PERGAMON

Annual Reviews in Control 26 (2002) 175–187

Annual Reviews in Control

www.elsevier.com/locate/arcontrol

CONTROL FROM COMPUTER SCIENCE[1]

Oded Maler

* CNRS-VERIMAG, *2, av. de Vignate, 38610, Gières, France.*
Oded.Maler@imag.fr
www-verimag.imag.fr/~maler/

Abstract: This paper presents some of the principles underlying verification and controller synthesis techniques for discrete dynamical systems developed within Computer Science along with some ideas to extend them to continuous and hybrid systems. Hopefully, this will provide control theorists and engineers with an additional perspective of their discipline as seen by a sympathetic outsider, uncommitted to the customs and traditions of the domain. Inter-cultural experience can be frustrating but sometimes fun.

1. WHAT AM I DOING HERE?

Being one of those who have chosen to study computer science partly due to an inability to understand differential equations, I feel a bit uncomfortable to publish a paper in a journal whose pages are probably full of occurrences of that terrifying $\int$ symbol. The scientific reason for my presence here is perhaps being one of those few computer scientists interested in the so-called *hybrid systems* research which was supposed to bring together the Computer Science and Control communities. So let me first speak about what I understand.

2. WHAT IS VERIFICATION?

Verification[2] like Control is concerned with a model-based design of systems. That is, we want to build something (a "controller") that makes some part of the real world (the "environment" or "plant") behave in a certain desired way. Instead of using trial-and-error methods we build a *mathematical model* which describes the combined dynamics of the environment and the controller. On this model we can make "gedanken experiments", e.g. manipulation of formulae or numerical simulations, to convince ourselves that the controller indeed makes the environment behave as required. If the model is a good approximation of the real world, there is a chance that a controller validated on the model will work properly when implemented.[3]

The description just given does not specify the type of dynamical models considered. In classical control these are models of continuous dynamical systems in either continuous or discrete time, and since examples of such systems appear in every decent control textbook, I will move directly to discrete systems of the type treated by the verification community and illustrate them via an example.

2.1 *The Coffee Machine*

Suppose we want to build a machine that distributes various hot drinks to customers who pay

[1] This research was supported in part by the European Community projects 26270 VHS (Verification of Hybrid Systems) and IST-2001-33520 CC (Control and Computation).

[2] The term "verification" is used as a short approximation for the disciplines and communities interested in "modeling, design and analysis of reactive systems" or "formal methods in system design". This is not "the" mainstream in Computer Science — I am not sure there is a mainstream in such a diverse domain.

[3] I mention this trivial fact because mathematicians, discrete and continuous alike, who spend much of their time in the abstract world, sometimes tend to forget it.

1367-5788/02/$20 © 2002 Published by Elsevier Science Ltd.
PII: S1367-5788(02)00030-5

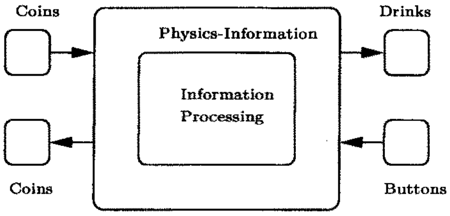

Fig. 1. The machine and its physical interface.

for them by inserting coins. Much of the interaction of the machine with its external environment is physical: users insert coins and press buttons and the machine heats water, mixes it with certain ingredients and releases plastic cups filled with the appropriate drink. In modern systems it is customary to decompose systems into two parts, the *physical interface* and the *information processing* component. The physical interface takes care of the transduction between energies of various forms and electronic signals. In our example it includes *sensors* that detect the pressing of a button or recognize the inserted coins, as well as *actuators* that do the opposite transformation and implement the "decisions" of the machine to heat water by turning a heater on or to release a cup by, say, a pneumatic device. When we remove this envelope we obtain the second component, the information processing system, a system that processes *information signals* regardless of the type of physical entities they represent.

Digression: Since information processing is perhaps the most important common aspect of control and computer science it is worth elaborating a bit about it. We can write a reactive computer program which responds to an input event a by an output event b. Only the connection of the computer I/O ports to sensors and actuators will give an external physical meaning to the symbols a and b and to the I/O relation defined by the program: e.g. "respond to a mouse click by starting to play a CD" or "respond to a pressed button by launching a missile". Similarly in the continuous world the same servo mechanism can be plugged into a temperature sensor and a furnace to regulate temperature, and equally well — to a velocity sensor and a motor to regulate speed. The essence in both types of systems is a *mathematical* relationship between input and output signals whose external physical meaning is defined by the physical interface of the system. For the information processing system the world consists of discrete or continuous signals at its I/O ports, realized by low-energy electricity.

In the past, the distinction between physics and information was not so sharp. For example, in Watt's governor the information about the rotational velocity was "transmitted" to the controller mechanically. Similarly, today when we press the throttle or the brakes of our car we still represent the instructions that we give to the car ("faster", "slower") by physical magnitudes which are just amplified along the way from the pedal downwards. In the near future, however, using drive-by-wire techniques, the distinction will become more apparent.[4] **End of Digression.**

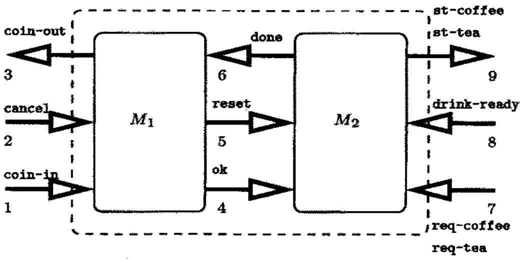

Fig. 2. The information-processing component of the machine.

From now on we restrict our attention to the information processing sub-system M of the machine and denote by E its environment, i.e. its physical interface. For simplicity we assume that there is only one type of a coin and two choices of drinks, coffee and tea, each costs one coin (the reader can make the exercise of extending the example to more complicated machines) and that there is a button for canceling the operation. We decompose M further into two sub-machines M_1 and M_2, the first interacts with the coin collection apparatus and the second with the choice and preparation of drinks. In addition to the interaction with the physical interface, the two machines should communicate: M_1 should inform M_2 about the reception of the required amount of money, while M_2 should tell M_1 that drink delivery has been accomplished. A block diagram of the machines appears in Figure 2. The transfer of information between the components is done via 9 communication ports described in the following table.

Port	From→To	Event types	Meaning
1	$E \to M_1$	coin-in	a coin was inserted
2	$E \to M_1$	cancel	cancel button pressed
3	$M_1 \to E$	coin-out	release the coin
4	$M_1 \to M_2$	ok	sufficient money inserted
5	$M_1 \to M_2$	reset	money returned to user
6	$M_2 \to M_1$	done	drink distribution ended
7	$E \to M_2$	req-coffee req-tea	coffee button pressed tea button pressed
8	$E \to M_2$	drink-ready	drink preparation ended
9	$M_2 \to E$	st-coffee st-tea	start preparing coffee start preparing tea

The dynamics of the two machines is depicted in Figure 3 using the formalism of *automata*, also known as *finite-state machines.*[5] Devices having

[4] This corresponds to the appearance of specialized nerve cells in living organisms. It may correspond to many other phenomena such as language, communication networks, etc. that take us further away from physics/geometry to information.

[5] In the sequel we will use also the terms *discrete dynamical systems* and *transition systems* for talking about the same objects.

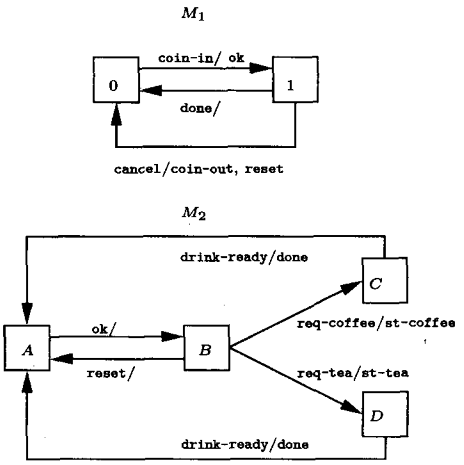

Fig. 3. The two machines M_1 and M_2.

several states, and which move from one state to another upon the occurrence of certain events, have become part of the daily life of people living in the beginning of the 21st century. Almost every one of us has experienced such machines while withdrawing cash, setting a digital clock or interacting with a graphical or vocal menu systems.

Machine M_1 has 2 states. In the initial state 0 it ignores all events except `coin-in` and upon its reception it moves to state 1 while emitting `ok`. This indicates to machine M_2 that the right amount of money has been inserted. Machine M_1 returns to state 0 upon receiving the signal `done` from machine M_2 (this means that the whole procedure is over and the machine is ready to accept money from the next customer). In addition, if the customer presses the `cancel` button while at state 1, the machine moves back to state 0 and emits two events: `coin-out` to release the money, and `reset` which returns machine M_2 into its initial state as well.

Machine M_2 stays in its initial state A and ignores all inputs as long as it does not receive the `ok` event from M_1. Once it receives the `ok` it moves to state B, and from there, upon reception of event `req-coffee` (resp. `req-tea`), it moves to state C (resp. D) and emits the event `st-coffee` (resp. `st-tea`) which initiates the physical process of preparing the respective drink. Upon receiving the event `drink-ready` from the preparation machine, machine M_2 moves from C or D back to A while sending the event `done` to M_1.

The transition arcs between states are sometimes labeled by `input/output` actions. This means that the machine in question can perform the transition only if it recieves `input` from its outside environment (which may include other machines) and while doing so it emits `output`. This is the way one machine (or the external evironment) can

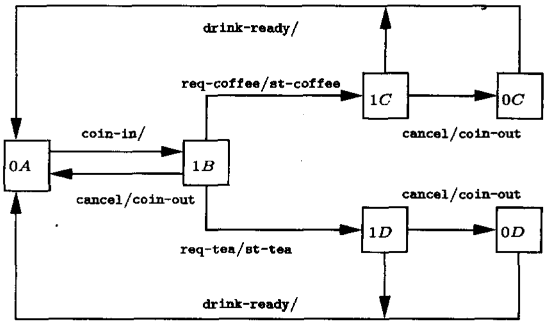

Fig. 4. The machine $M = M_1||M_2$.

influence the behavior of another machine. For example, M_1 can move from 0 to 1 only upon the reception of `coin-in` from E. Similarly it can move from 1 to 0 only if either it receives `done` from M_2 or `cancel` from E. Such means of coordinating the behaviors of several machines are called *synchronization* mechanisms.

When two or more machines are working together, they constitute a global system whose states are tuples consisting of the local states of each machine. For example, the composition[6] of M_1 and M_2, denoted by $M = M_1||M_2$, is an automaton whose initial state is $0A$. An automaton can move from one global state to another if all its components can take their corresponding local transitions. For example, M can move from $0A$ to $1B$ upon receiving `coin-in` because M_1 can move from 0 to 1 while emitting `ok` which makes M_2 move to B. For this reason, a global transition from $0A$ to $0B$ is impossible. Machine M appears in Figure 4, and by looking at it we can see paths that correspond to potential behaviors. For example the path

$0A$ `coin-in` $1B$ `cancel coin-out` $0A$

corresponds to a customer who changed his mind and got his money back. Similarly, the path

$0A$ `coin-in` $1B$ `req-coffee st-coffee`
$1C$ `drink-ready` $0A$

represents a full cycle of the normal operation of the machines. But looking at the state-transition graph we can see also unexpected behaviors. For example, what happens if the user enters `coin-in`,

[6] There are many forms of composition differing from each other in the form of interaction between components (via states or via events), in the nature of global time (synchronous or asynchronous) etc. Since I don't give here a formal definition of synchronization mechanisms and of composition, there are some imprecisions in the example which can be discovered by readers who try to build the product — a recommended activity by itself. They can be fixed by making the model a bit more complex and less intuitive. Note also that internal events such as ok that serve only for the interaction between M_1 and M_2, are not visible to the outside world and hence do not appear as labels on the transitions of $M_1||M_2$.

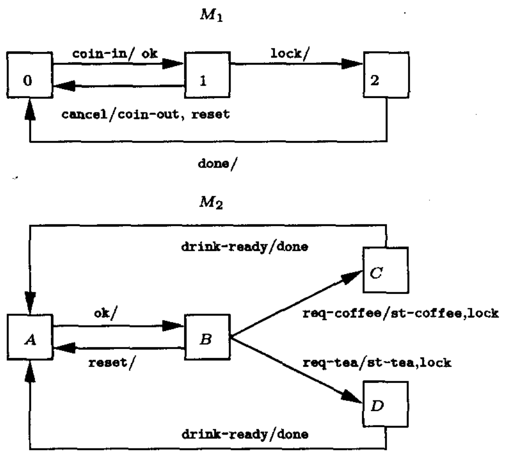

Fig. 5. The two machines M_1 and M_2 after fixing the bug.

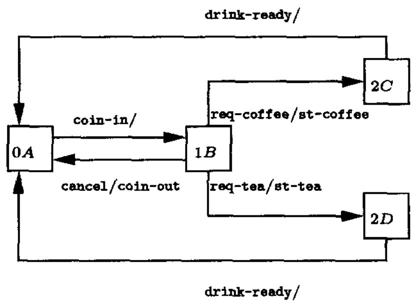

Fig. 6. The well-behaving product of M_1 and M_2.

then **req-coffee** and then, before the arrival of **drink-ready**, she pushes the **cancel** button? According to the path

$0A$ **coin-in** $1B$ **req-coffee st-coffee** $1C$ **cancel coin-out** $0C$ **drink-ready** $0A$

the machine will move to state $0C$ and the user will get the money back while the process initiated by **st-coffee** keeps on going. This bug can be quite unpleasant to the machine owner and its existence cannot be inferred immediately from by looking at the two machines separately. Imagine how hard it is to find such bugs in large systems composed of many interacting machines and whose sets of behaviors consist of huge numbers of very long sequences of events.

In order to fix the bug we add a new state 2 to machine M_1 (Figure 5). This is a "no-return" state into which M_1 enters upon receiving a **lock** message from M_2 after the user has selected the drink and the prepartion has started. In Figure 6 we can see the global system which, indeed, generates only acceptable behaviors.

The moral of this story is summarized as follows:

(1) There are numerous systems of practical interest that can be modeled as a product of many interacting discrete components. The global model for such a system is a finite but, possibly, very large automaton.
(2) The set of all possible behaviors of such a system, in the presence of all admissibe input sequences, is represented by paths in the global transtion graph.
(3) The desired behavior of such a system can be specified as a set of allowed sequences of states and events.
(4) Proving that the system is correct amounts to showing that all sequnces generated by the system are those allowed by the specification.

3. DISCRETE SYSTEMS

In this section I will present in a semi-formal manner some of the "systems theory" for discrete systems, especially those parts motivated by solving (4) above. Interested readers can consult books such as McMillan (1993); Kurshan (1994); Manna and Pnueli (1995); Clarke *et al.* (1999). I will consider three models of discrete systems which correspond roughly to the notions of *simulation*, *verification* and controller *synthesis*. At the first level of modeling we will consider closed systems such that given an initial state x_0, the state of the system is *determined* for every time t. At the second level, we add an input domain V, affecting the dynamics of the system. We interpret this domain as uncontrollable inputs (disturbances) to the system, i.e. influences coming from the external environment. Finally, at the third level of modeling we consider an additional input domain U, corresponding to the controller's actions. A system with two inputs can be seen as a two-person game where controller synthesis amounts to finding a winning strategy.

While I tell the discrete side of the story, the reader is aksed to think about the possible analogies with continuous systems, analogies that will be made explicit later (see also Maler (1998)).

3.1 *Model I: Closed Systems*

We start with systems not exposed to external influences and hence their future evolution depends exclusively on their current state.

Definition 1. (System D-I). A transition system is $S = (X, \delta)$ where X is a finite set and $\delta : X \to X$ is the transition function.

The state-space X of the system is usually a set without any additional structure, i.e. it does not admit metric or order. We keep in mind that it might be a Cartesian product of several domains

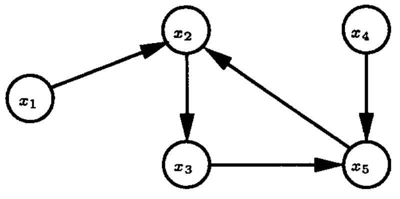

Fig. 7. A deterministic automaton.

but we do not take this fact into consideration. We use X^* to denote the set of all sequences (finite or infinite) over X and X^k for seqeunces of length k. The concatenation of two sequences $\xi_1, \xi_2 \in X^*$ is denoted by $\xi_1 \cdot \xi_2$. Automata are presented as directed graphs with states as nodes and with edges of the form (x, x') whenever $x' = \delta(x)$ (see Figure 7). We stress again that the embedding of this graph on the two-dimensional page is arbitrary and does not carry any geometrical meaning (unlike phase-portraits of continuous systems).

Definition 2. (Behavior). Given a system $S = (X, \delta)$, the behavior of S starting from an initial state $x_0 \in X$, is a sequence

$$\xi = \xi[0], \xi[1], \ldots \in X^*$$

such that $\xi[0] = x_0$ and for every i,

$$\xi[i+1] = \delta(\xi[i]).$$

Given a description of a dynamical system, the most natural thing to ask is how it will behave starting from some initial state. In many cases, we are particularly interested in reaching or avoiding a certain set of states. More complex desired properties of the system may involve the presence or absence of certain patterns in the set of system behaviors, for example, "after a there is no b until c". It is also possible to express properties concerning the behavior of a system extended to infinity "the pattern a after b repeats infinitely many times". Such properties can be expressed using various formalisms such as temporal logic, regular expressions or automata. I will restrict the discussion to the simplest of all properties, namely that the system never reaches a set P of "bad" states.

Definition 3. (Basic Reachability Problem). The basic reachability problem for a system S is: given x_0 and a set $P \subseteq X$, does the behavior of S starting at x_0 reach P? In other words: does there exist a time t such that $\xi[t] \in P$?.

For deterministic finite automata the problem appears to be trivial (just look at the automaton) but the reader should remember that, as in the coffee-machine example, S is not given *explicitly* but as a product of interacting automata, a description from which the answer cannot be derived just by visual inspection. The following simple algorithm solves this problem by computing all the states reachable from x_0. In fact, it is nothing but a "simulation" of the (single) behavior of S starting from x_0, combined with memorization of the visited states. The algorithm produces a set F_* consisting of all states reachable from x_0. This set can then be tested for intersection with P.

Algorithm 1. (Forward Simulation/Reachability).

```
$\xi[0] := x_0$
$F^0 := \{x_0\}$
**repeat**
  $\xi[k+1] := \delta(\xi[k])$
  $F^{k+1} := F^k \cup \{\xi[i+1]\}$
**until** $F^{k+1} = F^k$
$F_* := F^k$
```

For finite-state deterministic systems every behavior is ultimately-periodic, i.e. a sequence that can be written as $r \cdot s^\omega = r \cdot s \cdot s \cdots$ where r and s are finite sequences denoting, respectively, the *prefix* and the *period* of ξ. For the automaton of Figure 7, the behavior starting from x_1 is $x_1 \cdot (x_2 x_3 x_5)^\omega$ and the algorithm produces the sequence of sets

$$\{x_1\}, \{x_1, x_2\}, \{x_1, x_2, x_3\}, \{x_1, x_2, x_3, x_5\}.$$

Since $\delta(x_5) = x_2$, the next iteration does not add new states and the algorithm terminates.

Algorithm 1 solves the reachability problem by *forward* simulation. Alternatively we could start from P and go *backward* to determine all the states from which the system can reach P (a kind of "domain of attraction"). Going backwards may introduce non-determinism and we will discuss it in the next section. Note that unlike systems defined by differential equations, discrete transition systems are rarely reverse-deterministic (going backwards from x_5 you can reach both x_3 and x_4).

Finiteness plays an important role in this setting: the transition function, the set P, and the sets F^k of reachable states accumulated during the simulation can all be enumerated explicitly and can be stored in finite data-structures such as tables or lists. Finiteness also guarantees the termination of the algorithm. If we relax the finiteness condition and allow a *countable* state-space the above does not hold anymore. A discussion of infinite-state systems appears in the next section.

The analog problem for continuous systems would be to check whether the solution of the differential equation $\dot{x} = f(x)$ starting from x_0 intersects with some given subset $P \subseteq \mathbb{R}^n$. This subset can be, for example, a polyhedron or an ellipsoid. Note that we do not restrict the question to the *limit*

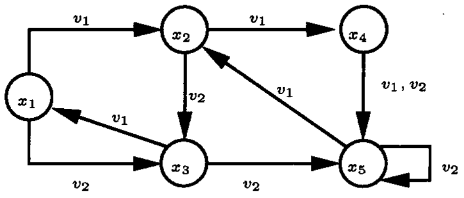

Fig. 8. An automaton with input.

behavior but ask also about *transient* states. A discrete-time version of this problem is concerned with the sequence $\xi[0], \xi[1], \ldots$ such that $\xi[0] = x_0$ and $\xi[k+1] = g(\xi[k])$ where g is a function from $\mathbb{R}^n$ to itself.

3.2 *Model II: Systems with One Input*

Definition 4. (System D-II). A one-input transition system is $S = (X, V, \delta)$ where X and V are finite sets and $\delta : X \times V \to X$ is the transition function.

The evolution of a type II system starting from a state depends on the external influence of the input. For example, in the system of Figure 8 $\delta(x_1, v_1) = x_2$ while $\delta(x_1, v_2) = x_3$. Hence there is not *one* behavior starting from any given state but rather a set of behaviors, each associated with an input sequence.

Definition 5. (Behavior Induced by Input). Given a system $S = (X, V, \delta)$ and an input sequence $\psi \in V^*$, the behavior of S starting from $x_0 \in X$ in the presence of ψ is a sequence

$$\xi(\psi) = \xi[0], \xi[1], \ldots \in X^*$$

such that $\xi[0] = x_0$ and for every i,

$$\xi[i+1] = \delta(\xi[i], \psi[i]).$$

In the automaton of Figure 8, an input starting with v_1, v_2, v_2, v_1, v_1 generates a behavior starting with $x_1, x_2, x_3, x_5, x_2, x_4$, a fact that can be denoted as:

$$x_1 \xrightarrow{v_1} x_2 \xrightarrow{v_2} x_3 \xrightarrow{v_2} x_5 \xrightarrow{v_1} x_2 \xrightarrow{v_1} x_4.$$

The simplest verification problem is the extension of the basic reachability problem to open systems:

Definition 6. (Reachability for Open Systems). Given a system $S = (X, V, \delta)$ and a set $P \subseteq X$, is there some input sequence $\psi \in V^*$ such that the behavior $\xi(\psi)$ reaches P?

This problem is a-priori much harder than the problem for a type I system due to the existential quantification of the infinite set of sequences.

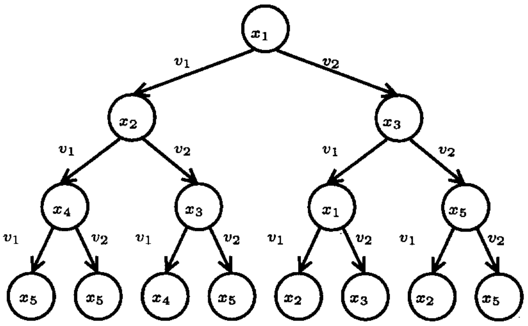

Fig. 9. An initial part of the execution tree of the type II system of Figure 8.

To understand the various approaches for solving this problem, let us look at the set of all behaviors of a type II system. This set admits a tree structure with the initial state at the root and where each branch represents the behavior induced by one input sequence (Figure 9).

When a specific sequence $\psi \in V^*$ is given, a trivial extension of Algorithm 1 can be used to simulate $\xi(\psi)$ and compute the set of states $F_*(\psi)$ it visits.[7] If we could repeat this procedure for all (infinitely-many) sequences in V^* we would have computed all reachable states. A basic result about finite automata allows us to use a finite set of sequences: In an n-state automaton, if a state is reachable from an initial state then it is reachable by a path of length smaller than n. So if we repeat Algorithm 1 with all finite sequences $\psi \in V^n$, we obtain the reachable states for *all* possible inputs as

$$F_* = \bigcup_{\psi \in V^n} F_*(\psi).$$

This exhaustive simulation technique can be seen as generating an input sequence for every branch of length n in the execution tree. Although finite, the number of such sequences is $|V|^n$ and, given that n itself might be prohibitively large (exponential in the number of system components), this option is not so attractive.

While the simulation approach is suitable for "black box" testing, it is rather wasteful when we have the structure of the automaton at our disposal. For the reachability problem we need not explore the successors of the same state more than once: since both v_2 and v_1v_2 lead to the same state x_3, we know that for every input ψ, the sequences $v_2 \cdot \psi$ and $v_1v_2 \cdot \psi$ will lead to the same state.[8] Hence we can apply more efficient search

[7] In fact, the technical story is a bit more complicated for infinite sequences, but for ultimately-periodic inputs it is known that the induced behaviors are ultimately-periodic as well.

[8] This is, in fact, the essence of the notion of a *state* in the modern theory of dynamical systems, and in particular in automata (Myhill-Nerode theorem).

algorithms to the transition graph at the price of losing some of the intuitive flavor of simulation.

To this end let us denote by $\delta(x)$ the set of all *immediate successors* of x, i.e.

$$\delta(x) = \{x' : \exists v\ \delta(x, v) = x'\}$$

and extend this notation to sets of states by letting $\delta(F) = \{\delta(x) : x \in F\}$. The following algorithm computes all reachable states of a type II system:

Algorithm 2. (Forward Reachability).

```
$F^0 := \{x_0\}$
repeat
  $F^{k+1} := F^k \cup \delta(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$
```

In essence this is a graph search algorithm and its complexity is $O(n \cdot \log n \cdot |V|)$ — much better than the simulation-based approach. This algorithm explores the transition graph in a *breadth-first* order and every F^k consists of the states reachable after at most k transitions. This can be viewed as running many simulations in parallel and aborting a simulation when it reaches a state already visited by another simulation. The set of tree nodes explored by this algorithm appears in Figure 10.

Unlike the simulation of a single behavior, when Algorithm 2 discovers that a bad state can be reached (a non-empty intersection of some F^k with P) we do not have immediately the input sequence and its induced behavior that led to P. But the extraction of such a sequence ("error trace", "counter-example") from the set sequence $\{F^j\}_{j=1..k}$ is easy to do.

One can write a depth-first variant of this algorithm which explores a branch of the tree until a previously-visited state is encountered and then backtracks ("rolling back" the simulation) and tries another branch. The behavior of a depth-first variant of the algorithm is depicted in Figure 11.

As mentioned earlier, verifying whether some behavior reaches a set P can also be done backwards. Let

$$\delta^{-1}(x) = \{x' : \exists v\ \delta(x', v) = x\}$$

be the set of *immediate predecessors* of x and let $\delta^{-1}(F)$ be its obvious extension to sets. The following algorithm computes the set of all states from which there is an input that drives the system into P. If this set includes x_0 the answer to the reachability problem is positive.

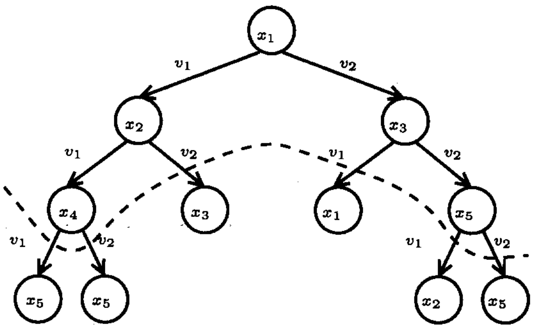

Fig. 10. Nodes explored by the forward reachability algorithm in breadth-first search regime. The dashed line indicates the frontier between the first and subsequent occurrences of states during the exploration.

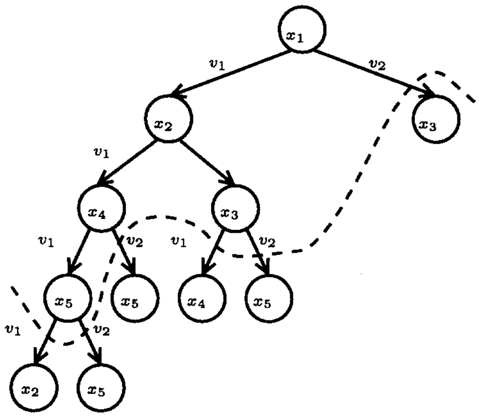

Fig. 11. Nodes explored by the forward reachability algorithm in depth-first search regime.

Algorithm 3. (Backward Reachability).

```
$F^0 := P$
repeat
  $F^{k+1} := F^k \cup \delta^{-1}(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$
```

Theorem 1. (Algorithmic Verification). There are algorithms that take a description of a type II system and verify whether any of the admissible inputs drives the system into a set P. Such algorithms always terminate after a finite number of steps.

Of course, "finite" can be very large and even too large, but the significance of this result is in its *generality*: it applies to *any* system that can be written as a product of finitely many finite automata. Variants of Algorithms 2 and 3 and their efficient implementations constitute most of what algorithmic verification is all about. A snapshot of the state-of-the-art in this domain

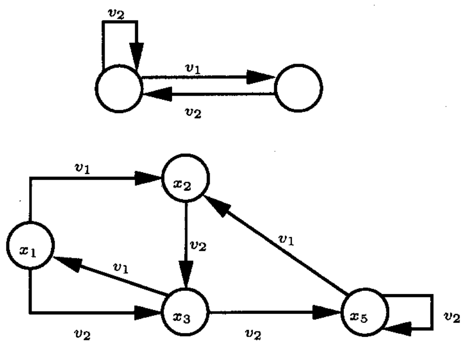

Fig. 12. A model of a restricted environment (above) and the result of composing it with the automaton of Figure 8, assuming x_1 as initial state.

can be observed in proceeding volumes such as Brinksma and Larsen (2002).

Before moving to controller synthesis let us discuss the question of *admissible inputs.* So far it was implicitly assumed that the external environment can produce any sequence in V^*. In many realistic situations the environment is constrained to follow some protocol and generate only a subset of V^*. An environment that, for example, does not produce two consecutive occurrences of v_1 can be modeled by an automaton, and the set of all behaviors in the presence of such inputs is captured by the composition of this automaton with the system (see Figure 12). In such an environment, state x_4 is not reachable from x_1. Likewise the coffee machine will never exhibit its bug in an environment where no user would press the cancel button once the coffee started pouring.

The analogous problem for continuous systems would be: given a system defined by the equation $\dot{x} = f(x, v)$ where v ranges over some set of admissible input signals, check whether there is some signal which drives the system into a set P. Recall that v stands for disturbance, not control.

3.3 *Model III: Systems with Two Inputs*

Definition 7. (System III-D). A two-input transition system is $S = (X, U, V, \delta)$ where X, U and V are finite sets and $\delta : X \times U \times V \to X$ is the transition function.

A type III system appears[9] in Figure 13. The behavior of the systems in the presence of two inputs, $\eta \in U^*$ and $\psi \in V^*$ can be characterized as before by letting $\xi(\eta, \psi)$ be a sequence satisfying

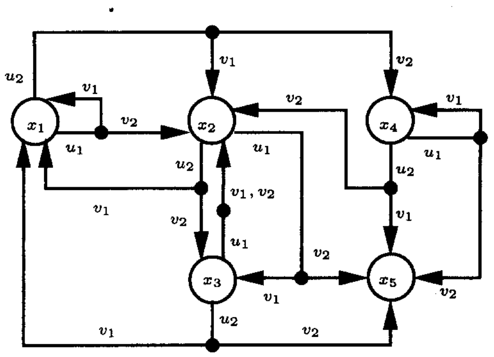

Fig. 13. A type III system with $U = \{u_1, u_2\}$ and $V = \{v_1, v_2\}$.

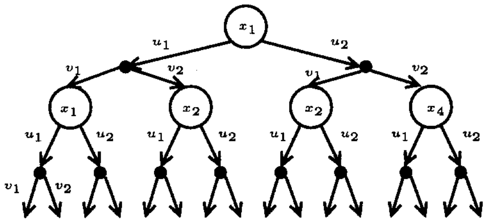

Fig. 14. The game tree for the system of Figure 13

$$\xi[i+1] = \delta(\xi[i], \eta[i], \psi[i])$$

for every i. The main novelty here is in the different interpretation we give to the two inputs. We interpret U as a set of control actions that we can select from and V as uncontrolled disturbances. This model can be viewed as a *game* between a controller U and an external environment V, each trying to steer the system toward other parts of the state-space. Our goal is to find a winning strategy, a rule that tells us which element of U to choose at every reachable situation in order to guarantee that whatever the adversary V does, the induced behaviors satisfy some property. This is essentially the controller synthesis problem.

Definition 8. (Strategies).
Let $S = (X, U, V, \delta)$ be a type III system. A strategy for U is a function $c : X^* \to U$. A state (or memoryless) strategy is a strategy satisfying $c(\xi \cdot x) = c(\xi' \cdot x)$ for every ξ and ξ' and hence it can be written as a function $c : X \to U$.

For this discussion we restrict ourselves to state strategies. Each strategy c converts a type III system into a type II system $S_c = (X, V, \delta_c)$ such that $\delta_c(x, v) = \delta(x, c(x), v)$.

Definition 9. (Synthesis for Reachability).
The controller synthesis problem for a system $S = (X, U, V, \delta)$ is: find a strategy c such that all the behaviors of the derived system $S_c = (X, V, \delta_c)$ avoid a set $P \subseteq X$ of "bad" states.

[9] To understand the graphical conventions note that $\delta(x_1, u_1, v_1) = x_1$, $\delta(x_1, u_1, v_2) = x_2$, $\delta(x_1, u_2, v_1) = x_2$ and $\delta(x_1, u_2, v_2) = x_4$. We assume that the choices of U and V are made simultaneously.

The set of behaviors of a type III system is structured as a game tree, also known as an alternating, AND/OR or min-max tree (see Figure 14). Due to space and time constraints I will only sketch the solution of the synthesis problem whose complete formalization is not easy due to the two types of branching and the use of feed-back in the definition of a behavior given a strategy c.

Consider the controller synthesis problem for the system of Figure 13 where the set of states to avoid is $P = \{x_5\}$. Looking closer we see that from state x_4 we cannot avoid the possibility of reaching x_5: if we choose u_1 the environment can choose v_2 and if we choose u_2 the environment can choose v_1 and in both cases the outcome will be x_5. On the other hand, from x_2 we can avoid reaching x_5, at least for one step, by taking u_2 rather than u_1. This motivates the following definition:

Definition 10. (Controllable Predecessors). Let $S = (X, U, V, \delta)$ be a type III system. The set of controllable predecessors of $F \subseteq X$ is

$$\pi(F) = \{x : \exists u \in U \;\; \forall v \in V \;\; \delta(x, u, v) \in F\}.$$

In other words, $\pi(F)$ denotes the states from which the controller, by properly selecting u, can force the system into F in the next step.

The following algorithm produces the set F_* of “winning states”, i.e. states from which reaching P can be forever avoided.

Algorithm 4. (Controller Synthesis).

```
$F^0 := X - P$
repeat
    $F^{k+1} := F^k \cap \pi(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$
```

This algorithm, a variant of dynamic programming, when applied to the system of Figure 13, produces the decreasing sequence of states

$$\{x_1, x_2, x_3, x_4\}, \{x_1, x_2, x_3\}$$

and converges. In control terms the set $\{x_1, x_2, x_3\}$ is the maximal control invariant set. The corresponding strategy is $c(x_1) = u_1$, $c(x_2) = u_2$ and $c(x_3) = u_1$ and it is computed by erasing transitions that can lead outside F_*. The resulting type II system is depicted in Figure 15. This is very similar to the supervisory control of Ramadge and Wonham (1989).

This concludes the story of finite-state discrete systems where simulation, verification and controller synthesis can all be performed exactly in a *fully-automatic manner* (modulo size limitations).

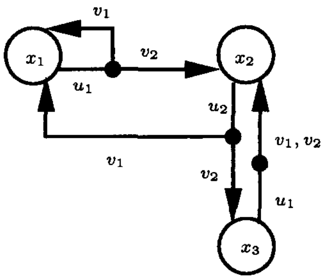

Fig. 15. The synthesized type II system which stays always within $\{x_1, x_2, x_3\}$.

The continuous analogues of type III systems are the *differential games* of Isaacs (1965) where the dynamics is of the form

$$\dot{x} = f(x, u, v)$$

and one wants to construct a continuous control law $c : X \to U$ such that all the behaviors of the induced type II system,

$$\dot{x} = f_c(x, v) = f(x, c(x), v)$$

satisfy the property in question. Traditionally in differential games the desired controller *optimizes* some performance measure over all the behaviors induced by V, but synthesis for reachability can be viewed as a special case of optimization with a $0-1$ cost function assigned to behaviors according to whether or not they reach P.

Before moving to continuous systems, let us see what happens to discrete systems if we consider an infinite state-space.

4. DISCRETE INFINITE-STATE SYSTEMS

Unlike finite transition systems which can be represented enumeratively by finite tables, infinite-state systems need richer description formalisms that express implicitly an infinite transition graph. The fact that computer programs can be viewed as representations of discrete dynamical systems is not part of the common knowledge of the general public, including control and even software engineers. In Computer Science, the dynamical system associated with a program is often referred to as its *operational semantics.* As an example, consider the following simple program which uses one integer variable y:

```
repeat
    $y := y + 1$
until $y = 4$
```

This program can be seen as a transition system over the state-space $X \times \mathbb{Z}$ where $X = \{x_1, x_2\}$ is the set of program locations (inside and after the loop) and $\mathbb{Z}$ is the set of possible values of

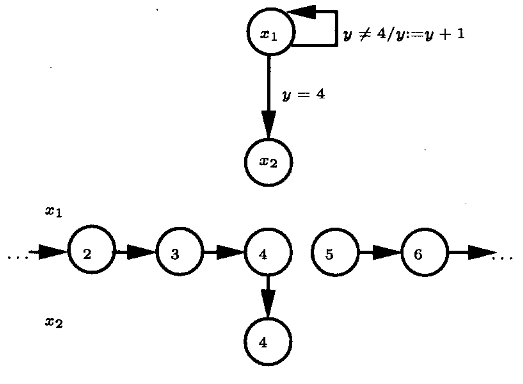

Fig. 16. An infinite-state system: an implicit representation (above) and a fragment of the explicit transition graph (below).

y_1. Such systems, although infinite, admit a finite *effective* representation such as the above program or the equivalent extended automaton[10] at the top of Figure 16. This is an automaton augmented with auxiliary variables which can be tested and modified by transitions. Such representations are effective in the sense that given any state it is possible to compute the next-state. However this local effectivity does not carry over to global properties such as reachability. For example, a forward simulation algorithm such as Algorithm 1, when started from state $(x_1, 2)$ will converge to the set $F_* = \{(x_1,2),(x_1,3),(x_1,4),(x_2,4)\}$. On the other hand, starting from $(x_1, 5)$ the algorithm will never terminate (see Figure 16).

In general the reachability problem for infinite-state systems is *undecidable*. This means that *there is no general algorithm* that takes *any* program with integer variables and solves its reachability problem. Note that the failure of Algorithm 1 to converge is *not* a proof of undecidability. The latter means that for *any conceivable algorithm* there will be a program for which it will fail to produce the answer. For such systems all you can do is to simulate forward until you reach P ("yes") or make a cycle ("no"), but none of these is guaranteed to happen. This notion allows theoretical computer scientists and logicians to publish *negative* results concerning the provable *inability* to produce certain algorithms.

There are two basic approaches to tackle such systems. If we want to stick to the algorithmic approach one needs to use *symbolic* rather then enumerative representations of the reachable states, that is, to encode sets of states using some formalism such as Boolean formulae combined with inequalities over numerical state variables. For example, the set of states reachable from $(x_1, 5)$ can be finitely represented by the formula $x = x_1 \wedge y \geq 5$. The computation of the reachable set is performed by doing *syntactic* operations on these formulae with some tricks to guarantee convergence, when possible. Even in the finite-state case symbolic techniques allow one to treat systems with a number of states which is otherwise prohibitively large.

Within the alternative *deductive* (or theorem-proving) framework, reachability properties are derived formally from axioms and rules concerning the dynamics of the system. The main disadvantage of this approach from the CAD point of view is that it is not *fully-automatic*, that is, one does not feed the computer with the description of the system, push a button and obtain the result. Even with the help of an automatic theorem prover, an active participation of a human user who understands the dynamics of the system in question is required. The analog of this approach in continuous systems would be, for example, proving a reachability property using a user-supplied Lyapunov function.

Verification of infinite-state systems is currently a very active domain of research, where combinations of algorithmic and deductive methods are investigated including questions of homomorphisms (called "abstraction") from infinite-state systems to finite ones. Some of the techniques for treating numerical variables are common to this domain and to continuous and hybrid systems.

5. CONTINUOUS SYSTEMS

In this section I sketch some of the problems encountered while trying to export algorithmic verification to continuous systems. A question that some readers will certainly pose is: *"Why bother?"* Indeed, with all this Control Theory, more than a century-old, employing all the accumulated knowledge of continuous mathematics, equation solving, optimization and more, why use these barbaric brute-force methods which do not exploit the special mathematical properties of the systems in question? My short answer[11] is that there are systems which cannot be modeled in a useful manner with purely continuous formalisms and which are more adequately modeled using *hybrid automata*, a combination of automata and differential equations where each state of the automaton represents one "mode" of operation. For such systems most "classical" methods fail while methods based on algorithmic reachability might work.

The state-space of continuous systems, $X = \mathbb{R}^n$, can be infinite in two senses: it can be unbounded, like the state-space of programs over the integers,

[10] Which is nothing but the good old flowchart.

[11] A longer answer can be found in the introduction to Asarin *et al.* (2000*b*).

but, even if we restrict the analysis to bounded subsets of $\mathbb{R}^n$, we have to face *dense* infinitude. The same goes for the time domain, $T = \mathbb{R}_+$. Moreover, inside the computer we cannot work with the ideal mathematical real numbers but rather with a finite (but large) subset of the rationals. This means that even the simulation of a single behavior is a non-trivial matter.

Consider the reachability problem for closed systems of the form $\dot{x} = f(x)$, whose discrete analogue has been shown to be trivially solvable using forward simulation. When we have a closed form solution, e.g. $\xi[t] = x_0 e^{At}$ for linear systems, we can claim to have "solved" the problem because $F_* = \{x_0 e^{At} : t \geq 0\}$ is a representation of all reachable states. But then, how can we check whether $F_* \cap P$ is empty where P is some simple subset of $\mathbb{R}^n$ defined by, say, combination of linear equalities? From the point of view of effective computation, such closed-form solutions are not much more explicit than the equations themselves.

Alternatively we can try forward simulation. For this we need first to discretize the time domain into a sequence $T_\Delta = \{n\Delta : n \in \mathbb{N}\}$ for some time step Δ and then produce a partial approximation of the solution ξ by a sequence $\xi' : T_\Delta \rightarrow X$ defined by some numerical integration scheme of the form

$$\xi'[(n+1)\Delta] = \xi'[n\Delta] + h(\xi'[n\Delta], \Delta).$$

Applying algorithm 1 we face two major problems:

(1) We are interested in the set

$$F_* = \{\xi[t] : t \in T\}$$

while what we compute is

$$F'_* = \{\xi'[t] : t \in T_\Delta\}.$$

Hence a non-empty intersection of F_* with P is not equivalent to such an intersection between F'_* and P (see Figure 17).

(2) The algorithm is not guaranteed to converge (like any infinite-state system), and if it converges, i.e. $\xi'[t] = \xi'[t']$ for some $t \neq t'$, this might be due to rounding errors and not because $\xi[t] = \xi[t']$.

It is clear from these observations that for continuous systems we cannot hope for the same strong and *exact* results as for finite automata. However, the situation is not so dramatic becasue the continuous world is less chaotic than the discrete one, and simulation can usually be used to increase our confidence in the correctness of a closed deterministic system. From now on we ignore the difference between ξ and ξ' and consider the simulation of a single behavior as a solved problem.

For type II systems of the form $\dot{x} = f(x, v)$ the situation is more complicated. The set of admissible inputs is typically the set of continuous signals of the form $\psi : T \rightarrow V$ over some bounded set V which we denote by V^T. As in the discrete case we can perform simulation for every individual input signal ψ and compute the set $F_*(\psi)$ of reachable states. However, unlike finite-state systems of size n where it is sufficient to simulate with all elements of $V^n \subseteq V^*$, there is no finite subset of V^T that "covers" all reachable states. The structure of this set is a "doubly-dense" tree, both in the vertical/temporal dimension (due to the density of T) and in the horizontal dimension (due to the density of V). Hence *exhaustive* generation of all inputs for simulation is not even an option.

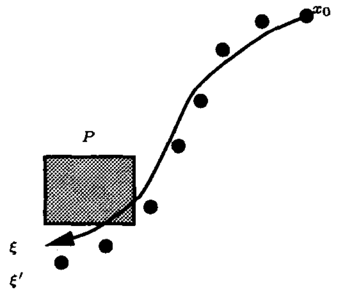

Fig. 17. A continuous behavior ξ that intersects P while its numerical approximation ξ' does not.

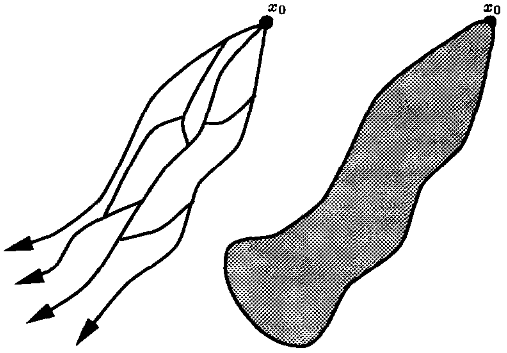

Fig. 18. The structure of the behavior of a continuous type II system: on the left we see some of the infinitely many behaviors generated by admissible inputs and on the right — the set of all states reachable by all the behaviors.

On the other hand, some approximate variants of Algorithm 2 are possible. To get the idea, let us look at Figure 18 where a sample of the behaviors induced by some inputs is shown. As in discrete systems, we need not explore all the (infinitely-many) visits of trajectories to the same state but rather find a way to construct F_* incrementally, not necesserily in a way that corresponds to the simulation of individual behaviors.

We use the notation $x \xrightarrow{t} x'$ to indicate the existence of an input signal $\psi : [0, t] \rightarrow V$ such that the behavior $\xi(\psi)$ starting at x reaches x'

at time t. Let F be a subset of X and let I be a time interval. The I-successors of F are all the states that can be reached from F within that time interval, i.e.

$$\delta_I(F) = \{x' : \exists x \in F\ \exists t \in I\ x \xrightarrow{t} x'\}.$$

Note that $\delta_{[0,\infty)}$ denotes all the states reachable from F. Assuming that admissible inputs do not depende on x, δ has the semi-group property, i.e.

$$\delta_{I_2}(\delta_{I_1}(F)) = \delta_{I_1 \oplus I_2}(F)$$

where $\oplus$ is the Minkowski sum and, in particular,

$$\delta_{[0,r_2]}(\delta_{[0,r_1]}(F)) = \delta_{[0,r_1+r_2]}(F).$$

If we had a procedure for computing $\delta_{[0,r]}$ we could construct incrementally the set of reachable states using the following algorithm:

Algorithm 5. (Continuous Reachability).

```
$F^0 := \{x_0\}$
repeat
    $F^{k+1} := F^k \cup \delta_{[0,r]}(F^k)$
until $F^{k+1} = F^k$
$F_* := F^k$
```

This algorithm suffers from the same problems as simulation, namely the inability to compute δ exactly and the lack of guarantee for finite convergence. In addition, it has to maintain representations of *subsets* of $\mathbb{R}^n$ to which the operation δ as well as union and equivalence testing should be applied. To overcome these problems we propose a pragmatic solution which is based on the following principles:

(1) The sets F^k are restricted to be polyhedra;[12]
(2) The successor operator δ is replaced by an an approximate version δ' such that $\delta'_{[0,r]}(F)$ is a polyhedron satisfying

$$\delta_{[0,r]}(F) \subseteq \delta'_{[0,r]}(F).$$

Under these conditions an approximate version of Algorithm 5 can be implemented whose outcome F'_* is an over-approximation of F_* (see Figure 19). Hence $F'_* \cap P = \emptyset$ implies that all behaviors of the system under all admissible inputs never reach P.

Variants of Algorithm 5 were implemented by Dang (2000) in a prototype tool called **d/dt**. These algorithms employ two techniques for approximating δ. One technique, inspired by Greenstreet (1996), is called "face lifting" and is based on maximizing normal derivatives of f on the faces of the polyhedron, see Dang and Maler (1998).

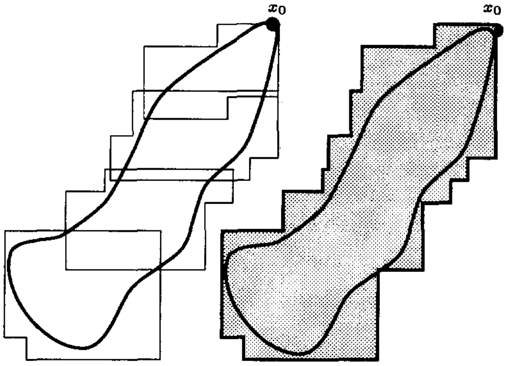

Fig. 19. The incremental construction of reachable sets using an approximate version of Algorithm 5 (left) and the final result, an over-approximation of the reachable states (right).

It applies to arbitrary non-linear systems. The other, more efficient technique is specialized for linear systems, see Asarin *et al.* (2000*a*), and uses some optimal control ideas due to Varaiya (1998). A similar technique was developed independently by Chutinan and Krogh (1998, 1999). Other approaches to solve reachability problems use ellipsoids instead of polyhedra, e.g. Kurzhanski and Valyi (1997); Botchkarev and Tripakis (2000) or try to apply ideas from the numerical solution of partial differential equations, e.g. Mitchell and Tomlin (2000). In all these techniques there is, of course, a trade-off between the over-approximation error and the efficiency of the algorithm as implied by parameters such as integration step size. It is worth mentioning that the whole approach for reachability computation was first developed in Alur *et al.* (1995) for a class of hybrid systems with piecewise-constant vector fields and implemented in the verification tool HyTech Henzinger *et al.* (1997).

For type III systems, differential games defined by $\dot{x} = f(x, u, v)$, no reachability based techniques have been developed yet, although there are some ideas. In Asarin *et al.* (2000*b*) a solution of the simpler problem of synthesizing a switching controller was proposed and implemented (see also Tomlin *et al.* (2000)). An experimental application of **d/dt** to control by switching was recently reported in Asarin *et al.* (2001). Other examples of work on related problems can be found in proceeding volumes such as Tomlin and Greenstreet (2002). To be honest, much work is still to be done before such techniques can be used in practice for systems of high dimension. Readers who want to experiment with these techniques are welcome to download **d/dt** from `www-verimag.imag.fr/~tdang/ddt.html` and apply it to their favorite examples.

[12] For technical reasons not to be discussed here, reachable sets are stored as *orthogonal polyhedra*, a sub-class of polyhedra consisting of those that can be written as finite unions of hyper-rectangles, see Bournez *et al.* (1999).

Acknowledgments:
Previous versions of this manuscript benefitted from discussions with E. Asarin, B. Krogh, A. Kurzhanski, J. Lygeros, G. Pappas, A. Pnueli, S. Sastry and P. Varaiya.

References

Alur, R., C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science* **138**, 3–34.

Asarin, E., O. Bournez, T. Dang and O. Maler (2000*a*). Approximate reachability analysis of piecewise-linear dynamical systems. In: *Hybrid Systems: Computation and Control* (B. Krogh and N. Lynch, Eds.). Lecture Notes in Computer Science 1790. Springer-Verlag. pp. 20–31.

Asarin, E., O. Bournez, T. Dang, O. Maler and A. Pnueli (2000*b*). Effective synthesis of switching controllers for linear systems. *Proceedings of the IEEE* **88**, 1011–1025.

Asarin, E., S. Bansal, B. Espiau, T. Dang and O. Maler (2001). On hybrid control of under-actuated mechanical systems. In: *Hybrid Systems: Computation and Control* (M.D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds.). Lecture Notes in Computer Science 2034. Springer-Verlag. pp. 77–88.

Botchkarev, O. and S. Tripakis (2000). Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In: *Hybrid Systems: Computation and Control* (B. Krogh and N. Lynch, Eds.). Lecture Notes in Computer Science 1790. Springer-Verlag. pp. 73–88.

Bournez, O., O. Maler and A. Pnueli (1999). Orthogonal polyhedra: Representation and computation. In: *Hybrid Systems: Computation and Control* (F.W. Vaandrager and J.H. van Schuppen, Eds.). Lecture Notes in Computer Science 1569. Springer-Verlag. pp. 46–60.

Brinksma, E. and Larsen, K.G., Eds. (2002). *Computer Aided Verification.* Lecture Notes in Computer Science 2404. Springer-Verlag.

Chutinan, A. and B.H. Krogh (1998). Computing polyhedral approximations to dynamic flow pipes. In: *Proc. of the 37th Annual International Conference on Decision and Control.* IEEE.

Chutinan, A. and B.H. Krogh (1999). Verification of polyhedral invariant hybrid automata using polygonal flow pipe approximations. In: *Hybrid Systems: Computation and Control* (F.W. Vaandrager and J.H. van Schuppen, Eds.). Lecture Notes in Computer Science 1569. Springer-Verlag. pp. 76–90.

Clarke, E.M., Orna Grumberg and Doron A. Peled (1999). *Model Checking.* The MIT Press. Cambridge, Massachusetts.

Dang, T. (2000). Verification and Synthesis of Hybrid Systems. PhD thesis. Institut National Polytechnique de Grenoble, Laboratoire Verimag.

Dang, T. and O. Maler (1998). Reachability analysis via face lifting. In: *Hybrid Systems: Computation and Control* (T.A. Henzinger and S. Sastry, Eds.). Lecture Notes in Computer Science 1386. Springer-Verlag. pp. 96–109.

Greenstreet, M.R. (1996). Verifying safety properties of differential equations. In: *Computer Aided Verification* (R. Alur and T.A. Henzinger, Eds.). Lecture Notes in Computer Science 1102. Springer-Verlag. pp. 277–287.

Henzinger, T.A., P.-H. Ho and H. Wong-Toi (1997). HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer* **1**, 110–122.

Isaacs, R. (1965). *Differential Games : A Mathematical Theory With Applications to Warfare and Pursuit, Control and Optimization.* Wiley.

Kurshan, R. (1994). *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press.

Kurzhanski, A. and I. Valyi (1997). *Ellipsoidal Calculus for Estimation and Control.* Birkhauser.

Maler, O. (1998). A unified approach for studying discrete and continuous dynamical systems. In: *Proc. of the 37th Annual International Conference on Decision and Control.* IEEE.

Manna, Z. and A. Pnueli (1995). *Temporal Verification of Reactive Systems: Safety.* Springer.

McMillan, K.L. (1993). *Symbolic Model Checking.* Kluwer Academic.

Mitchell, I. and C. Tomlin (2000). Level set method for computation in hybrid systems. In: *Hybrid Systems: Computation and Control* (B. Krogh and N. Lynch, Eds.). Lecture Notes in Computer Science 1790. Springer-Verlag. pp. 311–323.

Ramadge, P.J. and W.M. Wonham (1989). The control of discrete event systems. *Proceedings of the IEEE.* **77**, 81–97.

Tomlin, C. and Greenstreet, M.R., Eds. (2002). *Hybrid Systems: Computation and Control.* Lecture Notes in Computer Science 2289. Springer-Verlag.

Tomlin, C., J. Lygeros and S. Sastry (2000). A game-theoretic approach to controller design for hybrid systems. *Proc. of the IEEE* **88**, 940–970.

Varaiya, P. (1998). Reach set computation using optimal control. In: *Proc. KIT Workshop, Verimag, Grenoble.* pp. 377–383.