Qualitative modeling of hybrid systems*

Oleg Sokolsky and Hyoung Seok Hong
Department of Computer and Information Science
University of Pennsylvania
{sokolsky,hshong}@saul.cis.upenn.edu

Abstract

The paper discusses an approach to construct discrete abstractions of hybrid systems by means of qualitative reasoning. The work is performed in the context of a modeling language for hybrid systems CHARON. We introduce a qualitative version of the language and describe the abstraction technique using a motivational example. The resulting abstract model is conservative and can be used to analyze properties of the original hybrid system.

Keywords: hybrid systems, abstraction, qualitative reasoning.

1 Introduction

Distributed embedded control systems usually consist of multiple components that exhibit both continuous and discrete behavior. Hybrid systems is a widely-used mathematical model for such systems. Since many embedded systems are safety-critical, it is important to analyze hybrid systems for correctness. The combination of discrete and continuous state changes makes analysis of hybrid systems an extremely challenging task. Algorithmic verification techniques require that we work with a finite representation of the state space of a system. Abstractions and approximations are necessary to make algorithmic analysis possible. In this paper, we consider the construction of discrete approximations of hybrid systems by means of *qualitative reasoning*.

Qualitative reasoning [12, 7, 9] is a well-established technique in the Artificial Intelligence community. It allows researchers to model physical systems using incomplete information. Often, there is not enough information about the system to represent it by means of differential equations. However, the basic relations between the variables in the system are known. In this situation, qualitative models can be used to capture the incomplete knowledge in a model, which can be simulated to obtain a rough outline of the system behavior. Furthermore, as more information about the system becomes available, the qualitative model can be refined to provide a more accurate description.

An alternative role for qualitative reasoning has received much less attention. Qualitative models can be seen as discrete abstractions of continuous and hybrid systems. They provide a *conservative approximation* of the system behavior. That is, every possible behavior of a system is captured by some qualitative behavior, but not all qualitative behaviors necessarily correspond to a real system behavior. Qualitative models, which exhibit finite-state behavior, can be fully explored by a verification tool and thus provide a means of conservative analysis of hybrid systems.

We explore qualitative abstractions of hybrid systems in the context of CHARON [1], a recently introduced novel language for hybrid system modeling. The language supports specification of multi-threaded (parallel or distributed) systems as a hierarchy of concurrent agents and complex behaviors within one thread as a hierarchy of modes. CHARON has a number of high-level language features such as data encapsulation and scoping, exception handling, and instantiation of parameterized objects. CHARON has been given formal compositional semantics [2] that makes modular reasoning about hybrid systems possible. In this paper, we describe a qualitative variant of the CHARON language that will allow us to construct conservative qualitative approximations of CHARON models and analyze them using state-space exploration techniques.

*This work is supported in part by the NSF grant CCR-9988409, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA ITO MOBIES F33615-00-C-1707, and ONR N00014-97-1-0505 (MURI).

Related work. Qualitative reasoning has emerged in the past decade as a mature technique for approximate reasoning. Qualitative abstractions are primarily targeted at continuous systems expressed as differential equations. However, tools such as QSIM [12] are capable of modeling discrete transitions and are thus applicable to general hybrid systems. An application of qualitative reasoning to hybrid systems in the context of controller synthesis is discussed in [5]. Similar in spirit but different technically is recent work on verification of safety properties in continuous systems via qualitative abstractions [14, 13]. There, conservativeness of qualitative abstractions is used to prove that violations of safety properties is impossible in the concrete model. Analysis is based on reasoning about individual trajectories, while we are concentrating on more traditional in the verification are state-machine representations.

It is well-known that formal verification techniques such as reachability analysis and model checking are undecidable for hybrid systems in general [11]. Research has concentrated on decidable subclasses of hybrid systems, or on finding conservative approximations for hybrid systems. See [3] for a survey of state-of-the-art techniques.

The need to construct finite abstractions of infinite-state systems is not limited to the hybrid systems domain. Predicate abstraction [10] is a promising technique for reducing the range of a variable to a finite set of "important" values. Effectively, predicate abstraction determines appropriate landmark values for each variable in the program. The proposed approach can be seen as an extension of the predicate abstraction techniques for hybrid systems.

The paper is organized as follows: in Section 2 we introduce the language CHARON and informally describe its semantics. In Section 3, we present a framework for qualitative description of systems. Our approach follows the treatment of [12], which describes the simulation of qualitative models using a tool QSIM. Our approach is not based on simulation, however. We construct a qualitative model as a hierarchical state machine and explore its state space to determine its properties. The benefits of this approach are discussed in Section 5. Then, in Section 4 we present the qualitative variant of CHARON and its semantics. The semantics is compositional in the sense that behaviors of composite objects are computed from their components. A simple example is presented in Section 5 to illustrate the approach.

2 CHARON modeling language

CHARON is a high-level language for modular, hierarchical description of hybrid systems. CHARON describes a hybrid system as a collection of concurrent *agents* that interact with each other through shared variables and bounded-capacity channels[1]. Agents have well-defined interfaces, consisting of its input and output variables and channels. Sequential behavior is described in CHARON by means of *modes*. Modes also have interfaces, consisting of entry and exit *control points*, through which a thread of control enters and leaves the mode.

Intuitively, an execution of a CHARON specification is an alternating sequence of *discrete* and *continuous* steps. Discrete steps are instantaneous mode switches, while continuous steps take a finite amount of time when no control changes occur.

The hierarchy in CHARON is twofold. The *architectural hierarchy* describes how the *agents* in the system interact with each other, hiding the details of interaction between sub-agents. The *behavioral hierarchy* describes behavior of each agent as a collection of *modes*, hiding the low-level behavioral details. At the leaves of the architectural hierarchy are *primitive* agents that do not have concurrent sub-agents. Behaviors of primitive agents are captured by *modes*, described below.

Agents and modes operate on sets of typed variables. In each agent or mode, variables are partitioned into global and local variables. Global variables are further categorized into input and output variables. Also, variables can be either analog or discrete. Discrete variables are updated by discrete steps during the execution; analog variables are updated in a continuous fashion, but may also be reset by discrete steps. During a continuous step, analog variables follow a *flow*, a smooth continuous function of time. We assume that analog variables have type real.

A mode is a hierarchical hybrid state machine equipped with analog and discrete variables. While a mode stays in a state, its analog variables are updated continuously according to a set of constraints, which take the form of differential and algebraic equalities and inequalities. Taking transitions from one state to another, the mode updates its discrete variables. States of the mode are submodes that can have their own behavior. A mode has a number of control points, through which control enters and exits the mode. That is, to perform a computation in one of its submodes, a mode takes a transition to an entry point of that submode. When the computation in the submode is complete, a transition from an exit point of the submode is taken. The mode also has entry transitions, from

[1] Channels are not considered in this paper.

an entry point of the mode to an entry point of one of its submodes, and exit transitions, from an exit point of a submode to an exit point of the mode. Entry transitions specify initial states of a mode and may give initial values to the variables of the mode.

Primitive modes, which do not have any submodes, can have multiple entry points but only the default exit point. Since there are no internal control points in a primitive mode, every entry transition is also an exit transition. Intuitively, a primitive mode stays during its execution in its default exit point.

Transitions are labeled with *guards* and *actions*. A guard is a predicate on the values of the mode variables. A transition is *enabled* when its guard is true. An action is a partial *state transformer*: when a transition is taken, variables of the mode are updated according to the action of the transition.

Before the computation of a mode is completed, it may be interrupted by a group transition, originating from a default exit point of the mode. After an interrupt, control is restored to the mode via a default entry point. We use *invariants* to force one of the outgoing transitions. Control can reside in a mode only as long as its invariant is satisfied. As soon as an invariant is violated, control has to leave the mode by taking one of the enabled outgoing transitions.

Each primitive agent has an associated *top-level* mode that specifies its behavior. A top-level mode has a single non-default entry point *init*, which is used to initialize the mode before execution. Since agents never terminate, their top-level modes do not have non-default exit points.

An object-oriented feature of CHARON is that declarations of modes and agents act as classes. A parameterized declaration of a mode or an agent can be instantiated in a model multiple times with different values of parameters.

Semantics. CHARON is given formal compositional trace semantics. Each agent or mode is characterized by its interface and the set of traces it allows. Traces of a mode are formed by the flows defined by the mode constraints, interleaved with discrete steps of the mode, in which a mode transition is taken, updating local and output variables, and discrete *environment steps* that change the values of input variables. The set of traces of a composite mode can be computed from the traces of the submodes. While executing in one of the submodes, the mode follows a trace of the active submode that complies with the constraints of the mode.

A primitive agents has as its traces the traces of its top-level mode, restricted to the global variables of the agent. A trace of a composite agent is such that, when projected on the global variables of a sub-agent, it yields a trace of the sub-agent. Semantics of agents is also compositional. The set of traces of an agent can be computed from the sets of traces of the sub-agents.

A motivational example. We use a simple example throughout the paper to illustrate the facilities of CHARON. It represents a swimming pool equipped with a pump that controls the water level, and a sign that tells whether the water is deep enough to swim. The architecture of the model is shown in Figure 1. It consists of three agents, `Pool`, `Pump` and `Sign`. The first agent represents the water in the pool and its behavior is given by a single differential equation relating the flow of water and its level. Two other agents are instantiations of parameterized agents `WaterPump` and `Switch`. Their top-level modes are presented in Figure 2. The agent `WaterPump` controls the water flow. The pump can be turned on or off, maintaining constant flow: when the pump is on, water flows into the pool, when it is off, the water flows out of the pool. Modes `On` and `Off` are instances of the mode `SteadyMode` with different values of parameters. In addition, the pump has two transient modes, `TurnOn` and `TurnOff`. These modes are instances of the mode `TransientMode`, when the water flow smoothly changes from one steady-mode level to the other. Entry transitions of the primitive modes in the example are trivial and we omit them in the figures. Initially, the pump starts in the `On` or `TurnOff` submode depending on the water level, as prescribed by the entry transitions. Then, the pump cycles through on and off phases.

3 Fundamentals of qualitative reasoning

3.1 Qualitative variables

A qualitative variable has an associated type, or *quantity space*. A quantity space consists of a finite set of *landmarks*. A landmark represents an "interesting" value of the variable and may be a symbolic or integer constant. We assume that landmarks of a variable are completely ordered. That is, when we consider a variable v with the quantity space $\{v_1, v_2, ..., v_n\}$, we will always assume $v_1 < v_2 < ... < v_n$.

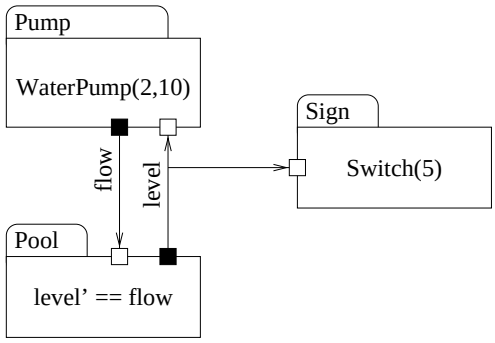


Figure 1: A swimming pool in CHARON

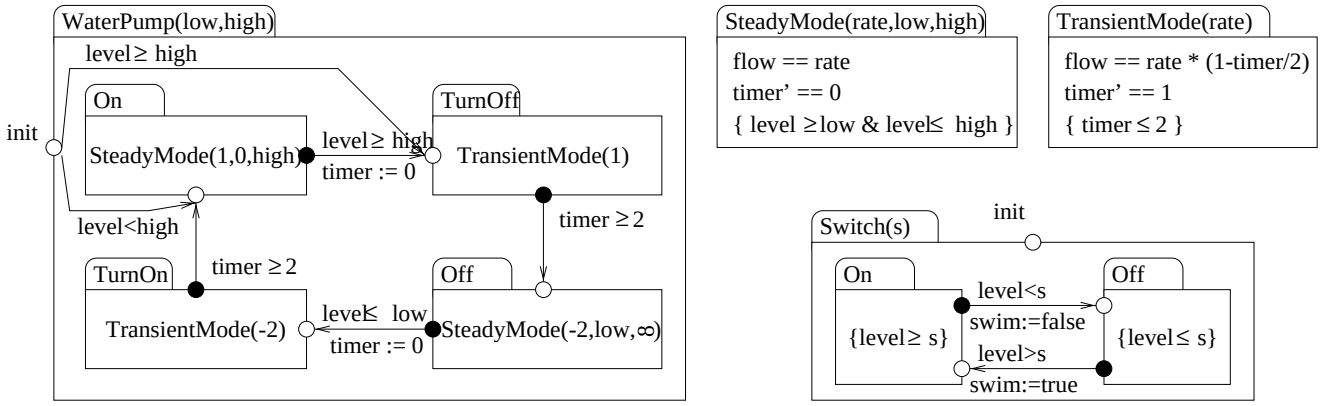


Figure 2: Declarations for the swimming pool

During an execution, we track not only the values of the qualitative variables, but also the directions of their change. It allows us to increase accuracy of qualitative behaviors. A *valuation* of a qualitative variable v is a pair (l, d), where l is a qualitative value of v and $d \in \{-, *, +\}$. The meaning of a valuation (l, d) for v is that the value of v is l and the first derivative with respect to time is negative if $d = -$, is zero if $d = *$, and is positive if $d = +$. For discrete qualitative variables the only possible value of d is their valuations is $*$.

During an execution, an increasing qualitative variable may either reach the next larger value or stop increasing. Similarly, a decreasing variable may reach the next smaller value or stop decreasing. A stationary variable may turn into either increasing or a decreasing one without changing its qualitative value. When the variable assumes its smallest landmark value, it cannot be decreasing; similarly, when it assumes the largest landmark value, it cannot be increasing. As an example, consider the possible evolutions of a single unconstrained qualitative variable with the quantity space $\{l_1, l_2, l_3\}$, represented as a state machine in Figure 3.

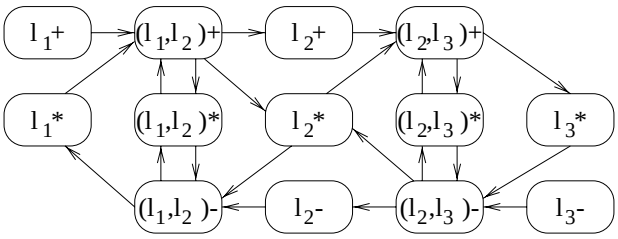


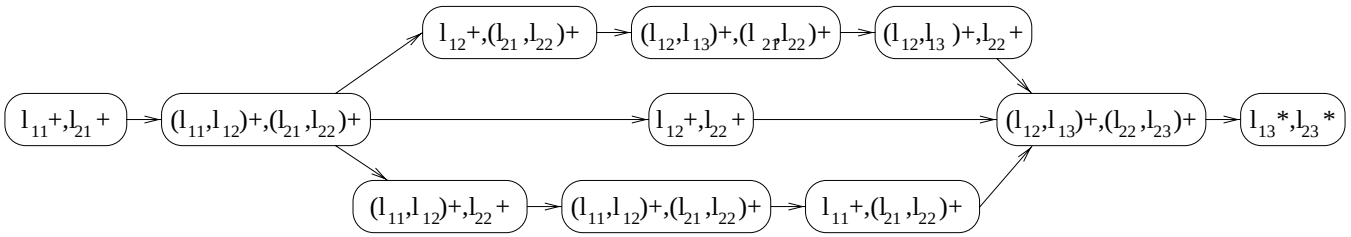
Figure 3: State machine for a single variable

3.2 Qualitative Constraints

Multiple variables in the same execution can evolve independently of each other. If variables are to evolve in a coordinated fashion, we need to introduce constraints.

Constraints over qualitative variables take the form $v \circ E_q$, where $\circ \in \{<, \leq, =, \neq, >, \geq\}$ and E_q is a *qualitative expression*. Qualitative expressions are constructed from qualitative variables and constants by means of qualitative operators described below. The quantity spaces of the left-hand and right-hand sides of a constraint must be the same.

Functional operators. A functional operator represents an underspecified function from a tuple of quantity spaces into a quantity space. An example of a functional operator is a monotonic function $M^+(v)$ for a qualitative variable v. A constraint of the form $v_1 = M^+(v_2)$ specifies that in every state of the execution, the directions of change in the valuations of v_1 and v_2 agree. In addition, the relation between some elements of the quantity spaces of v_1 and v_2 may be known. In this case, the valuations also have to agree on those elements. For example, consider a model with variables v_1 with the quantity space $\{l_{11}, l_{12}, l_{13}\}$ and v_2 with the quantity space $\{l_{21}, l_{22}, l_{23}\}$. Let the constraint be $v_1 = M^+(v_2)$ with the set of related values $\{(l_{11}, l_{21}), (l_{13}, l_{23})\}$. The "increasing" part of the state machine shown below. Symmetric "stable" and "decreasing" parts are omitted.



Arithmetic qualitative operators. Arithmetic qualitative operators are special cases of functional operators. An arithmetic qualitative operator is a mapping from a pair of quantity spaces to a quantity space. When components of the quantity spaces are integer constants, the natural rules can be used to define the arithmetic operators. For symbolic constants, the user must specify the mapping explicitly. For example, an addition operator from $\{0, On, Inf\}^2$ to $\{0, On1, On2, Inf\}$ may be given as $\{((On, 0), On1), ((0, On), On1), ((On, On), On2)\}$. Addition and multiplication operators are always commutative and monotonic in both arguments; the landmark value 0 is always used naturally in all arithmetic operators. If quantity spaces are signed (that is, 0 is an element of the quantity space), multiplication of positive values yields a positive value, etc. Other properties of the arithmetic operators (such as associativity) may not be satisfied.

Qualitative differential constraints. In addition to constraints on variables, qualitative constraints can apply to first derivatives of variables. Remember that a valuation of a qualitative variable includes a three-valued component representing the direction of its change. A qualitative differential constraint constrains this component of the valuation. Values of the right-hand side expression are taken in relation to 0, which must be contained in the quantity space of the expression. For example, constraint $v' = 1$ means that the direction-of-change component of the valuation for v is always +; that is, v monotonically increases.

4 Qualitative CHARON

In this section, we describe QCHARON, that replaces real variables of CHARON with qualitative variables. The change affects only the modes. The agent hierarchy and interfaces of agents are unaffected (except that types of agent variables change to qualitative types).

We introduce the following quantity spaces for the variables used in the swimming pool example:

level { 0, *Lo*, *Swim*, *Hi*, *Owf* }
flow { *mInf*, *Out*, 0, *In*, *Inf* }
timer { 0, *Ready*, *Inf* }

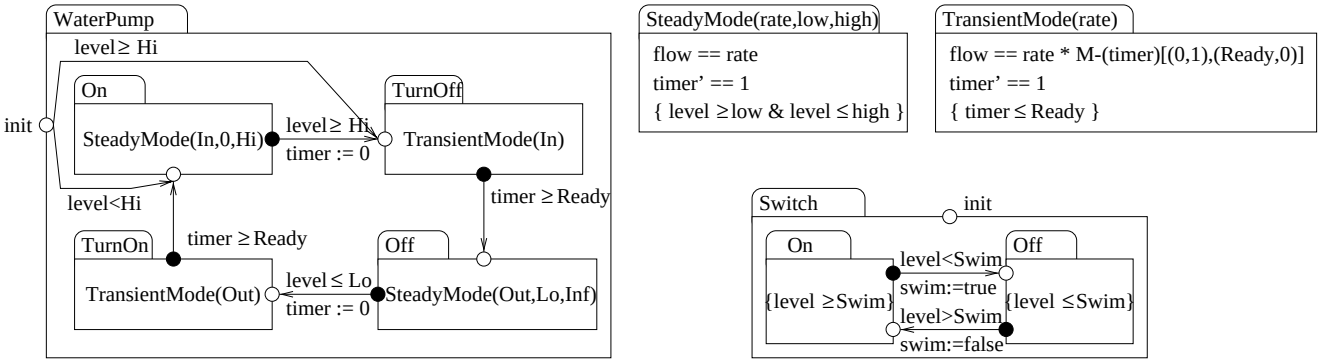


Figure 4: Declarations of qualitative agents and modes

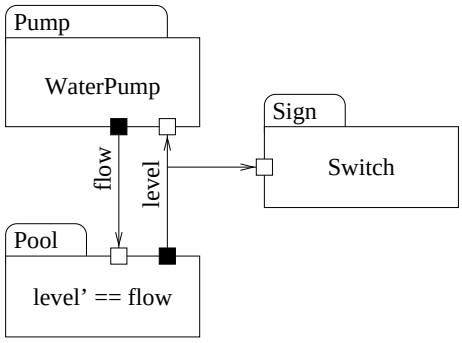


Figure 5: Architecture of the qualitative model

The landmarks for each variable are as follows: water level is at *Lo* when the pump has to be turned on, at level *High* the pump needs to be turned off, and it is safe to swim when the level is above *Swim*. When the flow of water is enough to make water level rise, the value of the flow is *In*, if the level is decreasing, the value is *Out*. The value 0 means that the level is constant. These landmark values for flow are chosen to be used in the differential constraint of the `Pool` agent. Finally, the timer has only one interesting value: duration of the interval that the pump spends in a transient state, denoted *Ready*. Note that all these values are either constants or parameters in the CHARON model.

We also need to turn the expressions of the CHARON model into qualitative expressions. Consider the differential equation of mode `TransientMode`, expression $rate * (1 - timer/2)$ becomes $rate * (M^-(timer)[(0,1)(Ready,0)])$. It represents a monotonically decreasing function of *timer*, which has value 1 when *timer* is 0, and 0 when *timer* has the qualitative value *Ready*. We do not need to specially define the multiplication operator in this expression, because we are interested only in the sign of the expression. All other expressions in the example are transformed into the qualitative form by replacing concrete constants and parameters with qualitative constants.

Figure 4 shows the qualitative version of the swimming pool example. Figure 5, which represents the architecture of the qualitative model is the same as Figure 1 with the parameters removed. The agents `WaterPump` and `Switch` are no longer parameterized, because their parameters are used in a qualitative way, and are now captured as types of qualitative variables. However, not all parameters in the model are removed. Submodes of `WaterPump` are still parameterized, since they are instantiated multiple times with different values of parameters.

Semantics. Following the setup of CHARON, semantics of a QCHARON specification is given by the interface of the mode (its control points and global variables), and set of traces that the specification can produce. We will define set of traces in a bottom-up fashion, starting from the leaves of a behavioral hierarchy. For each mode, we will first capture the set of its executions as a state machine. It is important to notice that this state machine is a semantic object and does not have to be constructed explicitly during analysis of a QCHARON specification. Executions are projected onto the global variables of the mode to yield the set of traces of the mode.

First, a state of a mode with variables $v_1, ..., v_n$ is a tuple of valuations for the variables of the mode. A mode

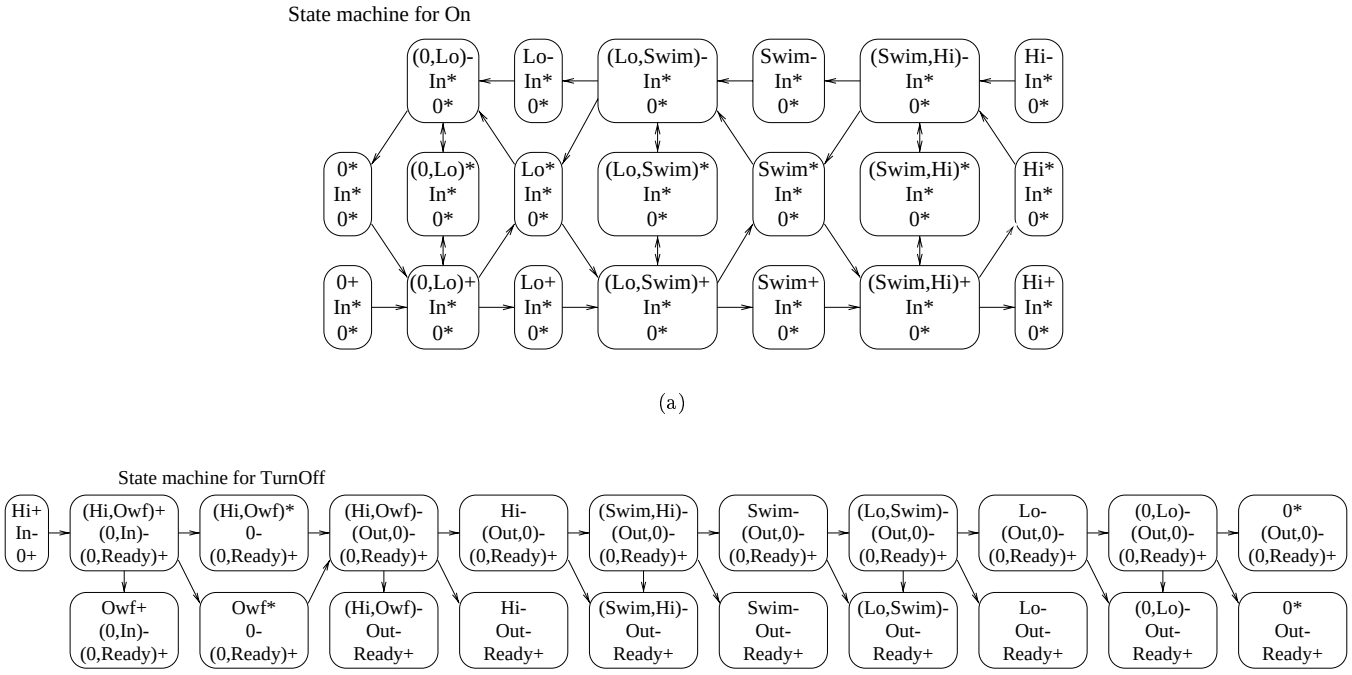


Figure 6: Runs of the submodes of `WaterPump`

variable can be updated either in a discrete or in a continuous fashion. Discrete variables are changed by the transitions of the mode. Discrete variables can assume only landmark values during an execution. A continuous variable follows a flow, i.e. a differentiable function, during an execution and thus can assume as values its landmarks and the intervals between adjacent landmarks.

Executions of a primitive mode are represented as a state machine, where each state corresponds to a state of the mode. Transitions represent possible changes in variable valuations, such that valuations in the states connected by a transition agree with all the constraints of the mode. A *run* of a state machine is a sequence of states such that every two consecutive states in a sequence are connected by a transition. To represent executions, we extend the state machine with special nodes that do not correspond to a state of the mode, but capture entry and exit points of the mode. Each entry point is connected by a transition to every state in which the values of variables agree with the guard and the action of the entry transition attached to the entry point. Every state is connected by a transition to the exit point node, since an execution can be interrupted at any time.

We show the state machine for `SteadyMode` (instantiated as `On`) in Figure 6(a). In the mode `On`, variables *flow* and *timer* are constant and variable *level* is an input variable whose value is constrained by the invariant of the mode and the direction of change is unconstrained. Figure 6(b) shows a fragment of the state machine for `TransientMode` (instantiated as `TurnOff`). In this mode, *timer* increases, *flow* decreases, and *level* is unconstrained. The fragment is chosen to comply with the constraint on level from the agent `Pool`, which will be applied when behaviors of individual components are composed into behaviors of the whole system. Since the entry transitions of the modes are trivial, the entry node of each state machine is connected to every state and we do not show them to avoid cluttering the figure.

We can now give semantics to composite modes by first extending the mode hierarchy at the leaves, replacing the primitive modes with their respective state machines. The state machine representing the executions of a composite mode m is obtained by "flattening" this hierarchical state machine into an ordinary state machine. To construct the flattened state machine of a mode from the flattened state machines of submodes, we perform the following steps.

1. Connect by transitions the states in the state machines of the submodes according to the transitions of the mode. Consider a mode m with submodes m_1 and m_2. Let m_1 have an exit point x and m_2 have an entry

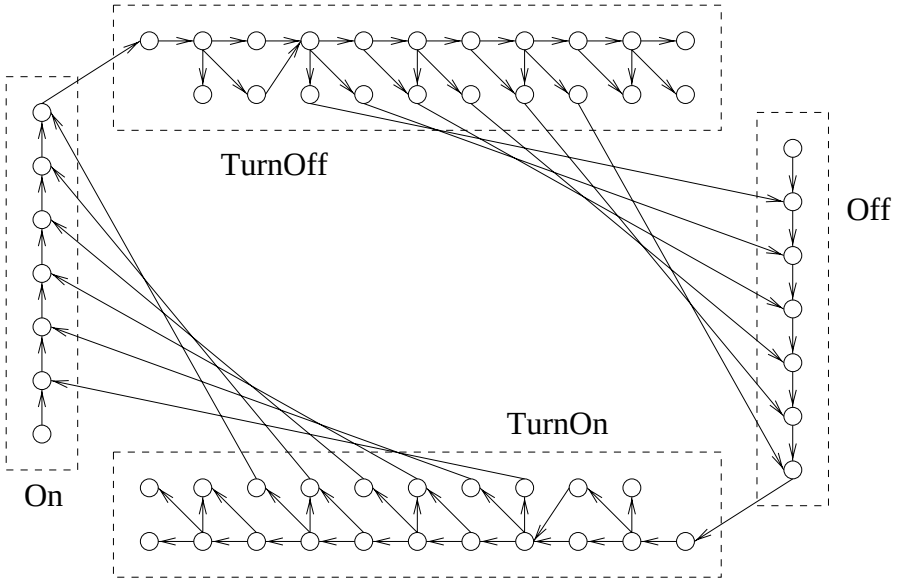


Figure 7: Qualitative runs of the swimming pool model

point e. Let t be a transition of m from x to e. Consider a state s_1 in the state machine of m_1 that is connected by a transition to the node corresponding to x, such that s_1 satisfies the guard of t. The state s_1 is connected by a transition to every state s_2 that agrees with the action of t. This operation is repeated for every s_1 in m_1 and for every t in m.

2. Next, we introduce the nodes for the control points of the mode and introduce transitions similarly way to the regular transitions: for each entry transition from an entry point e to an entry point e_1 of a submode m_1, we add a transition from the node for e to every state in the state machine of m_1 that is connected to e_1 and agrees with the guard and action of the entry transition.

3. After all transitions have been introduced, the nodes for the submode control points are removed.

Behaviors of a primitive agent are the same as the behaviors of its top-level mode. For a composite agent, we can compute the flattened state machine by taking a product of the state machines for the sub-agents, in which a transition is possible if and only if it is allowed by constraints in all the agents. In the swimming pool example, when we compose agents `WaterPump` and `Pool`, traces of `WaterPump` now have to satisfy the relationship between the variables *flow* and *level* prescribed by the `Pool`. In particular, this restriction effectively reduces the state machine of Figure 6(a) to the bottom row of states. We show the flattened state machine for the swimming pool example in Figure 7. To avoid cluttering the figure, we omit the labels of the states, but group together the states corresponding to executions within the same submode of `WaterPump`.

In the same way as CHARON, the semantics of QCHARON is compositional, making the construction of the flattened state machine unnecessary. The set of traces permitted by the state machine of a mode can be computed from the transitions and constraints of the mode and the sets of traces of the submodes.

5 Conclusions and Discussion

We have presented preliminary results on the construction of conservative approximations of CHARON specifications by means of qualitative reasoning. The approach differs both from the existing abstraction techniques for hybrid systems analysis and from traditional uses of qualitative reasoning. A lot remains to be done to turn this approach into an abstraction methodology for hybrid systems, but the first impression is encouraging.

Comparing our approach with that of qualitative simulation [12], we note that the hierarchical state machine yields a much more compact representation of the set of execution traces, in general, than an explicit representation. For comparison, we modeled our swimming pool example in QSIM, the foremost tool for qualitative simulation. Much to our surprise, the problem turned out to be intractable for QSIM. It exceeded the limit of 500 traces that

we set for the simulation, and ran out of memory with the trace limit removed. The reason for this explosion of behaviors seems to lie in the way QSIM introduces dynamic landmarks into execution traces. When the mechanism of dynamic landmarks is turned off, QSIM produces an *envisionment* (a tree view of the state machine) which is similar in the number of states to our state machine. At the same time, dynamic introduction of landmarks may allow us to improve the accuracy of our qualitative models (see below), as long as we keep them from exploding the state space.

Comparing the proposed technique to the abstraction techniques for hybrid systems described in [3], it is clear that qualitative reasoning yields coarser abstractions than other existing techniques. This has its advantages and disadvantages. On the one hand, qualitative abstractions are much easier to compute and manipulate. This allows us to handle larger specifications. On the other hand, qualitative abstract models are much less precise than state-of-the-art techniques. They admit many behaviors that the original system cannot exhibit. In the discussion below, we consider ways to improve precision of the abstraction without incurring too much overhead.

Our future work on this topic will concentrate on the following aspects:

- **Improving accuracy of abstractions.** A qualitative description of a mode or an agent represents all possible values parameters and constants in the model, because they are now assume qualitative values. This makes it more difficult to check properties of concrete systems. In effect, the question "does model A have property B?" in qualitative analysis becomes "can we select the values for parameters and constants in A such that the resulting model has property B?" As a result, a qualitative model will allow more qualitative behaviors than the original hybrid model, instantiated with a fixed set of parameters, would.

  In terms of our swimming pool example, the question whether the pool can overflow is answered positively. Indeed, if we choose the value of `High` too close to `Owf`, an overflow is possible in the `TurnOff` mode, while the water level is still rising above `High`. At the same time, parameters in the original swimming pool example were chosen so that overflow cannot occur. In order to get a more precise answer, we need to constrain the model further to express the relative values of qualitative landmarks. The problem can be addressed from two directions. On the one hand, *semi-quantitative* reasoning [4] extends purely qualitative reasoning with partial numerical information. The second approach involves model refinement techniques such as proposed in [6]. These two approaches will be the main direction of our future research in this area.

- **Local landmarks.** We observe that not every landmark value of a variable needs to be considered in every mode. For example, the value *Swim* of the variable *level* is used only in the agent `Switch`, and values *Low* and *High* are used only in the agent `WaterPump`. We can use this fact to reduce the sizes of mode state machines, and refine them as needed during analysis.

- **More complex functions.** Functions used in our example are very simple, which makes the qualitative abstraction easy to perform. In general, providing accurate qualitative representations for a complex function may be difficult. The problem has been studied in the context of qualitative simulation previously [8]. We will explore *mode splitting* to make construction of qualitative representations simpler. With this technique, we partition the ranges of variables in a CHARON model in such a way that in each block of the partition the function can be simplified or approximated differently, yielding expressions with simpler qualitative form in each case. Then, we introduce a separate mode for each block of the partition, with additional invariants to ensure that the values of variables are within the block. Transitions between the new modes will correspond to the execution moving from one block of the partition to another. In this way, a mode in a CHARON model will correspond to a number of modes in QCHARON, but each mode will provide a more precise approximation.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Proceedings of Hybrid Systems: Computation and Control, Third International Workshop*, volume 1790 of *LNCS*, pages 6–19. Springer-Verlag, 2000.

[2] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of Hybrid Systems: Computation and Control, Fourth International Workshop*, March 2001.

[3] R. Alur, T.A. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 2000.

[4] D. Berleant and B. Kuipers. Qualitative and quantitative simulation: Bridging the gap. *Artificial Intelligence Journal*, 95(2):215–255, 1997.

[5] G. Brajnik and D.J. Clancy. Control of hybrid systems using qualitative simulation. In *Working notes from the 11th International Workshop on Qualitative Reasoning about Physical Systems (QR-97)*, June 1997.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV '00*, July 2000.

[7] J. De Kleer and J.S. Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[8] A. Farquhar and G. Brajnik. A semi-quantitative physics compiler. In *Working Papers of the International Workshop on Qualitative Reasoning (QR-94)*, 1994.

[9] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[10] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of CAV '97*, pages 72–83. Springer-Verlag, July 1997. introduced predicate abstraction.

[11] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. *Journal of Computer and System Sciences*, 57:94–124, 1998.

[12] B. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.

[13] T. Loeser, Y. Iwasaki, and R. Fikes. Safety verification proofs for physical systems. In *12th International Workshop on Qualitative Reasoning*, pages 88–95. AAAI Press, May 1998.

[14] B. Schults and B. Kuipers. Proving properties of continuous systems: Qualitative simulation and temporal logic. *AI Journal*, 92:91–129, 1997.