Specification and Synthesis of Hybrid Automata for Physics-Based Animation

Thomas Ellman
ellman@cs.vassar.edu
Department of Computer Science
Vassar College, Poughkeepsie, NY 12601

Abstract

Physics-based animation programs can often be modeled in terms of hybrid automata. A hybrid automaton includes both discrete and continuous dynamical variables. The discrete variables define the automaton's modes of behavior. The continuous variables are governed by mode-dependent differential equations. This paper describes a system for specifying and automatically synthesizing physics-based animation programs based on hybrid automata. The system presents a program developer with a family of parameterized specification schemata. Each scheme describes a pattern of behavior as a hybrid automaton passes through a sequence of modes. The developer specifies a program by selecting one or more schemata and supplying application-specific instantiation parameters for each of them. Each scheme is associated with a set of axioms in a logic of hybrid automata. The axioms serve to document the semantics of the specification scheme. Each scheme is also associated with a set of implementation rules. The rules synthesize program components implementing the specification in a general physics-based animation architecture. The system allows animation programs to be developed and tested in an incremental manner. The system itself can be extended to incorporate additional schemata for specifying new patterns of behavior, along with new sets of axioms and implementation rules. It has been implemented and tested on over a dozen examples. We believe this research is a significant step toward a specification and synthesis system that is flexible enough to handle a wide variety of animation programs, yet restricted enough to permit programs to be synthesized automatically.

1. Introduction

Physics-based animation programs are useful in a variety of contexts, including science, engineering, education and entertainment. For example, in science, they are used to investigate the behavior of dynamical systems. In engineering, they are used to help design vehicles, machinery and other mechanical devices. In education, they are used to teach basic principles of physics. In entertainment, they are used in games involving cars, planes, spaceships and other moving objects. Such programs are usually constructed by hand, in conventional programming languages, such as C++, possibly augmented with a physics-based animation toolkit. Unfortunately, manual construction of physics-based animation programs is expensive, time-consuming and highly prone to error.

A considerable portion of the difficulty results from the need to track and manage instantaneous changes in the states of objects and the equations and constraints that govern their behavior. For example, when one rigid body collides with another, the objects' states of motion may change instantaneously. If the contact persists for a period of time, the governing equations of motion and constraints may change as well. Similar instantaneous changes occur when an autonomous agent switches from one control mode to another. For example, when the driver of a car depresses or releases the accelerator or brake, the car's acceleration may change instantaneously.

In previous work, the author and his students developed a system for specification and synthesis of numerical simulation programs for physics-based animation applications. [Ellman et al, 2002], [Ellman et al, 2003]. The system allows a developer to specify the geometry, shading, lighting and camera angles of a scene in 3D Studio Max® and specify the dynamics of the scene in Mathematica®. A Mathematica program processes these specifications and generates a numerical C++ program that interleaves simulation and rendering to generate a real-time animation of the specified scene. This work drew upon the field of analytical dynamics [Baruh, 1999], in which motion is governed by differential equations involving forces and constraints, for modeling physical systems. The equations were assumed to remain fixed over time, resulting in continuous and smooth motion, ignoring the complexities described above.

We now report on research extending the system to hybrid automata [Van Der Schaft and Schumacher, 2000] A hybrid automaton includes both discrete and continuous dynamical variables. The discrete variables define the automaton's modes of behavior. The continuous variables are governed by mode-dependent differential equations. Hybrid automata are suited to modeling physical systems with instantaneous changes in forces, constraints and equations. They are also suited to modeling some types of autonomous behavior by agents operating in a physics-governed environment.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

The long-term goal of this research is to develop a specification language and synthesis system that is general enough to handle a wide variety of animation programs, yet restricted enough to permit programs to be synthesized automatically. We are taking an incremental approach in working toward this goal. We began by developing a family of parameterized specification schemata. Each scheme describes a pattern of behavior as a hybrid automaton passes through a sequence of modes. A developer can specify a program by selecting one or more schemata and supplying application-specific instantiation parameters for each of them. Each scheme has a declarative interpretation, and can be combined with other schemata in an order-independent fashion. We have associated each scheme with a parameterized sentence in a logic of hybrid systems. The sentence serves to document the semantics of the specification scheme. We have also developed sets of rewrite rules that implement each of the specification schemata in a general physics-based animation architecture. Finally, we have tested the system on over a dozen examples. Our approach is extendable. New specification schemata and synthesis rules can be added as they are developed, without impacting the semantics of existing schemata and the functionality of existing synthesis techniques. We intend to expand, generalize and unify the specification schemata and synthesis techniques over time. We expect the process will lead eventually to a logic-based specification language and synthesis system with a combination of expressiveness and automation that makes it useful for developing animation programs in real-life applications.

2. Examples

Ball on Steps: A ball bounces and rolls down a gradually inclined series of steps. The system has two modes of operation: bouncing and rolling. In the bouncing mode there are no constraints on the ball's motion. In the rolling mode, the ball is constrained to roll on a step without skidding. The system begins in a bouncing mode. Each of the first several times the ball strikes a step, it undergoes a transition in which the bouncing mode is preserved and the ball's linear and angular velocities undergo instantaneous changes as a result of the collision. The velocities are updated according to equations expressing conservation of linear and angular momentum and loss of energy according to the Coulomb friction model. When the ball's kinetic energy at impact falls below a threshold, it makes a transition from the bouncing mode to the rolling mode. Eventually the ball rolls off the step and undergoes a transition back into the bouncing mode as it proceeds to bounce on the next step, repeating the cycle.

Robot on Track: A robot rides on a four-wheeled cart moving around a circular track. The robot has one arm, consisting of an upper arm, a forearm and a hand. A ball sits on the track directly in the path of the robot. Each time the robot encounters the ball, it stops, reaches out, picks up the ball, and places it directly behind itself on the track. It proceeds on its way around the track until it encounters the ball again and repeats the cycle, over and over, forever. The robot has ten modes of operation. It cycles through these modes in a fixed order. In each mode, the robot's joints and axles are constrained to rotate with fixed (possibly zero) angular velocities. The angular velocity of each joint depends on the current mode. Another mode-dependent constraint requires the ball to translate and rotate along with the robot's hand (while the ball is in the hand) and to remain motionless relative to the track (while the ball is on the track).

Dueling Spaceships: Two spaceships travel around a planet in elliptical orbits according to Newton's law of gravitation. Each spaceship carries a torpedo inside it. When one ship moves within its firing range of the other, it launches its torpedo with a carefully chosen speed and direction. The torpedo moves under the influence of gravity along its own elliptical orbit until it collides with the other spaceship. After the collision, the torpedo returns instantly to its resting point inside its parent spaceship, ready to be fired again. Each torpedo has two modes of operation. In its rest mode, a torpedo is constrained to move with the same velocity as its parent spaceship. It its launched mode, a torpedo's motion is unconstrained, subject only to the force of gravity.

3. Hybrid Automata

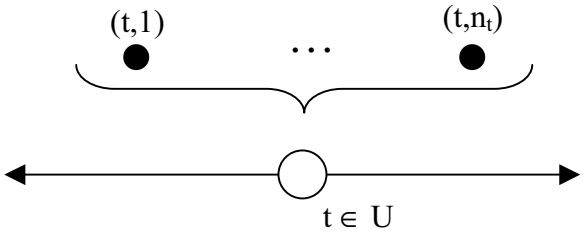
A hybrid automaton consists of a finite set M of modes, (parameterized by a set of nominal-valued mode variables), a set $S \subseteq R^N$ of states (parameterized by a set of real-valued state variables), a "flow" function $f:M\times S\times R\rightarrow S$; and a set P of transition operators $p=(g,r_m,r_s)$, in which $g:M\times S\rightarrow\{True,False\}$ is a "guard" and $r_m:M\rightarrow M$ and $r_s:S\rightarrow S$ are mode and state "reset" functions.[1]

A behavior of a hybrid automaton is a path through $M\times S$ governed by the flow function and the transition operators. As long as automaton A resides in a given mode, its behavior is described by the flow function. Thus if A is in mode m and state s at time t, and it remains in mode m for time Δ, it will be in state $f(m,s,\Delta)$ at time $t+\Delta$. A flow function is usually represented by a set of differential equations. The transition operators describe

[1]The hybrid automaton model presented here is motivated by the presentation in [Van Der Schaft and Schumacher, 2000]; however, some details are simplified or modified.

when and how the mode and state variables can change instantaneously. If an automaton A is in mode m and state s at time t, and for some transition operator $p=(g,r_m,r_s)$, $g(m,s)$=True, then A can make a transition to mode $r_m(m)$ and state $r_s(s)$ at time t. Whenever one or more guards are true, the automaton makes one of the allowed transitions.

Formally, a behavior β is a tuple $(m_\beta, s_\beta, T_\beta, E_\beta, \leq_\beta, N_\beta)$ where $m_\beta:T_\beta \rightarrow M_\beta$; $s_\beta:T_\beta \rightarrow S_\beta$, and the set T_β is a *time evolution,* constructed from the real numbers R by removing a subset U of R and replacing each element t of U with a set $S_t=\{(t,1),\ldots,(t,n_t)\}$. This device allows the path functions m and s to take multiple values at the points in time at which transitions occur. The set E_β of *event times* is the union of S_t for all $t \in U$. The relation $\leq_\beta$ is the smallest total ordering on T_β that includes the restriction of the usual ordering $\leq$ of R to R-U, and such that for all t and u in R-U, and all integers i and j in $[1..n_t]$, $(t,i) \leq_\beta (t,j)$ if $i < j$; $u \leq_\beta (t,i)$ if $u < t$; and $(t,i) \leq_\beta u$ if $t < u$. The *immediate successor* relation N_β on $T_\beta \times T_\beta$ is defined in terms of $\leq_\beta$ so that $N_\beta(u,v)$ if and only if u=(t,i) and v=(t,i+1) for some t in R-U and integer i in $[1..n_t-1]$. If $N_\beta(u,v)$ then we sometimes use the notation $t^\flat$ for u and $t^\sharp$ for v. The symbols $\flat$ and $\sharp$ are mnemonics for the relationship between an event time $t^\flat$ and its (unique) immediate successor $t^\sharp$, when an immediate successor exists. A time evolution in the neighborhood of a set E_t of event times in shown below.



The general scheme of a program for simulation of hybrid automata is shown below. The procedure Simulate(p,c,m,s,t_1,t_2,Δ) takes the following inputs: the program initialization parameters p, the state c of the interactive user's control device, and the mode m and state s at time t_1. It returns the updated mode and state at time t_2 using Δ as its initial step size in a search for transition points. It begins by integrating a mode-dependent system derivative function from t to $t+\Delta$. Then it applies each element of an array of Boolean-valued guard functions to the resulting state. If any of the guards return true, the program conducts a bisection search between t and $t+\Delta$ to find an approximation of the earliest time τ and state σ at which some guard Guard[i] became true. Next it uses ModeResetFunction[i] and StateResetFunction[i] to update the mode m and state σ at time τ. In addition, the procedure repeatedly selects and applies another pair of mode and state reset functions as long as there exists a pair with a true guard. Finally, the algorithm proceeds to continue the simulation from τ to t_2 using the updated mode and state. The simulation architecture can be instantiated in the context of a given application problem by synthesizing the system derivative function Derivative(p,c,m), guard predicates Guard[i], and mode and state functions ModeResetFunction[i] and StateResetFunction[i].

```
Simulate(p,c,m,s,t1,t2,Δ):
  If (t1≥t2) Then Return((m,s)).
  Else 1. Let r = Integral(Derivative(p,c,m),s,t1,t1+Δ).
       2. If (∃ i ∈ [1..n]) Guard[i](p,c,m,r,t1+Δ)
          Then  a. Let (σ, τ , i) = Locate(p,c,m,s,t1,Δ, r, i).
                b. Let (m',s') = Transition(i,p,c,m,σ,τ).
                c. Simulate(p,c,m', s', τ, t2, Min[t2-τ, Δ]).
          Else  Simulate(p,c,m,r,t1+Δ1,t2,Min[t2-(t1+Δ), Δ]).

Locate(p,c,m,s,t,Δ,r,i):
  If  (Δ < ε)  Then Return(r,t,i).
    Else  a. Let q = Integral(Derivative(p,c,m),s,t,t+Δ/2).
          b. If  (∃ j ∈ [1..n]) Guard[j](p,c,m,q,t+Δ/2)
                Then Locate(p,c,m,s,t,Δ/2, q, j).
                Else  Locate(p,c,m,q,t+Δ/2,Δ/2, r, i).

Transition(i,p,c,m,s,t):
  1. Let m' = ModeResetFunction[i](p,c,m,s,t).
  2. Let s' = StateResetFunction[i](p,c,m,s,t).
  3. If  (∃ j ∈ [1..n])  Guard[j](p,c,m',s',t)
       Then Return(Transition(j,p,c,m', s', t')).
       Else  Return((m', s')).
```

4. Specification

The system presents a human developer with a family of schemata from which he/she can construct a specification of a physics-based animation program. Each scheme represents a pattern of behavior that occurs commonly in physics-based animation programs. In particular, each scheme describes a sequence of one or more transitions through partially specified modes. The schemata have parameters that allow a developer to instantiate them in the context of an application problem. These parameters describe conditions for activating the sequence, passing through each mode in the sequence, and reinitializing the mode and state at each transition point.[2]

[2] We discuss only specifications of guards and mode and state reset functions in this paper. Specifications of the system derivative function are discussed in [Ellman et al, 2002] [Ellman et al, 2003].

4.1 A Logic of Hybrid Automata

Each scheme is associated with one or more axioms. The axioms serve to document the semantics of the scheme. They are expressed in a *timed hybrid linear temporal logic*. Sentences are constructed from primitive formulae; Boolean operators; universal $\forall$ and existential $\exists$ quantification over the real numbers R; and the following temporal operators: **.** (freeze), χ (next), $\Box$ (always), $\Diamond$ (eventually) and *U* (until*)*. A primitive formula $\varphi(m,s,t_1,\ldots,t_k,u_1,\ldots,u_n)$ is an equation or inequality of expressions over mode (m) and state (s) functions, time-valued variables $(t_1,\ldots,t_k)$ and real-valued variables $(u_1,\ldots,u_n)$. If $\beta=(m_\beta,s_\beta,T_\beta,E_\beta,\leq_\beta,N_\beta)$ and $t\in T_\beta$ then $\beta\vDash_t\psi[\rho]$ means that behavior β satisfies formula ψ at time $t\in T_\beta$ with assignments $\rho:V\rightarrow R\cup T_\beta$ to the universe V of free variables, some of which may occur in ψ. The expression $\rho(v\mapsto t)$ is the result of updating ρ with the assignment $v\mapsto t$. The expression $\beta\vDash\psi$ means that behavior β satisfies ψ. The expression $A\vDash\psi$ means that hybrid automaton A satisfies ψ. The satisfaction relations are defined as follows:[3]

- $\beta\vDash_t\varphi(m,s,t_1,\ldots,t_k,u_1,\ldots,u_n)[\rho]$ if and only if $\varphi(m_\beta,s_\beta,\rho(t_1),\ldots,\rho(t_k),\rho(u_1),\ldots,\rho(u_n))$ is true in the standard[4] model of real arithmetic.
- $\beta\vDash_t(\varphi\rightarrow\psi)[\rho]$, $\beta\vDash_t(\varphi\wedge\psi)[\rho]$, $\beta\vDash_t(\varphi\vee\psi)[\rho]$ and $\beta\vDash_t(\neg\psi)[\rho]$ are defined as in classical logic.
- $\beta\vDash_t((\exists u)\psi)[\rho]$ if and only if there exists some $r\in R$ such that $\beta\vDash_t\psi[\rho(u\mapsto r)]$.
- $\beta\vDash_t((\forall u)\psi)[\rho]$ if and only if for all $r\in R$, $\beta\vDash_t\psi[\rho(u\mapsto r)]$.
- $\beta\vDash_t(v\ .\ \psi)[\rho]$ if and only if $\beta\vDash_t\psi[\rho(v\mapsto t)]$.
- $\beta\vDash_t(\chi\ \psi)[\rho]$ if and only if there exists some $v\in E_\beta$ such that $N_\beta(t,v)$ and $\beta\vDash_v\psi[\rho]$.
- $\beta\vDash_t(\Diamond\psi)[\rho]$ if and only if there exists some $v\in T_\beta$ such that $v\geq_\beta t$ and $\beta\vDash_v\psi[\rho]$.
- $\beta\vDash_t(\Box\psi)[\rho]$ if and only if for all $v\in T_\beta$ such that $v\geq_\beta t$, $\beta\vDash_v\psi[\rho]$.
- $\beta\vDash_t(\varphi\ U\ \psi)[\rho]$ if and only if there exists $u\in T_\beta$ such that $u\geq_\beta t$ and $\beta\vDash_u\psi[\rho]$ and for all $v\in T_\beta$ such that $t\leq_\beta v<_\beta u$, $\beta\vDash_v\varphi[\rho]$, or else for all $v\in T_\beta$ such that $t\leq_\beta v$, $\beta\vDash_v\varphi[\rho]$.
- $\beta\vDash\psi$ if and only if $\beta\vDash_t\psi[\rho]$ for all ρ and $t\in T_\beta$.
- $A\vDash\psi$ if and only if $\beta\vDash\psi$ for all behaviors β of A.

[3] The logic presented here draws upon features of logics discussed in [Alur and Henzinger, 1992] and [Davoren and Nerode, 2000].

[4] The standard model of real arithmetic must be extended to include a binary time difference operation such that if $t_1\in R$ and $u=(t_2,i)\in T_\beta$, then $t_1-u=t_1-t_2$ and $u-t_1=t_2-t_1$.

In the following sections, we define each specification scheme in terms of the logic presented above, and provide a brief paraphrase of its semantics. We also show how each scheme can be used in specifying one of the example animation scenarios introduced above.

4.2 Equational Reset Scheme

The *EquationalReset* scheme allows a developer to specify a single transition operator. The developer supplies a Boolean valued expression *g* over state variables, mode variables and time, to specify the guard of the transition operator. He/she also supplies a set *e* of equations that relate the values of mode and state variables before and after the transition, to specify the operator's mode and state reset functions.

EquationalReset(g, e):

$$(t^\flat\ .\ g(m,s,t^\flat)\rightarrow\chi\ (t^\sharp\ .\ e(m,\ s,\ t^\flat,\ t^\sharp)))$$

Paraphrase: If the guard $g(m,s,t^\flat)$ is satisfied at time $t^\flat$, then the system undergoes a transition to a time $t^\sharp$ such that $N(t^\flat,t^\sharp)$ and equations $e(m,s,t^\flat,t^\sharp)$ are satisfied

The *EquationalReset* scheme can be illustrated in terms of the "Ball on Steps" example[5]. (See Appendix.) The ball has two modes of operation, bouncing and rolling, and three transitions: (1) *Bouncing→Bouncing*; (2) *Bouncing→Rolling*; (3) *Rolling→Bouncing*. The specification of the first transition is shown in the Appendix. In this instance of the *EquationalReset*

[5] Keywords of the specification language are shown in boldface. The language includes terms for the linear and angular positions and velocities of objects. (E.g., *AbsTran*s and *AbsLV* respectively refer to an object's absolute position and absolute linear velocity). The language also includes predicates for asserting various levels of continuity and smoothness in reset equations (e.g., *Continuity*, *PositionContinuity*, *VelocityContinuity*), as well as predicates for asserting that a transition must enforce the position or velocity constraints that apply in the new mode. (E.g.,*PositionConstraintsInMode*, *VelocityConstraintsInMode*). The examples also use the following Mathematica notation: The operator [ ] indicates function application. A vector is represented as {x, y, z}. The x, y and z components of vector v are v[[1]], v[[2]] and v[[3]]. A transformation rule Lhs->Rhs describes how an expression matching Lhs is replaced with the instantiation of Rhs. A pattern variable in Lhs has an underscore at the end of its name. The expression E /. R indicates the application of rule R to expression E.

scheme, the *Guard* asserts that the transition will occur when the ball is bouncing on the first step; the ball is moving downward; the lowest point of the ball is below the height of the step; and the ball's energy at impact is above a threshold. The *ResetEquations* assert that after the transition, the ball continues to bounce on the first step, and the state variables are reset so that the ball recoils from the collision losing a small amount of energy, but conserving linear and angular momentum. The "Ball on Steps" example includes three instances of the *EquationalReset* scheme, corresponding to the three transitions described above.

4.3 Specified Termination Scheme

The *SpecifiedTermination* scheme allows a developer to specify two or more transition operators that take the system through a sequence of partially specified modes. The developer supplies a guard expression *g* and a set *e* of reset equations to define a transition operator to initiate the sequence. He/she also supplies an additional termination condition *c* and set *f* of reset equations to end each of the phases of the sequence. A termination condition *c* or reset equation set *f* may involve the time, mode and state values at present and earlier event times in the sequence, and relationships among them. In the single phase version of this scheme, the developer provides only one (c,f) pair. In the multi-phase version he/she provides one or more (c,f) pairs.

SpecifiedTermination(g, e, (c, f) $^{+}$):

$$(v^{\flat} \,.\, g(m,s,v^{\flat}) \rightarrow \chi(v^{\sharp} \,.\, e(m,s,v^{\flat},v^{\sharp}) \wedge ((t \,.\neg c(m,s,v^{\flat},v^{\sharp},t))\ U\ (t^{\flat} \,.\, c(m,s,v^{\flat},v^{\sharp},t^{\flat}) \wedge \chi(t^{\sharp} \,.\, f(m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})))))$$

Paraphrase: If the guard $g(m,s,v^{\flat})$ is satisfied at time $v^{\flat}$ then the system undergoes a transition to a time $v^{\sharp}$ such that $N(v^{\flat},v^{\sharp})$ and equations $e(m,s,v^{\flat},v^{\sharp})$ are satisfied. If condition $c(m,s,v^{\flat},v^{\sharp},t)$ is true at $t=v^{\sharp}$ or some future time, then at the earliest $t^{\flat}\geq v^{\sharp}$ satisfying $c(m,s,v^{\flat},v^{\sharp},t^{\flat})$, the system undergoes a second transition to a time $t^{\sharp}$ such that $N(t^{\flat},t^{\sharp})$ and equations $f(m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})$ are satisfied.

The *SpecifiedTermination* scheme can be illustrated in terms of the "Robot on Track" example. (See Appendix.) This instance of the *SpecifiedTermination* scheme describes one phase of the robot's behavior. The *Guard* tests that the robot is in the *Drive* mode (indicating the robot is driving around the track) and the mode variable *Status[Clock]* is *Stopped* (indicating that the robot has not yet started driving). The *InitializationEquations* set *Status[Clock]* to *Running*, and keep the robot in the *Drive* mode. They preserve the positions of all the robot's state variables; however, they modify the velocities of the state variables to accord with the velocity constraints that hold in the *Drive* mode. The *TerminationCondition* asserts that the robot remains in the *Drive* mode until it is close enough to the ball to pick it up. The *FinalizationEquations* put the robot in the *Swing* mode (rotating its shoulder joint to move its arm toward the ball), and stop the clock, thus preparing for the initiation of transitions governing the *Swing* phase. The complete specification of the robot uses ten single-phase instantiations of the *SpecifiedTermination* scheme. The clock status variable is used to ensure that each phase is initiated only once per trip around the track. An alternative "Robot on Track" specification uses a single ten-phase instantiation of the *SpecifiedTermination* scheme, and avoids the use of the clock status variable, resulting in fewer transitions per cycle. Unfortunately, this specification is too lengthy to include here.

4.4 Temporal Projection Scheme

The *TemporalProjection* scheme allows a developer to specify a limited type of planned action by an autonomous agent, as it passes through a sequence of behavior modes. The developer supplies a guard expression *g* and a set *e* of reset equations to define a transition operator to initiate the sequence. In addition, he/she defines a vector **u** of unknown control variables. Finally, the developer also supplies a duration condition *d*, target condition *k* and reset equation set *f* for each phase in the sequence. The specification asserts that the control values will be chosen to make each target condition true at the time specified by the corresponding duration condition, i.e., at the end of the corresponding phase. A duration condition *d*, target condition *k* or reset equation set *f* may involve time, mode and state values at present and earlier event times in the sequence, and relationships among them. In the single phase version of this scheme, the developer provides only one (d,k,f) triple. In the multi-phase version he/she provides one or more (d,k,f) triples.

TemporalProjection(g, **u**, e, (d, k , f) $^{+}$):

$$(v^{\flat} \,.\, g(m,s,v^{\flat}) \rightarrow (\exists\ \mathbf{u})\ \chi(v^{\sharp} \,.\, e(\mathbf{u},m,s,v^{\flat},v^{\sharp}) \wedge \Diamond\,(t^{\flat} \,.\, d(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat}) \wedge k(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat}) \wedge \chi(t^{\sharp} \,.\, f(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})))))$$

Paraphrase: If the guard $g(m,s,v^{\flat})$ is satisfied at time $v^{\flat}$ then there exists vector **u** of values such that the system undergoes a transition to a time $v^{\sharp}$ such that $N(v^{\flat},v^{\sharp})$ and equations $e(\mathbf{u},m,s,v^{\flat},v^{\sharp})$ are satisfied Eventually the system reaches a time $t^{\flat}$ satisfying duration condition

$d(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat})$ and target condition $k(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat})$. Finally, the system undergoes a transition to a time $t^{\sharp}$ such that $N(t^{\flat},t^{\sharp})$ and equations $f(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})$ are satisfied.

The *TemporalProjection* scheme can be illustrated in terms of the "Dueling Spaceships" example. (See Appendix.) Each torpedo has two modes of operation (*Rest* and *Launched*) and two transitions (R→L and L→R). These two pairs of transitions (one pair for each torpedo) are specified by two instances of the *TemporalProjection* scheme (one instance for each torpedo). The scheme instance for *Torpedo1* is shown in the Appendix. The *Guard* is true when *SpaceShip1* is within its firing range of *SpaceShip2*, and *Torpedo1* is in its *Rest* mode. The *UnknownValues* (*vX*, *vY*, and *vZ*) represent the velocity with which *Torpedo1* will be launched. The *SeedEquations* specify the initial values to be used in an iterative numerical routine for determining the *UnknownValues*. The *InitializationEquations* assert that after the R→L transition, *Torpedo1* will be in the *Launched* mode; *Torpedo2* will be in the same mode as before the transition; the positions of all objects will be unchanged; the velocities of *SpaceShip1*, *SpaceShip2* and *Torpedo2* will be unchanged; and the velocity of *Torpedo1* will be (*vX,vY,vZ*). The *DurationCondition* asserts that *Torpedo1* takes a specified period of time to reach its target. The *TargetConditions* assert that the positions of *Torpedo1* and *SpaceShip2* will be the same at the time the torpedo is supposed to reach its target. The *FinalizationEquations* assert that after the L→R transition, *Torpedo1* will be in the *Rest* mode; *Torpedo2* will be in the same mode as before the transition; the positions and velocities of *SpaceShip1*, *SpaceShip2* and *Torpedo2* will be unchanged; and the position and velocity of *Torpedo1* will be the same as the position and velocity of *TorpedoDock1*, the point inside *SpaceShip1* at which it rests.

The *TemporalProjection* scheme supports a considerable number of variations in the choice of *UnknownValues*, and the way in which these values appear in other parts of the instantiated specification scheme. In the "Dueling Spaceships" example, the *UnknownValues* appear in the *InitializationEquations*, but not in the duration condition. Thus the duration of the *Launched* phase is known statically. In an alternative formulation of this problem, the unknown quantities are defined to include the direction of the torpedo's initial velocity and the duration of its flight in the *Launched* phase, while the magnitude of its initial velocity is known statically. In the multi-phase version of the *TemporalProjection* scheme, *TargetConditions* may be placed at the end of any of the phases. In the resulting problem, phase durations or phase initialization values must be chosen to achieve target conditions at multiple points in time.

Parameters to the each of the specification schemata must meet some conditions. Reset equations $e(m,s,t^{\flat},t^{\sharp})$, $e(\mathbf{u},m,s,t^{\flat},t^{\sharp})$, $f(m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})$ and $f(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat},t^{\sharp})$ must each be solvable for $m(t^{\sharp})$ and $s(t^{\sharp})$. Termination condition $c(m,s,v^{\flat},v^{\sharp},t^{\flat})$ must be composed of inequalities, since exact numeric equalities cannot be reliably tested at run time. Duration condition $d(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat})$ must be solvable for $t^{\flat}$ such that $t^{\flat} \geq v^{\sharp}$. Target condition $k(\mathbf{u},m,s,v^{\flat},v^{\sharp},t^{\flat})$ must be a conjunction of *n* equations, where *n* is the length of the vector **u** of unknown values.

5. Synthesis

The program synthesis procedure is divided into three stages. Each stage is implemented as a collection of rewrite rules in the Mathematica programming language. The first stage takes the program specification as input and generates a functional program as output. The functional program is expressed in a language with operations for higher-order numerical procedures such as integration and root extraction, as well as a variety of array operations, among others. The language is in Mathematica syntax and is not executable. It serves only as an intermediate stage in the program synthesis process. The functional program is constructed using a combination of program scheme instantiation and specialized rules for generating expressions solving systems of linear and nonlinear algebraic equations. The second stage takes the initial functional program as input and generates an optimized functional program as output. It uses a variety of program transformation rules to decompose numerical program components and optimize the flow of data to avoid unnecessary computation. Finally, the third stage takes the optimized functional program as input and generates a C^{++} program as output. It defines C^{++} function object classes implementing functional parameters to higher-order numerical procedures, and generates additional ordinary C^{++} functions that implement the optimized functional program. The underlying numerical routines are taken from the Numerical Recipes library [Press, et al. 1986]. The synthesized C^{++} program fits into a program architecture in which simulation is interleaved with rendering to generate real-time animation.

Each specification scheme is associated with a set of rewrite rules that synthesize the program components that comprise its implementation. Each rule set performs the following tasks: (1) Synthesize one or more transition operators, each composed of a guard predicate, a mode reset function and a state reset function; (2) Define any new mode or state variables needed to support the transition operators; (3) Define algebraic or differential constraints that govern the evolution of newly defined

state variables. The implementation rules generate programs that obey the *Persistent Mode Convention* [Van Der Schaft and Schumacher, 2000]. They synthesize transition rules that fire only when a transition is required by the specification.[6]

EquationalReset: The rules implementing this specification scheme generate only one transition operator. The guard predicate is obtained from the definitional expansion of the *Guard* parameter of the scheme instance. The mode reset function is obtained by symbolically solving the *ResetEquations* parameter to obtain an expression for the new values of the mode variables in terms of their old values. The state reset function is likewise synthesized by generating an expression for the new values of the state variables in terms of their old values. In this case, the system uses a set of rules for decomposing systems of equations into components, solving each component symbolically when possible, and otherwise generating expressions that solve the component using an LU decomposition routine (for linear equations) or a Newton-Raphson routine (for nonlinear equations).

SpecifiedTermination: The rules implementing this specification scheme generate N+1 transition operators, where N is the number of phases in the instantiated scheme. In some respects, the rules implementing this scheme are similar to the rules implementing the *EquationalReset* scheme described above. Each guard predicate is obtained from the definitional expansion of a *Guard* or *TerminationCondition* parameter of the scheme instance. Likewise, each mode or state reset function operates by symbolically or numerically solving equations supplied in the *InitializationEquations* or *FinalizationEquations* parameters. The process is complicated by the fact that a termination condition or finalization equation may refer to mode variables, state variables and time values at earlier event times. The referenced values are accessed in the following way: Whenever a value at event time *t* is referenced by a subsequent guard or reset equation, the value is stored in a synthesized state variable by the state reset function that fires at event time *t*. The value of the synthesized state variable remains fixed as the system passes through the sequence of phases, until the termination condition or finalization equation obtains the needed value from the state variable in which it was stored. Another complication results from the way in which the scheme semantics define a context in which transitions should fire. A guard that terminates the first phase, or a subsequent phase, should only fire when the corresponding phase is running, and not in any other context. This condition is enforced by rules synthesizing a mode variable to represent the number of the currently executing phase (if any) and arranging that each guard check the synthesized mode variable for an appropriate value before firing.

TemporalProjection: The rules implementing this specification scheme also generate N+1 transition operators, where N is the number of phases in the instantiated scheme. The state reset function initializing the first phase must determine *UnknownValues* that will guarantee satisfaction of *TargetConditions* that apply at the ends of one or more phases. It synthesizes a shooting method for this purpose [Press, et al. 1986]. The shooting method uses a Newton-Raphson routine to solve a set of simultaneous nonlinear equations. The equations involve expressions that repeatedly initialize each phase; use a Runge-Kutta routine to integrate the system derivative to the end of the phase; and evaluate the target conditions in the resulting state. After computing the *UnknownValues*, the state reset function uses them to initialize synthesized state variables. These state variables carry the *UnknownValues* forward in time to the points at which they are used by state reset functions to initialize phases. The duration of each phase is always determined in advance by the first state reset function via the shooting method. The phase durations are used to initialize synthesized clock variables. The termination of each phase is controlled by a guard predicate that references a clock variable as well as a synthesized phase number variable. In most other respects, the guard predicates and mode and state reset functions for each phase are implemented by rules that operate in a manner similar to those implementing the *EquationalReset* and *SpecifiedTermination* schemata described above.

Some of the rules implementing the first state reset function, for a single-phase *TemporalProjection* scheme, are shown in the Appendix. These rules instantiate a program scheme for a shooting method. They refer to primitives such as *NonLinearSolution* (implementing the Newton-Raphson routine) and *Integral* (implementing the Runge-Kutta routine). The expressions in bold face are expanded by the synthesis rules. The expressions in normal face appear in the synthesized functional program.

The implementation of the *TemporalProjection* scheme depends on an assumption about the way the system interacts with a user when the animation program is running. Correctness of the implementation requires that either the target conditions do not depend on user controlled variables, or the state of the user's input device

[6] In order to instantiate the simulation program scheme, we must also synthesize the system derivative function. Our procedure for synthesizing the system derivative is described in [Ellman et al, 2002] and [Ellman et al, 2003].

does not change between the time the system carries out the temporal projection, and the time the target conditions are all achieved.

One may reasonably ask under what conditions the synthesized automation will satisfy its specification. We address this issue informally as follows. To begin with, suppose that automaton A is generated by the implementation rules associated with a single instantiated specification scheme S. Then it is not hard to see that $A \models S$, as long as the code solving the reset equations, or the temporal projection equations, is guaranteed to find a solution. Now suppose that $S=S_1 \wedge \ldots \wedge S_n$ and that automaton A is constructed by separately applying the appropriate set of implementation rules to each S_i and forming the union of the synthesized transition operators. Is it the case that $A \models S$? The answer is "yes", provided (1) each S_i is an instance of either the *EquationalReset* scheme or the *ConditionalTermination* scheme; (2) all the transition equations are linear and nonsingular (so their unique solutions are found reliably); (3) the entire specification is consistent. The *EquationalReset* and *ConditionalTermination* schemata require only that specified transitions fire under specified conditions. The synthesized automaton fails to meet its specification only if some operator fails to fire when its guard is true. This can happen only if some other operator with a different effect fires instead; however, if two synthesized operators with different effects are simultaneously enabled, then the specification is inconsistent. So $A \models S$ if S is consistent.

Unfortunately, we have no such guarantee if S includes instances of the *TemporalProjection* scheme. The *TemporalProjection* scheme asserts that specified transitions will occur, and that target conditions will hold at a subsequent time. The mere firing of the synthesized transitions does not guarantee satisfaction of the specification. One can easily construct an example $S=S_1 \wedge S_2$ in which S_1 is an instance of *TemporalProjection* and S_2 specifies a transition operator that interferes with the implemented solution to the temporal projection problem. For example, in the dueling spaceships example, S_2 might specify a transition operator that implements a collision between a torpedo and an asteroid and moves a launched torpedo off its projected course. Furthermore, some other implementation of *TemporalProjection* might avoid or correct the interference. For example, the alternative implementation of S_1 might include yet another transition that immediately puts the torpedo on a new course toward its target. In such a case, we cannot blame the developer by saying the specification is inconsistent. In order to guarantee correctness under these circumstances, the rules implementing *TemporalProjection* would have to examine and take into account the entire specification. This observation would seem to be at odds with the incremental, modular approach we have taken in developing parameterized specification schemata, and associating each with its own set of implementation rules.

A developer would probably not intend for operators synthesized from two different schemata to be inconsistent or interfere with each other as described above. This suggests a practical solution. The system would separately synthesize an implementation of each instantiated specification scheme, and afterward attempt to verify consistency and absence of interference. For example, the system could check consistency by examining each pair of synthesized transition operators and trying to determine if the two guards can be simultaneously true, e.g., by linear programming for guards composed from linear inequalities, or cylindrical algebraic decomposition for guards containing polynomial inequalities. Absence of interference with temporal projection could be verified by doing a dependency analysis of the system derivative function to determine which state and mode variables influence the target conditions. If the relevant variables are modified only by transition operators implementing the temporal projection instance, then the implementation is free from interference. The verification process could result in three different answers: "correct", "incorrect" and "unknown". In the latter two cases, the appropriate remedial action would depend on the context in which the application program would be used.

In practice, there are many reasons why the synthesized program might fail to meet expectations. The specification might include nonlinear equations. Numerical solution of these equations might fail to find a root when one exists, or might find the wrong root. This is often a possibility with the TemporalProjection scheme, since the projection equations are usually nonlinear. Another problem is inherent in the nature of numerical simulation. A transition may be entirely missed if its guard is true for a period that is shorter than the step size used in searching for transition points. Two transition operators that are enabled at nearly the same time might be fired in the wrong order. Finally, a developer may write specifications that are inconsistent, e.g., two instances of the EquationalReset scheme with guards that can be simultaneously true, and reset equations that lead to different successor states.

6. Results

The specification and synthesis techniques have been implemented and used successfully to generate over a dozen different animation programs, each of which runs in our physics-based animation architecture and generates

a real-time animation of a physical scenario. These results include the examples discussed above (**Ball on Steps**, **Robot on Track**, and **Dueling Spaceships**). Some of the additional examples are described below, along with a description of the system features they demonstrate:

Ball in Box: A ball bounces around the inside of a box. Several different versions of this example demonstrate use of the *EquationalReset* scheme to describe a variety of ways of modeling collisions between a rigid body and another object.

Ball in Canal: A ball rolls without skidding around the inside of a closed track made from two cylindrical surfaces and two toroidal surfaces. The ball repeatedly breaks contact with one type surface as it makes contact with another. This example demonstrates use of mode-dependent constraints, and use of the *EquationalReset* scheme to describe discontinuous changes in velocity that occur when a mode is changed.

Car on Road Network: A car drives and coasts on an interconnected set of roads with a variety of branching points, under user control. The car is constrained to remain on each road segment as it curves, rises, falls and banks in various directions. This example also demonstrates use of mode-dependent constraints, and use of the *EquationalReset* scheme to describe discontinuous changes in velocity that occur when a mode is changed.

Traffic Control: Two cars drive around on separate circular tracks. The tracks are tangent to each other at one point. The cars must avoid colliding with each other. If a car reaches the border of the intersection region, and the other car is in the region, the first car slows to a stop at a specified point. It remains there until the other car leaves the region, and then proceeds through the intersection. This example demonstrates a single phase instance of the *SpecifiedTermination* scheme to describe the process of waiting for the intersection to be clear, and two separate single phase instances of the *TemporalProjection* scheme to describe the process of decelerating to stop at a specified point, and accelerating to a specified speed at a specified point.

Car on Stunt Track: A car drives around a track with an up-ramp, an open space and a down-ramp. Upon reaching the up-ramp, the car accelerates up the ramp, flies through the open space, and lands precisely at the start of the down-ramp, after which it decelerates to its original speed. This example demonstrates use of a two-phase version of the *TemporalProjection* scheme to describe the acceleration and flying phases; and a one-phase version of the same scheme to describe the deceleration phase.

Docking Spaceship: The motion of a spaceship is controlled by the direction and magnitude of the thrust generated by its engine. The spaceship undocks and accelerates away from a space station until reaching a specified exit point with a specified velocity. It then accelerates, coasts and decelerates along a curved path to reach the entry point of a second space station at a specified velocity. It then decelerates and docks with the second station at the instant its velocity reaches zero. This example demonstrates the use of two separate one-phase versions of the *TemporalProjection* scheme to describe the undocking and docking operations, and a three-phase version of the same scheme to describe the flight from the exit point of the first station to the entry point of the second station.

7. Related Work

Shift [Deshpande, et al. 1997] and Charon [Alur, et al. 2000] are general languages for defining, compiling and simulating hybrid systems. The research underlying these systems is focused mainly on issues of hierarchical or compositional modeling, i.e., constructing hybrid systems from large numbers of fairly simple parts. These systems provide some specification tools similar to our *EquationalReset* scheme; however, they appear to provide little or no support for specifying the kinds of temporal relationships that can be expressed using our *ConditionalTermination* and *TemporalProjection* schemata. Applications of Charon to computer animation are discussed in [Aaron, et al. 2001] and [Aaron, et al. 2002]. This work uses Charon to model low-level and high-level navigation strategies for virtual agents operating in a two-dimensional world. It is based on a model of (2D) continuous dynamics that is different from the (3D) analytical dynamics model we used in our research. Its focus is verification, rather than synthesis. It presents results of experiments using a hybrid system model checker (HyTech, [Alur, et al. 1996]) to debug an animation program.

Research on synthesis and verification of controllers for hybrid systems is surveyed in [Van Der Schaft and Schumacher, 2000] and [Labinaz, et al. 1997]. Much of this work deals with global safety properties of hybrid systems, i.e., assertions that certain undesirable behaviors can never occur [Asarin, et al. 2000], [Tomlin, et al. 2000]. For example, a safety condition might assert that an undesirable region of the mode and state space cannot be reached from other, desirable regions. In contrast to this, our work focuses on synthesis of hybrid automata satisfying local properties, i.e., sequences of transitions that occur under specified mode, state and time

conditions. Furthermore, our specifications affirmatively describe the desirable behaviors the developer wishes will occur, such as achieving a target condition, rather than negatively proscribing behaviors he/she wishes will not occur. Finally, our work also deals with the problem of synthesizing an efficient program implementing the hybrid automaton. Most research on hybrid control synthesis tends to ignore issues of implementation.

Logics of hybrid systems are discussed in [Davoren and Nerode, 2000]. This work is based on a semantic model that is somewhat different from the one presented here. A satisfaction relation between formulae and hybrid automata is defined in a two step process: (1) A hybrid automaton is interpreted as labeled transition system (LTS); (2) The LTS is deemed to satisfy or not satisfy a formula according to semantic machinery used in temporal logics of discrete systems. The resulting logics appear to be useful for describing global safety properties of hybrid systems; however, they do not appear to suit our purposes. The main problem is the absence of a "next" (χ) operator. In our specifications, we need this operator to assert the occurrence of transition sequences under specified conditions. The LTS-based semantics of [Davoren and Nerode, 2000] could probably be extended to define the "next" (χ) operator; however, for our purposes, it seemed simpler to define our own logic from scratch.

A number of other investigators have developed automated program synthesis techniques for numerical computation problems. Some use program scheme instantiation and transformation techniques that are similar to the methods of our implementation rules; however, the applications are generally quite different. SciNapse uses a knowledge base of transformation rules implemented in Mathematica to generate programs that solve partial differential equations [Kant, 1993], [Akers et al., 1998]. Amphion uses deductive synthesis to generate programs utilizing libraries of astronomical software [Lowry et al., 1994]. AutoBayes generates statistical data analysis programs from declarative descriptions of problem variables and probability distributions [Gray et al., 2003], [Fischer and Schumann, 2003]. It uses schema-guided deductive synthesis, augmented by symbolic-algebraic computation techniques. AutoFilter synthesizes programs for state estimation problems [Rosu and Whittle, 2002]. It generates programs by recursive instantiation of parameterized program-component schemata. AutoBayes and AutoFilter also construct proofs certifying key properties of synthesized programs [Schumann et al., 2003].

8. Future Work

Many variations on our specification schemata can be imagined and probably implemented using synthesis techniques similar to the ones we have developed to date. One possible extension would develop a scheme for specifying optimal control strategies based on the Pontryagin maximum principle [Hartl, et al., 1995]. Another possible extension would define tools for parallel and sequential composition of specification schemata. Such tools would allow a developer to define a scheme instantiation once, and use it repeatedly in several different contexts. The developer would gradually construct more and more complex behaviors by combining simpler ones. Yet another extension would apply our methods to animation of articulated figures engaged in walking, running, jumping and similar actions. Systems of this sort have considerably more variables than the ones we have investigated to date. It would be worthwhile to investigate problems of scale that might arise in specifying and synthesizing programs that animate such complex articulated figures.

9. Contributions

The long-term goal of this research is to develop a specification language and synthesis system that is flexible enough to handle a wide variety of animation programs, yet restricted enough to permit programs to be synthesized automatically. We have defined a set of specification schemata that represent prototype fragments of the desired language. We have demonstrated that a number of interesting animation programs can be specified in terms of these schemata, and synthesized automatically using rewrite rules that instantiate program schemata. Finally, we have suggested ways in which our specification language and synthesis system might be extended. For these reasons, we believe the research we have reported in this paper is a significant step toward our goal.

10. Acknowledgements

The research reported in this paper is supported by Vassar College. The anonymous referees provided comments that the author found quite helpful in preparing the final version of this paper.

11. References

[Aaron, et al. 2001] E. Aaron, F. Ivancic, O. Sokolsky and D. Metaxas, "A Framework for Reasoning about Animation Systems". In *Intelligent Virtual Agents*, LNCS, 2190, Springer, 2001.

[Aaron, et al. 2002] E. Aaron, F. Ivancic and D. Metaxas, "Hybrid System Models of Navigation Strategies for Games and Animations". In *Hybrid Systems: Computation and Control*, LNCS, 2289, Springer, 2002.

[Akers et al., 1998] R. Akers, P. Baffes, E. Kant, C. Randall, S. Steinberg and R. Young, "Automatic Synthesis of Numerical Codes for Solving Partial Differential Equations", Mathematics and Computers in Simulation, 45, 1998.

[Alur and Henzinger, 1992] R. Alur and T. Henzinger. "Logics and Models of Real-Time: A Survey". In *Real Time: Theory in Practice*, LNCS, 600, Springer, 1992.

[Alur, et al. 1996] R. Alur, T. Henzinger and P. Ho., "Automatic Symbolic Verification of Embedded Systems". IEEE Transactions on Software Engineering", 22, 1996.

[Alur, et al. 2000] R. Alur, R. Grosu, Y. Hur, V. Kumar and I. Lee, "Modular Specification of Hybrid Systems in Charon". In *Hybrid Systems: Computation and Control*, LNCS, 1790 Springer, 2000.

[Asarin, et al. 2000] E. Asarin, O. Bournez, T. Dang, O. Maler and A. Pnueli, "Effective Synthesis of Switching Controllers for Linear Systems". Proceedings of the IEEE, 88, 7, 2000.

[Baruh, 1999] H. Baruh, *Analytical Dynamics*. WCB/McGraw-Hill, 1999.

[Davoren and Nerode, 2000] J. Davoren and A. Nerode, "Logics for Hybrid Systems". Proceedings of the IEEE, 88, 7, 2000.

[Deshpande, et al. 1997] A. Deshpande, A. Gollu, and P. Varaiya, "SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata". In *Hybrid Systems IV*, LNCS, 1273, Springer, 1997.

[Ellman et al, 2002] T. Ellman, R. Deak and J. Fotinatos, "Knowledge-Based Synthesis of Numerical Simulation Programs for Rigid-Body Systems in Physics-Based Animation". Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002.

[Ellman et al, 2003] T. Ellman, R. Deak and J. Fotinatos, "Automated Synthesis of Numerical Programs for Simulation of Rigid Mechanical Systems in Physics-Based Animation". Automated Software Engineering, In Press.

[Fischer and Schumann, 2003] B. Fischer and J. Schumann, "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models". Journal of Functional Programming, 2003, In Press.

[Gray et al., 2003] A. Gray, B. Fischer, J. Schumann and W. Buntine, "Deriving Statistical Algorithms Automatically: The EM Family and Beyond". Proceedings of the Conference on Neural Information Processing Systems (NIPS2002), 2003.

[Hartl, et al., 1995] "A Survey of the Maximum Principles for Optimal Control Problems with State Constraints", SIAM Review 37, 2, 1995.

[Kant, 1993] E. Kant, "Synthesis of Mathematical Modeling Software". IEEE Software, 10, 3, 1993.

[Labinaz, et al. 1997] G. Labinaz, M. Bayoumi, and K. Rudie, "A Survey of Modeling and Control of Hybrid Systems". Annual Reviews in Control, 21, 1997.

[Lowry et al., 1994] M. Lowry, A. Philpot, T. Pressberger, and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments". Proceedings of the Ninth Knowledge-Based Software Engineering Conference, Monterey, CA, 1994

[Press et al., 1986] W. Press, W. Vetterling, S. Teukolsky, and B. Flannery, *Numerical Recipes*. Cambridge University Press, New York, NY, 1986.

[Rosu and Whittle, 2002] G. Rosu and J. Whittle, "Towards Certifying Domain Specific Properties of Synthesized Code". Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, Edinburgh, UK.

[Schumann et al., 2003] J. Schumann, B. Fischer, M. Whalen and J. Whittle, "Certification Support for Automatically Generated Programs". Proceedings of the 36th Hawaii International Conference on System Sciences, 2003.

[Tomlin, et al. 2000] C. Tomlin, J. Lygeros and S. Sastry, "A Game Theoretic Approach to Controller Design for Hybrid Systems". Proceedings of the IEEE, 88, 7, 2000.

[Van Der Schaft and Schumacher, 2000], A. Van Der Schaft and H. Schumacher, "An Introduction to Hybrid Dynamical Systems". Lecture Notes in Control and Information Sciences, 251, Springer, 2000.

Appendix

Example Specifications:

```
EquationalReset[        (* Ball on Steps *)
  EventTimes -> {t1,t2},
  Guard -> Mode[Ball][t1] == Bouncing  &&  Region[Ball][t1] == Step1 &&
           AbsTrans[Contact][t1][[3]]<=AbsTrans[Point1][t1][[3]] && AbsLV[Ball][t1][[3]]<0.0 &&
           RelativeEnergy[Ball, Point1, t1] > EnergyThreshold,
  ResetEquations -> BouncingResetEquations[Step1, t1, t2]]

SpecifiedTermination[   (* Robot on Track *)
  InitialEventTimes -> {t1a,t1b}, FinalEventTimes -> {t2a,2b},
  Guard -> Status[Robot][t1a] == Drive && Status[Clock][t1a] == Stopped,
  InitializationEquations -> Status[Robot][t1b] == Drive && Status[Clock][t1a] == Running &&
                    PositionContinuity[t1a,t1b]&&VelocityConstraintsInMode[t1b,{Status[Robot][t1b]->Drive}],
  TerminationConditions -> VectorLength[AbsTrans[Robot][t2a] - AbsTrans[Ball][t2a]] <= RobotArmRange,
  FinalizationEquations -> Status[Robot][t2b] == Swing && Status[Clock][t2b] == Stopped &&
                           PositionContinuity[t2a, t2b] && VelocityContinuity[t2a, t2b]]

TemporalProjection[   (* Dueling Spaceships *)
  InitialEventTimes -> {t1a,t1b}, FinalEventTimes -> {t2a,2b},
  Guard -> Status[Torpedo1][t1a] == Rest && InRange1[SpaceShip1, SpaceShip2, t1a],
  UnknownValues -> {vX, vY, vZ}, SeedEquations -> {vX == 0.0, vY == 0.0, vZ == 0.0},
  InitializationEquations -> Status[Torpedo1][t1b]==Launched&&Status[Torpedo2][t1b]==Status[Torpedo2][t1a] &&
    Continuity[SpaceShip1,t1a,t1b] && Continuity[SpaceShip2,t1a,t1b] &&
    Continuity[Torpedo2,t1a,t1b] && PositionContinuity[Torpedo1, t1a, t1b] &&
    AbsLV[Torpedo1][t1b][[1]] == vX && AbsLV[Torpedo1][t1b][[2]] == vY && AbsLV[Torpedo1][t1b][[3]] == vZ,
  DurationCondition -> t1a + TorpedoTransitTime1 == t2a,
  TargetConditions -> PositionEquality[Torpedo1, SpaceShip2, t2a],
  FinalizationEquations -> Status[Torpedo1][t2b] == Rest && Status[Torpedo2][t2b] == Status[Torpedo2][t2a] &&
           Continuity[SpaceShip1,t2a,t2b] && Continuity[SpaceShip2,t2a,t2b] && Continuity[Torpedo2,t2a,t2b] &&
           PositionEquality[Torpedo1, TorpedoDock1, t2b] && VelocityEquality[Torpedo1, TorpedoDock1, t2b]]
```

Example Synthesis Rules:

```
TPInit[initTs_, finalTs_, unknowns_, seedEqns_, initEqns_, durCond_, targetConds_] ->
  Function[{pgmParms, ctlParms, mode, state, time},
    Let[{{seed, EqnSolution[seedEqns, unknowns]},
         {solution, NonLinearSolution[
                      Function[{guess},
                        Let[{{projectedState, ProjectedState[initTs, finalTs, unknowns, initEqns, durCond]
                                                [pgmParms, ctlParms, mode, state, time, guess]}},
                          Residual[unknowns, finalTs, targetConds]
                                [pgmParms, ctlParms, guess, projectedState]]],
                      seed]}},
        InitialState[initTs, unknowns, initEqns][pgmParms,ctlParms,mode,state,time,solution]]]

ProjectedState[initTs_, finalTs_, unknowns_, initEqns_, durCond_] ->
   Function[{pgmParms, ctlParms, mode, state, time, guess},
            Let[{{initialState, InitialState[initTs,unknowns,initEqns][pgmParms,ctlParms,mode,state,time,guess]},
                 {duration, EqnSolution[durCond /. Second[initTs]->0.0, First[finalTs]]}}]
              Integral[DerivativeFunction[pgmParms, ctlParms, mode], state, time, time + duration]]
```