Distributed Reactive Systems are Hard to Synthesize*

Amir Pnueli and Roni Rosner†
The Weizmann Institute of Science‡

Abstract

This paper considers the problem of synthesizing a finite-state *distributed* reactive system. Given a distributed architecture $\mathcal{A}$, identifying several processors $P_1, \ldots, P_k$, and their interconnection scheme, and a propositional temporal specification φ, a solution to the synthesis problem consists of finite-state programs $\Pi_1, \ldots, \Pi_k$ (one for each processor), whose joint (synchronous) behavior maintains φ against all possible inputs from the environment. We refer to such a solution as the *realization* of the specification φ over the architecture $\mathcal{A}$. The work reported here extends the finite-state reactive synthesis studies reported in [PR88, PR89a] and also in [ALW89], that did not impose a given architecture, and hence standardly yielded a solution based on the *easiest* architecture, that of a single processor.

Specifically, we show that the problem of *realizing* a given propositional specification over a given architecture is undecidable, and it is nonelementarily decidable for the very restricted class of *hierarchical* architectures. These results are based on Peterson and Reif's [PR79] work on games of incomplete information. We further give an extensive characterization of architecture classes for which the realizability problem is elementarily decidable, and of classes for which it is undecidable.

Another approach to the implementation of φ over $\mathcal{A}$ is to synthesize first (using the technique of [PR89a]) a single processor strategy that satisfies φ, and then to *decompose* this specific strategy into a set of programs $\Pi_1, \ldots, \Pi_k$ for $\mathcal{A}$. The problem of *decomposing* a given finite-state program for a single processor into a set of programs over a given architecture $\mathcal{A}$ is shown to be decidable for all acyclic architectures.

*This research was supported in part by the European Community ESPRIT Basic Research Action project 3096 (SPEC).

†This author's research was partially supported by a grant from the Israel Ministry of Science and Development, the National Council for Research and Development.

‡Address: Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel. E-mail: amir@wisdom.weizmann.ac.IL, roni@wisdom.weizmann.ac.IL .

1 Introduction

In this paper we continue the investigation, initiated in [PR88, PR89a], into the question of automatic synthesis of open reactive systems from their logical specifications. *Reactive* systems are those systems whose role is to maintain an ongoing interaction with their environment, rather than to yield a final value upon termination. Such systems must therefore be specified in terms of their behaviors. In contrast, the works of [MW80] and [Con85] consider the synthesis of nonreactive, or *transformational* programs.

The first attempts to synthesize finite-state reactive systems from specifications formulated in propositional temporal logic were developed in [MW84] and [EC82]. These two papers present algorithms, based on tableaux construction, which check whether a given temporal logic specification is implementable, and if it is implementable, the algorithms produce a finite-state program that satisfies (implements) the specification.

The synthesis method proposed in these two pioneering papers suffers from two limitations. The first limitation is that it can only be applied to the synthesis of *closed* reactive systems, i.e., systems in which the implementor can also construct the component representing the activity of the environment. Thus, if the specification is given by the temporal formula φ, the basic question asked for such systems is whether the environment and the system components can *cooperate* in order to satisfy φ. In contrast, in an *open* system context the environment and the systems components *compete* rather than cooperate, and the basic question asked is whether we can construct a *strategy* for the system's component that will win, by maintaining φ, against all possible moves of the environment.

This limitation was removed in [PR89a], which considers the synthesis problem for *open* reactive systems (called *modular synthesis* there). The paper presented a logical formulation of the implementability problem. When considering the case of finite-state systems that are specified by propositional temporal formulae, the paper suggested an algorithm, based on a reduction to automata on infinite trees, that produces an implementing strategy, whenever one exists. Not surprisingly, the approach proposed there relied heavily on game theoretical methods. These results were later extended to the model of asynchronous systems in [PR89b], where an arbitrary delay may occur between an input and its corresponding output. Another study of the implementability problem of open systems is reported in [ALW89], where additional aspects of the problem are studied, but a very similar algorithm for the finite-state case is recommended.

Unfortunately, the original approaches of [MW84] and [EC82] still suffer from a second limitation, that has not yet been removed by any of the subsequent works mentioned above. The limitation is that all the synthesis algorithms produce a program (strategy) for a *single* module (processor) that receives all the inputs to the system and generates all the outputs. This is particularly embarrassing in cases that the problem we set out to solve is meaningful only in a distributed context, such as the mutual exclusion problem, and a centralized single module solution does not seem very relevant. The somewhat ad-hoc solution to this difficulty that has been suggested in these works is to use first the general algorithm to produce a single module program, and then to *decompose* this program into a set of programs, one for each distributed component of the system.

In this paper we mount a direct attack on this last limitation, by considering the synthesis problem of *distributed* open reactive systems. As input to the synthesis problem we consider a distributed system architecture $\mathcal{A}$ and a temporal formula φ. The architecture $\mathcal{A}$ specifies a set of processors $P_1, \ldots, P_k$, and an interconnecting scheme for the communication among the processors and between the processors and the environment. Our model is based on communication by single-writer single-reader shared variables. However, we believe that the results are extendable to other communication models such as message-passing. The formula φ, expressed in linear temporal logic, specifies the set of behaviors that are admissible for the system. For the finite-state case we assume that the temporal formula φ is *propositional.*

The corresponding synthesis problem, to which we refer as the *realization* problem, is whether there exist programs (strategies) $\Pi_1, \ldots, \Pi_k$, such that if each processor P_i follows the program Π_i, for $i = 1, \ldots, k$, then the joint behavior of the system maintains φ against all possible inputs from the environment. We can view the problem solved in [PR89a] as a special case of the realization problem, where the architecture specifies a single processor P, to which all the external inputs and outputs are channeled.

The main results obtained in this paper for the general realization problem are rather pessimistic and can be summarized by:

- The general realization problem is undecidable, but is semi-decidable (r.e.). It is non-elementarily decidable for the restricted class of *hierarchical* architectures. These results are based on the work of Peterson and Reif in [PR79] on games of incomplete information.

- Characterizations are given for architectures for which the realization problem is elementarily decidable, and for undecidable architectures.

On the other hand, we show that the decomposition of a single processor finite-state program into a set of distributed programs $\Pi_1, \ldots, \Pi_k$ is decidable. This establishes the soundness and effectiveness of the methodology of deriving first a single processor program for φ and then decomposing it over $\mathcal{A}$. This methodology is obviously not complete in the sense that φ may be realizable over $\mathcal{A}$ and yet the methodology may fail to produce such a realization.

In Section 2 we introduce a general framework for studying the synthesis aspects, and formalize the basic relevant problems. Section 3 relates the implementability and realization issues to the connectivity properties of system architectures. Section 4 and Section 5 contain the main results concerning realization and decomposition, respectively.

2 The Framework

A *system architecture* (an *architecture* for short) is a tuple $\mathcal{A} = (\mathcal{P}, X, Y, T, r, w)$. $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a finite, non-empty set of *processors*. $X = \{x_1, \ldots, x_l\}$, $Y = \{y_1, \ldots, y_m\}$ and $T = \{t_1, \ldots, t_n\}$ are finite, possiblly empty, sets of respectively input, output and internal communication variables, i.e., shared variables used for communication. $r : X \cup T \rightarrow \mathcal{P}$ and $w : T \cup Y \rightarrow \mathcal{P}$ identify for each variable at most one reading and one writing processor, and we assume that for every internal variable $t \in T$, $r(t) \neq w(t)$. Note that for the input (respectively, output) variables, the environment is the writer (respectively, reader). Thus the specified variables are single-reader single-writer shared variables, serving for communication among the processors and between the processors and the environment, but not for the processors internal storage.

A sub-architecture of $\mathcal{A}$ consists of a subset of processors $\mathcal{Q} \subseteq \mathcal{P}$, and the appropriate restriction of the variables and their interconnection functions r and w to $\mathcal{Q}$. The following definition and notations concerning architectures apply implicitly to sub-architectures as well.

We shall usually represent an architecture by a directed graph whose nodes are the processors, and every variable $z \in X \cup Y \cup T$ corresponds to an edge (pictorially an arrow) from $w(z)$ to $r(z)$. Of course, the edges representing output variables have no destination, while those representing input variables have no source. An edge is labeled by the name of the variable it represents. Consequently, we will speak about paths, cycles, etc., in an architecture, referring to the corresponding paths, cycles, etc., in the associated graph.

In the simplified model considered here we allow only architectures whose graphs are *acyclic*. In a subsequent paper we will consider a model in which there is a non-negative delay associated with each variable. For such a model the requirement forbidding all cycles is relaxed to the exclusion of cycles whose accumulated delay is zero. We believe that the results obtained for the present model can be appropriately extended for the more general model.

For a processor P, $in(P) = r^{-1}(P)$ is the set of variables read by P, called the *in-variables* of P, and similarly $out(P) = w^{-1}(P)$ is the set of *out-variables* of P. For a set of processors $\mathcal{Q} \subseteq \mathcal{P}$ we further define $extin(\mathcal{Q}) = in(\mathcal{Q}) - out(\mathcal{Q})$, the variables read by processors of $\mathcal{Q}$ but written outside of $\mathcal{Q}$, and $extout(\mathcal{Q}) = out(\mathcal{Q}) - in(\mathcal{Q})$.

We refer to the trivial architecture consisting of a single processor that accepts all the inputs and generates all the outputs as the *singleton architecture*. $\mathcal{A}$ is said to be *connected*, if the underlying undirected graph is strongly connected. It is said to be *routed*, if for every input variable $x \in X$, and every output variable $y \in Y$, there is a path from $r(x)$ to $w(y)$. If for every pair of variables $x \in X$ and $y \in Y$ there is no path connecting $r(x)$ to $w(y)$, the architecture is said to be ***totally oblivious*** (notice that when either X or Y is empty, the architecture is both routed and totally oblivious).

Define an ***aggregation homomorphism*** of an architecture $\mathcal{A}'$ into an architecture $\mathcal{A}$ as a surjective mapping $h : \mathcal{A}' \rightarrow \mathcal{A}$, such that $X = X'$, $Y = Y'$, $T \subseteq T'$, and for each processor $P \in \mathcal{A}$,

$$in(P) = extin(h^{-1}(P))$$

and

$$out(P) = extout(h^{-1}(P)).$$

Thus, $\mathcal{A}$ can be obtained from $\mathcal{A}'$ by the aggregation (or collapsing) of the sets of processors $h^{-1}(P)$ into single processors P, for all processors P of $\mathcal{A}$, and the deletion of all variables which are internal to such a set. If such an aggregation homomorphism $h : \mathcal{A}' \rightarrow \mathcal{A}$ exists, we write $\mathcal{A}' \geq \mathcal{A}$, and say that $\mathcal{A}$ is an ***aggregation*** of $\mathcal{A}'$, or equivalently, that $\mathcal{A}'$ is a ***decomposition*** of $\mathcal{A}$ and $\mathcal{A}'$ ***decomposes*** $\mathcal{A}$. It is easy to see that the decomposition relation is a partial order on architectures.

We shall use the notations $\vec{x}$ for $x_1, \ldots, x_l$, $\vec{y}$ for $y_1, \ldots, y_m$, and $\vec{t}$ for $t_1, \ldots, t_n$. We assume that all variables of the architecture, namely $X \cup Y \cup T$, are oredered in some fixed given order, so that whenever we consider tuples of variables (or tuples of values for variables), the elements of the tuples are accordingly ordered.

Without loss of generality, we let all variables range over a single finite domain D, and let $d_0 \in D$ be some fixed initial value. A *reactive synchronous system* (a *system* for short), is an architecture $\mathcal{A}$ as above, together with a *semantic program*

$$f = \{f_z \mid z \in T \cup Y\},$$

which assigns to every variable $z \in T \cup Y$ that is produced by the processor $P = w(z)$ with input set $in(P) = \{z_1, \ldots, z_p\}$, a function $f_z : (D^p)^+ \rightarrow D$. We also refer to such a program as a ***strategy*** for $\mathcal{A}$. Thus, if P produces z, f_z is a program mapping non-empty histories of all P's input variables into single values that P writes on z.

Consider an infinite sequence of states

$$\sigma : s_0, s_1, \ldots,$$

where each state s_i consists of an interpretation over D for the variables in $V = X \cup T \cup Y$. We denote by $s_i[Z]$ the restriction of s_i to the subset $Z \subseteq V$. This definition extends naturally to restrictions of sequences of states and sets of such sequences.

The sequence σ is defined to be a *synchronous behavior* of the system $(\mathcal{A}, f)$, if for every variable $z \in T \cup Y$ written by the processor $P = w(z)$ with input set $in(P) = Z \subseteq X \cup T$, and each step $q = 0, 1, \ldots$, the value of z at step q satisfies

$$s_q[z] = f_z(s_0[Z] \cdot \ldots \cdot s_q[Z]).$$

Thus, the values of the writable variables are compatible with (determined by) the semantic function f. We denote by $B(\mathcal{A}, f)$ the set of behaviors of the system $(\mathcal{A}, f)$. The state s_i in a behavior represents the values of the variables $\vec{x}, \vec{y}$ and $\vec{t}$, at the ith step of the computation.

We extend the decomposition ordering to programs and systems as follows. We say that the system $(\mathcal{A}', f')$ *decomposes* the system $(\mathcal{A}, f)$, and write $(\mathcal{A}', f') \geq (\mathcal{A}, f)$, if $\mathcal{A}'$ decomposes $\mathcal{A}$, and $B(\mathcal{A}, f) = B(\mathcal{A}', f')[U]$, where U is the set of all variables of the aggregated architecture $\mathcal{A}$.

A *behavioral specification* is a linear temporal logic formula $\varphi(X, Y)$, in which, as a convention, X is a set of input variables and Y is a set of output variables. An architecture $\mathcal{A}$ (or a system $(\mathcal{A}, f)$) and a behavioral specification φ are said to be *compatible* with each other, if both have precisely the same input variables and the same output variables. A pair $(\mathcal{A}, \varphi)$ consisting of an architecture and a compatible behavioral specification is called a *complete specification.* A system $(\mathcal{A}, f)$ *(synchronously) implements* a compatible behavioral specification $\varphi(X, Y)$, written $\mathcal{A}, f \models \varphi(X, Y)$, if every behavior σ of $(\mathcal{A}, f)$ satisfies $\sigma \models \varphi(X, Y)$. Specifications for which an implementing system exists are called *implementable*, and the system implementing them is called an *implementation* of the corresponding behavioral specification. If some system $(\mathcal{A}, f)$ implements φ, then φ is called *$\mathcal{A}$-realizable*, and the program f is called an *$\mathcal{A}$-realization* of φ.

We now consider the following generic problems.

Problem 2.1 (Implementation) *Given a behavioral specification φ, does there exist a compatible implementation $(\mathcal{A}, f)$ for φ?*

Problem 2.2 (Realization) *Given a complete specification $(\mathcal{A}, \varphi)$, does there exist an $\mathcal{A}$-realization (program) f of φ?*

Problem 2.3 (Decomposition) *Given a system $(\mathcal{A}, f)$ and an architectural decomposition $\mathcal{A}' \geq \mathcal{A}$, does there exist a program f' compatible with $\mathcal{A}'$ such that $(\mathcal{A}', f') \geq (\mathcal{A}, f)$?*

A particularly interesting variant of the decomposition problem considers finite-state programs, that is, programs representable by finite-state automata (sometimes termed finite-state deterministic transducers).

Problem 2.4 (Finite-State Decomposition) *Given a system $(\mathcal{A}, f)$ where f is finite-state, and an architectural decomposition $\mathcal{A}' \geq \mathcal{A}$, does there exist a finite-state program f' compatible with $\mathcal{A}'$ such that $(\mathcal{A}', f') \geq (\mathcal{A}, f)$?*

For every architecture $\mathcal{A}$, we may also consider the restricted variants of the previous problems, such as the $\mathcal{A}$-realization and $\mathcal{A}$-decomposition problems, in which the considered architecture $\mathcal{A}$ is not part of a given instance of the problem but rather fixed. In the sequel we will refer to both behavioral and complete specifications simply as specifications, whenever the intention is clear form the context.

In the next sections we consider the mutual relations between these problems and their decidability and complexity aspects.

3 Implementability, Realization and Connectivity

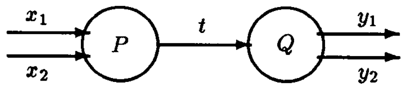
In this section we study some of the general relations holding between implementability and realization of a specification.

The first proposition characterizes the relation between implementability and realization in a singleton architecture. A realization on a singleton architecture is by definition an implementation. On the other hand, from any implementation of the specification on φ, say a realization f on some architecture $\mathcal{A}$, it is easy to derive a realization f_0 on the compatible singleton architecture. The function f_0 simply *emulates* the behavior of f on $\mathcal{A}$, using a set of compound states, accounting for the states of all the (finitely many) components of $\mathcal{A}$. Moreover, if f is finite-state, then f_0 is finite-state too.

Proposition 3.1 (Singleton-Realization) *A specification φ is implementable if and only if it is realizable on the singleton architecture.*

One conclusion of this proposition is that implementability can be reduced to a special case of realization. We next analyze the reasons why the other direction does not hold, thus demonstrating that realization is, in some precise sense, a *strictly harder problem* than implementability. One of the main obstacles to realization over a distributed architecture is the lack of adequate connectivity. Consider the architectures $\mathcal{A}_1$ and $\mathcal{A}_2$ below.

$\mathcal{A}_1$



$\mathcal{A}_2$

For example, in architecture $\mathcal{A}_1$ processor Q is charged with outputting y which in general may depend on the input x. However, x is connected to processor P which has no way to communicate with processor Q. Clearly, any non-trivial specification will be unrealizable on architecture $\mathcal{A}_1$. The case of architecture $\mathcal{A}_2$ is slightly more complex. Here there is a connection between P and Q but it is not wide enough to transmit both x_1 and x_2 so that Q can produce both y_1 and y_2.

To measure the adequacy of connectivity available in a given architecture, we introduce the following notions. An input-labeling is a mapping $\lambda : X \cup T \rightarrow X$ satisfying

1. for each $x \in X$, $\lambda(x) = x$;
2. if $t \in out(P)$ then $\lambda(t) = \lambda(z)$ for some $z \in in(P)$.

The intended meaning of the labeling λ is to assign to each variable $z \in X \cup T$ the role of communicating the value of the input $\lambda(z) \in X$. Requirement 2 demands that if P outputs the value of $x = \lambda(t)$ on t, it must have obtained it from some incoming variable $z \in in(P)$, in which case we say that P is ***informed about*** x. P is said to be ***informed about*** $X' \subseteq X$, if it is informed about every $x \in X'$. An input-labeling λ is defined to be ***adequate*** for a subset $X' \subseteq X$, if every output processor $P \in w^{-1}(Y)$ is informed about X'. Thus, adequacy for X' demands that, if P is responsible for generating some external output $y \in Y$, then it must have all the values of X' communicated to it.

We define the ***transmission width*** of the architecture $\mathcal{A}$, denoted by $tw(\mathcal{A})$, to be the size of the largest subset $X' \subseteq X$ for which $\mathcal{A}$ has an adequate input-labeling. An architecture $\mathcal{A}$ is called ***full*** if $tw(\mathcal{A}) = |X|$, i.e., it has an adequate input-labeling for the complete input set. Clearly, in a full architecture we can communicate each input value to each output processor, which ensures adequate connectivity.

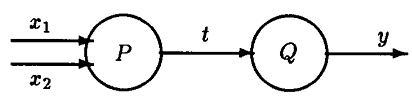
The computational complexity of the problems related to checking transmission-width is established in the next proposition.

Proposition 3.2 *The following problems are NP-complete:*

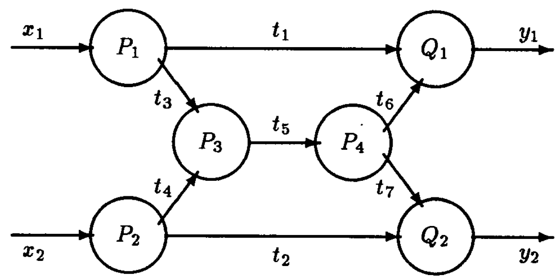
- *Given an architecture $\mathcal{A}$ and a positive integer N, is $tw(\mathcal{A}) \geq N$?*
- *Given an architecture $\mathcal{A}$, is $\mathcal{A}$ full?*

Sketch of Proof:
Obviously, both problems can be solved by non-deterministic polynomial-time algorithms. For NP-hardness, it is enough to show that checking fullness of an architecture is NP-hard. This can be done by quite a standard reduction from SAT, the satisfiability problem for the propositional calculus. ∎

While fullness is a sufficient condition for adequate connectivity, it is by no means a necessary condition. Consider the architectures $\mathcal{A}_3$ and $\mathcal{A}_4$ below.



$\mathcal{A}_3$



$\mathcal{A}_4$

Clearly, neither $\mathcal{A}_3$ nor $\mathcal{A}_4$ have an adequate input-labeling for $\{x_1, x_2\}$. Yet $\mathcal{A}_3$ can function properly by P computing the output corresponding to the inputs and transmitting it on t, while Q only copies t to y. Moreover, even $\mathcal{A}_4$ can function properly, and it needs just a fixed scheme for routing information: At any state, P_1 transmits x_1 on both t_1 and t_3, P_2 transmits x_2 on t_2 and t_4, while P_3 computes the value of $t_3 \oplus t_4$ (the *exclusive or* of t_3 and t_4) which is equal to $x_1 \oplus x_2$, and transmits it on t_5. Finally, P_4 copies $x_1 \oplus x_2$ to both t_6 and t_7, from which Q_1 can compute x_2 as $x_1 \oplus (x_1 \oplus x_2)$, and Q_2 can compute x_1 as $x_2 \oplus (x_1 \oplus x_2)$.

We therefore define the more general notion of *adequate input-output labeling* (for X and Y). This is a labeling $\lambda : X \cup T \cup Y \rightarrow$ Bool-Exp$(X) \cup Y$, (i.e, an assignment of either an output variable or a Boolean expressions over X to every variable) such that

1. $\lambda(z) = z$, for each $z \in X \cup Y$; and
2. if $z \in out(P)$, where $z \in T \cup Y$, then either
   (a) $\lambda(z) \in$ Bool-Exp$(\lambda(in(P)))$, i.e., $\lambda(z)$ is a Boolean expression over the labels of the variables in $in(P)$; or
   (b) $\lambda(z) \in Y$, and for every $x \in X$, there is a Boolean expression over $\lambda(in(P))$, which is logically equivalent to x.

Requirement 2(b) allows an intermediate processor, such as P in $\mathcal{A}_3$, to generate an output y which is then transmitted towards its final destination. Requirement 2(a) allows an intermediate processor such as P_3 in $\mathcal{A}_4$, to compute a Boolean combination of the values it reads and transmit it for further computation. Notice that requirement 2(a) permits the transmission of an already computed output, but no further computation with such a value is allowed, since by definition the labeling can not assign expressions over Y other than *single* variables $y \in Y$. We formally define an architecture to be *adequately connected*, if it has an adequate input-output labeling.

We can now propose a characterization of the architectures for which the realization problem is *not* harder than implementability. We define two compatible architectures $\mathcal{A}$ and $\mathcal{A}'$ to be *realization-equivalent*, if for every compatible specification φ, φ is $\mathcal{A}$-realizable iff it is $\mathcal{A}'$-realizable. That is, there does not exist a compatible specification which is realizable over one of them but not over the other.

Proposition 3.3 (Realization-Equivalence) *Every adequately connected architecture is realization-equivalent to the singleton architecture. In particular, any full architecture is realization-equivalent to the singleton architecture.*

We conjecture that the other direction of the proposition is also true. That is, an architecture is realization-equivalent to the singleton architecture only if it is adequately connected, i.e., possesses an adequate input-output labeling. If this conjecture is true it provides a complete characterization of the architectures for which the realization problem is *not* harder than implementability.

The results of Proposition 3.3 can be somewhat generalized. Consider two architectures $\mathcal{A}$ and $\mathcal{A}'$, such that $\mathcal{A}'$ decomposes $\mathcal{A}$. It is not difficult to see that this implies that realization on $\mathcal{A}$ is at least as easy as realization on $\mathcal{A}'$ in the sense that

if φ is realizable on $\mathcal{A}'$ it is also realizable on $\mathcal{A}$.

The interesting question is when are such $\mathcal{A}$ and $\mathcal{A}'$ realization- equivalent. Proposition 3.3 and the associated conjecture provide a complete answer for the special case that $\mathcal{A}$ is the singleton architecture. We refer to this case as a *basic decomposition*. Considering the more general case, any decomposition of $\mathcal{A}$ into $\mathcal{A}'$ can always be presented as a set of basic decompositions in which *some* (singleton) processors $Q_1, \ldots, Q_m$ of $\mathcal{A}$ are basically decomposed into sub-architectures $\mathcal{A}'_1, \ldots, \mathcal{A}'_m$ of $\mathcal{A}'$. We may examine each of the basic decompositions, say Q_i into $\mathcal{A}'_i$, for being adequately connected relative to $in(Q_i)$ and $out(Q_i)$. The following proposition provides an appropriate generalization of Proposition 3.3.

Proposition 3.4 (Generalized Equivalence) *If $\mathcal{A}$ is decomposed into $\mathcal{A}'$ by a set of basic decompositions, decomposing $Q_1, \ldots, Q_m$ into $\mathcal{A}'_1, \ldots, \mathcal{A}'_m$ of $\mathcal{A}'$, respectively, and each basic decomposition is adequately connected, then $\mathcal{A}$ is realization-equivalent to $\mathcal{A}'$.*

The reason that, for the general case, adequate connectivity is only a sufficient condition but not a necessary one is that one of the processors, Q_1 say, may be decomposed in an inadequate manner, but may be redundant to the main functioning of $\mathcal{A}$. In this case the fact that $\mathcal{A}'_1$ cannot faithfully emulate Q_1, may not detract from the ability of $\mathcal{A}'$ to emulate $\mathcal{A}$ in all cases.

4 Realization

In this section we consider the realization problem. The first observation towards the extension of implementability results into realization over distributed systems is concerned with the effect the addition of a processor P with $out(P) = \phi$ to any singleton architecture may have on the realization problem. This type of architectures corresponds to Reif's 2-player games of incomplete information ([Rei84]). The corresponding realization problem can be logically formalized in the sense of [PR88, PR89a], by prefixing the specification with universal quantification over the external input to the additional processor. Moshe Vardi showed that this case can be solved relatively easily, and when the specification is given by a Büchi automaton, the realization problem is not harder than that of realization in a singleton architecture ([Var89]). This observation can be generalized as follows.

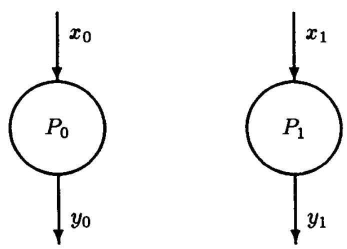
Proposition 4.1 *Let $\mathcal{A}$ be an architecture, and $\mathcal{A}'$ be $\mathcal{A}$ augmented with some totally oblivious architecture. Then, the $\mathcal{A}$-realization problem is decidable if-and-only-if the $\mathcal{A}'$-realization problem is.*

On the other hand, Peterson and Reif's results on constant-space bounded, multiple-person games of incomplete information can be extended to yield the following undecidability results.

Proposition 4.2 (Realization is Undecidable) *Finite-state realization is Σ_1^0-complete (recursively enumerable complete).*

Sketch of Proof:
We present several typical architectures for which the realization problem is undecidable, and from which it can be deduced that realization is undecidable for 'almost all' architectures.

The first architecture $\mathcal{A}_0$ presented below, serves for our basic undecidability proof. The other proofs are essentially variations of the fundamental ideas appearing in it.



Given any deterministic Turing machine M, we describe a specification φ_M which is $\mathcal{A}_0$-realizable *iff* M halts on the empty input-tape.

Assume that the outputs on channels y_0, y_1 encode a vocabulary sufficient to describe legal configurations of M, namely all tape symbols, internal states, and the special symbol \$. The output on channel y_i will represent a legal sequence of configurations $C^0, C^1, \ldots$ of M, delimited by non-empty strings of \$ symbols. Let the input on channels x_0, x_1 encode the vocabulary $\{S, N\}$, where S stands for *Start* (a new configuration) and N for *Next* (symbol of the current configuration). Let $\sigma : s_0, s_1, \ldots$ be a computation (behavior) of $\mathcal{A}_0$. A state s_j in σ is called a \$$_i$-state if y_i is outputting \$ in this state, i.e., $s_j[y_i] = \$$, and it is called an S_i-state if the input on x_i is S, i.e., $s_j[x_i] = S$. For a state s_j in σ, denote by $L_i(s_j)$ (or simply by L_i if the state is implicitly understood), for $i = 0, 1$, the *level* of processor P_i in state s_j, defined to be the number of S_i-states that preceded or coincide with the state s_j, i.e., $L_i(s_j) = |\{k : k \leq j, s_k[x_i] = S\}|$.

We first describe three assertions, ψ_1, ψ_2, and ψ_3. Their precise statement in propositional linear temporal logic is straightforward. The formula ψ_1 asserts that the following requirements on the input always hold:

- an S_i-state can only appear following a state in which both y_0 and y_1 contain \$;
- $|L_0 - L_1| \leq 1$.

The formula ψ_2 asserts that the following requirements on the output always hold:

- initially y_i outputs \$'s until the first S_i-state, where the initial configuration of M on the empty tape is outputted;
- beginning at every S_i-state, y_i outputs a legal configuration of M followed by one or more \$ symbols which extend until the next S_i-state;
- from any state which is both an S_0- and S_1-state, denote the configurations outputted on y_0 and y_1 by C_0 and C_1 respectively; then it is required that
  - * $C_0 = C_1$ if $L_0 = L_1$;
  - * $C_0 \vdash C_1$ if $L_1 = L_0 + 1$;
  - * $C_1 \vdash C_0$ if $L_0 = L_1 + 1$.

  Here $C \vdash C'$ denotes the fact that C' is the configuration which is the successor to the configuration C according to the rules of M.

The formula ψ_3 asserts that if L_i is unbounded, i.e., there are infinitely many S_i-states, then eventually y_i encodes a terminating configuration. Let $\psi_{0,M}$ be $\psi_1 \rightarrow \psi_2$, and φ_M be $\psi_1 \rightarrow (\psi_2 \wedge \psi_3)$.

We describe a *canonical strategy* $f^M = \{f_0, f_1\}$ for the architecture $\mathcal{A}_0$ (and the given machine M). Each f_i, $i = 0, 1$, operates as follows:

1. Output \$ symbols until the first S_i-state is reached.

2. At each S_i-state, with $L_i = k$, $k \geq 0$, let $C_1 \vdash C_2 \vdash \ldots \vdash C_k$ be the first k configurations of M on the empty input-tape — then output C_k followed by \$ symbols until the next S_i-state.

The following lemma says that f^M is the only $\mathcal{A}_0$-realization for $\psi_{0,M}$.

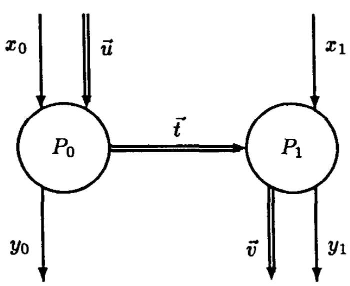
Lemma 4.3 *For architecture $\mathcal{A}_0$, strategy f^M, and specifications φ_M and $\psi_{0,M}$ as above,*

1. *f^M is an $\mathcal{A}_0$-realization of $\psi_{0,M}$; and*
2. *every $\mathcal{A}_0$-realization f of $\psi_{0,M}$ is identical to f^M; and thus, every $\mathcal{A}_0$-realization of φ_M is identical to f^M too.*

Now since ψ_3 essentially asserts the convergence of M (to be demonstrated by the output in y_i for appropriate input in x_i), we have our desireed result:

Lemma 4.4 *φ_M is $\mathcal{A}_0$-realizable if and only if M halts on the empty input-tape.*

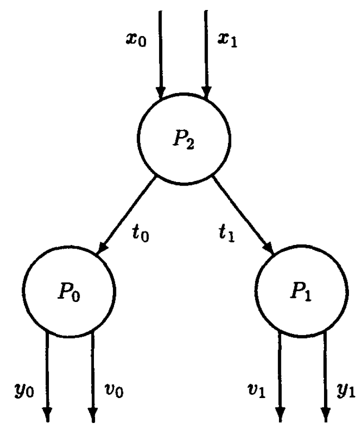
Consider now a more general case, consisting of the following connected architecture $\mathcal{A}_1$:



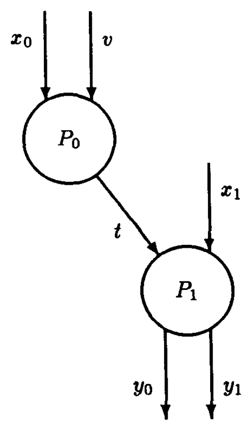
Here, all of $\vec{u}, \vec{t}, \vec{v}$, denote vectors consisting precisely of l variables, for some $l \geq 0$. (in the case $l = 0$, $\mathcal{A}_1$ is precisely the previous architecture $\mathcal{A}_0$.) Given a deterministic Turing machine M we again describe a specification $\varphi_{1,M}$ which is $\mathcal{A}_1$-realizable iff M halts on the empty input-tape. $\varphi_{1,M}$ extends φ_M, in that it asserts precisely the same requirements on the channels x_i and y_i, with the additional requirement, demanding that at every time instance $\vec{v}$ is equal to $\vec{u}$. Thus, P_0 is forced to use all the variables of $\vec{t}$ to transmit the current values of $\vec{u}$ to P_1, for these values to be emitted on $\vec{v}$.

Hence, the arguments showing the undecidability of $\mathcal{A}_0$-realization can be used to show the undecidability of $\mathcal{A}_1$-realization too.

By similar techniques we can show the undecidability of the realization problem for the architectures $\mathcal{A}_2$ and $\mathcal{A}_3$ below. Details will be given in the full version of the paper.



$\mathcal{A}_2$



$\mathcal{A}_3$

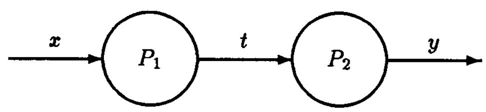
Nevertheless, decidability can be achieved by restricting the structure of the architectures considered. We call an architecture $\mathcal{A}$ with set of processors $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ a *pipeline* of *length* m, if it satisfies

- $X = in(Q_1)$;
- $in(Q_{j+1}) \subseteq out(Q_j)$, for $j = 1, \ldots, m-1$.

That is, the processors are arranged in a sequence $Q_1, \ldots, Q_m$, with the input to each processor in the sequence coming only from its predecessor, except for the first processor Q_1, whose input is external. A pipeline $\mathcal{A}$ of length m is *strict* if no pipeline $\mathcal{B} < \mathcal{A}$ is realization equivalent to $\mathcal{A}$. We now extend the hierarchical structures of [PR79], and define an architecture $\mathcal{A}'$ to be ***hierarchical of length m***, if there exists a strict pipeline $\mathcal{A}$ with processors $Q = \{Q_1, \ldots, Q_m\}$, such that $\mathcal{A}$ is an aggregation of $\mathcal{A}'$ via the homomorphism $h : \mathcal{A}' \to \mathcal{A}$, and each sub-architecture $h^{-1}(Q_i)$ is adequately connected, for $i = 1, \ldots, m$.

Proposition 4.5 *The problem of propositional $\mathcal{A}$-realization for hierarchical architectures $\mathcal{A}$ is decidable in time nonelementary (upper and lower bound) in the length of $\mathcal{A}$.*

Sketch of Proof:
It is enough to prove the claim for strict pipelines. The lower bound follows from the one proved in [PR79]. For the upper bound, consider the following strict pipeline $\mathcal{A}$ of length 2 (In order to simplify notation, we use single variables x, t, and y assumed to range over some finite domains D_x, D_t, and D_y, respectively, such that the strictness requirement of the pipeline is satisfied):



$\mathcal{A}$

Let φ_1 be a given specification for $\mathcal{A}$. From [PR89a] we have that φ_1 is implementable iff a certain Rabin tree automaton M on infinite trees is non-empty (accepts at least one tree), and moreover, there is a one-one correspondence between the implementations of φ_1 on the singleton architecture and the trees accepted by M. We call M the *specification automaton*. That is, each program for φ_1 corresponds to a tree in which every node n has degree $|D_x|$, where the descendants of n correspond to the various values the variable x may assume in the next step, and the labels of the descendants represent the output y produced by the program in the next step in response to these inputs. Thus, each such tree represents a program mapping finite histories of the input variable x to values of the output variable y. We refer to such a program as an x-y-strategy.

We define a *folded tree* to be a finite tree with a mapping that identifies each leaf of the tree with one of its ancestors. By unwinding a folded tree we obtain an infinite tree. An infinite tree T that can be obtained by unwinding a folded tree $\mathcal{F}$ is called *regular*, and $\mathcal{F}$ provides a finite representation of the infinite tree T. We call $\mathcal{F}$ the *scaffold* of the infinite regular tree T.

It is proved in [HR72] and [Rab72] that the specification automaton M accepts an infinite tree iff it accepts a regular one. There is a one-one correspondence between the scaffolds of the regular trees accepted by M and the *finite-state* implementations (programs) of φ_1 on the singleton architecture. It is also shown in [HR72] that there exists a finite automaton on finite trees M_1 that accepts precisely the set of scaffolds of the regular trees (perhaps with some finite unwinding) accepted by M. An efficient construction of such an automaton M_1 is given in [PR89a]. It should be noted that M_1, can also be viewed as a *deterministic* automaton on *strings* over the alphabet $D_x \times D_y$. For the purposes of the construction proposed below, note that M_1 can be constructed so as to satisfy the following two properties:

- All transitions that exit an accepting state, lead to an accepting state as well.
- For every state there is a single output value associated with the state.

We will show how to construct a branching formula ψ, and a corresponding automaton on finite trees M_2, with the following property: M_2 recognizes a finite-state program f_2 for P_2 (i.e., a finite tree T_2 representing a t-y-strategy) iff there exists a finite-state program f_1 for P_1 (an x-t-strategy), such that $B(\mathcal{A}, \{f_1, f_2\})[x, y]$, the joint behavior of the program composed of both f_1 and f_2 restricted to the external variables (which is a finite-state x-y-strategy), satisfy φ_1. Thus, the non-emptiness of M_2 is equivalent to the realization of φ_1 on the architecture $\mathcal{A}$.

In order to describe ψ, we consider several finite domains and two *dynamic* variables S and N which, in every node n of T_2 are assigned values S_n and N_n, respectively, ranging over the following domains. Let Q_1 be the set of states of M_1, with $q_0 \in Q_1$ the initial state, and $A_1 \subseteq Q_1$ the set of accepting states. S ranges over 2^{Q_1}, i.e., subsets of states of M_1. N ranges over (some enumeration of) the functions of type $Q_1 \times D_x \to 2^{Q_1 \times D_t}$, and is assumed to satisfy $N(s, a) \neq \phi$ iff $s \in S$. That is, in any node n of T_2, N_n is a function that maps every pair $\langle s, a\rangle$ consisting of a state $s \in S_n$ and a value $a \in D_x$, to a nonempty subset of pairs of the form $\langle s', b\rangle$, consisting of a state $s' \in Q_1$ and a value $b \in D_t$.

Assume that we are given a finite-state program Π_1 for f_1. Such a program can be represented by a finite set of (program) states $\hat{Q}$, an initial state $\hat{q}_0 \in \hat{Q}$, and a transition function $\hat{N} : \hat{Q} \times D_x \to \hat{Q} \times D_t$. This function specifies for each program state $q \in \hat{Q}$ and input value $a \in D_x$ a pair $\langle q', b\rangle = \hat{N}(q,a)$, consisting of a new state $q' \in \hat{Q}$ to which the program moves, and an output value $b \in D_t$. For a given input history $h_x \in D_x^+$, the program Π_1 produces an output history $h_t \in D_t^+$ of length equal to that of h_x, and reaches a certain program state $\hat{q}$. We denote $h_t = output_{\Pi_1}(h_x)$ and $\hat{q} = final_{\Pi_1}(h_x)$.

The variables S and N introduced above are intended to superimpose a program for f_1, such as Π_1, on the t-y-strategy for f_2. Let us see how it is done for the case that the program Π_1 is given. For this case the variable S ranges over the given $\hat{Q}$ (rather than Q_1) and N ranges over functions in $\hat{Q} \times D_x \to \hat{Q} \times D_t$. In the non-trivial case $|D_t| < |D_x|$, and therefore several different x-histories may cause Π_1 to produce the same t-history. Consider a node n in the program tree for f_2. This node corresponds to some t-history $h_t \in D_t^+$. We define $S_n \subseteq \hat{Q}$ to be the set of all states $\hat{q} \in \hat{Q}$ such that $\hat{q} = final_{\Pi_1}(h_x)$ for some h_x, such that $h_t = output_{\Pi_1}(h_x)$. Thus, the variable S_n holds all the program states of Π_1 that can be reached by reading an x-history that produces the t-history h_t which is associated with the node n. The variable N_n holds the current transition function, defining the local behavior of the program Π_1 when it is in one of the possible states (for the history h_t) $\hat{q} \in S_n$, and it reads the next input $a \in D_x$. The local behavior $\langle q', b\rangle$ is specified in terms of a next state $q' \in \hat{Q}$ and a current output $b \in D_t$.

In the case we consider here, the program Π_1 is unknown, and consequently we do not know the set of program states $\hat{Q}$. One of the main claims of the proposition we are considering is that it is sufficient to take for $\hat{Q}$ the set Q_1 of states of the automaton M_1, even though M_1 accepts x-y-strategies, while Π_1 defines a single x-t-strategy.

The *branching* formula ψ has the form

$$(\forall t)(\exists S, N)A\varphi_2(t, S, N),$$

where $\varphi_2(t, S, N)$ is a *linear* formula stating that a path satisfies the following requirements:

1. Initially $S = \{q_0\}$.
2. At any node n along the path N_n assigns correct next states according to the transitions of M_1.
3. For any node n and its successor n' along the path, $S_{n'}$ is determined by S_n, N_n and $t_{n'}$.
4. At any node n along the path, all state $q \in S_n$ agree on some output value $c \in D_y$.
5. There exists a node n along the path with $S_n \subseteq A_1$.

Let M_2 be the automaton on finite trees whose language consists of the scaffolds of regular trees satisfying ψ. We can now prove the following main lemma.

Lemma 4.6 *For φ_1, ψ, and M_2 as above,*

1. *φ_1 is $\mathcal{A}$-realizable iff ψ is valid over all tree models.*
2. *A finite-state program f_2 is accepted by M_2 iff there exists a finite-state program f_1 such that $\{f_1, f_2\}$ is an $\mathcal{A}$-realization of φ_1.*

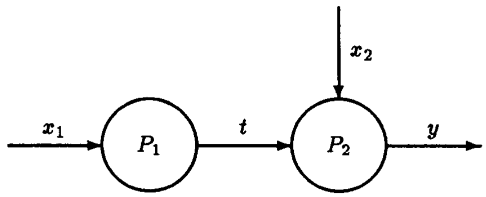
Now, if P_2 is decomposed into a deeper pipeline, we can carry on the above construction recursively, taking φ_2 to specify the tail of the pipeline, and M_2 to specify its regular implementations. ∎

5 Finite-State Decomposition

In order to achieve decidability, one may want to restrict the expressive power of the specifications considered. More specifically, we may approximate a solution to the realization problem by considering the restricted problem of *finite-state* decomposition. We believe this approach to be more pragmatic, especially if some knowledge about the required realization is available, and this knowledge can be used to focus the attention to a restricted number of implementations out of the possibly infinite number of different ones. A fast demonstration that decomposition is an *easier* problem than realization, is obtained by considering the architecture $\mathcal{A}_0$ used to prove the basic undecidability result for realization in Proposition 4.2. We leave it as an exercise to devise an algorithm which for any given finite-state program f, checks whether f is $\mathcal{A}_0$-decomposable. Actually, the next proposition shows that this is true of any (acyclic) architecture, namely, the finite-state decomposition problem is decidable.

Proposition 5.1 (Finite-State Decomposition) *Finite-state decomposition is nonelementarily decidable.*

Sketch of Proof:
Instead of a full proof we will sketch the decomposition procedure for a typical architecture, for which the realization problem is undecidable. Consider the following architecture $\mathcal{A}$:



$\mathcal{A}$

Assume that the variables x_1, x_2, t and y range over finite domains D_1, D_2, D_t and D_y, respectively, such that $|D_t| < |D_1| = |D_y|$, thus guaranteeing that the $\mathcal{A}$-realization problem is undecidable. Let $f : (D_1 \times D_2)^+ \rightarrow D_y$ be given by a finite-state program Π for the singleton aggregation of $\mathcal{A}$, represented by a set of states Q, an initial state $q_0 \in Q$, a transition function $\delta : Q \times D_1 \times D_2 \rightarrow Q$, and an output function $\eta : Q \rightarrow D_y$. We will show how to construct a branching formula ψ and a corresponding automaton M_2 on finite trees, such that M_2 accepts a finite-state program f_2 for P_2 (a (t, x_2)-y-strategy) iff there exists a finite-state program f_1 for P_1 (an x_1-t-strategy), such that the composed program $f_1 \circ f_2$ is equal to f.

Let D be 2^Q, the power set of Q, and let S_1, S_2 and N_1 be dynamic variables ranging as follows. S_1 ranges over 2^D, i.e., consists of sets of subsets of Q. S_2 ranges over $D = 2^Q$. N_1 ranges over the domain of functions of type $D \times D_1 \rightarrow D_t$.

To explain the intuition behind the described constructions, assume that we have actual programs for P_1 and P_2 that decompose Π. Let h_1 be a finite x_1-history, i.e., a sequence of D_1-values. We define $K_1(h_1)$ to be the set of all Q-states that the program Π may reach on reading the x_1-history h_1 and some x_2-history of the same length. The set $K_1(h_1)$ represents what P_1, that only sees the x_1-inputs, knows about the current state of Π.

Let h_1 and h_1' be two *infinite* x_1-histories. We say that h_1 and h_1' are Π-*equivalent* if Π generates the same output sequence on the input (h_1, h_2) and the input (h_1', h_2) for every infinite x_2-history $h_2 \in D_2^\omega$. Let h_1 and h_1' be two finite x_1-histories of equal length. We say that h_1 and h_1' are Π-*equivalent* if $h_1 \cdot h$ is Π-equivalent to $h_1' \cdot h$ for every infinite x_1 history $h \in D_1^\omega$.

Let Q_1 be the set of states of P_1, δ_1 its *state-reachability* function, and e_1 its *emission* function. Thus, we write $q_1 = \delta_1(h_1)$ and $h_t = e_1(h_1)$ to denote that the x_1-history h_1 causes Q_1 to move to state $q_1 \in Q_1$ and to emit the t-history $h_t \in D_t^+$.

We say that the program P_1 is Π-*normal* if, for every two x_1-histories h_1 and h_1', $e_1(h_1) = e_1(h_1')$ and $K_1(h_1) = K_1(h_1')$ imply $\delta_1(h_1) = \delta_1(h_1')$. Thus, a Π-normal P_1 does not unnecessarily distinguish between two states that are reached by two histories that emit the same output and lead to identical K_1-sets.

It is not very difficult to show that if Π is $\mathcal{A}$-decomposable, it is decomposable into a pair (P_1, P_2) such that P_1 is Π-normal. This is based on the observation that if $e_1(h_1) = e_1(h_1')$ and $K_1(h_1) = K_1(h_1')$, then h_1 is Π-equivalent to h_1'. From now on, we assume that P_1 is Π-normal.

Let h_t be a t-history. We define $K_t(h_t)$ to be the set of subsets $\{K_1(h_1^1), \ldots, K_1(h_1^m)\}$, where $h_1^1, \ldots, h_1^m$ are all the x_1-histories that cause P_1 to emit h_t. This represents the knowledge of P_2, that only sees t-histories, about what P_1 may think is the current state of Π.

We also define $K_2(h_t, h_2)$, for a t-history h_t and an x_2-history h_2, to be the set of all states that Π may assume after reading the x_1-history h_1 and the x_2-history h_2, where h_1 causes P_1 to emit h_t. Note that while $K_t(h_t)$ is a set of sets of Q-states, i.e., an element of 2^D, $K_2(h_t, h_2)$ is a set of Q-states, i.e., an element of $D = 2^Q$.

When P_1 and P_2 are known, we expect that the values assumed by S_1 and S_2 at a node n in a (t, x_2)-tree be $K_t(h_t)$ and $K_2(h_t, h_2)$, respectively, where h_t and h_2 are the t-history and x_2-history determined by the values of t and x_2 on the path leading from the root of the tree to node n.

Due to the assumption that P_1 is Π-normal, each element $s_1 \in S_1 = K_t(h_t)$ represents a unique Q_1-state that can be reached by some x_1-history that emits h_t. Consequently, for each s_1 and each next x_1-value $a \in D_1$, there exists a t-value $c \in D_t$ which is the output emitted by P_1 when in state q_1 and seeing the next input a. We therefore expect the function N_1 defined at the current node to yield $N_1(s_1, a) = c$.

Since our problem is to find out whether P_1 and P_2 exist for the finite state program Π, we cannot assume that they are known. Instead, we put into the formula ψ clauses requiring that S_1, S_2, and N_1 behave as implied by the existence of the decomposition (P_1, P_2).

The branching formula ψ has the form

$$(\forall t)(\exists S_1, S_2, N_1) A \varphi_2(t, S_1, S_2, N_1),$$

where $\varphi_2(t, S_1, S_2, N_1)$ is a *linear* formula stating that a path satisfies the following requirements:

1. Initially $S_1 = \{\{q_0\}\}$ and $S_2 = \{q_0\}$.
2. All states $q \in S_2$ agree on some output values $e \in D_y$.

3. If, at the next node on the path, $t = c$ for some $c \in D_t$, then the next value of S_1 is

$$\left\{ s_1^a \;\middle|\; \begin{array}{l} s_1 \in S_1, \\ a \in D_1, \\ \text{such that } N_1(s_1, a) = c \end{array} \right\} - \{\phi\}$$

where $s_1^a = \{\delta(q, a, b) \mid q \in s_1,\ b \in D_2\}$.

4. If, at the next node on the path, $t = c$ for some $c \in D_t$ and $y = b \in D_2$, then the next value of S_2 is

$$\left\{ \delta(q, a, b) \;\middle|\; \begin{array}{l} q \in s_1 \cap S_2 \text{ for some } s_1 \in S_1, \\ a \in D_1 \text{ with } N_1(s_1, a) = c \end{array} \right\}.$$

Let M_2 be the automaton on finite trees whose language consists of the scaffolds of regular trees satisfying ψ. Then, the following lemma provides the connection between ψ validity, M_2 acceptance and $\mathcal{A}$-decomposition of f.

Lemma 5.2 *For f, ψ and M_2 as above,*

1. *The program f is $\mathcal{A}$-decomposable iff ψ is valid over all tree models.*
2. *A finite-state program f_2 is accepted by M_2 iff there exists a finite-state program f_1 such that $\{f_1, f_2\}$ is an $\mathcal{A}$-decomposition of f.*

Actually, the $\mathcal{A}$-decomposition of f could be easily expressed by the validity of a *linear* formula, somewhat more complex than the above φ_2. Nevertheless, the construction of a linear *decomposition formula* can be generalized to all acyclic architectures. ∎

Acknowledgements

We thank Moshe Vardi for pointing out the relevance of the work of Reif and Peterson on games of incomplete information to this research, and for helpful suggestions.

References

[ALW89] M. Abadi, L. Lamport, and P. Wolper, Realizable and unrealizable concurrent program specifications, *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, Lec. Notes in Comp. Sci. 372, Springer, 1989, pp. 1–17.

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

[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.

[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.

[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.

[PR79] G.L. Peterson and J.H. Reif, Multiple-person alternation, *Proc. 20th IEEE Symp. Found. of Comp. Sci.*, 1979, pp. 348–363.

[PR88] A. Pnueli and R. Rosner, A framework for the synthesis of reactive modules, *Proc. Intl. Conf. on Concurrency: Concurrency 88* (F.H. Vogt, ed.), Lec. Notes in Comp. Sci. 335, Springer, 1988, pp. 4–17.

[PR89a] A. Pnueli and R. Rosner, On the synthesis of a reactive module, *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, 1989, pp. 179–190.

[PR89b] A. Pnueli and R. Rosner, On the synthesis of an asynchronous reactive module, *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, Lec. Notes in Comp. Sci. 372, Springer, 1989, pp. 652–671.

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

[Rei84] J.H. Reif, The complexity of two-player games of incomplete information, *J. Comp. Sys. Sci.* **29**, 1984, pp. 274–301.

[Var89] M.Y. Vardi, Personal Communication, 1989.