SAN098-1338C
SAND--98-1338C
CONF-990101--

A Synchronous Paradigm for Modeling Stable Reactive Systems

Victor L. Winter*
Sandia National Laboratories
vlwinte@sandia.gov

Abstract

This paper describes a modeling technnique for *single-agent* reactive systems, that is influenced by the modeling paradigm of Parnas [5] as well as by the synchronous paradigms of LUSTRE[4] and ESTEREL[1].

In this paradigm, single-agent reactive systems are modeled in a universe having a discrete clock. This discretization of time greatly reduces the temporal complexity of the model. We believe that the advantage of this reduction in temporal complexity is that the resulting model is in many ways better suited to automated software construction and analysis techniques (e.g., deductive synthesis, transformation, and verification) than models that are based on continuous representations of time.

1 Motivation

In [5], real-time systems are elegantly modeled in terms of a collection of controlled and monitored variables. System *behaviors* are then described in terms of *time functions* that describe the values of controlled and monitored variables over time. Specifications are then defined as sets of acceptable or desirable behaviors.

While the model and notation given in [5] is quite intuitive, we believe that for reactive systems, it is not ideally suited to automated software construction technologies such as deductive synthesis and transformation. At the root of this difficulty lies the continuous nature of time that is expressed within the model. For many systems, this richness of expression is unnecessary and can "confuse" automated analysis and construction techniques.

Consider for a moment how one might define a system "state change" with respect to a paradigm of the type described above. Initially, at time t_0, the system will be in some state that is possibly defined as a tuple of monitored and controlled variables, $(\vec{m}, \vec{c})$. Then at some time, t_1, the system will enter a new state, $(\vec{m}', \vec{c})$.

If such an approach is taken it will, in general, often be the case that a control vector, $\vec{c}$, can cause a system to pass through a number of states over time.

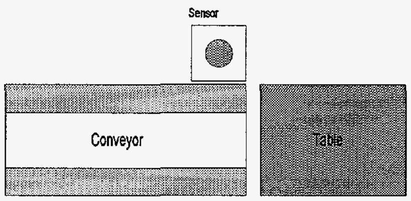
$$(\vec{m}_1, \vec{c}) \xrightarrow{t_1} (\vec{m}_2, \vec{c}) \xrightarrow{t_2} (\vec{m}_2, \vec{c}) \xrightarrow{t_3} (\vec{m}_3, \vec{c}) \rightarrow \ldots$$

Some of the states that the system passes through can be directly sensed and will be reflected by changing values in the vector of monitored variables (e.g., m_1, m_2, m_3). Other states, cannot be explicitly sensed

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

(e.g., $(\vec{m}_2, \vec{c}) \xrightarrow{t_2} (\vec{m}_2, \vec{c})$) but are implied by the trace history of the system. Because of the potential, that a single control vector has, to cause the system to pass through a number of states over time, it becomes difficult to distinguish a transition sequence from an individual transition without addressing time in some nontrivial fashion.

For example, consider a robotic system consisting of a conveyor belt and a rotating table. In this system a metal plate can be passed from the conveyor to the table as follows: (1) Assume a plate is at the left side of the conveyor, (2) turn the conveyor motor on, (3) keep the conveyor running until the photoelectric sensor located at the right side of the conveyor senses the plate, (4) if the table is in its leftmost position **and** the conveyor motor is running **and** a plate is sensed by the photoelectric sensor, then when the photoelectric sensor no longer senses the plate, the plate will have been transfered from the conveyor to the table.



A simple robotic system

Consider the following attempt to formally express the plate transfer just explained:

$$\forall t_1, t_2 : \quad (t_1 < t_2) \wedge \textit{sensor\_on}(t_1) \wedge \textit{table\_left}(t_1) \wedge \textit{sensor\_off}(t_2) \wedge \textit{conveyor\_on}(t_1)$$
$$\Rightarrow$$
$$\textit{plate\_on\_table}(t_2)$$

This formalization is incorrect for a number of reasons. First of all, missing is the fact that t_2 is the earliest time after t_1 that the predicate *sensor_ off* becomes true. Also missing is the fact that the table is in its leftmost position during the entire time interval $[t_1...t_2]$. Clearly, if the table moves out of its leftmost position during the interval $[t_1...t_2]$ then one cannot assume, from the English explanation given, that a "proper" plate transfer has occurred. And finally, the predicate *conveyor_ on* is not very useful in the sense that it does not convey any relevant information. That is, if we removed the *conveyor_ on* predicate from the above description its meaning would essentially remain unchanged in as much as that the same behavior is still being described.

The problem in the example just given stems from the fact that it is difficult, within the formal model, to distinguish state change sequences from individual state changes. In the model, this distinction is made through variables that describe discrete points with respect to a continuous model of time. In such a model a system description can easily become overly complex. The complexity arises because the notion of time essentially falls outside of the paradigm and must be accounted for (i.e., modeled) using brute force low-level mechanisms (e.g., variables quantified over a continuous range of values). The end result is that time is not modeled in a very abstract manner.

Another difficulty here is that in addition to modeling problems there are also algorithm optimization/manipulation problems. Consider the following informally described algorithm consisting of the sequential composition of two behaviors:

move the table to its left position **and then** *turn the conveyor on*

Suppose that we wanted to perform these behaviors in parallel. From a correctness point of view we need to be able to formally verify that parallelization of these behaviors is indeed possible and correctness preserving. In general, reasoning about this kind of optimization involves (1) showing that, during the relevant time intervals, the behaviors do not intersect or interfere with each other in some "behavior altering" manner (e.g., a safety violation such as a collision), and (2) showing that behaviors will complete at the proper time (e.g., the table must be in its left position **before** the plate leaves the right edge of the conveyor).

Let us look at this general problem from a more implementation oriented perspective. Consider the following code fragement which is an example of a control algorithm for achieving a specific behavior.

```
begin loop
  $\vec{m}$ := poll sensors;
  if $p_1(\vec{m}) \rightarrow \vec{c}_1$
    [] $p_2(\vec{m}) \rightarrow \vec{c}_2$
    [] ...
    [] $p_n(\vec{m}) \rightarrow \vec{c}_n$
  endif
end loop
```

Here the *if-endif* construct is a guarded command. Recall that the guard expressions in such a command are complete. That is,

$$\forall \vec{m} : p_1(\vec{m}) \vee p_2(\vec{m}) \vee ... \vee p_n(\vec{m}) = true$$

Because of its completeness, care must be taken when manipulating an if-endif construct via automated mechanisms such as transformation. An additional difficulty here is that in the general case a control vector, $\vec{c}_i$, can effect numerous components in the system simultaneously. For example, rotating a table to the left and engaging a conveyor. And to make matters worse simultaneously operating components can complete various objectives at different times.

An approach that is much more suited to transformation is to begin with simple control vectors that "just do one thing" and that are embedded within their own sense-reace loop. Given such an initial form, correctness preserving transformations can be used to merge control loops and compose control vectors into more complex vectors. However, in order for such a bottom-up compositional approach to be practical, the modeling framework must readily support it.

We would like to point out that other types of sense/react loops can be constructed (e.g., we don't need a guarded command construct with its completeness requirement). The purpose of the above example is simply to draw attention to some of the difficulties that can be encountered in constructing software for reactive systems and how the complexity of the underlying formal model can impact this process.

To address the difficulties described above, we have developed a synchronous paradigm for modeling reactive systems. Though we only briefly touch upon it in this paper, the paradigm we present is well suited to a bottom-up software development process where simple behaviors (i.e., simple control vectors and simple control structures) can be synthesized deductively and then transformed into complex efficient behaviors in a correctness preserving manner. The paradigm we present somewhat limits the kinds of systems that can be modeled. However, we believe that a large class of real world problems can be described by this model. Our goal is to create a formal model in which, during the algorithm design phases of software development, time is a discrete and essentially unitless quantity (all atomic behaviors take 1 unit of time regardless of their actual duration).

To date we have used the synchronous model described in this paper to model a moderate sized robotic system. This model together with a suitable specification has been formally abstracted and given to a deductive synthesis system. The result from the synthesis stage is an inefficent correct-by-construction abstract algorithm that can then be optimized and targeted to a specific computing platform through correctness preserving transformations[2][14][15].

2 *Single-Agent* Reactive Systems

Definition 1 ***Reactive System*** *— a system in which the metric (i.e., duration) of time is not important. A system's behavior can be described by considering only temporal (e.g., before, after, eventually) qualities of events.*

Reactive systems are often formally described in terms of a collection of controlled and monitored variables. The values that are bound to the monitored variables are generally supplied by various sensors, and the values bound to the controlled variables are used to produce various system behaviors (e.g., operation of mechanical devices such as opening and closing of valves, operation of electrical devices such as engaging electromagnets, etc.) Given this view, the function of a controller is to (1) use the information provided by the monitored variables in order to determine what state the system is in, and (2) adjust the values of the controlled variables in order to achieve a desired system behavior.

Since the controller can change the state of the system, it is considered to be an *agent* of that system. Similarly, if the environment has the ability to change the state of the system it is also considered to be an agent of that system.

Definition 2 ***Single-Agent Reactive System*** *— a reactive system in which the controller is the only* ***agent*** *(i.e., a reactive system in which the environment cannot initiate state changes in the system).*

For practical reasons our research is currently limited to *deterministic single-agent systems*. These are single-agent systems in which all *atomic state transitions* (defined in Section 5) produce a state change whose outcome can be determined *apriori*. For the remainder of this paper, we will use the term "system" to refer to a deterministic single-agent reactive system.

3 A Basic System Model

We begin by modeling a system, s, as a vector of monitored variables $\vec{m} \stackrel{\text{def}}{=} (m_1, m_2, ..., m_n)$ and a vector of controlled variables $\vec{c} \stackrel{\text{def}}{=} (c_1, c_2, ..., c_k)$. In our model, we require that the controlled variables be independent of one another in the sense that assigning a value to a variable c_i should not restrict[1] the value that can be assigned to a variable c_j when $i \neq j$. Let M and C respectively denote the sets of all possible configurations of the monitored and controlled variables that are allowed by the environmental constraints. The set $S \stackrel{\text{def}}{=} \{(\vec{m}, \vec{c}) \mid \vec{m} \in M \wedge \vec{c} \in C\}$ then describes the state space of the system.

Observable State Space — Given a system together with a behavior that we would like this system to satisfy, the *observable state space* of the system is the collection of states that the controller must be able to distinguish in order produce the desired system behavior. For example, if a controller needs to be able to distinguish distances on a robot arm to a resolution of 1 cm, then the sensors of the system must be able to directly (or indirectly) provide this level of resolution, otherwise the capabilities of the system will not be sufficient to produce the desired behavior.

[1]This does not subsume the possibility of a safety violation.

Control State Space — The *control state space* is the set of all possible values that the control vector can assume given the environmental constraints that the system is subjected to.

System Behavior — Let $\mathcal{M}$ denote the observable state space of a system. A *system behavior* is a sequence of elements $b_m = < m'_1, m'_2, m'_3, ... >$ such that each $m'_i \in \mathcal{M}$. We say a system implementation, s, satisfies the system behavior b_m iff the system **eventually** enters all of the states $m'_i \in b_m$ in the order defined by the behavior. For example, to satisfy b_m, the system s must eventually enter m'_1, then it must eventually enter m'_2, and so on. Systems behaviors can be at various levels of detail. At one extreme, they can provide so much information that they become algorithmic in nature. At the other end of the spectrum, elements in a system behavior sequence can be used to describe attributes of interest such as invariants for infinite behaviors and pre and postconditions for finite behaviors.

System-Controller Behavior — Let $S \overset{\text{def}}{=} \{(m, c) \mid m \in \mathcal{M} \wedge c \in C\}$ denote the state space of a system. A *system-controller behavior* is essentially the same as a system behavior except for the fact that system-controller behaviors also contain information about the control state of the system. That is, a system-controller behavior is a sequence of elements $b_s = < (m_1, c_1), (m_2, c_2), (m_3, c_3)... >$ such that each $(m_i, c_i) \in S$.

Specifications — A *specification* defines a set of system behaviors. We also say that a specification defines a problem. Given a specification, **spec**, and a system behavior **f**, if **f** $\in$ **spec**, then **f** is said to solve the problem defined by **spec**.

Software Construction — Given a system, s, with the characteristics described above together with a statement of a problem, p, that we would like the system to solve (i.e., a specification), our objective is to construct a software controller that monitors the system via $\vec{m}$ and controls the system via $\vec{c}$ in order to solve p.

3.1 Virtual Sensors

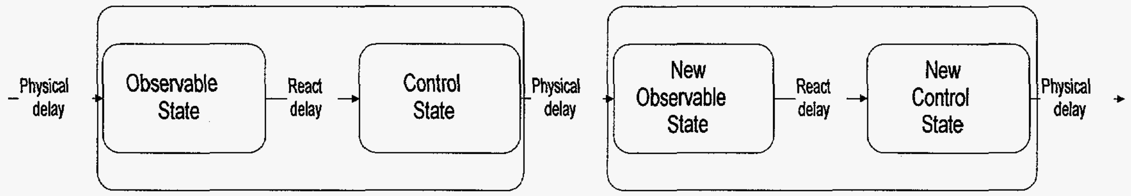
Let *spec* be a problem specification. In general, $\vec{m}$ will not directly provide enough information to distinguish all states in the observable state space of the system. In other words, in practice it is often the case that by using only the information provided by $\vec{m}$ it will not be possible for a controller to achieve a behavior $f \in$ *spec*. However, by combining (1) knowledge of the systems initial state, (2) the information provided by $\vec{m}$, together with (3) an historical trace of the system, the controller will have sufficient information to solve p.[2]

A more unified perspective of the system state can be obtained by viewing the historical trace as a collection of *virtual monitored variables*, $(v_1, v_2, ...v_j)$, which represent the cumulative information provided by the trace. Such a view extends the monitored variables from $\vec{m} \overset{\text{def}}{=} (m_1, m_2, ..., m_n)$ to $\vec{m}_v \overset{\text{def}}{=} (m_1, m_2, ..., m_n, v_{n+1}, v_{n+2}, ...v_{n+j})$. This correspondingly modifies the definition of the system state space to: $S_v \overset{\text{def}}{=} \{(\vec{m}_v, \vec{c}) \mid \vec{m} \in M_v \wedge \vec{c} \in C\}$, where M_v is observable state space of the system (i.e., $M_v = \mathcal{M}$).

[2]Note that we are assuming that a solution to p is possible given the controller's capabilities.

4 A Synchronous Modeling Paradigm

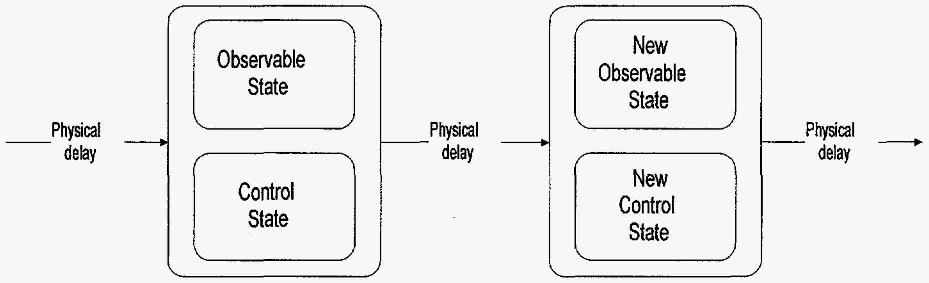
On close inspection it becomes apparent that there is a delay between (1) the time that the system enters an observable state and the time it takes for the controller to recognize that the system is in that state, as well as (2) the time that the controller changes one or more values of the controlled variables and the time it takes for the system to enter into the (new) observable state as dictated by the control state.



The time delays between observable states and control states.

It is worth noting that the time it takes for the observable state to reflect the changes brought about by a control state can vary greatly. For example, turning an electromagnet on in order to "grab" a metal object results in an almost instantaneous observable state change. On the other hand, moving an object down a conveyor can require a significantly longer amount of time to produce an observable state change. The question that arises here is: "To what extent does this temporal aspect of the system need to be modeled?"

synchronous hypothesis — *The synchronous hypothesis assumes that the controller is fast enough to react to every observable state change. By accepting this hypothesis we can, without loss of generality, set the time between "observable state" and "control state" (i.e., the react delay) to **zero!***



A Synchronous Behavior Perspective

If we accept the *synchronous hypothesis* as a proof obligation (i.e., a hypothesis whose proof we defer to a later point in the software construction process), we are able to abstract time in a very elegant manner.

4.1 Parallel Systems

It is worth mentioning that in a system in which parallel activities can take place (e.g., two conveyor belts operating at the same time), one must be careful when assuming a synchronous hypothesis. A little thought will show that in most cases it is unrealistic to unconditionally assume a synchronous hypothesis for systems in which parallel behaviors are possible.

For example, in a two conveyor system, let t_1 and t_2 respectively denote the time at which the photocell of the first conveyor senses an object and the time at which the photocell of the second conveyor senses an object. Without additional assumptions one cannot conclude that the difference between t_1 and t_2 will not become vanishingly small at some point during the operation of the system. Thus for many parallel systems one might think that the synchronous hypothesis does not hold.

It turns out that the difficulty described in the previous paragraph can be easily avoided by first constructing a sequential controller and then introducing parallel behavior only when it can formally be verified to be correct (i.e., the event times **between** parallel activities can safely be ignored). In an upcoming paper we describe how *correctness-preserving* transformations can be used to optimize (i.e., parallelize) controller algorithms in this manner.

4.2 Atomic (sequential) System Behaviors

Due to the difficulties that can be encountered with parallel controller states, we are interested in limiting our attention to states that “just do one thing”. In Section 5 we define *atomic control states* in terms of their nearness to *stable* control states. However, until that section is reached, the informal — “just do one thing” — definition should suffice. In order to avoid potential difficulties that can be encountered in parallel systems, we will assume from here on out that all control states are atomic (i.e., they just do one thing).

The (atomic) restriction on control states together with our synchronous hypothesis allows us to model time in terms of discrete ticks of a universal clock in the following manner:

- the controller always reacts instantly to the observable state of the system (i.e., react delay = 0), and
- given an atomic control state it will take exactly one clock tick for the observable state to change in a manner corresponding to the control state (i.e., physical delay = 1).

4.3 Synchronous Notation

We write the expression $[\vec{m}\colon\vec{c}]_{t_i}$ to denote that at time t_i the system is in observable state $\vec{m}$ and control state $\vec{c}$. Informally, the relationship between $\vec{m}$ and $\vec{c}$ is as follows: at the instant of time denoted by t_i, the controlled variables have been set to $\vec{c}$ and the observable state has not yet been given an opportunity to respond to (i.e., to be affected by) the new control state. For convenience we will omit the time subscript when it is unimportant with respect to what we wish to express.

Given an atomic control state, $\vec{c}$, our objective is to construct a system model in which the change to the observable state is accomplished during one tick of the universal clock. The expression $[\vec{m_1}\colon\vec{c}] \div \vec{m_2}$ asserts that if at some (arbitrary) time the system is in observable state $\vec{m_1}$ and the controller is in atomic state $\vec{c}$, then at the next clock tick the observable state will be $\vec{m_2}$. It is important to remember that this notation is only defined when $\vec{c}$ is an atomic state. This restriction allows us to deal with the issues of parallel behaviors completing at different times within a transformational framework.

5 Stable Systems

In order for a single-agent system to be reactive, it must have the property that it can be brought to rest in any observable state. That is, for every observable state there exists a control state that will cause the observable state to remain unchanged over time. We call control states that have this property *stable control states.* With some thought it can be seen that for systems where this is not the case the metric of time is important — hence, such a system is not strictly reactive.

At this point, we assume the existence of a function, $\mathcal{E}$, that measures the energy needed to maintain a control state. For example, keeping an electromagnet on is a control state that requires energy. Similarly, keeping a conveyor belt running requires energy.

Definition 3 *Given a system,* s, *let* $\mathcal{M}$ *and* C *denote the observable state space and the control state space of* s *respectively. Then*

- ***stable***$(s) \overset{def}{=} \forall \vec{m} \in \mathcal{M}, \exists \vec{c} \in C : [\vec{m}{:}\,\vec{c}] \doteqdot \vec{m}$. *Though we do not go into it in any great detail in this paper, systems that are stable are interesting because they have a temporal simplicity that is well suited to automated construction and analysis techniques.*
- ***passive***$(s) \Leftrightarrow (\forall \vec{m} \in \mathcal{M}, \exists \vec{c} \in C : [\vec{m}{:}\,\vec{c}] \doteqdot \vec{m} \wedge \mathcal{E}(\vec{c}) = 0)$.
- ***stable control state*** — *given a tuple* $(\vec{m}, \vec{c})$, $\vec{c}$ *is stable with respect to* $\vec{m}$ *iff* $[\vec{m}{:}\,\vec{c}] \doteqdot \vec{m}$.

Definition 4 ***Atomic*** — *A control state* $\vec{c}$ *is* ***atomic*** *with respect to an observable state* $\vec{m}$ *iff it is possible to change the value of a single element,* c_i, *in* $\vec{c}$ *and thereby produce a control state that is stable with respect to* $\vec{m}$. *In this case we say that the value* c_i *makes the control state* ***active***. *Note that as a result an atomic control state can simply be denoted by the value of the element* c_i. *This observation allows us to describe an atomic control state compactly in terms of the value* c_i *(rather than the vector* $\vec{c}$*).*

5.1 An FSM-based System Model

As we have already mentioned, a stable control state causes the observable state of the system to "freeze". Mathematically, stable control states can be thought of as temporal fixed-points. Metaphorically they are stepping stones across the river of temporal complexity and as such can be used to limit the scope of temporal consideration during software construction and analysis.

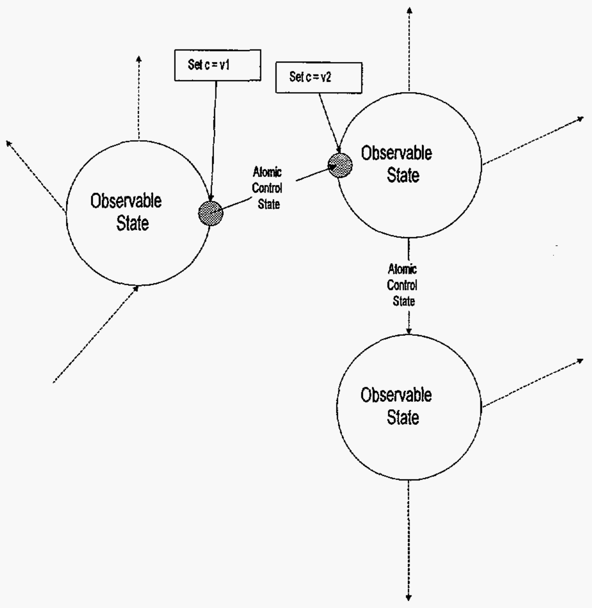
Definition 5 ***minimal***$(\vec{c}, \vec{m}) = ([\vec{m}{:}\,\vec{c}] \doteqdot \vec{m}) \wedge \neg(\exists \vec{c_2}{:}\ [\vec{m}{:}\vec{c_2}] \doteqdot \vec{m}) \wedge (\mathcal{E}(\vec{c_2}) < \mathcal{E}(\vec{c}))$

At this point, our objective is to define a *move* on an observable state, $\vec{m}$, as a control tuple $(\vec{c_1}, \vec{c_2})$ where $\vec{c_2}$ is a stable control state that is minimal with respect to the observable state $\vec{m_2}$, and $[\vec{m}{:}\vec{c_1}] \doteqdot \vec{m_2}$ where $\vec{c_1}$ is an atomic control state. Given the observable state $\vec{m}$, the *move* $(\vec{c_1}, \vec{c_2})$ defines the system-controller behavior:

$$< ((\vec{m}, \vec{c_1})), ((\vec{m_2}, \vec{c_2})) > \text{ where } [\vec{m}{:}\vec{c_1}] \doteqdot \ \vec{m_2}$$

We can further simplify this model by implicitly assuming that at this stage in the software development process, every atomic control state will be followed by a minimal control state. Thus an atomic control state is simply a transition from one stable control state to another. Furthermore, since stable control states are functions of the observable state, we simply use the observable state $\vec{m}$ to denote the state $(\vec{m}, \vec{c_2})$ where $\vec{c_2}$ is a stable control state. At this point we have an FSM-based[3] model of our system that is well suited to deductive synthesis and transformation.

[3]We say "FSM-based" model because it might be desirable to model system properties like *liveness* in terms of virtual monitored sensors over infinite domains. Such a model would not be a pure FSM.



An FSM-based System Model

6 Algorithm Synthesis

At this point we have system model that elegantly supports an incremental approach to algorithm construction. This particular incremental algorithm framework is appealing because algorithm construction can be realized using deductive synthesis techniques.

Given a (non-algorithmic) specification of a set of *system behaviors*, deductive synthesis can be used as a technique to "discover" an abstract algorithm satisfying the specification. More precisely, we are interested in using deductive synthesis to compose a sequence of *moves* such that the resulting system behavior is an element of the set of behaviors that is described by the specification.

We would like to mention that from a correctness standpoint, deductive synthesis is attractive for algorithm construction because (1) it can be automated, and (2) since this form of synthesis creates an algorithm using sound reasoning steps, it produces correct-by-construction algorithms.

In spite of the attractiveness of deductive synthesis from a correctness perspective, in practice the number of *moves* that are possible in a system model can present a major obstacle to effective use of deductive synthesis. Theory resolution [9][12] and abstraction[3] are two complementary and promising techniques that can be tailored to specific problem domains to greatly enhance the capabilities of deductive synthesis.

We are presently developing a theory for constructing an abstraction hierarchy suitable for our formalization of single-agent reactive systems. We envision a hierarchy where an algorithm at one level of abstraction can be utilized to assist a deductive synthesis engine in constructing a consistent algorithm at a lower level of abstraction. From a correctness standpoint, a very attractive feature of this hierarchy is that it is created in a formal framework and **not** as part of the formalization process. Note that a traditional top-down or object oriented approach tends to encourage the construction of a similar abstraction hierarchy as part of the formalization process itself. Such a construction cannot be recommended for high consequence software construction because it falls outside of the scope of verification.

7 Transformation

After an abstract algorithm has been synthesized, transformations can applied with the purpose of optimizing controller behaviors. An brief example of this is given in the appendix. For the general synchronous modeling paradigm that we have presented in this paper, we have discovered numerous problem independent theorems. These theorems encapsulate general model knowledge sufficient to enable automated verification of the correctness of numerous optimizing transformations. To implement these transformations we have developed a language independent, syntax-derivation-tree based transformation system called HATS [15] which can apply transformations in a fully automated fashion. Additionally, in [14] we develop a general framework for proving the correctness of transformations that introduce implementation level constructs.

8 Summary

In this paper we have presented a synchronous paradigm for modeling deterministic single-agent reactive systems. Systems are modeled in terms of a vector of monitored variables and controlled variables. The monitored variables are extended with virtual sensors to capture the information provided by the trace of the system. Through these extensions, the monitored variables describe the observable state space of the system.

Next we accepted a synchronous hypothesis as a proof obligation for the model. This allowed the abstraction of time as a discrete quantity represented by ticks of a universal clock.

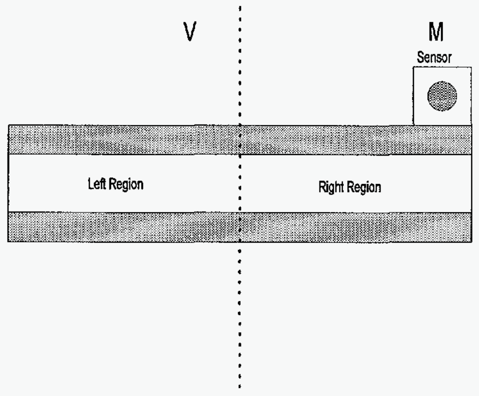
We then argued that strictly reactive systems must have a stable state(s) for every observable state. This lead to further simplifications that resulted in an FSM-based model of the system. We then briefly discussed the issues surrounding the use of deductive synthesis and transformation as a means to construct system behaviors based on these models.

In closing we would like to mention that the development of the model presented in this article forms a cornerstone of a formal method called AST[4] that is being developed within the High Integrity Software (HIS) program at Sandia National Laboratories. The goal in AST [15] is to develop a software construction methodology based on **A**bstraction, (deductive) **S**ynthesis, and **T**ransformation (hence the acronym AST) that can be used to construct ultra high-assurance software.

Appendix A: A Simple Example

Consider a system that consists of a single conveyor belt. In this system, issuing the command *add_object* will cause an object to be placed on the left end of the conveyor. This conveyor belt has a motor that can be turned on and off. When this motor is turned on, the conveyor belt can transport an object from the left end of the conveyor to the right end. In addition, there is a photoelectric sensor that can sense when an object reaches the right end of the conveyor. Furthermore, let us assume that the conveyor belt is long enough so that two objects can fit on the conveyor with enough space between them so that the photoelectric sensor can distinguish both objects as they pass off the right end of the conveyor (e.g., the moment the photoelectric sensor detects an object another object can safely be placed on the left end of the conveyor via the *add_object* command).

[4]see *http://www.sandia.gov/ast/*



A Conveyor Belt

The standard model of this system consists of a single boolean-valued monitored variable (the photoelectric sensor) and two controlled variables (the motor, and the variable associated with the *add_object* command).

Note that the monitored variable does not capture the observable state space of the system. In particular, the monitored variable cannot determine when the left side of the conveyor has an object on it. However, given an historical trace of the system together with the initial state of the system, it is possible to determine whether the left side of the conveyor has on object on it. This implicit state information can be modeled by creating a boolean-valued virtual sensor v that stores the information explicitly. When taken together the monitored variables m and v now can distinguish all states in the observable state space of the system. This gives us the following model of the system:

$$\begin{aligned} \text{controlled variables} &= (c_1, c_2) \\ \text{monitored variables} &= (v, m) \end{aligned}$$

$$\begin{aligned} c_1 &= \{\textit{motor\_on, motor\_off}\} \\ c_2 &= \{\textit{add\_object, do\_nothing}\} \end{aligned}$$

$$\begin{aligned} v &= \{\textit{object\_in\_left\_region, object\_not\_in\_left\_region}\} \\ m &= \{\textit{object\_in\_right\_region, object\_not\_in\_right\_region}\} \end{aligned}$$

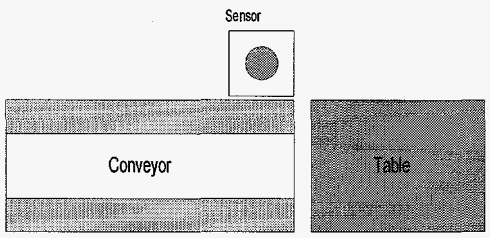
At this point it is worth asking the question: "If the conveyor was long enough to hold 3 objects would we need to introduce another virtual sensor into the vector of monitored variables?" The answer is no! The reason being that, given a single sensor, the trace alone cannot provide enough information to enable a controller to safely introduce three objects onto the conveyor. To accomplish this real-time considerations must be taken into account — which would violate our assumption that the system is (strictly) reactive.

Appendix B: A Simple Synthesis and Transformation Example

In this section we give an overview of how a simple robotic cell can be modeled, and specified and how this formalization can be used to construct an abstract controller algorithm.

Informal System Description

The robotic cell in this example consists of a conveyor belt (like the kind described in Appendix A) and an elevating-rotating table. The table can be in two rotational positions: a left position and a right position. Setting the rotational-controlled variable to turn_left/turn_right will cause the table to rotate into its left/right position. The rotational motor can also be stopped. In addition, the table can also be in two elevational positions: top and bottom. Setting the elevation-controlled variable to table_up/table_down will cause the table to elevate to the top/bottom position. The elevational motor can also be stopped. There is a sensor that senses if the table is in the right or left position. There is a sensor that senses if the table is in the bottom or top position.



A simple robotic cell

An object can be transferred from the conveyor to the table by placing the table in its lower left position and passing an object through the photoelectric sensor. That is, when an object is sensed by the photoelectric sensor this implies that the object is in the right region of the conveyor belts observable state space. If such a state is reached and the table is in the lower left position, then if the conveyor is turned on and the photoelectric cell no longer senses the object, we can conclude that the object is now on the table. This of course assumes that the table does not move away from the lower left position at any time during the exchange process. Note that, as we mentioned in the beginning of this paper, expressing this assumption can be awkward in non-synchronous paradigms.

Finally, if the table has an object on it and is then moved to its upper right position we assume that the object is *consumed* (i.e., it leaves the system).

Informal Specification

We would like a controller to orchestrate the behavior of this cell so that the cell will "consume" an infinite number of objects.

Synchronous Model

This system has four controlled variables.

$$\begin{array}{lcll} C_1 & = & \{\textit{do\_nothing, add\_object}\} & = \{0,1\} \\ C_2 & = & \{\textit{ motor\_off, motor\_on}\} & = \{0,1\} \\ C_3 & = & \{\textit{stop\_turn, left\_turn, right\_turn}\} & = \{0,1,2\} \\ C_4 & = & \{\textit{stop\_elevate, table\_down, table\_up}\} & = \{0,1,2\} \end{array}$$

This system has the following observable state space.

$$\begin{array}{lcll}
D_1 & = & \{\mathit{object\_not\_in\_left\_region,\ object\_in\_left\_region}\} & = \{0,1\} \\
D_2 & = & \{\mathit{object\_not\_in\_right\_region,\ object\_in\_right\_region}\} & = \{0,1\} \\
D_3 & = & \{\mathit{left\_position,\ right\_position}\} = & = \{0,1\} \\
D_4 & = & \{\mathit{bottom\_position, top\_position}\} & = \{0,1\} \\
D_5 & = & \{\mathit{object\_not\_on\_table,\ object\_on\_table}\} & = \{0,1\}
\end{array}$$

Let $\vec{c} = (a,b,c,d,e)$ denote the vector of controlled variables and let $\vec{m} = (v,w,x,y,z)$ denote the vector of monitored variables whose elements are quantified over the domains C_1, C_2, ..., C_4 and D_1, D_2, ..., D_5 respectively. For human readability we might want to consider rewriting the controlled and monitored vectors as follows:

- $\vec{c} =$ **control_state**($\mathit{conveyor}(a,b)$, $\mathit{table}(c,d)$)
- $\vec{m} =$ **observable_state**($\mathit{conveyor}(v,w)$, $\mathit{table}(x,y,z)$)

The Atomic Transition Set

For space considerations we drop human readable state annotations in this section. The transitions given here are by no means complete. In particular, the failure space of the system is omitted. The notation that we are using to describe the atomic transition set is an exhaustive enumeration where states are described by very simple predicates. Predicates so simple, that unification alone enables one to determine whether a concrete state matches a specific transition definition. Note that this simple (and verbose) notation is generally not practical for modeling larger state spaces.

$$\begin{array}{lcl}
[(v,w,x,y,z) : (\mathit{do\_nothing},0,0,0)] & \doteqdot & (v,w,x,y,z) \\
[(0,w,x,y,z) : (\mathit{add\_object},0,0,0)] & \doteqdot & (1,w,x,y,z)
\end{array}$$

$$\begin{array}{lcl}
[(v,w,x,y,z) : (0,\mathit{motor\_off},0,0)] & \doteqdot & (v,w,x,y,z) \\
[(0,0,x,y,z) : (0,\mathit{motor\_on},0,0)] & \doteqdot & (0,0,x,y,z) \\
[(0,1,0,0,0) : (0,\mathit{motor\_on},0,0)] & \doteqdot & (0,0,0,0,1) \\
[(1,0,x,y,z) : (0,\mathit{motor\_on},0,0)] & \doteqdot & (0,1,x,y,z) \\
[(1,1,0,0,0) : (0,\mathit{motor\_on},0,0)] & \doteqdot & (0,1,0,0,1)
\end{array}$$

$$\begin{array}{lcl}
[(v,w,x,y,z) : (0,0,\ \mathit{stop\_turn},0)] & \doteqdot & (v,w,x,y,z) \\
[(v,w,1,y,z) : (0,0,\mathit{left\_turn},0)] & \doteqdot & (v,w,0,y,z) \\
[(v,w,0,1,1) : (0,0,\ \mathit{right\_turn},\ 0)] & \doteqdot & (v,w,1,1,0) \\
[(v,w,0,1,0) : (0,0,\mathit{right\_turn},\ 0)] & \doteqdot & (v,w,1,1,0) \\
[(v,w,0,0,0) : (0,0,\ \mathit{right\_turn},\ 0)] & \doteqdot & (v,w,1,0,0) \\
[(v,w,0,0,1) : (0,0,\ \mathit{right\_turn},\ 0)] & \doteqdot & (v,w,1,0,1)
\end{array}$$

$$\begin{array}{lcl}
[(v,w,x,y,z) : (0,0,0,\ \mathit{stop\_elevate})] & \doteqdot & (v,w,x,y,z) \\
[(v,w,x,1,z) : (0,0,0,\mathit{table\_down})] & \doteqdot & (v,w,x,0,z) \\
[(v,w,1,0,1) : (0,0,0,\ \mathit{table\_up})] & \doteqdot & (v,w,1,1,0) \\
[(v,w,1,0,0) : (0,0,0,\mathit{right\_turn})] & \doteqdot & (v,w,1,1,0) \\
[(v,w,0,0,0) : (0,0,0,\ \mathit{right\_turn})] & \doteqdot & (v,w,0,1,0) \\
[(v,w,0,0,1) : (0,0,0,\ \mathit{right\_turn})] & \doteqdot & (v,w,0,1,1)
\end{array}$$

Specification

Let $<\vec{m}_1, \vec{m}_2>$ specify a system behavior in terms of its pre and postcondition [10]. Furthermore, let $b_1; b_2$ denote the sequential composition of behaviors b_1 and b_2. Then assuming that the table initially does not hold an object, the following recursively defined system behavior equation formally specifies the behavior we want the controller to satisfy:

$$
\begin{aligned}
spec \; = \; & < \mathbf{observable\_state}(conveyor(v_1, w_1), table(\mathbf{object\_not\_on\_table}, y_1, z_1)), \\
& \mathbf{observable\_state}(conveyor(v_2, w_2), table(\mathbf{object\_on\_table}, y_2, z_2)) > \\
& ; \\
& < \mathbf{observable\_state}(conveyor(v_2, w_2), table(\mathbf{object\_on\_table}, y_2, z_2)), \\
& \mathbf{observable\_state}(conveyor(v_1, w_1), table(\mathbf{object\_not\_on\_table}, y_1, z_1)) > \\
& ; \\
& spec
\end{aligned}
$$

Note that the object of specification is to describe the set of all possible system behaviors that would be considered acceptable.

An Abstract Controller

Given the initial state **observable_state**(*conveyor*(0, 0),*table*(0, 0, 0)) the following abstract controller can be synthesized:

```
spec =  add_object;    /* place an object on the left end of the conveyor */
        motor_on;      /* move the object to the right end of the conveyor */
        motor_on;      /* transfer the object to the table */
        table_right;
        table_up;      /* consume the object */
        table_left;
        table_down;    /* table is ready for next object transfer */
        spec           /* repeat */
```

Note that in the algorithm above, we are denoting atomic control states by giving the value (e.g., add_object) of the element of the control vector that makes the control state non-passive. The algorithm above describes an infinite path through the FSM-based model of the system, and since it was constructed using deductive synthesis the algorithm is correct by construction.

We now can apply correctness preserving transformations to produce the following optimized abstract algorithm:

```
spec =  (((table_right || table_up) ; (table_left || table_down)) || (add_blank ; motor_on)) ;
        motor_on ; spec
```

Where the "||" denotes parallel composition and parenthesis denote the scope of expressions.

References

[1] F. Boussinot and R. De Simone. *The ESTEREL Language.* Proceedings of the IEEE, Vol. 79, No. 9, September 1991.

[2] J. M. Boyle, R. D. Resler, and V. L. Winter. *Do You Trust Your Compiler? Applying Formal Methods to Constructing High-Assurance Compilers.* Proceedings of the IEEE High-Assurance Systems Engineering Workshop, 1997.

[3] F. Giunchiglia and A. Villafiorita. *ABSFOL: a proof checker with abstraction.* Proceedings of the 13th International Conference on Automated Deduction, 1996.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. *The Synchronous Data Flow Programming Language LUSTRE.* Proceedings of the IEEE, Vol. 79, No. 9, September 1991.

[5] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. *Tabular Representations in Relational Documents.* Relational Methods in Computer Science, C. Brink & G. (eds.) in cooperation with R. Albrecht, Springer-Verlag, 1996.

[6] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Series in Computer Science, 1994.

[7] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science Vol. 891, Springer-Verlag.

[8] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. *AMPHION: Automatic Programming for Scientific Subroutine Libraries.*

[9] M. Stickel. *Automated Deduction by Theory Resolution.* Automated Reasoning, Vol. 1, 1985, pp. 333-355.

[10] Carroll Morgan. *Programming from Specifications.* Prentice Hall International Series in Computer Science, 1990.

[11] M. Stickel, R. Waldinger, M. Lowery, T. Pressburger, and I. Underwood. *Deductive Composition of Astronomical Software from Subroutine Libraries.*

[12] J. Van Baalen. *Automated design of specialized representations.* Artificial Intelligence, Vol. 54, pp 121-198, 1992.

[13] Veroff, R. and Wos, L. *The Linked Inference Principle, I: The Formal Treatment.* Journal of Automated Reasoning, Vol. 8, no. 2, pp. 213–274 (1992).

[14] V. L. Winter and J. M. Boyle. *Proving Refinement Transformations for Deriving High-Assurance Software.* Proceedings of the IEEE High-Assurance Systems Engineering Workshop, 1996.

[15] V. L. Winter. *Software Construction via Abstraction, Synthesis, and Transformation (AST).* Proceedings of the IEEE High Integrity Software Conference, 1997.