Safety Verification Proofs for Physical Systems

Tony Loeser, Yumi Iwasaki, Richard Fikes
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

While much progress has been made on verification of discrete systems such as computer programs, work on formal verification of continuous, physical systems has been limited. We present a technique for verification of safety properties of such systems. Our algorithm treats safety as a reachability problem, and attempts to prove that a system cannot evolve from an abstract initial state into a state in which the safety condition does not hold. This approach is inspired by qualitative simulation techniques and makes use of trajectories comprised of a sequence of qualitative states and state transitions. The applicability of the technique, however, is not limited to qualitative problems, as we can use any amount of quantitative mathematics in the system description. This paper describes the technique, presents example problems, and discusses its limitations as well as potential for use in device engineering.

Introduction

Computer simulation is a common tool for evaluating designs of physical systems. Simulations are used to discover a device's behavior, and more specifically, to verify that it meets certain criteria. Ideally, the designer would like to obtain a guarantee of some aspects of the expected behavior and do so as efficiently as possible.

For the most part, behavior simulations are computed by numeric integration. While obviously powerful, this approach does not always satisfy the engineer's needs. In particular, when the starting conditions represent a region in state space rather than a point, numeric simulations will not provide a guarantee that any safety condition is true throughout all possible behaviors of the device. Numeric simulation can be used to find and describe unsafe behavior trajectories (Neller 1998), but since one cannot generate all behavior trajectories from a region in state space numerically, the validity of safety conditions cannot in general be verified using only numeric simulation.

In addition, numeric simulations will always contain precision errors. While such errors can be bounded by analysis or decreased through more detailed computations, situations will occur in which precise guarantees are hidden by that error.

A more abstract approach to simulation is that of qualitative physics (Kuipers 1994). In qualitative mathematics, variable domains are divided by discrete, ordered landmarks, and the variables assume values that are either landmarks or intervals between them. An assignment to some state vector will define a qualitative state that corresponds to a range of quantitative states. To describe physics, the relationships between the variables are expressed by qualitative constraints, which are then used to determine whether or not states are mathematically consistent. Simulations are produced under the assumption that the physical variables and their first derivatives remain continuous. Given a system of qualitative constraints and a starting state, the simulation will produce qualitative state trajectories. In general, there is much branching on the trajectories because state transitions are limited only by variable continuity and consistency of the target state.

To perform qualitative safety verification, the simplest aproach would be to build an envisionment, or graph of all consistent states, with directed edges representing possible transitions. Since each state variable has a finite number of landmarks in its value space, the envisionment graph is finite but exponentially large. Given this graph, safety verification is straightforward; one simply marks all of the states which satisfy the initial condition, and then marks all of the states which violate the safety condition. If one cannot then find a path from one set of states to the other, then one has proved the safety property; the system cannot

evolve into an unsafe state.

An improvement would be to use an attainable envisionment, or behavior tree. In this case, the envisionment is built incrementally, from some starting state. States are added as needed for the targets of the transition arrows; in the end, the graph contains only attainable states by construction. If any state in the graph violates the safety condition, then the behavior is unsafe. This is the approach taken by Shults et al. (Shults & Kuipers 1997); in this work they formalize the verification of temporal logic formulae on attainable envisionments.

The qualitative model of a physical system is abstract, but it provides an important kind of upward solution property. Namely, if a trajectory is a solution of the quantitative equations, then the abstracted trajectory is a solution of the corresponding qualitative system. This property was proved in the context of the qualitative simulator QSIM by Kuipers et al. (Kuipers 1994). The envisionment is therefore useful in our case because if it shows that a qualitative trajectory is invalid, then we know that there is no corresponding exact trajectory, either.

We present a technique that attempts formal verification of safety problems, but that tries to avoid paying the cost of a complete envisionment. As in qualitative simulation, our technique relies on tracking continuous variables, and abstracts their values to landmarks and intervals. In addition, we are not limited to qualitative mathematics; we can use as many quantitative values and mathematical sophistications as are available.

More specifically, given a physical system, its initial conditions, and a safety condition, we will try to produce a proof that the safety condition holds in every reachable state, given those initial conditions. The physical system is represented by a set of variables of state, and equations that describe their interactions and evolution through time. The conditions are relations (equalities or inequalities) over the variable set; the initial conditions define the starting state of the system, and the safety condition defines the safety property. Viewed as a question of reachability, we have an abstract initial state S_i which includes those states in which the initial conditions are true, and we have an abstract goal state S_g in which the safety condition is false. We attempt to prove that S_g is unreachable.

Our algorithm incrementally builds an abstract trajectory and attempts at each step to prove safety. Thus, although our algorithm will eventually build a trajectory that is similar in size to a full envisionment, it can complete a proof at any stage in the trajectory building process and thereby avoid building a complete envisionment. When the algorithm succeeds the system will have the stated safety properties, to the extent that the mathematical model represents the true physical system.

Abstract Trajectories

Our approach is to keep as abstract a description of the system's behavior as is possible, gradually increasing the level of detail but attempting the proof at each level. Our basic unit of information is the abstract state, a partially specified qualitative state. The state is abstract because neither all of the variable values nor the exact time need be specified. For example, consider a system with variables x, y, and z. An abstract state may be defined just by $x = (x_1\ x_2)$, representing all states in which x has that assignment. A more specific substate might be the *first* time that x takes that value. Another substate might be the predecessor of $x = x_2$. Not the only predecessor of the latter state, but one of the possibilities. In the course of reasoning the state trajectory will be refined and extended, but the idea is to do so only as necessary.

We will assert an abstract trajectory that the system must follow in order to evolve from S_i to S_g, and then reason about the trajectory's validity. After asserting the trajectory endpoints, the intermediate states are filled in by application of the rules of continuity and the intermediate value theorem. We simply need to find a continuous variable which has different values at the endpoints of the trajectory. Each of these states on the trajectory is defined by an assignment of that variable to an intermediate value. For example, consider the continuous variable x. If $x < 3$ in S_i and $x > 5$ in S_g, then we know that there is a state in between for which $x = 4$. In addition, we know that there was an intermediate state in which x was increasing. After deducing these features of the system's trajectory purely from the property of continuity, we then use the domain knowledge contained in the equations of state to reason about the trajectory. Perhaps the model's equations show us that it is inconsistent for x to be increasing, or that the system cannot reach $x = 4$. In these cases, we can conclude that the system will not reach S_g, and the safety condition $x \leq 5$ holds.

For a more complete example, consider a ball thrown upwards into the air (fig. 1). Take for a safety condition that the ball is always above the ground, $h > 0$. In the initial state, the ball has height $h = h_0$, and h is increasing. For our goal state, use $h = 0$, when the ball

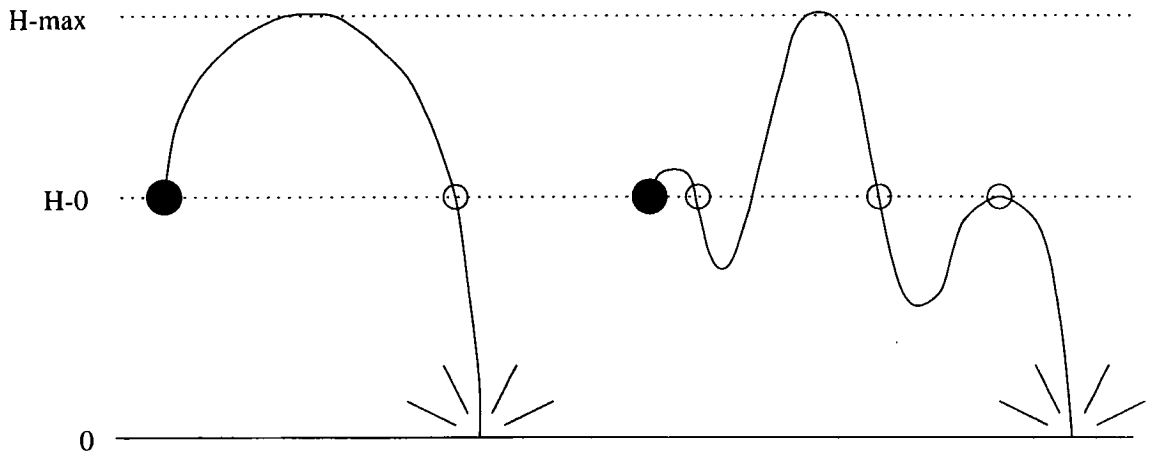


Figure 1: Two possible trajectories for a ball thrown up in the air. The one on the left seems most reasonable, but the abstract trajectory does not rule out the one on the right. The circles indicate points which correspond to state S_5.

hits the ground. We will name the highest value of h h_{max}, but all we know is that it is $h_{max} > h_0$. For the system (the ball) to evolve from the start to the goal state, it must pass through some trajectory. What can we say about the system's evolution? Of course, we could use our knowledge of physics to say that, barring any unexpected collisions (or very strong winds, etc.), the ball will rise, be still for an instant, and fall past h_0 until it hits the ground. Instead, our algorithm will produce an abstract trajectory which relies on continuity, not domain knowledge. The knowledge will be useful later for deducing properties of the trajectory and its states.

Given the start and end points of the trajectory, and given that the height of the ball h is a continuous variable, we know that h will pass through the following seven intermediate states:

S_1: $h = h_0$, increasing

S_2: $h \in (h_0\ h_{max})$, increasing

S_3: $h = h_{max}$, steady

S_4: $h \in (h_0\ h_{max})$, decreasing

S_5: $h = h_0$, not increasing

S_6: $h \in (0\ h_0)$, decreasing

S_7: $h = h_0$

Each of these states is abstract, corresponding to every specific state in which the variable h has the given qualitative value. When and how often each state occurs is unspecified; for example in fig. 1, one trajectory passes through S_5 once, while the other visits that state three times. Continuity simply requires that the system pass through that state at least once. S_i is a more specific version of S_1; it is the first time (in this simulation) that $h = h_0$ and is increasing. Similarly, S_g is the first occurence of S_7.

We have a couple of useful facts about this trajectory. First, the states must occur in order. From any continuous trajectory, we can pick out a set of seven states S_1' to S_7', temporally ordered by subscript, which are single occurences of the abstract states S_1 to S_7. Second, we know that each pair of consecutive states (S_1 and S_2, S_2 and S_3, etc.) must correspond to temporally consecutive states somewhere in the actual trajectory. So, there are states S_1'' and S_2'' which elaborate S_1 and S_2 respectively, such that S_1'' and S_2'' are each other's predecessor and successor. So, there may be many actual states corresponding to S_5, with all sorts of successors. At least once, though, there must occur in temporal succession actual states corresponding to S_5 and S_6.

Note that, in S_5, the direction of change isn't "decreasing", but rather it is "not increasing". This is because there are two ways for a continuous variable to pass from above to below a landmark: the way $y = -x$ passes through zero, and the way $y = -x^3$ does. Since we are not using any domain knowledge in the construction of the trajectory, we must account for both possibilities.

Also note that all of the states mentioned above are ones through which the system must pass. This is important when we try to show that the trajectory is invalid, or can not be followed. Still, we may have cause to reason about states which do not have this property. For example, there are a couple of possibilities for the predecessor of S_4. The system must

pass through one or the other, but not necessarily both. (The other one is defined by $h \in (h_0\ h_{max})$ and steady.) Unless stated otherwise, when we refer to a state on the trajectory we mean one through which the system must pass.

We call the construction of this abstract trajectory the expansion for h between the specified endpoint states. After we expand for h between S_i and S_g, the verification task becomes one of showing that this trajectory is somehow invalid, so the system can not actually follow it. We search for a *block*, a state or transition on the trajectory which is inconsistent. In the thrown ball example, we do not have any information which implies a block.

While the abstract trajectory is asserted using variable continuity between different states, the block will be found by considering the equations of state that govern the system. If a state from the abstract trajectory turns out to be mathematically inconsistent, then we know that the system will not evolve into any concrete realization of that state. In turn, this proves that the system can not follow the trajectory from beginning to end. Since the trajectory was constructed to be one through which the system must evolve in order to reach S_g, this is a proof that the system cannot reach S_g, and thereby verifies our safety property.

The block does not have to be an inconsistent state. For example, we could infer from the system's equations that some state is quiescent, i.e. that the system will cease to evolve after reaching that state. In this case, the transition out of that state is disallowed. Another simple blocking transition would be an asymptotic one, in which the predecessor state can never push a variable quite far enough to enter the successor state. (This case should be rare in real world examples.)

Algorithm

The procedure thus far is to pick a variable which differs in S_i and S_g, expand an abstract trajectory for that variable, and check for blocks on that trajectory. If no block is found, then there are a couple of ways to continue. One is to choose a different variable, expand its abstract trajectory, and look there for a block. The other is to refine the existing trajectory. While the abstract trajectory defines its states with constraints on only one variable, the system's equations may allow us to infer constraints on others. If one of the other variables is constrained to different values on the first trajectory's intermediate values, then this other variable can be expanded between those states, producing a refinement on the original trajectory. Trajectories can be repeatedly refined in this way, in principle until every variable has been expanded between every value, producing a structure with exponential size similar to the qualitative envisionment. Every step of the way, the new states are checked for blocks, and the algorithm finishes as soon as one is found.

If the safety condition cannot be proved in this manner, then the algorithm must eventually give up. The most obvious time is when there are no new opportunities for expanding a trajectory for some variable between two states. This point will eventually be reached, once the set of trajectories has been refined for every variable between all of its possible values. There is, of course, the possibility that heuristics and other reasoning techniques can guide the search for the block, possibly considering more promising expansions first, possibly stopping the search when success is not possible. This is currently an area of investigation.

In order that the search be kept finite, the variable values must be expressed as landmarks and intervals. This form is given in the case where one is using qualitative mathematics, but needs to be derived for variables with quantitative information. We envision a simple set of heuristics similar to those already in use for the dynamic identification of landmarks in qualitative mathematics (Kuipers 1994). The idea is to assert landmarks which correspond to other variables' landmarks, or inflection points and transitions. The value does not have to be known even for a quantitative variable's landmark; for example, the ball we tossed earlier reached a maximum height h_{max}, a landmark which is asserted along with the trajectory, and then assigned a definite value later if necessary. Note that in creating new landmarks, one must take care to keep each variable's value space finite, so that if the algorithm fails to find a proof, it will be sure to terminate.

To be more precise, the algorithm as it stands in our initial implementation is listed in figure 2. First, we identify the endpoints S_i and S_g of the reachability problem, called `start` and `end` in the listing. The function `build-state` simply returns an abstract state based on the assignments and constraints given as inputs. The `trajectory` initially has just those two states. The algorithm then goes into a loop, refining or adding to the trajectory until a block is found or it gives up. In this implementation, there is one trajectory, with only a partial ordering over its intermediate states. For example, the trajectory can contain the expansion for, say, x between `start` and `end`, as well as the expansion for y between the same

```
function FIND-BLOCK (equations,
                     initial_conditions,
                     verification_condition)
  start := build_state(initial_conditions)
  end := build_state(not(verification_condition))
  trajectory := { start, end }
  loop until ( contains-block(trajectory, equations) or
               give-up(trajectory) )
       trajectory := expand-trajectory(trajectory, equations)
```

Figure 2: Listing of main algorithm - FIND-BLOCK. The inputs are equations which describe some physical system, initial conditions that describe the starting state of that system, and a safety condition which defines the safety query.

two states.

Iteration will stop when one of two functions returns true. The first of these functions, `contains-block` examines the new or refined states in `trajectory`, and using the system's `equations`, returns true if it can find a block. Current tests include, for the state consistency, a simple check that the equations allow the asserted variable values. When variables are at a landmark, we check to make sure there is a legal predecessor and successor. We also filter for quiescent states. This sort of reasoning can become arbitrarily complex, possibly involving more than one state at a time to try and prove that some state or transition is invalid. The second check to cease iteration is `give-up`, which currently returns true if there are no more trajectory expansions available, or on user prompt.

Not shown explicitly in the listing is a global set of mathematical facts that `contains-block` uses to save information from one call to the next. While it may not be possible to prove that a certain state is inconsistent, assuming that it is consistent may lead to a constraint on some constant variables, or to some other math fact. These facts are then stored and used when checking the other states and transitions. This is a way to check the trajectory for self-consistency, in addition to simple consistency with the system's equation set. Note that assertion of a new math fact may necessitate checking old states and transitions for blocks.

On each iteration, `expand-trajectory` will add to the trajectory by expanding a variable between two states as described above. Assuming a given set of landmark values for a variable, a trajectory is quite easy to generate; simple templates encode how a variable moves from one value to another, and possibly turns around. Currently, the user identifies which variable to expand next. Another obvious strategy is to do all possible expansions. Again, here is more room for elaboration of the algorithm.

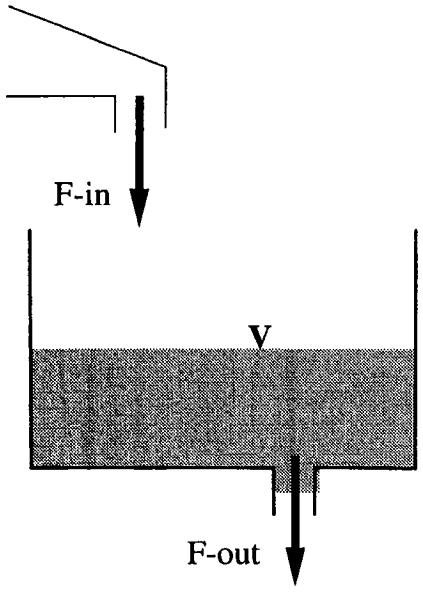
If a block is found, then the algorithm returns successfully, having proved that the safety condition will always hold, given the starting conditions. If there is no block, then there is no negative answer either; the system may be safe or unsafe.

Examples

For an example, consider the problem of the filling bathtub shown in figure 3. The bathtub holds water; water flows in through the faucet and out through the drain. The variables that describe this system are V, the volume of water in the tub, F_{in}, the flow in through the faucet, F_{out}, the flow out through the drain, and k, a constant describing the drain size. Four equations serve to describe the behavior of this system: first, the statement that water volume is always positive, as is the drain size; then, the rate of change of the volume is the difference in the flows; finally, a very simple minded approximation of the flow out through the drain, namely that the flow is proportional to V.

In the initial conditions, V is greater than zero and increasing. We also state that F_{in} is a constant, or alternatively that its time derivative is always zero. The safety condition is that there is water in the tub. Our algorithm will try to disprove the reachability problem, i.e. show that the system can not evolve from the initial state into one with $V = 0$.

We start by creating the abstract start and end states. S_i is defined simply by $V = V_0$ and increasing, where V_0 is a label for the initial value. S_g is defined by $V = 0$. The initial trajectory is $\{S_i\ S_g\}$. As it turns out, this proof is quick because there is already a block in the trajectory; S_g does not have a legal predecessor, and since it is not the start state, we will show that this is an inconsistency. From eqn. 4, we derive that $F_{out} = 0$. Since the initial conditions ensure that F_{in} is always positive,



Variables:
V, F-in, F-out, k

Equations:
1. V >= 0
2. k >= 0
3. d/dt V = F-in - F-out
4. F-out = kV

Initial Conditions:
F-in constant
V > 0
d/dt V > 0

Safety Condition:
V > 0

Figure 3: The bathtub example.

eqn. 3 now tells us that V is increasing. So, we know that in S_g, V is increasing. Now continuity of V tells us that the predecessor to S_g has $V < 0$, which contradicts eqn. 1. So a block is found, and the function FIND-BLOCK returns successfully.

For argument's sake, assume this block is not found, or is ignored. The next step would be to expand a trajectory for some variable, between the two state endpoints. We will then continue searching for blocks, as there are examples of several kinds in this problem. (Of course, in practice a proof requires only one block.) For the expansion, V is an obvious choice; to keep things simple, we will assume that the bathtub does not overflow, i.e. $V \leq V_{full}$. The abstract trajectory becomes:

S_1: $V = V_0$, increasing (corresponds to S_i)
S_2: $V \in (V_0\ V_{full})$, increasing
S_3: $V \in (V_0\ V_{full}]$, steady
S_4: $V \in (V_0\ V_{full})$, decreasing
S_5: $V = V_0$, not increasing
S_6: $V \in (0\ V_0)$, decreasing
S_7: $V = 0$, not increasing (corresponds to S_g)

We now search for blocks in the expanded trajectory. S_1 is not necessarily inconsistent, so we assume that it is definitely consistent, and continue. That assumption produces a relation on the constants, $kV_0 < F_{in}$. The next block is difficult to find automatically; the transition from S_2 to S_3 is asymptotic, meaning that the system will never actually reach S_3. An easier block is the transition out of S_3, as that state is quiescent. All derivatives go to zero in S_3, so the system stops evolving. Pushing forward to find a third block, we can show that S_5 is inconsistent. More specifically, S_1 and S_5 cannot both be consistent; when we assumed that S_1 was not a block, we got a relation over the constants which in turn shows that V must be increasing in S_5, a contradiction with the qualitative value of V. Now that it has provided us with many different examples of blocks, we can be quite sure that the bathtub will not become empty in this scenario.

To show a more complex form of block, we will alter the example somewhat. First, a quick repair to eqn. 4; to keep the transition from S_6 to S_7 from being asymptotic, we use a low water level aproximation $F_{out} = kV^{\frac{1}{2}}$. This time, the initial conditions are that $F_{in} = 0$ and is constant, and $V = V_{full}$ and is decreasing. Again, ask if the tub will ever be empty. Clearly it will, until we add one more modification; at time $t = t_1$ ($t = 0$ in the initial state), we change the value of k to zero as the drain is closed. (Admittedly, the bathtub is now a hybrid system, but this change is not important for the blocking example.) Now, it is a simple race between the drain and the clock; this fits

in a generalized way into our formalism. The trajectory eventually will be:

S_1: $V = V_{full}$, decreasing

S_2: $V \in (0\ V_{full})$, decreasing

S_3: $V = 0$, decreasing

Since all three states are consistent, we are checking the transitions; the transition from S_2 to S_3 may be a block. To be precise, there are substates S_2' and S_3' of S_2 and S_3 respectively, such that S_2' is the predecessor of S_3'. We analyze the transition between these two states. To show that the transition in question is blocking, we attempt to show that V cannot evolve from its initial value in S_2' to its value at the transition between S_2' and S_3', $V = 0$. The basic idea is that we can place a constraint on how some variables change in the state, namely time, and might propagate that constraint to other variables.

First, we need to find the initial value of V in S_2'. We can get information about this value from the predecessors of S_2'. Lump them into an abstract state and call them S_4. Due to continuity of V, the possible values of V in S_4 are $V \in (0\ V_{full})$ and steady, and $V = V_{full}$ and decreasing. Because $F_{in} = 0$, the substate defined by the former value is inconsistent. So in S_4, $V = V_{full}$ and decreasing. (S_4 is an elaboration of S_1) From S_4 we see that the initial value for V in S_2' is $V = V_{full}$ and decreasing.

To show that the transition in question is blocking, we need to show that V cannot move from V_{full} to 0 in the duration of S_2'. With our current algorithm, we do not know the duration of an abstract state right away; we may pass in and out of S_2 ten times before the tub is empty. We do know, however, that the duration is less than t_1, and this leads us to the obvious test. If the drain is big enough or t_1 is small enough, the transition becomes blocking.

The method for transition analysis demonstrated in the preceding paragraph is not very strong; there are plenty of examples that a human can solve but where the algorithm fails to find a proof. However, one could devise a more complete procedure for checking the transition. Our general approach is to assert the abstract trajectory through which the system must pass, and then look for blocks on that trajectory. This general approach will still be a framework for the reachability proof when the procedures to assert and check the trajectory become more sophisticated.

Discussion

We have described a procedure for addressing safety queries in continuous, physical systems, using algebra rather than numeric integration. Instead of producing a simulation, a successful run of the algorithm gives a formal proof that the safety condition will be valid in all trajectories. For example, in the bathtub problem, we can extract a formal proof that the tub will never be empty, given those initial conditions. The uncertainties of the verification are then limited to the those associated with the underlying mathematical model, the equations of state.

Our approach also supports the analysis of partially specified designs and behaviors. Since the underlying technique uses landmarks and intervals in the manner as qualitative mathematics, it can handle qualitative specifications, and mixtures of qualitative and quantitative mathematics. One expects to have better success generating proofs when the landmarks are known, or tightly constrained, as will be the case more often with quantitatively valued variables. Still, the ability to reason with varying amounts of ambiguity in the specification is an important one.

We have shown that the algorithm is sound, but it is not complete. In this case, we take completeness to mean that the algorithm will prove the safety condition whenever it is true. It is not the mathematics which are lacking; any mathematical proof of unreachability will boil down to an inconsistency supported by this method. Rather, it is the search for this inconsistency in the trajectory which is difficult. If a human is guiding the variable expansions, for example, then the human needs to make the right choices. This is similar to many automated theorem provers, in which the user must specify the methods used at many steps on order to achieve success. Even if completely automatic, this is not an algorithm for searching all of "mathematical proof space".

There are a couple of ways in which this search is difficult. One is the problem of too much abstraction. (This is shared in part with envisionment proofs.) There may be an inconsistent state through which the system must pass, but there are not sufficient landmarks to isolate it. Or in evolving from $x = 3$ to $x = 5$, the system may need to visit $x = 0$ for complicated reasons; this state may be inconsistent, but it will not even be represented on just a simple expansion for x. Another problem is with the elaboration path. The inconsistency may involve values of many variables, without being obvious in a way which guides the system to generate a trajectory through that state.

Although incomplete, our algorithm has been useful on the practical examples we have tried so far. In our limited experience, the

proof tends not to be all that complicated. Perhaps this is biased by the examples we choose. Also, it may be a characteristic of good design. If an engineer needs to produce a design with certain safety properties, it is perhaps better practice to make the mechanisms that guarantee the safety simple rather than convoluted. The automatically generated proof then acts as both a sanity check and valuable documentation for the design.

It is useful to consider the possible role of this verification tool in engineering design, with the idea that rigorous verification may be a useful part of the device design process. We envision this sort of automated assistance as playing a role in a design cycle of refinement and testing. A candidate device design is given to the verification tool in an attempt to guarantee certain properties. If the proof is not possible, then the design can be modified. If there is a proof, then the engineer has confidence that the component will have the desired properties. In the end, the design is accompanied by a theoretical guarantee that the device will meet certain specifications. The cycle can be one of changing the design and the parameter values, or just as easily can be one of changing or narrowing the range in which some values are constrained to lie. If the tool can, through acceptance of qualitative specifications, allow the finished design to be more abstract, then it has helped to produce a more versatile design.

There is also the opportunity to use feedback from the algorithm to assist the design process. First, there is the simple possibility that looking at descriptions of some states that the system must pass through before violating the safety condition may trigger design ideas. In a similar way, some basic failure analysis for the proof procedure may be helpful. We could, for example, give lists of constraints over the variables which, if they were true, would allow the algorithm to show that some state or transition in the trajectory was a block. Most of the constraints would not lead to feasible design refinements, but the list may contain something useful or inspire a good idea.

In a more sophisticated way, we can generate constraints on unspecified variables. Say, for example, exogenous parameter x were unspecified in some design. At each blocking test, we try to generate conditions on x which would create a block. At the end, output the least restrictive such conditions as possible ways to guarantee safety. There has been work on this sort of parameter setting before, using causal ordering to find the right parameter settings (Hibler & Biswas 1993), but this was for the case of static equations. We have a framework in which to generalize this to dynamic behaviors.

While the proof procedure will have exponential complexity if carried to completion on a problem for which it fails, there are also problems which it will solve quickly. These simple problems may be more frequent in actual engineering designs, as properties in straightforward designs tend to be caused directly, rather than from details far upstream in the behavior. If we are automatically searching for variables upon which to expand, a likely aid would be the causal ordering tree for the variable(s) mentioned in the query. In practice, the behavior features that validate the safety property may involve variables which are close to the root of that tree. When designing a device, one wants to achieve the required behavior in as simple a manner as possible, rather than relying on more indirect effects. This in turn would tend to produce designs which our algorithm could handle efficiently.

In general, we have built a simple framework upon which to build verification techniques for reasoning with physical systems. Future work will involve building algorithms to deal with more complicated and practical example problems.

References

Hibler, D., and Biswas, G. 1993. Restriction of qualitative models to ensure more specific behavior. *Intelligent Systems Engineering* 2:133–44.

Kuipers, B. 1994. *Qualitative Reasoning.* Cambridge, Massachusetts: The MIT Press.

Neller, T. 1998. Information based optimization approaches to dynamical system safety verification. In *Proceedings of Hybrid Systems VI (HS98).* Springer Verlag.

Shults, B., and Kuipers, B. 1997. Proving properties of continuous systems; qualitative simulation and temporal logic. *AI Journal* 92:91–129.