B) \stackrel{?}{=} \emptyset$ *(in cubic time).* ∎

Decidability of the emptiness problem for automata on infinite trees is discussed in [Rab69,Rab72, HR72,GH82,MS85,Eme85,VS85,Tho]. Let $A = \langle Q, \delta, Q_0, \Omega \rangle$ be a t.a. on infinite trees with $|Q| = n$ and $|\Omega| = m$. The complexities of the algorithms given in the above papers vary from non-elementary time in the original [Rab69] paper, deterministic exponential time(n) in [Rab72] and deterministic double-exponential time(n) in [HR72] to nondeterministic polynomial time(n) plus deterministic polynomial time(m) in [VS85,Eme85]. We shall present here a deterministic algorithm which solves (constructively) the emptiness problem of automata on infinite trees and runs in polynomial time($(nm)^m$). This complexity turns out to be significant for the class of t.a.'s for which m, the number of pairs in the acceptance condition, is much smaller (polylogarithmic) than n — the number of states. Another algorithm which yields similar complexity, has been recently reported in [EJ88], and is used there to achieve better upper bounds on the complexities of some logics of programs.

The following proposition and its proof essentially consist of delicate modifications of the result reported in [HR72] as theorem 1 and its proof.

Proposition 2 (Non-Emptiness Condition) *Let $A = \langle Q, \delta, Q_0, \Omega \rangle$ be a t.a. with $\Omega = \{\langle L_i, U_i \rangle \mid 1 \leq i \leq m\}$ and $|Q| = n$. Then, $T_\omega(A) \neq \emptyset$ if and only if for some finite tree E and a run $r = (v, E')$ of A on E, for every path π in E, there exist an integer i, $1 \leq i \leq m$, and strings $x_0, x_1, \ldots, x_{n+1}$, such that the following hold:*

1. $x_0 \leq x_1 < \ldots < x_n < x_{n+1} \in \pi \cap Ft(E)$.
2. *For all l, $0 < l \leq n+1$, $v(x_l) \in U_i$.*
3. *For all j, $1 \leq j \leq m$, $v[x_1, x_{n+1}] \cap L_j \neq \emptyset \Rightarrow v[x_0, x_1] \cap L_j \neq \emptyset$.*
4. $v[x_0, x_{n+1}] \cap L_i = \emptyset$.

Proposition 3 (Non-Emptiness Algorithm) *The emptiness problem for automata on infinite trees, is decidable in deterministic time $O((nm)^{cm})$ for some constant c, where n is the number of states and m is the number of pairs.*

Proof: Given a t.a. A as in proposition 2, we construct a *deterministic* t.a. B on *finite* trees, which accepts some finite tree E satisfying the conditions of proposition 2, if such a tree exists. B is the cross-product of A and a finite automaton C which recognizes the leaves of E.
A state of C should record, for every i, $1 \leq i \leq m$, whether for some string x and integer l', there exist strings $x_0 \leq x_1 < \ldots < x_{l'} \leq x$, satisfying the above mentioned conditions. The state is final if $l' = n+1$ and $x_{l'} = x$, for some i. It can be shown that sufficient information for such a state (and for determining the appropriate transition function) can be coded into a vector of at most $2m$ counters of size $log(m) + log(n)$ each, and m pointers into this vector. A counter (i, l) records that a potential x_l has been encountered with respect to i (and with respect to the internal order of the vector). Also, if (i, l) is added to the vector, all previous (i, l') with $0 < l' < l$ can be eliminated from it. The pointers record for each $j \leq m$, which is the last segment (with respect to the vector) in which some state of L_j has occurred. Acceptance is announced when some counter approaches $(i, n+1)$. Clearly, the size of B is at most $(nm)^{3m}$, hence by

proposition 1, emptiness can be tested in deterministic time $O((nm)^{cm})$ for some constant $c \leq 9$.

If $T_f(B) \neq \emptyset$, we may apply some simple modifications to B, which do not change the complexity of the algorithm, yielding a *deterministic transducer* D with $T_\omega(D) = \{t\}$ for some regular $t \in T_\omega(A)$. Moreover, $|D| \leq |B|$, that is, $|D| = O((nm)^{3m})$. ∎

9 Directions for Further Research

We propose two directions for extending the results of this paper. We are currently developing a similar framework for the specification and synthesis of *asynchronous* systems. Secondly, there is need for strategies and techniques for the systematic development of *general* modules (specified by first-order formulae).

References

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

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

[CE81] E.M. Clarke and E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, *Proc. IBM Workshop on Logics of Programs*, Lec. Notes in Comp. Sci. 131, Springer, 1981, pp. 52–71.

[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. Syst.* **8**, 1986, pp. 244–263.

[Chu63] A. Church, Logic, arithmetic and automata, *Proc. 1962 Int. Congr. Math.*, Upsala, 1963, pp. 23–25.

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

[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. Comp. Sci.*, 1988.

[Elg61] C.C. Elgot, Decision problems of finite automata design and related arithmetics, *Trans. Amer. Math. Soc.* **98**, 1961, pp. 21–52.

[Eme85] E.A. Emerson, Automata, tableaux and temporal logics, *Proc. Conf. Logic of Programs*, Lec. Notes in Comp. Sci. 193, Springer, 1985, pp. 79–88.

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

[GH82] Y. Gurevich and L.A. Harrington, Automata, trees and games, *Proc. 14th ACM Symp. Theory of Computing*, 1982, pp. 60–65.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, On the temporal analysis of fairness, *Proc. 7th ACM Symp. Princ. Prog. Lang.*, 1980, pp. 163–173.

[Har87] D. Harel, Statecharts: a visual formalism for complex syatems, *Sci. Comp. Prog.* **8**, 1987, pp. 231–274.

[HP85] D. Harel and A. Pnueli, On the development of reactive systems, *Logics and Models of Concurrent Systems*, Springer, 1985, pp. 477–498.

[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.

[MP82] Z. Manna and A. Pnueli, Verification of concurrent programs: the temporal framework, *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), Academic Press, London, 1982, pp. 215–273.

[MS85] D.E. Muller and P.E. Schupp, Alternating automata on infinite objects, determinancy and rabin's theorem, *Automata on Infinite Words*, Lec. Notes in Comp. Sci. 192, Springer, 1985, pp. 100–107.

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

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

[Pnu77] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Symp. Found. Comp. Sci.*, 1977, pp. 46–57.

[Pnu85] A. Pnueli, In transition from global to modular temporal reasoning about programs, *Logics and Models of Concurrent Systems*, Springer, 1985, pp. 123–144.

[Pnu86] A. Pnueli, Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends, *Current Trends in Concurrency*, Lec. Notes in Comp. Sci. 224, Springer, 1986, pp. 510–584.

[Rab69] M.O. Rabin, Decidability of second order theories and automata on infinite trees, *Trans. Amer. Math. Soc.* **141**, 1969, pp. 1–35.

[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. Comp. Sci.*, 1988.

[SC85] A.P. Sistla and E.M. Clarke, The complexity of propositional linear time logics, *J. ACM* **32**, 1985, pp. 733–749.

[Tho] W. Thomas, Automata on infinite objects, *Handbook of Theoretical Computer Science*, North-Holland. To appear.

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

[WL69] R.J. Waldinger and R.C.T. Lee, PROW: a step towrds automatic program writing, *Proc. First Int. Joint Conf. on Artificial Inteligence*, 1969, pp. 241–252.