Solving Complexity and Ambiguity Problems within Qualitative Simulation

by

Daniel Joseph Clancy, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1997

Solving Complexity and Ambiguity Problems within Qualitative Simulation

Approved by
Dissertation Committee:

I dedicate this dissertation to the many people who have touched my life. In particular, my loving wife Suzanne, my child to be "Peanut", my ever–patient and supportive parents Mary Ethel and Jerry, and my father Joeseph Ignatius Clancy, who was taken before his time. Thanks for all love.

Acknowledgments

I am grateful to the family, friends and professional colleagues who have supported me throughout the long process required to obtain my PhD. My advisor Benjamin Kuipers provided countless insights, financial support, and more importantly friendship throughout. In addition, I would like to thank my dissertation committee Elizabeth Bradley, Ray Mooney, Bruce Porter and Risto Miikkulainen for their support. In particular, Elizabeth Bradley performed the impressive task of reading the entire dissertation and offering countless comments to greatly improve the quality of the document. Other colleagues and friends that have supported me throughout this process include Ken Forbus, Rich Doyle, Brian Falkenhainer, Giorgio Brajnik, and all of the members of the qualitative reasoning community.

I would like to thank my fellow students Tara Estlin, Dave Moriarty, John Zelle, Paul Baffes, and Jeff Mahoney for the many conversations and the hours wasted playing darts and participating in stupid arguments. Other students who have supported me and provided useful insights into my research include Jeff Rickel, Adam Farquhar, and Rich Mallory along with many others. I also would like to make a special note of the contributions of Bert Kay to the work presented here. A friend and colleague whose life was recently cut short, Bert and I spent 6 years together working together under Ben and for the last few years was my sole companion in the QR group at UT. My thoughts and prayers go out to his wife Meg and his twin daughters Sonia and Nina.

Besides my professional colleagues, I would like to thank my many friends from Duke, Boston and Austin. They have provided countless hours of joy doing little more than hanging out, drinking beers and playing basketball. In particular, I would like to thank those friends who have been closest throughout the years, "D", Dom and Doug, for their invaluable friendship.

Last and obviously not least, I would like to thank my loving wife Suzanne, my supportive parents, Mary Ethel and Jerry, my many siblings, Joe, Mary, Paul, Bob, Carolyn and Barbara, my mother-in-law Janet and my faithful dog Thor. All

of these people have meant a great deal to me throughout the years. My parents have always offered guidance, support and a place to call home. Joe provided me with many years of enjoyment when we were kids and a foil to compete against as we grew older while the rest of my siblings provided me with constant entertainment and companionship ensuring that there was never a dull moment around the house. Similarly, Thor, has ensured that there is never a dull moment around our house here in Austin. Finally, my wife Suzanne has been eternally patient as I have worked over the years to obtain my PhD. She is my wife, best friend and companion and I look forward to the many years that we have left together and the children to come.

Daniel Joseph Clancy

The University of Texas at Austin
December 1997

Solving Complexity and Ambiguity Problems within Qualitative Simulation

Publication No. ____________________

Daniel Joseph Clancy, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Benjamin Kuipers

Qualitative simulation is used to reason about the behavior of imprecisely defined dynamical systems for tasks such as monitoring, diagnosis, or design. Often, however, simulation of complex dynamical systems results in either an intractable simulation or an ambiguous behavioral description. These results have caused concern regarding the scalability of techniques based upon qualitative simulation to real–world problems. Two different approaches are used to solve these problems: 1) abstraction and problem decomposition techniques are used during simulation to focus on relevant distinctions thus reducing the overall complexity of the simulation; and 2) the expressiveness of the modeling language is extended to allow the modeler to incorporate additional information.

Model decomposition and simulation (DecSIM) uses a component–based simulation algorithm to reason about the complex interactions within a sub–system independent of the interactions between subsystems. Variables within the model are partitioned into closely related components and a separate behavioral description is generated for each component. Links are maintained between related components to ensure that all of the constraints in the model are satisfied. DecSIM results in exponential speed–up for models that lend themselves to decomposition. Furthermore, DecSIM is guaranteed to generate a behavioral description that is equivalent to the description generated by a standard QSIM simulation modulo the temporal

ordering of events for variables in separate components.

A common source of irrelevant distinctions within qualitative simulation is intractable branching due to *chatter*. Chatter occurs when the derivative of a variable is constrained only by continuity within a restricted region of the state space. We present two different abstraction techniques that completely eliminate the problem of chatter by abstracting a chattering region into a single abstract state. Both techniques retain the QSIM soundness guarantee and eliminate all instances of chatter without over–abstracting.

To address the problem of an ambiguous behavioral description, Temporally Constrained QSIM (TeQSIM) integrates temporal logic model checking into the qualitative simulation process allowing the modeler to specify behavioral information via *trajectory constraints*. Trajectory constraints, specified using an extension of a propositional linear–time temporal logic, can be used to focus the simulation, simulate non–autonomous systems and reason about boundary condition problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

What happens when you throw a ball in the air? Well, unless you are in the Mir space station, it probably goes up and then comes back down. Or, what happens when you put a pot of water on the stove? It boils. For humans these are simple questions that can often be answered with little knowledge about the dynamical properties of the system being reasoned about and certainly without requiring a precise numerical model. While many people may answer these questions simply based upon experience, others may use knowledge about the qualitative relationships between the variables to reason about the potential behavior of the system or explain particular behaviors.

These examples demonstrate the use of qualitative information in "common sense reasoning"; however, this type of information is also quite useful within the domain of engineering problem solving. The world is infinitely complex and thus when reasoning about a system an engineer must use abstraction, approximation, aggregation and other techniques to generate a manageable representation of the forces and factors that are relevant to the problem being addressed. While a host of numerical techniques exist for reasoning about the behavior of a physical system, often precise numerical information may not be available thus making it difficult to automate the reasoning process.

But what type of knowledge do people use when they reason abstractly about the behavior of the system and how can this knowledge be represented and used in an automated fashion? Over the last 15 years, the field of qualitative reasoning has attempted to answer these and other questions trying to gain an understanding of how people reason about autonomous change in the physical world. Of particular interest within the field are techniques for deriving behavioral information about a system from a structural model describing the relationships between the variables.

Qualitative simulation (Kuipers, 1994; Forbus, 1984; de Kleer & Brown, 1985) allows the modeler to explicitly represent and reason about an imprecisely defined dynamical system using an abstract structural model to derive a description of all possible qualitatively distinct behaviors. These techniques have been used for tasks such as monitoring, diagnosis, design and explanation.

Traditionally, qualitative simulation uses a state-based representation to describe the behavior of the system via a tree or graph of alternating time-point and time-interval states. A state provides a qualitative value for each variable within the model. A branch results within this description when a state has multiple potential successor states due to the imprecision within the model. For example, figure 1.1 shows the results from the simulation of a simple bathtub model. The behavior tree contains three distinct paths corresponding to the three qualitatively distinct behaviors consistent with the constraints provided within the model: the bathtub can overflow, reach a steady state prior to filling, or it can reach reach a steady state and become full at the same time. In addition, the model can be augmented by quantitative information in the form of ranges on landmark values and envelopes for monotonic functions to refine the description. Such a model is called a semi-quantitative model (Kay, 1991; Kay & Kuipers, 1993; Berleant & Kuipers, 1988).

Qualitative simulation provides a guarantee that the behavioral description generated provides a representation of all potential real-valued trajectories for the class of dynamical systems defined by the model. Any given instance of this class, however, will only exhibit one of these behaviors. In the bathtub example, the actual behavior exhibited by a given bathtub depends upon the size and shape of the tank, the inflow rate, and the size of the drain.

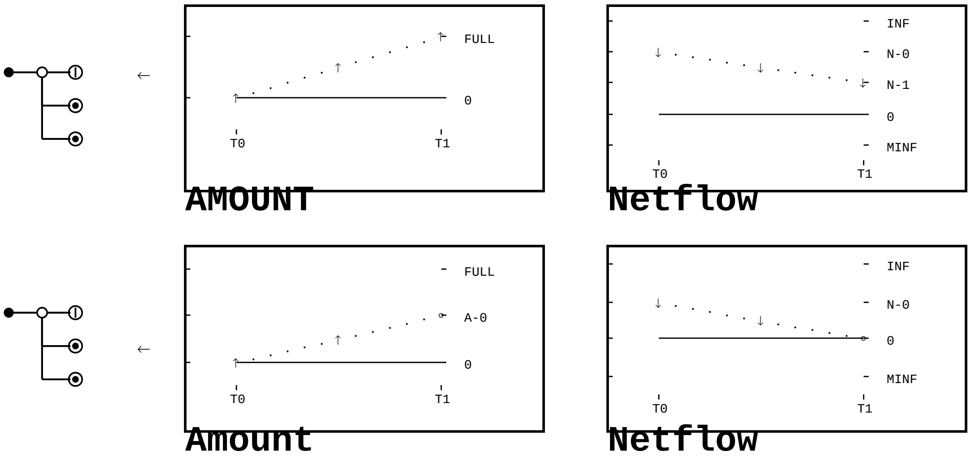
Recently the efficacy of these techniques for solving real–world problems has been questioned due to the complexity of the simulation process and the ambiguity of the behavioral description (Doyle & Sacks, 1992). Simulation of larger, more complex models often results in an intractable simulation that may or may not contain the discriminatory information required to address the task at hand. Furthermore, it is often quite difficult to develop a model and extract information from the results due to the complexity of the behavioral description.

At times, an ambiguous description or complex simulation are inherent limitations of the abstract representation used to describe the system. Imprecision within the model can result in a wide range of qualitatively distinct behaviors. Often, however, many of the distinctions are artifacts of the simulation algorithm and irrelevant to the current task. In addition, the modeler may possess additional information not contained within the model that could constrain the space of consistent

ODE Equation	Model	
	Constraints	Variables

Amount′ = *Inflow* − *Outflow*	`(M+ amount outflow)`
	`(ADD netflow outflow inflow)`
	`(d/dt amount netflow)`
	`(constant inflow)`
		`(amount (0 FULL))`
		`(outflow (0 inf))`
		`(inflow (0 if* inf))`
		`(netflow (minf 0 inf))`

(a) Qualitative model



(b) Results

Simulation of a simple bathtub model (a) with four variables results in a total of three behaviors. The behavior of Amount and `Netflow` are displayed for the first two behaviors (b). The third behavior is simply a combination of these behaviors in that `Amount` becomes steady when it reaches `Full`.

Figure 1.1: Qualitative model and simulation of a simple bathtub

behaviors. Ideally, the techniques used for simulation should mimic a human's ability to reason at multiple levels of abstraction within different parts of the system depending upon the information available and the distinctions of interest. Furthermore, the system should be able to reason with different types of information in an integrated fashion and to systematically apply simplifying assumptions when needed.

The research described within this dissertation builds upon existing work within the field to provide a flexible, efficient qualitative simulation algorithm that reasons at multiple levels of abstraction and allows the modeler to describe the system with various types of information. The following contributions are provided.

1) Decomposition and abstraction techniques reduce the complexity of the simulation by eliminating irrelevant distinctions that lead to combinatoric branching. These techniques result in an exponential speed–up for certain models and ensure that the complexity of the simulation is a function of the problem specification as opposed an artifact of the simulation algorithm.

(2) Temporal logic–based *trajectory constraints* extend the expressiveness of the modeling language allowing the modeler to directly specify time–varying constraints on the behavior of the system. Traditionally, qualitative simulation only allows the modeler to specify atemporal, structural constraints. The qualitative simulation is restricted to the region of the trajectory space identified by the trajectory constraints. This extension provides the modeler with a declarative language for specifying assumptions, controlling the simulation and incorporating additional information.

All of the techniques described here have been developed as extensions to the QSIM qualitative simulation algorithm (Kuipers, 1994). The problems addressed, however, are general problems that are encountered when using other qualitative simulation algorithms (Forbus, 1984; Bredeweg, 1992; de Kleer & Brown, 1985) and the solutions are equally relevant to these algorithms.

1.1 Reducing the Complexity of a Simulation

Qualitative simulation traditionally uses a state–based representation that provides a single level of detail and highlights a fixed set of distinctions. Each qualitative state provides unique values for all of the variables within the model. A branch results in the behavioral description when the model fails to constrain the valid combinations of variable values that can follow from a given state. The complexity of a simulation is dominated by the complexity of this representation and thus the degree to which the model restricts branching. Many of these distinctions, however, are often irrelevant or of secondary importance to the current task thus needlessly increasing the complexity of the simulation. To reduce the simulation complexity, we must first understand why branching occurs and then modify the representation so that the simulation focuses on relevant distinctions.

The types of branches within a simulation are determined by the transition values allowed by continuity (see appendix D for a listing of the valid transitions). Branches can be divided into two categories: event and chatter branches.

Definition 1.1 (Event branching) *An* event *occurs when a variable reaches a landmark or becomes steady (i.e. its derivative crosses zero). An* event branch *occurs when there are multiple events following a time–interval state whose ordering is unconstrained by the model.*

For example, in the bathtub model (see figure 1.1), a three way event branch occurs following state S_1 due to the complete temporal ordering of two events: the bathtub becoming full and the amount of water in the tub becoming steady.

Definition 1.2 (Chatter branching) *A* chatter branch *occurs following time-point t_i if there exists a variable v that is currently steady whose direction of change is constrained only by continuity. A three way branch*[1] *occurs depending upon whether the variable is increasing, decreasing or steady in the interval $(t_i\ t_{i+1})$.*

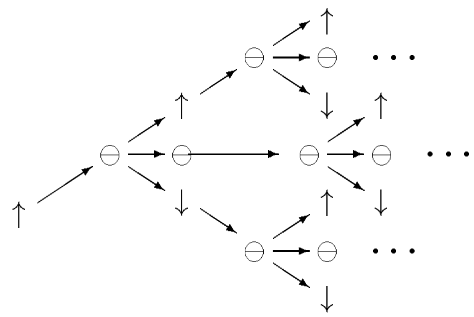
Since the variable's direction of change is unconstrained, the variable is free to become steady again and the process repeats itself for an arbitrary number of qualitative states. (See figure 1.2.) While some behaviors exit the unconstrained region of the state space, others will continue cycling between different values for the direction of change resulting in an infinite number of behaviors that remain within this region of the state space.[2]

Figure 1.3 shows a simple example of chatter when simulating a model of a two tank cascade. The direction of change for tank B's netflow (i.e. `netflowB`) is influenced by two opposing forces following the initial state. Both the inflow and the

[1]A two way branch occurs if the analytic function constraint is applied since the branch where the variable remains steady over the ensuing interval is filtered. The analytic function filter assumes that if a variable is constant over an interval, then it must always be constant (Kuipers, 1994).

[2]The behavioral description is infinite due to the introduction of landmarks during the simulation. If landmarks are not introduced, then the number of behaviors within the chattering region is exponential in the number of chattering variables. The introduction of landmarks is discussed in section 2.

outflow are increasing and thus the direction of change for `netflowB` is unconstrained resulting in a series of chatter branches. In more complicated systems, multiple variables can chatter simultaneously thus complicating the description further.

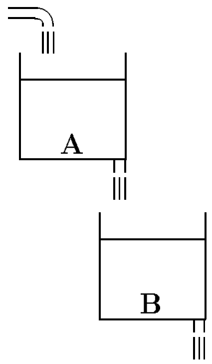


- Once a variable whose derivative is unconstrained becomes steady, it can increase, decrease or remain steady in the following time–interval state. A three way branch occurs in the behavioral description. This process continues as the variable returns to steady and another three way branch occurs.

Figure 1.2: Repeating three way chatter branch

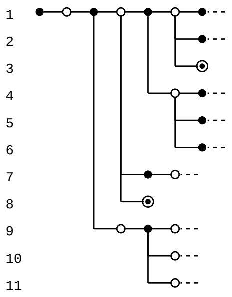
Event branching and chatter branching pose different problems with respect to the complexity of the simulation and the information provided by the distinctions. In the case of chatter, distinctions in a variable's direction of change within a chattering region of the state space provide no additional information since the simulation simply computes all possible trajectories consistent with continuity. Furthermore, chatter results in exponential branching that makes the results of the simulation unusable. Thus, chatter branching must be eliminated from the behavioral representation.

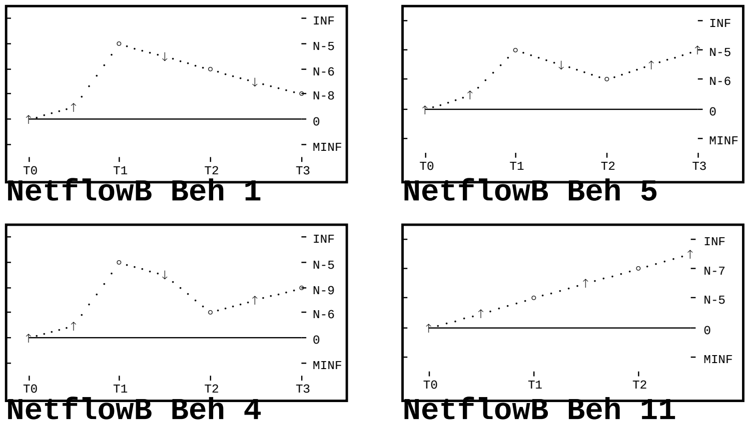
Event branches, on the other hand, often provide useful information that corresponds to qualitative distinctions specified within the model. In the bathtub example, the event branch differentiates between the tank reaching quiescence versus an overflow condition. Event branching, however, can also result in the complete temporal ordering of a set of unrelated events. As the size of the model grows, the likelihood of any two events being unrelated increases and event branching becomes more problematic. For example, simulation of a qualitative model of a V-8 automobile engine containing a description of all eight spark plugs could exhibit over 500,000 behaviors simply describing the order in which the eight spark plugs may have fired if all possible orderings of these eight events are allowed. Thus, to provide



$$
\begin{aligned}
A' &= in - f(A) \\
B' &= f(A) - g(B) \\
& \quad f, g \in M^+
\end{aligned}
$$

(a) Cascaded Tanks





(b) Behavior Tree **(c) Assorted Behaviors for NetflowB**

`NetflowB` is influenced by two opposing forces: the inflow is increasing, but so is the outflow (a). (*i.e.* Both $f(A)$ and $g(B)$ are increasing and thus the sign of B' is ambiguous since there are two opposing forces.) Thus, its direction of change is unconstrained resulting in an infinite behavioral description (b) as the derivative of `NetflowB` moves freely. Four of the behaviors are displayed (c).

Figure 1.3: Simulation of a Two Tank Cascade

a tractable simulation, certain event branches must be eliminated

1.1.1 Model Decomposition and Simulation (DecSIM)

A divide and conquer approach is used to address the problem of combinatoric event branching via the model decomposition and simulation (DecSIM) algorithm (Clancy & Kuipers, 1997b). The variables within the model are partitioned into components so that closely related variables are contained within the same component. Each component is simulated independently to eliminate event branching between variables in separate components. A separate behavioral description is generated for each component. Links are maintained between the separate behavioral descriptions to reason about the interactions between the components using a causal analysis of the model.

The degree to which the model constrains the behavior of the system determines the complexity of the qualitative simulation. By partitioning the model into closely related components, DecSIM divides the problem into smaller, more tightly constrained sub-problems. In addition to reducing the overall size of the problem, this process eliminates a primary source of complexity by separating unrelated variables into distinct components. For models that lend themselves to decomposition, this process results in an exponential speedup in simulation time as well as a more compact and understandable description of the potential system behaviors.

This dissertation provides a detailed description of the DecSIM simulation algorithm. Theoretical results show that the behavioral description generated describes the same set of real–valued trajectories as a standard qualitative simulation except for the temporal ordering of events in separate components. Theoretical results are also presented demonstrating the benefits of the DecSIM algorithm in terms of the complexity of the simulation. Finally, an empirical analysis is provided demonstrating the performance of DecSIM on a variety of models.

1.1.2 Chatter abstraction techniques

The phenomenon of chatter is a difficult problem that has addressed by a variety of techniques. In general, these techniques either filter spurious behaviors or eliminate distinctions in a variable's direction of change throughout the simulation. While existing techniques work well in certain situations, they do not provide a general solution that eliminates all instances of chatter without reducing the constraining power of the model. The algorithms presented here not only provide such a technique, but they also handle more complex forms of chatter not addressed by existing techniques.

Two separate abstraction techniques (Clancy & Kuipers, 1997a, 1997c) have been developed to eliminate chatter: chatter box abstraction and dynamic chatter abstraction. Both of these techniques abstract chattering regions of the state space into a single qualitative state within the behavioral description. The techniques differ in the manner in which the boundaries of the chattering region are identified and how the successors of the abstracted state are computed.

Chatter box abstraction (Clancy & Kuipers, 1993) uses static information contained within the model to identify potentially chattering variables over an open time interval. A recursive call to the simulation algorithm restricted to the potentially chattering region of the state space to determine which of these variables actually exhibit chatter and to identify the states exiting the chattering region. The behavioral description generated by this simulation is abstracted into a single qualitative state within the main description and its successors are derived from the results of the recursive simulation.

Dynamic chatter abstraction provides a more efficient solution that extends chatter box abstraction by eliminating the need to perform a simulation to determine the behavior of the system over the potentially chattering region. Instead, the algorithm uses a dynamic analysis of the model along with the current qualitative state to determine which variables, if any, chatter and to create the abstract state. The algorithm computes the successors of this abstract state using an extension to the standard QSIM successor generation algorithm.

By recursively calling the simulation algorithm, chatter box abstraction exploits the inference capabilities already contained within the algorithm. This facilitates the integration of chatter box abstraction with other extensions to the basic simulation algorithm and lends itself to a straight–forward proof of the ability of chatter box abstraction to eliminate all instances of chatter without over–abstracting.

Dynamic chatter abstraction, on the other hand, provides a scalable solution to chatter elimination that exploits knowledge about the type of inferences made by the simulation algorithm. While it is more efficient than chatter box abstraction, it is not as extensible. Dynamic chatter abstraction is evaluated both theoretically and empirically. We show that it eliminates all instances of chatter without over–abstracting given two clearly stated assumptions. The empirical evaluation, using a corpus of over 20 models collected from various researchers within the qualitative reasoning community, validates the theoretical results and demonstrates the benefits provided by dynamic chatter abstraction when compared to chatter box abstraction with respect to the complexity of the simulation.

1.2 Focusing and Refining Ambiguous Behavioral Descriptions

The detail contained within the original model limits the behavioral information computed by the simulation algorithm. Multiple behaviors result when there is insufficient information to discriminate between alternative behaviors. Additional information that distinguishes these behaviors may be available to the modeler, however, it may be difficult or impossible to represent this information via structural constraints. For example, when simulating a pot of water on the stove, the modeler may know that the water boils within five minutes; however, this information cannot be incorporated into the model. This additional information can refine the behavioral description generated during the simulation. Alternatively, the modeler may be interested in focusing the simulation on certain regions of the trajectory space. For example, when designing a nuclear power plant, a modeler may be interested in focusing his attention on behaviors in which the plant can explode.

Traditionally, qualitative simulation techniques do not allow the modeler to represent behavioral information within the model. Structural equations are used to constrain the simultaneous values of related variables. Non–local information constraining the behavior of the system across time is not used except through the implicit application of continuity during the simulation or through specific extensions to the simulation algorithm.[3]

1.2.1 Temporally Constrained QSIM (TeQSIM)

The Temporally Constrained QSIM (TeQSIM, pronounced *tek'sim*) (Brajnik & Clancy, 1996a, 1996b, 1997) algorithm extends the expressiveness of the modeling language allowing the modeler to specify both continuous and discontinuous behavioral information via *trajectory constraints*[4] to focus the simulation and refine the

[3] For example, the QSIM energy and non–intersection constraints both use information contained within an entire behavior (*i.e.* non–local information) as opposed to limiting inferences to information within a single state (*i.e.* local information).

[4] A *trajectory* for a tuple of variables $<v_1, \ldots, v_n>$ over a time interval $[a, b] \subseteq \Re^+ \cup \{0, +\infty\}$ is defined as a function τ mapping time to variable values defined over the set of the extended reals, *i.e.* $\tau : [a, b] \rightarrow (\Re \cup \{-\infty, +\infty\})^n$.

behavioral description. Trajectory constraints contain both qualitative and quantitative information and are formulated using a combination of temporal logic expressions, a specification of discontinuous changes and a declaration of external events. TeQSIM integrates an incremental model checking algorithm into the qualitative simulation process to eliminate partial behaviors that fail to model the trajectory constraints and to refine behaviors using qualitative information contained within the trajectory constraints.

A billiards shot provides a simple example of how trajectory information can be used when reasoning about the behavior of a dynamical system. The objective is to derive quantitative bounds on the velocity required to hit a successful billiards shot given an initial description of the table. (*i.e.* How hard and in which direction does the white ball need to be hit so that it strikes a designated ball and propels it into a particular pocket?) A standard qualitative simulation derives a description of all potential behaviors including behaviors in which the shot is not successful. The modeler then needs to identify those behaviors that are consistent with the desired shot and extract the required information. TeQSIM, on the other hand, allows the modeler to use trajectory constraints to specify boundary conditions on the desired behaviors to restrict the simulation to the region of the state space in which the cue ball strikes the target ball (before stopping or striking a cushion) and the target ball hits the desired pocket (before stopping or striking a cushion). This example demonstrates only some of the benefits provided by trajectory constraints within TeQSIM. In addition, trajectory constraints can can be used to focus the simulation for larger, more complex simulations, simulate non–autonomous systems and incorporate observations into the simulation.

This dissertation provides a detailed description of the TeQSIM algorithm along with examples demonstrating how trajectory constraints can be used within a simulation. In addition, theoretical results are presented showing that a behavior is included within the behavioral description if and only if there exists an extension of the behavior that can model all of the trajectory constraints.

1.3 Evaluation

We use both theoretical and empirical results to validate and evaluate the techniques presented within this dissertation. The theoretical results guarentee that the behavioral description generated by the qualitative simulation contains all real–valued trajectories consistent with the information contained within the model. (*i.e.* We retain the QSIM soundness guarentee.) In addition, we show that the abstraction and

Model	Description
Plant Physiology	Various models and scenarios generated via automated modeling from a large–scale botany knowledge–base (Rickel & Porter, 1994).
Reaction Control System	Navigation system used on the NASA Space Shuttle
Glucose-Insulin Interaction	Variety of models describing various physiological scenarios with respect to the relationship between glucose and insulin in the human body.
Continuously Stirred Tank Reactions (CSTR)	A number of CSTR models are available for various chemical reactions.
Three tank cascade	Three tank cascade model described in section 4.
Mansfield dam control	Dam outflow control for Lake Travis, that uses real–world quantitative information
Graft versus host disease (GVHD)	Model of the physiological process leading to GVHD in humans using various scenarios.
Proportional Integral Controller	Various controller models.
Heart Model	Model of the human cardiovascular system.
Iron Metabolism	Model of the effects of external perturbations on the human iron-metabolism process.
Van der Pol equation	Model of the Van der Pol equation
Predator–Prey	Model describing the cyclic relationship between a predator population and a prey population.

Table 1.1: Models used for evaluation.

decomposition algorithms do not introduce new behaviors within the description. (*i.e.* The constraining power of the model is not reduced.)

The empirical evaluation reinforces the theoretical results and demonstrates the effectiveness of these techniques at reducing the complexity of a simulation. To validate the theoretical results, we tested each algorithm on a corpus of models collected from various researchers throughout the field of qualitative reasoning. Figure 1.1 contains a brief description of some of these models. In general, the corpus contains all relevant medium–to–large sized QSIM models that we could obtain. Thus, the corpus reflects the type of problems currently addressed using qualitative simulation.

The empirical evaluation uses "extendible" models to evaluate the asymptotic performance of the algorithms presented with respect to execution time. An extendible model is a model whose size can be incrementally increased by adding variables in a controlled manner. For example, an N tank cascade is an extendible model since the size of the model can be increased by adding tanks to the bottom of the cascade. Extendible models allow us to determine how the algorithms scale as the size of the model grows.

1.4 Overview

The next chapter provides an introduction to qualitative simulation, relevant definitions and other background information. The rest of the dissertation is divided into three self–contained chapters describing the DecSIM, chatter abstraction, and TeQSIM algortihms respectively. Each chapter contains additional background information, a detailed presentation of the algorithm, theoretical results, evaluation, related work, and future extensions. These chapters can be read in any order. The final chapter contains concluding remarks along with a more general discussion of how qualitative simulation can be used as one component in a larger problem solving context. This chapter includes specific proposals for future research to facilitate the application of qualitative simulation to larger, more realitic problems.

Chapter 2

Qualitative Simulation

2.1 Qualitative models

Qualitative simulation uses a structural model of a dynamical system to derive a behavioral description. In the QSIM representation, the model is represented by a *qualitative differential equation* (QDE). A QDE is an abstraction of an ordinary differential equation (ODE) that specifies qualitatively significant distinctions and relationships within the model.

Definition 2.1 (Qualitative differential equation) *A QDE is a tuple of four elements* $<V, Q, C, T>$ *where the elements are defined as follows:*

- V *is a set of* variables.
- Q *is a set of* quantity spaces*. A separate quantity space is defined for each variable in* V*.*
- C *is a set of* constraints *on the variables in* V*.*
- T *is a set of* transitions *defining the boundary of the domain of applicability of the QDE.*

Definition 2.2 (Consistent model) *A QDE is considered consistent if and only if there exists at least one qualitative behavior describing the variables within the model that is consistent with the constraints. Otherwise the model is considered inconsistent.*

2.2 Qualitative states

A symbolic language is defined to represent the state of the system at either a time–point or time–interval. A qualitative state provides a qualitative value for all of the variables within the model along with a value for the special variable `TIME`.

Definition 2.3 (Qualitative value) *A* qualitative value *for the variable v is described by a magnitude (*qmag*) and a direction of change (*qdir*). The qmag is described with respect to a finite, totally ordered* quantity space $l_1 < l_2 < \ldots < l_k$ *defined for v.*

- *The qmag is either a landmark value l_j or an interval between two adjacent landmarks (l_j, l_{j+1}).*
- *The qdir represents the sign of the derivative of the variable where*

$$qdir = \begin{cases} inc & \text{if } \dot{v} > 0 \\ std & \text{if } \dot{v} = 0 \\ dec & \text{if } \dot{v} < 0 \end{cases}$$

The qualitative value, magnitude, and derivative for a variable v within state S are written $Qval(v, S)$, $Qmag(v, S)$ and $Qdir(v, S)$ respectively.

Each qualitative state corresponds to a point or region within the real–valued state space for the dynamical system being modeled. However, the symbolic language defined by the model allows each state to be viewed as a point in a symbolic state space of dimensionality $2n$ where n is the number of variables within the model. (The dimensionality is $2n$ since each variable has a magnitude and direction of change.) Throughout the dissertation references to a "region of the state space" refers to the qualitative state space which in turn maps to a corresponding region of the real–valued state space for the dynamical system.

Viewing a qualitative state in this manner allows us to extend the concept of a qualitative state to include regions within the qualitative state space by extending the definition of a qualitative value.

Definition 2.4 (Abstract qualitative value) *An* abstract qualitative value *is a qualitative value containing either an abstract qmag, an abstract qdir or both. An abstract qmag is defined as an interval between two non-adjacent landmarks within the quantity space while an abstract qdir is defined as a list of qdirs where*

(dec std inc)	*corresponds to*	$\dot{v}$ *being unspecified*
(dec std)	*corresponds to*	$\dot{v} \leq 0$
(std inc)	*corresponds to*	$\dot{v} \geq 0$

Note that the value (inc dec) is not allowed since it does not correspond to a continuous interval. A value of `nil` *is interpreted as abbreviation for (dec std inc) for a qdir and* $(-\infty\ \infty)$ *for a qmag.*

An *abstract qualitative state* is a qualitative state containing at least one abstract qval.[1] In general, term *qualitative state* will be used for the generalized concept of a qualitative state (i.e. either an abstracted or a non-abstracted state) and the terms *concise* or *non–abstracted* will be used to refer to a qualitative state or value that has not been abstracted.

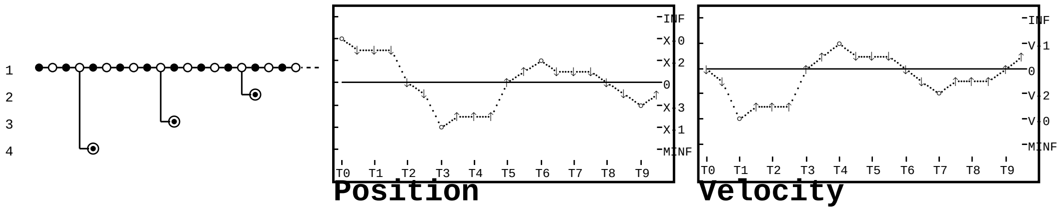
Thus, a qualitative state refers to a region of the qualitative state space. As such, two qualitative states can be related through a subset/superset relation while a concise qualitative state can be described as an *element* of the region of the state space described by an abstract qualitative state.

2.3 Behavioral description

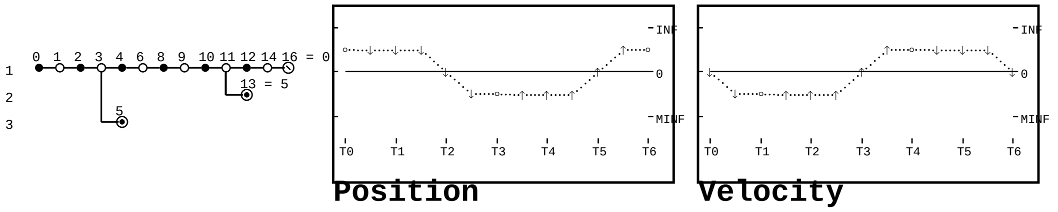
Qualitative simulation describes the dynamic behavior of the system by either a graph or tree of alternating time–point and time–interval points. An envisionment graph represents the behavior as a state-transition graph of qualitative states in which each unique set of qualitative values corresponds to a single, unique state within the representation. Each path within the graph corresponds to a qualitatively distinct behavior. A behavior tree, on the other hand, represents the behavior of the system as a *tree* of qualitative states[2]. Similarly, each path within the tree corresponds to a distinct qualitative behavior. In addition, a behavior tree representation allows for the introduction of new, qualitatively interesting distinctions throughout the simulation. QSIM inserts new landmarks into a quantity space to represent points at which the variable reaches a critical value (*i.e.* its derivative becomes zero). The introduction of landmarks allows QSIM to represent behaviors such as decreasing and increasing oscillations and to infer additional quantitative information thus providing a more refined behavioral description. Furthermore, it

[1] An *abstract* qualitative state is conceptually equivalent to the idea of a *partial* qualitative state as used by Kuipers(Kuipers, 1994).

[2] The representation is not strictly a tree since cycles are represented via links from leaf states to preceding states within a behavior. The representation, however, is referred to as a *tree* throughout the literature and thus we will continue to refer to it in this manner



(a) Behavior tree (b) Behavior plot from behavior tree simulation



(c) Envisionment graph (d) Behavior plot from envisionment simulation

- A behavior tree simulation of the damped spring describes the initial segment of a potentially infinite behavior tree (a). The behavior plot (b) of behavior 1 describes a decreasing oscillation. Landmarks X-0 through X-3 and V-0 through V-2 are introduced during the simulation.
- An envisionment generates a finite behavioral description (c) with links pointing to other equivalent states within the tree (state 16). The behavior plot (d) is unable to describe a decreasing oscillation since landmarks are not introduced.

Figure 2.1: Damped Spring Simulation

is easier to understand the overall behavior of the system using a behavior tree since it explicitly represents each distinct behavior. Within an envisionment graph, inferring patterns within the behavioral description and obtaining a comprehensive understanding of the potential system behavior may require an analysis of each path within the graph. This analysis becomes difficult when multiple cycles occur within the description.

Although QSIM can generate either representation, the default representation is a behavior tree description. Figure 2.1 presents the results from both a behavior tree and an envisionment simulation for a model of a damped spring.

2.4 Qualitative constraints

The dynamic behavior of the system is restricted by the constraints specified within the model. A qualitative constraint, an abstraction of a mathematical relation, specifies valid combinations of simultaneous values for a set of variables. The semantics for these constraints are specified using a qualitative sign algebra (Williams, 1991). A brief presentation of some of the commonly used constraints is provided below. Please refer to Kuipers (Kuipers, 1994) for a more thorough discussion. The expression $[x]_{x_0}$ corresponds to the sign of x with respect to some landmark value x_0 and $[\dot{x}] = sign(dx/dt)$. $[x]$ is used as an abbreviation for $[x]_0$.

Constraint	Algebraic Restrictions
`(ADD x y z)`	$[x] + [y] = [z] \ \wedge$
	$[\dot{x}] + [\dot{y}] = [\dot{z}]$
`(MULT x y z)`	$[x] * [y] = [z] \ \wedge$
	$[\dot{x}][y] + [\dot{y}][x] = [\dot{z}]$
`(M+ x y)`	$[\dot{x}] = [\dot{y}]$
`(M- x y)`	$[\dot{x}] = -[\dot{y}]$
`(D/DT x y)`	$[\dot{x}] = [y]_0$
`(CONSTANT x)`	$[\dot{x}] = 0$

For many of these constraints, it is also possible to specify *corresponding values* that describe a tuple of landmark values that the variables in the constraint can have at the same time to further constrain the set of values allowed by each constraint.

In addition to the basic constraints listed above, the following more sophisticated constraints will be referred to within this dissertation:

- The QSIM multivariate monotonic function constraint (called the "multivariate M constraint") generalizes the concept of the monotonic function constraint to handle functions of several variables. The format of the constraint is as follows:

  ```
  (((M s1 ...  sn) x1 ...  xn y) (a1 ...  an a0) ...  )
  ```

  where s_i specifies the sign of the influence of x_i on y and $a_1, \ldots, a_n, a_0$ specifies a tuple of corresponding values.

- $U^{+/-}$ and $S^{+/-}$ correspond to non-monotonic generalizations of $M^{+/-}$. U^+ corresponds to a concave up (U^- is concave down) function that is monotonic on both sides of the extreme point while S^+ corresponds to functions that are monotonically increasing within a given interval (S^- decreases), but constant outside of the interval.

2.5 Dynamic simulation

QSIM generates the behavioral description using an iterative process of computing the successors for each non-terminal leaf state within the description. The successors of a qualitative state are computed using a combination of continuity and the constraints within the model. Initially, a list of potential successor values for each variable are computed using continuity. A successor state is formed for each combination of these variable values consistent with the constraints contained within the model. A branch results in the description when there are multiple consistent successor states.

2.5.1 Simulation complexity

The primary source of complexity within a qualitative simulation is the branching factor within the tree. In its most basic form, qualitative simulation attempts to compute all consistent solutions to the constraint satisfaction problem (CSP) defined by the constraints within the model coupled with the continuity constraint applied during simulation. Thus, each behavior corresponds to a unique solution to the CSP.

Viewing each behavior as a separate solution to the CSP requires a sightly non–standard characterization of the CSP since the solution being computed is potentially infinite. A straight–forward characterization of qualitative simulation as a CSP simply maps the variables and constraints within the model directly to the CSP (Kuipers, 1994). In this characterization, however, a *solution* is simply a qualitative state. Alternatively, the CSP can be characterized such that each behavior corresponds to a solution by viewing continuity as a constraint in the CSP. Thus, each variable within the model actually corresponds to a vector of variables within the CSP with a value for each time–point and each time–interval. While formally characterizing the CSP that is being solved, at this point, it is more important to simply see how qualitative simulation can be viewed in this manner.

Since qualitative simulation computes all possible solutions, there are two choices for reducing the complexity of a simulation:

1. increase the constraining power of the model, or

2. modify the representation used to describe the solutions.

The degree to which additional constraints can be added to the model depends upon the information available to the modeler. TeQSIM extends the expressiveness of the modeling language, thus providing the modeler with more flexibility when trying to incorporate additional information or assumptions. DecSIM and chatter abstraction, on the other hand, modify the representation used to describe the system behavior providing a more compact representation that eliminates irrelevant distinctions. DecSIM does this by generating a separate, state–based representation for each component while chatter elimination abstracts intractable regions of the trajectory state into a single qualitative state within the description.

Chapter 3

Model Decomposition and Simulation

Qualitative simulation (de Kleer & Brown, 1985; Forbus, 1984; Kuipers, 1994) traditionally describes a dynamical system via a single model of interacting constraints reasoning about its behavior using a global, state–based behavioral representation. For smaller, more tightly constrained models, this representation proves adequate, often resulting in a small number of behaviors and a tractable simulation (see the bathtub example in figure 1.1.) As the size of a model increases, however, the model tends to become more loosely constrained. Unrelated distinctions in distant variables combine combinatorially often resulting in intractable event branching. Often, these distinctions are irrelevant to the current task and tend to obscure other relevant distinctions in the behavioral representation.

Combinatoric branching due to unrelated events is an inherent limitation of a state–based representation. An alternative representation first proposed by Hayes (Hayes, 1985) and later used by Williams (Williams, 1986) describes the behavior of each variable as a set of independent variable *histories*. Relevant temporal relations between variables are described separately. For a set of closely related variables, however, the number of temporal relations may be quite large resulting in a more complex representation if a history–based description is used instead of a state–based representation.

The Model Decomposition and Simulation (DecSIM) qualitative simulation algorithm bridges the gap between a history–based and a state–based representation. Variables within the model are partitioned into loosely coupled components and each component is simulated separately using a state–based simulation algorithm. DecSIM reasons about interactions between components as needed to constrain each component. Thus, the relationship between variables within the same component is state–based while the relationship between different components is history–based.

The DecSIM algorithm can be separated into the simulation algorithm and the variable partitioning algorithm. This dissertation focuses on the simulation algorithm and assumes that the variable partitioning is provided by the modeler. For an arbitrary variable partitioning, the DecSIM simulation algorithm provides the same soundness guarantees and the same degree of constraining power as a standard state–based qualitative simulation.

By partitioning the model into smaller chunks, DecSIM significantly reduces the overall complexity of the simulation by eliminating the temporal correlations between variables in different components. For models that lend themselves to decomposition, partitioning the model results in an exponential speed–up in simulation time and a much more compact representation of the system behavior. Given an appropriate partitioning of the variables, the complexity of the DecSIM algorithm becomes a function of the problem specification rather than an artifact of the simulation algorithm.

3.1 DecSIM Overview

DecSIM uses a divide and conquer approach to control the problem of event branching by exploiting structure within the qualitative model. To understand the general approach taken in DecSIM, it is best to view qualitative simulation as a constraint satisfaction problem (CSP) in which each behavior corresponds to a unique solution to the CSP. Qualitative simulation explicitly computes all possible solutions to the CSP. As with any CSP, if two variables are completely unconstrained with respect to each other, then the set of all possible solutions will contain the cross product of the possible values for each variable. For QSIM this results in combinatoric branching when the temporal ordering of a set of events is unconstrained. DecSIM avoids explicitly computing this cross product by breaking the CSP problem P_M into a set of smaller sub–problems $\{p_1, p_2, \ldots, p_n\}$. Each sub–problem contains a subset of the variables in P_M while shared variables represent the constraints between sub–problems.

DecSIM explicitly computes all solutions for each sub–problem. Each solution to a sub–problem p_i provides a partial solution to P_M since it provides an assignment of values to a subset of the variables. Two partial solutions are consistent if they assign the same value to any shared variables. For a partial solution to be *globally consistent*, it must be consistent with a complete assignment of the variables from P_M. Determining if a partial solution is globally consistent is itself a constraint satisfaction problem.

By decomposing the model into smaller problems, DecSIM is able to exploit structure within the model in two ways.

(1) It avoids explicitly computing all possible solutions to the original CSP. Instead, it computes all solutions for each sub–problem and then for each of these solutions it computes a single solution to the global CSP.

(2) DecSIM uses causality to identify an ordering between the sub–problems allowing solutions to the causally upstream sub–problems to be identified as globally consistent without finding a solution to the global CSP.

The characterization of qualitative simulation as a non–standard CSP with a potentially infinite number of variables and constraints poses a number of interesting problems when refining this general approach. Each partial solution is a sequence of qualitative states. To ensure that a partial solution for sub–problem p_i is consistent with a solution for a related sub–problem p_j, each state within the solution for p_i must be matched against a state in the solution for p_j. Validating a partial solution as globally consistent must be performed incrementally as the partial solution is generated via simulation.

Describing the DecSIM algorithm from the perspective of constraint satisfaction demonstrates the basic approach that is being used. For the rest of the presentation, however, we focus specifically on the component–based qualitative simulation algorithm. The more general problem of constraint satisfaction is only referenced when it serves to clarify the discussion. The next section provides an overview of the algorithm. This is followed by an example and then a more detailed presentation of the algorithm in the following sections.

3.1.1 DecSIM: A component based qualitative simulation algorithm

DecSIM decomposes a given model M into sub–problems by partitioning the variables in M so that closely related variables are grouped together. Currently, DecSIM requires the modeler to provide a partitioning of the variables $\{V_1, V_2, \ldots, V_n\}$. For each partition V_i, a *component* or *sub-QDE* is generated. Two types of variables are contained within each component.

Within-partition variables are the variables specified in the corresponding partition. A variable can only be classified as a within–partition variable within a single component.

Boundary variables are variables contained in other partitions that *causally influence* the within–partition variables..

Each component is designed to derive a behavioral description for the within–partition variables of the component. Boundary variables, however, must be included since the within–partition variables are constrained by the behavior of variables in other components.

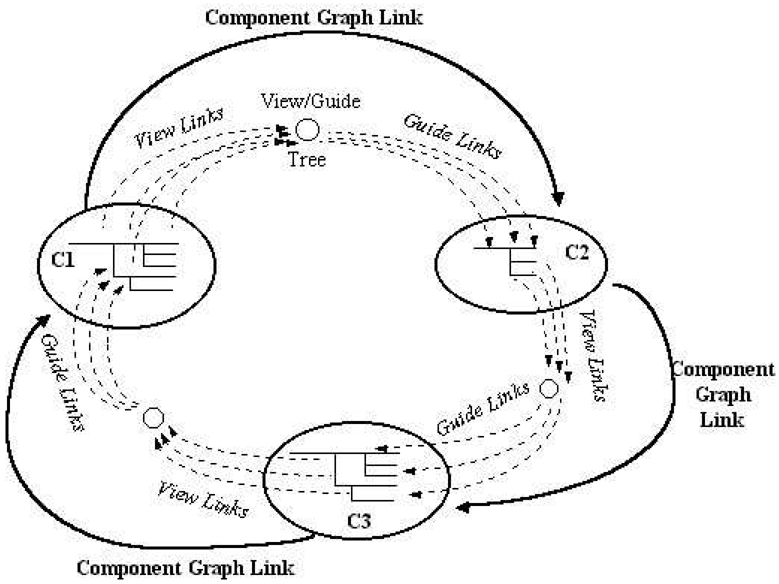
Boundary variables for a component are identified using a causal analysis (Iwasaki, 1988; Nayak, 1994) of the QDE. A variable v is a boundary variable for a component C if and only if v is directly upstream with respect to the causal ordering and related to a within–partition variable in C via a constraint within the original model. Boundary variables are the manner in which two components are related. A component C_1 is said to be a *boundary component* of component C_2 if there exists a variable v that is shared by the two components such that v is a within–partition variable in C_1 and a boundary variable in C_2. Note that the categorization of a variable as a within–partition or boundary variable is relative to a specific component.

The relationships between components defined by the boundary variables defines a labeled, directed graph of components called the *component graph* in which each node corresponds to a component. A directed edge exists between components C_1 and C_2 if and only if C_1 is a boundary component with respect to C_2. The edge is labeled with the boundary variables for C_2 that are contained in C_1.

QSIM is used to derive a separate behavioral description, called a *component tree*, for each component. The terms *component behavior* and *component state* are used to refer to a behavior and a state within a component tree respectively. A component behavior is *locally consistent* if it is consistent with all of the constraints specified within the component. QSIM guarantees that all of the component behaviors are locally consistent. In addition, however, each component behavior must be consistent with respect to the the rest of the model. A component behavior b is *globally consistent* if there exists a compatible combination of behaviors from each component that includes b. A combination of component behaviors is compatible if and only if the resulting behavior corresponds to at least one behavior within a non–decomposed behavior tree.

DecSIM maintains links between matching states within related component trees via an intermediate *view/guide tree*[1]

[1]The tree is called a *view/guide* tree because it provides a *view* of the shared variables in the component tree of the upstream component that is used to *guide* the behavior of the variables in the downstream component.



View/guide trees are used to relate states between components that share variables. The upstream component is used to derive a description of the behavior of the shared variables (i.e. the view/guide tree). This description is used to guide the behavior of the boundary variables in the simulation of the downstream component. Each consistent state within the downstream component must map to at least one component in the view/guide tree.

- Links from the upstream tree to a view/guide tree are called view links while links from the view/guide tree to the downstream component are called guide links.
- Each view/guide tree corresponds to a link in the component graph.
- If component C_1 contains boundary variables in C_2 and C_2 contains boundary variables in C_1, then two links exist within the component graph relating these components. Thus, two separate view/guide trees are maintained since the directionality of the influence in the two sets is different.

Figure 3.1: View/guide links and the component graph

describing the behavior of the shared variables (see figure 3.1). The *view/guide links* specify constraints on how behaviors in related component trees can be combined. Information about the temporal ordering of unrelated events is represented within these links. Determining if a component behavior is globally consistent is equivalent to finding a solution to the constraint satisfaction problem defined by the component graph coupled with the view/guide links. The constraints within this CSP are defined by the view/guide links while the components correspond to variables and component behaviors to variable values.

A tree–clustering algorithm (Dechter & Pearl, 1988b, 1989) is used to exploit structure and causality within the component graph to reduce the complexity of the CSP when determining if a component behavior is globally consistent. A tree-clustering algorithm converts a CSP into a tree–based representation by identifying clusters within the component graph. A cluster identifies a set of components such that a cycle exists containing any two components within the cluster. The consistency of a component behavior is only dependent upon components within the same cluster and non–cluster components that are upstream in the component graph. Figure 3.2 presents an example component graph and demonstrates how structure and causality are applied for this graph.

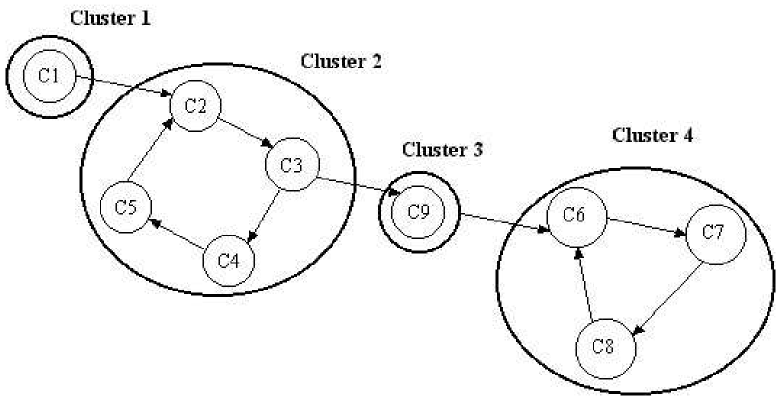
3.2 Two tank cascade example

Figure 3.3(a) describes the model for a simple two tank cascade. The variables are partitioned into two components, A and B, corresponding to the two tanks. The sub–QDE for component B includes *OutflowA* as a boundary variable since it is causally upstream and related to *InflowB* through a constraint. *InflowB* is *not* a boundary variable in component A since it is causally downstream. Thus, the constraint relating these two variables is only represented within component B.

Figure 3.3(b) shows the topology of the component graph. Since component A is strictly upstream from component B, the consistency of each component behavior for A is independent of B. Thus, component A can be simulated to completion prior to simulating component B.[2]

Each state within the component tree for B is linked to a set of corresponding states within the component tree for A. The assertion of the links is determined by the behavior of *OutflowA* and continuity. A component B state is only placed on

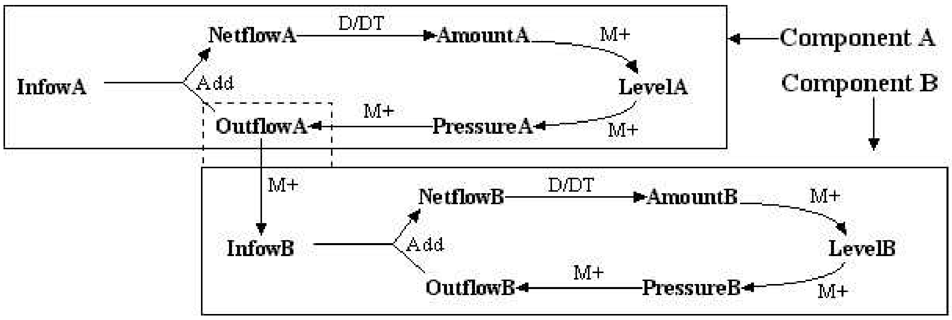
[2]The ordering of the simulation is slightly different since the description for component A could be infinite. For this reason, the simulation of component A is extended as additional information is required to drive the simulation of component B.



In the example above, a total of nine components, identified by small circles, are grouped into 4 clusters.

- The global consistency of behaviors in component $C1$ can be determined independent of the rest of the model.
- Component $C9$ separates the components in cluster 4 from cluster 2. If a component behavior in $C6$ is supported by a globally consistent behavior in $C9$, then it is consistent with all of the component graph upstream from $C9$.
- Constraint satisfaction must be used to determine the global consistency of a component behavior within clusters 2 and 4. However, this CSP only needs to "assign a value" for the components in the cluster. Thus, the complexity of the CSP is determined by the size of the cluster.

Figure 3.2: Exploiting structure in the component graph



(a) Model of two tank cascade

(b) Component graph of two tank cascade

- The model of a simple two tank cascade is represented using a directed graph (a) describing a causal ordering of the variables within the model. Nodes within the graph correspond to variables and edges to constraints. Arrows indicate the direction of causality between variables with respect to the constraint.
- The boxed region identifies the partitioning of the model provided as an input to DecSIM. The dotted line extension to the box around component B indicates that *OutflowA* is a boundary variable for component B.
- The topology of the component graph (b) can be derived from the relationship between the component via boundary variables. Since the topology is a tree structure, simulation of the upstream component is independent of the results from the simulation of the downstream component.

Figure 3.3: Simple Two Tank Cascade

the agenda for simulation (i.e. to have its successors computed), if it is *supported* by a state in component A. Figure 3.4 shows the results from the DecSIM simulation.

The cascaded tanks example provides a simple demonstration of the DecSIM algorithm. In many cases, however, a feedback loop exists between a set of related components. Figure 3.5(a) describes a version of the two tank cascade in which the level in tank B controls the inflow to tank A. The type of controller used is irrelevant to the critical features of the example so a detailed model of the controller is omitted. It is assumed that *InflowA* is influenced by a control variable u in some manner. In this example, *LevelB* is a boundary variable in the controller component while u is a boundary variable in component A. Figure 3.5(b) shows how a cycle exists within the corresponding component graph.

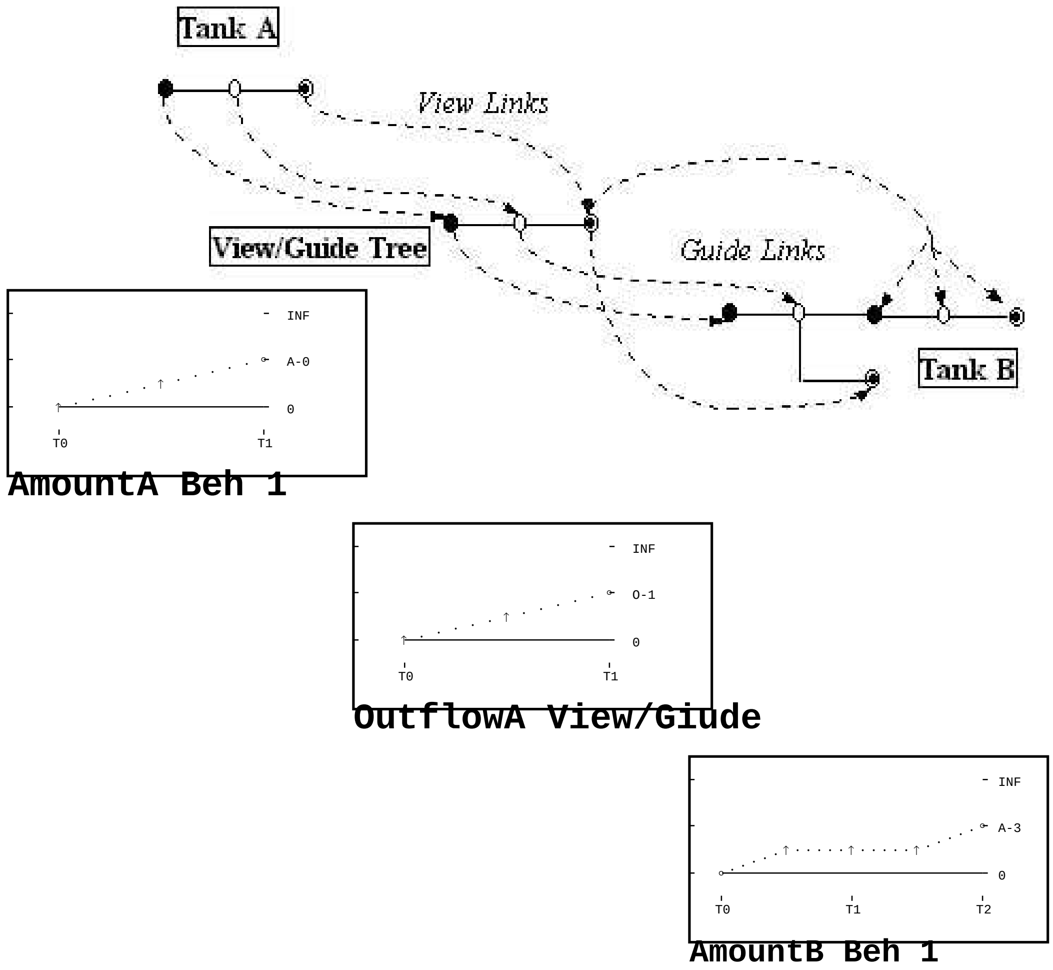
Simulation of this example generates three separate component trees with three view/guide trees providing a mapping from states in component A to states in the controller component, from the controller component to B and finally from B back to A (figure 3.4.) For a state to be globally consistent, a cycle must exist within this mapping. This cycle corresponds to a *solution* to the component graph since it contains a state from each component and each state is consistent with the other component states. In this example, each component tree is infinite. A DecSIM simulation hides the details of each component from the rest of the simulation. In a traditional QSIM simulation, combinatoric branching would result if an event any one component was unrelated to an event in the other components.

3.2.1 Presentation Overview

In the next section, we provide a formal definition for the concepts and terms used up to this point followed by a detailed presentation of the core DecSIM simulation algorithm. Section 3.6 discusses how DecSIM handles certain extensions to QSIM along with other issues that are not addressed in the presentation of the core algorithm. This is followed by theoretical and empirical results demonstrating the benefits provided by DecSIM with respect to the complexity of the simulation and proving that DecSIM generates an equivalent behavioral description as QSIM. Finally, in sections 3.9 and 3.10 we discuss related and future work respectively.

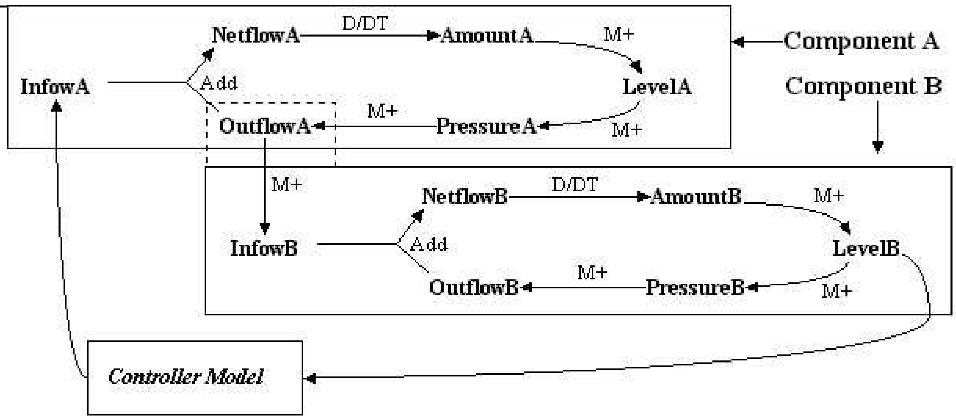
3.3 Components, the component graph and clusters

Given a QSIM model M, a partitioning of the variables $\{V_1, V_2, \ldots, V_n\}$, and an initial state S, the component generation algorithm returns a component, or *sub-QDE* for each partition along with a component graph augmented with a grouping

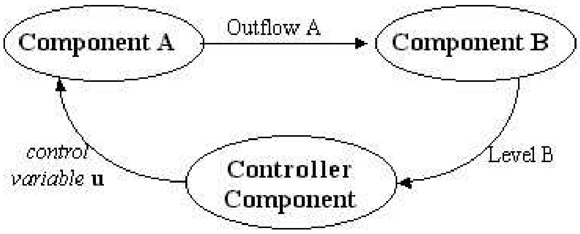


- DecSIM generates a *view/guide* tree when simulating component A. This tree is used to control the behavior of *OutflowA* in the simulation of component B.
- DecSIM maintains view links from the component tree for A to the view/guide tree and also asserts guide links from the view/guide tree to component B. These links identify compatible states.
- For component B, a branch occurs depending upon whether or not B reaches quiescence at the same time or after A reaches quiescence. The final quiescent state in component A maps to all of the states in component B in which *OutflowA* has become steady.

Figure 3.4: DecSIM simulation of the two tank cascade



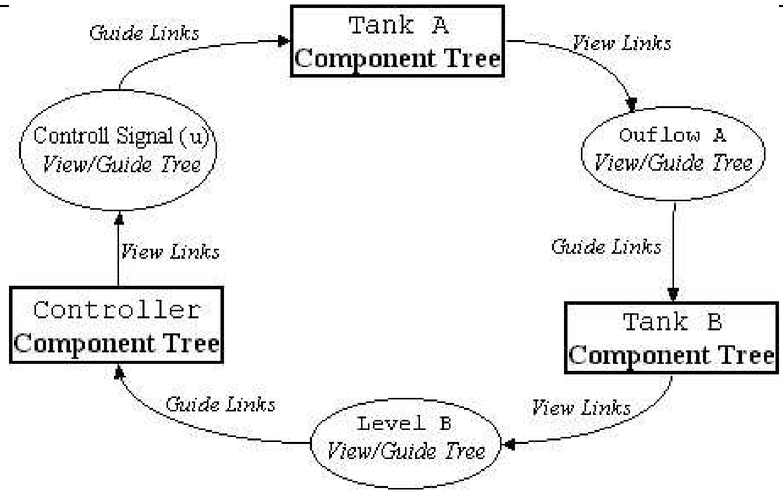
(a) Controlled two tank cascade



(b) Component graph

- Adding a controller to the two tank cascade (a) links *LevelB* to *InflowA* through a controller component and results in a cycle within the component graph (b).

Figure 3.5: Controlled Two Tank Cascade



- A decomposition of a controlled two tank cascade results in a feedback loop within the component graph. Simulation results in an infinite component tree for each sub-QDE. We have omitted the actual trees from the figure due to the complexity of the view/guide link mapping.

Figure 3.6: DecSIM simulation of controlled two tank cascade

of the components into clusters. For each variable partition V_i, a component C_i is defined by the tuple $<V_i, BV, Q, \texttt{Con}, T>$ where

- V_i is a set of *within–partition variables*,
- BV is a set of *boundary variables*,
- Q is a set of quantity spaces, one for each of the variables,
- `Con` is the maximum set of constraints within M such that if $Variables(\texttt{Con})$ represents the set of variables within `Con`, $Variables(\texttt{Con}) \subseteq (V_i \cup BV)$.
- T is a set of transition.[3]

During the simulation a behavior tree, called a *component tree*, is generated for each component. The component tree is consistent with the QDE defined by the set of variables $V_i \cup BV$, the quantity spaces for each of these variables and the set of constraints `Con`. Each state within a component tree is called a *component*

[3]Currently, we do not handle transitions within DecSIM; section 3.6.4 discusses extensions to DecSIM to handle transitions.

state. The initial state for each component is simply defined as the projection of the complete initial state onto the variables defined by the component.[4] Throughout our discussion a qualitative state with a subscript c_i (*i.e.* s_{c_i}) corresponds to a component state within the component tree for component C_i. For simplicity, we may omit this reference and simply describe s_{c_i} as a component state.

3.3.1 Causal ordering

Boundary variables are identified using a causal ordering of the variables in the model via an extension of the causal ordering algorithm first described by Iwasaki and Simon (1988) and later refined by Nayak (1992). The causal ordering algorithm is a conservative algorithm that converts the constraint graph within the model into a directed hypergraph called the *causal graph.*

Definition 3.1 (Causal graph) *A* causal graph *is a hybrid directed/non–directed, hypergraph defined for a model M such that*

(1) a node is generated for each variable within M,

(2) a hyperedge is generated for each constraint C relating the variables within C,

(3) each hyperedge can have at most one outgoing link, and

(4) each node can have at most one incoming link from a hyperedge.

A variable v is said to *immediately influence* a variable v' if there exists a hyper–edge relating v and v' with an outgoing edge directed at v'. If two the causal relationship specified by a constraint are ambiguous, then the corresponding hyperedge will not contain an outgoing link.

DecSIM uses a conservative causal ordering algorithm that does not guarentee the assertion of a causal relationship for each of the constraints in the model. DecSIM, however, does not depend upon the causal ordering algorithm at all. This information simply improves the efficiency of the algorithm. Furthermore, the modeler is free to specify the causal relationship between components by directly specifying the boundary variables for each partition as an input.

3.3.2 Boundary variables

Given a causal graph of the model, a variable v is considered a boundary variable in component C_i if and only if

[4]A formal definition for *projection* is included later in the discussion.

(1) v is not a within–partition variable in C_i

(2) v is related to a variable v' via a constraint `Con` such that v' is a within–partition variable in C_i, and

(3) the hyperedge in the causal graph corresponding to `Con` is either acausal (i.e. a causal direction was not inferred) or it has an outgoing edge pointing to v'.

3.3.3 Constraints

The constraints from the model are assigned to components such that a constraint `Con` is included within a component C if and only if all of the variables in `Con` are either within–partition or boundary variables in C.

Lemma 3.1 *Each constraint within the model is assigned to at least one component.*

Proof: For each constraint `Con` two cases must be considered:

(1) If all of the variables in `Con` are contained in a single component, then `Con` is by definition assigned to that component.

(2) Otherwise, `Con` spans multiple partitions. If the hyperedge corresponding to `Con` has a single outgoing link, then component pointed to by this link will contain the other variables within the constraint as boundary variables. Thus, `Con` will be assigned to this component. If the hyperedge corresponding to `Con` does not have an outgoing edge, then each of the components containing a variable in `Con` will contain the rest of the variables as boundary variables. Thus, `Con` will be assigned to all of these components. □

3.3.4 Component graph

Definition 3.2 (Component graph) *Given a set of related components* $\{C_1, C_2, \ldots, C_n\}$ *the* component graph *is a labeled, directed graph with a node corresponding to each component. The edges are defined as follows:*

- *An edge exists from component C_i to node C_j if and only if there exists a variable v such that v is a within–partition variable in C_i and a boundary variable in C_j.*

- *An edge from C_i to C_j is labeled with the set of boundary variables in C_j that are classified as within–partition variables in C_i.*

Definition 3.3 (Directly influence) *A component* C directly influences C' *if and only if there exists an edge within the component graph from* C *to* C'.

The expression $A \overset{v_{ab}}{\rightarrow} B$ is used throughout the rest of the dissertation as an abbreviation for the expression: *an edge labeled vars exists within the component graph connecting component* A *to component* B.

Note that for two components A and B, it is possible for both A to directly influence B and for B to directly influence A. This occurs if either the causal ordering is incomplete or if two separate constraints with different causal directions relate the variables within the two components. However, there can only exist one link within each direction. Thus, each link is uniquely defined by the direction of the relation.

Definition 3.4 (Upstream/downstream) *If there exists a path within the component graph from component* C_i *to component* C_j *then* C_i *is considered* causally upstream *from* C_j *while* C_j *is* causally downstream *from* C_i. C_i *is* strictly upstream *if and only if* C_i *is causally upstream from* C_j, *but not also causally downstream.*[5]

3.3.5 Identifying clusters

DecSIM uses a non–directed version of the component graph to identify *clusters* of related components within the component graph.

Definition 3.5 (Cluster) *The component graph is partitioned into clusters such that each cluster is the maximum set of components* $\{C_1, C_2, \ldots C_n\}$ *such that two components* C_i *and* C_j *are contained within the same cluster if and only if there exists a cycle containing both* C_i *and* C_j *within the non–directed version of the component graph. The function* Cluster : $C \rightarrow 2^C$ *maps a component to the cluster (i.e. set of components) containing it.*

Lemma 3.2 (Single cluster) *A component is included in one and only one cluster.*

Proof: First we will show that a component is included in at least one cluster and then we will show that it is included in at most one cluster.

(1) By definition, a component is included in at least one cluster. If the component A is not contained in a cycle, then the cluster simply contains A.

[5] C_i can be both upstream and downstream from C_j if they are contained within the same cluster.

(2) Suppose component A is contained in clusters C_1 and C_2. Then there exist two distinct paths p_1 and p_2 corresponding to the two clusters such that both paths begin and end in A. (*i.e.* both paths are cycles.) For any given pair of nodes N_1 and N_2 within the respective paths, there exist two alternative paths connecting them: the suffix of p_1 starting at N_1 composed with the prefix of p_2 ending at N_2 and the suffix of p_2 combined with the prefix of p_1. Thus, N_1 and N_2 must be contained within the same cluster. This is a contradiction. □

Definition 3.6 (Cluster graph) *The partitioning of the component graph into clusters defines a directed graph such that each node within the graph corresponds to a cluster and two nodes N_i and N_j are connected with a directed edge if and only if an edge exists within the component graph from a component in N_i to a component in N_j.*

Lemma 3.3 *The cluster graph is acyclic.*

Proof: Follows directly from the definition of the partitioning of the component graph into clusters. If a cycle exists between two components, they are contained within the same cluster. Thus, the cluster graph must be acyclic. □

3.4 A Component–Based Simulation Algorithm

DecSIM uses the core QSIM algorithm to compute a behavioral description for each component. DecSIM repeatedly iterates through the components using the QSIM successor generation algorithm to extend each leaf state within a component tree a single time–step. The algorithm terminates once all of the component trees are fully extended or if the state–limit is reached. QSIM guarantees that each component behavior is consistent with respect to the constraints and variables contained within the component. In addition, DecSIM must determine whether each component behavior is consistent with the rest of the model. Constraints between components occur due to shared variables and thus correspond to the edges within the component graph.

For each edge within the component graph such that $A \overset{v_{ab}}{\to} B$, DecSIM generates a *view/guide tree* that focuses on the behavior of the subset of variables v_{ab} within the component tree for A.

Definition 3.7 (View/guide tree) *A* view tree *T_{vars} is a projection of a behavior tree T onto a subset of the variables v_{ab} described within T. Thus, $T_{vars} = \Pi_{v_{ab}}(T)$*

where Π *is the projection operator as traditionally defined within the relational database literature. The upstream component is referred to as the* viewed *component. The next section contains a formal definition of a* projection.

DecSIM maintains *view links* relating states within the component tree for the upstream component to states within the view tree. The view tree is then used to *guide* the behavior of the boundary variables v_{ab} within the downstream component. The behavior of these variables within the downstream component is completely determined by their behavior within the view/guide tree. DecSIM also maintains *guide links* relating states in the view/guide tree to states in the downstream component. Note that the directionality of the edge relating two components determines the manner in which the view/guide tree is generated. Causality allows the structure of the view/guide tree to be completely determined by the upstream component tree. Later in the presentation we will discuss how an acausal relation between variables is handled.

3.4.1 View/guide tree generation

DecSIM incrementally generates both the view links and the view/guide tree as the upstream component tree is extended. A *projection* of a qualitative state and of a behavior tree are defined as follows (see figure 3.7 for a more intuitive, graphical description of a behavior tree projection). The symbol $S =_{vars} S'$ is used for short hand to mean that S and S' are equivalent with respect to the set of variables $vars$. More formally, for all v such that $v \in vars :: Qval(v, S) = Qval(v, S')$. This abbreviation will be used throughout the dissertation.

Definition 3.8 (Projection of a qualitative state) *A qualitative state* S_{vars} *is said to be a projection of the qualitative state* S *with respect to the set of variables* $vars$, *written* $\Pi^s_{vars}(S) = S_{vars}$ *if and only if,*

(1) if V *represents the set of variables described by* S *then* $vars \subset V$,

(2) $S =_{vars} S_{vars}$, *and*

(3) for all v *such that* $v \in V$ *and* $v \notin vars$, $Qval(v, S_{vars})$ *is undefined*

where

Definition 3.9 (Projection of a behavior tree) *A behavior tree* T_{vars} *is said to be a* projection of T with respect to $vars$, *written* $\Pi_{vars}(T) = T_{vars}$, *if and only if the there exists a surjective function* $P_{vars} : S_T \rightarrow S_{T_{vars}}$ *mapping states from* T *to states in* T_{vars} *such that:*

(1) for all s such that s is in T, $P_{vars}(s) = \Pi^s_{vars}(s)$.

(2) if s is an initial state of T then $P_{vars}(s)$ is an initial state of T_{vars}

(3) either s is a time–point or a quiescent state or $P_{vars}(s)$ is a time–interval state

(4) if $P_{vars}(s) = s'$ then for all s_{succ} such that s_{succ} is a successor of s either $P_{vars}(s_{succ}) = s'$, or $P_{vars}(succ(s)) \in succ(s')$.

Furthermore, if T_{vars} is a projection of T with a mapping function P_{vars} then the function $\mathcal{P}^{-1} : S_{T_{vars}} \rightarrow 2^{s_T}$ defines a mapping from a state in T_{vars} to the set of corresponding states in T such that $s \in \mathcal{P}^{-1}(P_{vars}(s))$ and for all qualitative states S_T and $S_{T_{vars}}$ from T and T_{vars} respectively $S_T \in \mathcal{P}^{-1}(S_{T_{vars}})$ if and only if $P_{vars}(S_T) = S_{T_{vars}}$.

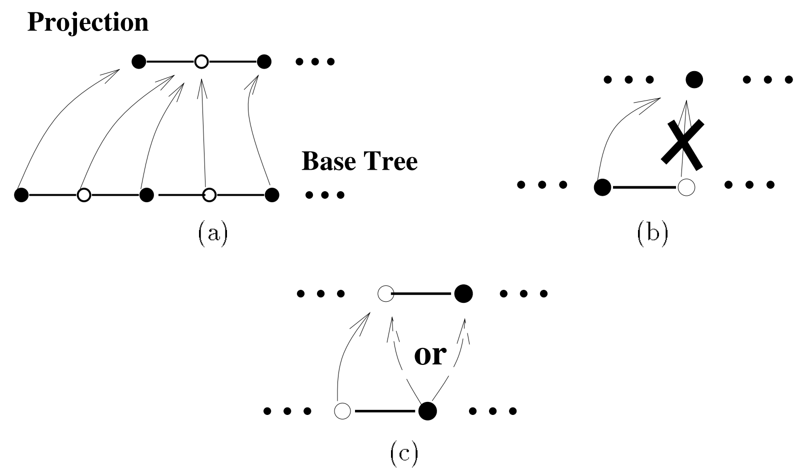
Each state added to the upstream component tree is either mapped to an existing state within the view/guide tree or the view/guide tree is extended and a new state is introduced. Formally, the mapping defined by the view/guide link defines a function $ViewedBy_{v_{ab}} : S_u \rightarrow S_{v/g}$ where S_u and $S_{v/g}$ correspond to states in the upstream component tree and the view/guide tree respectively. $ViewedBy_{v_{ab}}$ corresponds to the mapping function $\mathcal{P}_{v_{ab}}$ defined by the projection operator. The *variable view* algorithm (Clancy, Brajnik, & Kay, 1997), described in figure 3.1, generates the view/guide tree and defines the function $ViewedBy_{v_{ab}}$.

3.4.2 Generating guide links

Guide links are generated as either the view/guide tree or the downstream component tree is extended. These links define a partial function $GuideOf : S_{v/g} \rightarrow 2^{S_d}$ that maps a state in the view/guide tree to a set of states in the downstream component tree. The function is partial since a state may be added to either the view/guide tree or to the downstream component tree even if a corresponding state does not exist in the other tree. This is because the extension of the view/guide tree is driven completely by the simulation of the upstream component. For those states that are mapped, the function $GuideOf(s)$ is equivalent to the inverse mapping function $\mathcal{P}^{-1}$ defined by the projection operator. Figure 3.2 describes the *guide link mapping algorithm* used to maintain the guide mapping links.

The mapping function $GuideOf$ defined by the guide link mapping algorithm satisfies the following properties where $T_{v/g}$ corresponds to the view/guide tree, T_d to the downstream component tree, and v_{ab} to the set of variables described by $T_{v/g}$.

(1) for all $s_{v/g}, s_d$ and v such that $s_{v/g} \in T_{v/g}, s_d \in T_d$, and $v \in v_{ab}$ ::
if $s_d \in GuideOf(s_{v/g})$ then $Qval(v, s_{v/g}) = Qval(v, s_d)$,



For $T_{v_{ab}}$ to be a projection of T the many–to–one mapping from states in T to states in $T_{v_{ab}}$ must satisfy the following properties ($\circ$ represents a time–interval state while $\bullet$ represents a time–point state):

(1) if two states are mapped together, they must be equivalent with respect to v_{ab}

(2) initial states are mapped to initial states

(3) since transitions within T may correspond to changes in other variables, a time–interval state in the projection can map to multiple states in the original tree (a), but a time–point state can only map to a single time–point state (b)

(4) if a state A in T maps to B then each successor of A must either map to B or to a successor of B (c).

Figure 3.7: Projection of a behavior tree

Variable View Algorithm:

Given a state s and a set of variables v_{ab} such that v_{ab} is a subset of the variables described by s, the following algorithm updates the view/guide tree for v_{ab} and modifies the function $ViewedBy_{v_{ab}}$. The symbol $=_{v_{ab}}$ is used to mean that two states are equivalent with respect to the variables v_{ab}.

1. If s is an initial state then generate a new state $s_{v_{ab}}$ such that $s_{v_{ab}} = \Pi^s_{v_{ab}}(s)$ and assert a view link so that $ViewedBy_{v_{ab}}(s) = s_{v_{ab}}$ and return.
2. If $s =_{v_{ab}} Pred(s)$ and either s is a time–point state or $ViewedBy_{v_{ab}}(Pred(s))$ is a time–interval state, then set $ViewedBy_{v_{ab}}(s)$ to $ViewedBy_{v_{ab}}(Pred(s))$ and return.
3. If there exists an s' such that $s' \in Succ(ViewedBy_{v_{ab}}(Pred(s)))$ and $s' =_{v_{ab}} s$, then set $ViewedBy_{v_{ab}}$ to s' and return.
4. Otherwise, generate a new state $s_{v_{ab}}$ such that $s_{v_{ab}} = \Pi^s_{v_{ab}}(s)$, add $s_{v_{ab}}$ as a successor of $ViewedBy_{v_{ab}}(Pred(s))$, set $ViewedBy_{v_{ab}}(s)$ to $s_{v_{ab}}$ and return.

Table 3.1: Variable View Algorithm

(2) if $s_{v/g}$ is an initial state of $T_{v/g}$ then for all $s_d \in \mathit{GuideOf}(s_{v/g})$ s_d is an initial state of T_d

(3) either $s_{v/g}$ is a time–interval or a quiescent state or for all $s_d \in \mathit{GuideOf}(s_{v/g})$ s_d is a time–point state

(4) for all $s_{v/g} \in T_{v/g}$ and $s_d \in T_d$ if $s_d \in \mathit{GuideOf}(s_{v/g})$ then either $s_{v/g}$ and s_d are initial states, $pred(s_d) \in \mathit{GuideOf}(pred(s_{v/g}))$, or $pred(s_d) \in \mathit{GuideOf}(s_{v/g})$.

3.4.3 Compatible behaviors

For all edges $A \overset{v_{ab}}{\rightarrow} B$ within the component graph, the predicate $M_{A \overset{v_{ab}}{\rightarrow} B}(s_a, s_b)$ relating states within A and B can be defined by composing the mapping functions $\mathit{ViewedBy}_{v_{ab}}$ and $\mathit{GuideOf}$ defined with respect to the view/guide tree relating A and B. $M_{A \overset{v_{ab}}{\rightarrow} B}(s_a, s_b)$ is called a *component edge predicate.*

Definition 3.10 (Component–edge predicate) *For each edge within the component graph such that $A \overset{v_{ab}}{\rightarrow} B$, the* component–edge predicate $M_{A \overset{v_{ab}}{\rightarrow} B}(s_a, s_b)$*, where s_a and s_b are a states in the component trees for components A and B respectively, is true if and only if*

$$s_b \in \mathrm{GuideOf}(\mathrm{ViewedBy}_{v_{ab}}(s_a))$$

Lemma 3.4 (component–edge lemma) *For each edge within the component graph such that $A \overset{v_{ab}}{\rightarrow} B$, the* component–edge predicate $M_{A \overset{v_{ab}}{\rightarrow} B}(a_i, b_j)$ *is true if and only if the following statements are satisfied:*

- $a_i =_{v_{ab}} b_j$,
- *a_i is an initial state iff b_j is an initial state,*
- *If a_i and b_j are not initial states, $(M_{A \overset{v_{ab}}{\rightarrow} B}(a_{i-1}, b_{j-1}) \cup (M_{A \overset{v_{ab}}{\rightarrow} B}(a_i, b_{j-1}) \cup (M_{A \overset{v_{ab}}{\rightarrow} B}(a_{i-1}, b_j)$*

The proof for this lemma is contained in appendix A.

While the component–edge predicates define relations on qualitative *states*, these predicates allow us to efficiently determine whether behaviors from two related components are *compatible*. Intuitively, two behaviors are compatible if they can be combined to form a single composite behavior describing all of the variables within both components. It is possible to determine whether two behaviors are compatible

Guide Link Mapping Algorithm:

Given a state s that is added to a component tree that is *guided by* a view/guide tree describing the set of variables v_{ab} such that v_{ab} is a subset of the variables described by s, update the function *GuideOf* if there exists a state corresponding to s in the view/guide tree. $GuidedBy_{v_{ab}}$ is the inverse mapping function of *GuideOf*

1. If s is an initial state and there exists an initial state of the view/guide tree s' such that $s' =_{v_{ab}} s$, then add s to the set returned by *GuideOf* and return. If there is not an initial state of the view/guide tree matching s then return an error.
2. If s is a time–point,
   - if $s =_{v_{ab}} Pred(s)$ then add s to $GuideOf(Pred(s))$.
   - otherwise, if there exists an s' such that $s' \in Succ(GuidedBy_{v_{ab}}(Pred(s)))$ then add s to $GuideOf(s')$.
   - Return.
3. If s is a time–interval,
   - if $GuidedBy_{v_{ab}}(Pred(s))$ is a time–point, then if there exists a state s' such that $s' \in Succ(GuidedBy_{v_{ab}}(Pred(s)))$ where $s' =_{v_{ab}} s$, then add s to $GuideOf(s')$.
   - otherwise, if $s =_{v_{ab}} GuidedBy_{v_{ab}}(Pred(s))$, then add s to $GuideOf(Pred(s))$.
   - return.

Given a state s that is added to a view/guide tree, if $GuideOf(Pred(s))$ maps to a state that has a successor s' such that $s =_{v_{ab}} s'$ and if so update the function $GuideOf(s)$ accordingly.

Table 3.2: Guide link mapping algorithm

by comparing the behaviors beginning at the initial states. By incrementally maintaining the mapping between states in related components, however, we are able to determine whether two behavior segments are compatible simply by comparing the relation between the two terminal states within the behaviors.

Definition 3.11 (Compatible behaviors) *A set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ each from separate components are compatible if and only if the behaviors can be composed to form a single behavior describing all of the variables within each component behavior.*

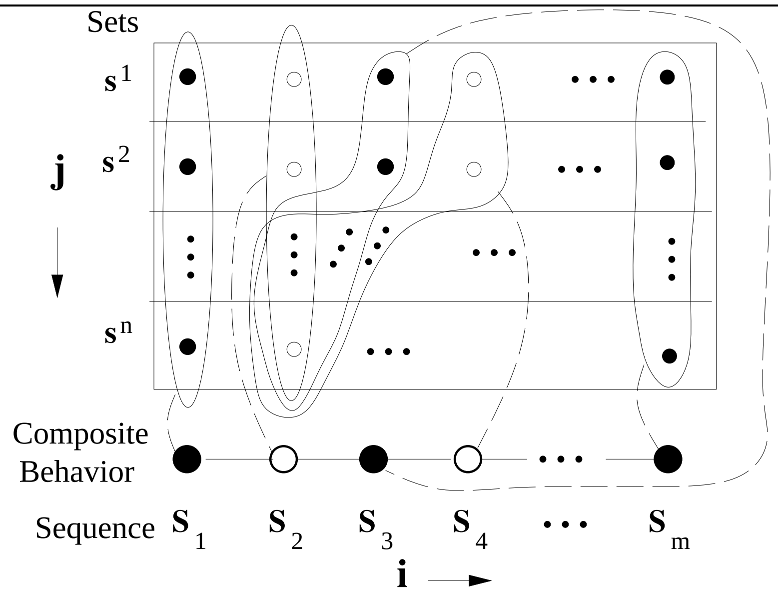
Definition 3.12 (Behavior composition) *A set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ can be* composed *if and only if there exists a* composite behavior *B such that for all $1 \leq i \leq n$, $\Pi_{v_i}(B) = b_i$ where v_i corresponds to the set of variables described by b_i. Furthermore, the join operator, $\bowtie$ when applied to a set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ defines the maximal set of composite behaviors which in effect is a behavior tree.*

Thus, for a set of behaviors to be composed there must exist an ordered sequence of sets $(S_1, S_2, \ldots S_m)$ such that each set $S_i = \{s_i^1, s_i^2, \ldots s_i^n\}$ has a single state[6] from each component tree such that any pair of states in a set are equivalent with respect to the shared variables and the ordering of the sets satisfies the following constraints:

- the states within S_1 are all time–point, initial states,
- if set S_i contains a time–point state then set S_{i+1} cannot contain a time–point state,
- for all i, j where $1 \leq i \leq m$ and $1 \leq j \leq n$, s_{i+1}^j is either the same as s_i^j or the successor of s_i^j within the component behaviors.

The sequence of sets corresponds to a qualitative behavior that is consistent with the component behaviors. If a set contains a time–point state, then the corresponding qualitative state is a time–point state. Otherwise, it is a time–interval state. Figure 3.8 provides a graphical description of this mapping. Note that in general there may exist multiple sequences of sets that satisfy this property. Thus, each sequence corresponds to a different complete behavior while the set of all such sequences corresponds to the maximal set of composite behaviors.

[6] In the notation used, the superscript corresponds to the component while the subscript corresponds to the ordering of the state within a component behavior.



The definition of a composite behavior provides a grouping of states, one from each component behavior, that are sequenced in a manner that allows the component behaviors to be combined to form a consistent qualitative state.

- The corresponding complete behavior is listed below and the states are labeled with the name of each sequence (*i.e.* S_i). In the definition from the text, i identifies the sequence while j identifies the state from the behavior within the set.
- The states contained within each set are identified by an oblong circle surrounding them.
- Note how sets with time–point states map to time–point states within the composite behavior.

Figure 3.8: Composite behavior mapping

Note that if the set of variables described by behaviors b_1 and b_2 do not intersect, then the two behaviors are by definition compatible since any state from b_1 can be combined with any state from b_2.

Theorem 3.1 (Compatible behavior theorem) *A given pair of component behavior segments $b_i = \{s^i_1, s^i_2, \ldots s^i_n\}$ and $b_j = \{s^j_1, s^j_2, \ldots s^j_m\}$ from components C_i and C_j, respectively, are compatible if and only if*

- *if $C_i \stackrel{v_{ij}}{\rightarrow} C_j$ is an edge in the component graph then $M_{C_i \stackrel{v_{ij}}{\rightarrow} C_j}(s^i_n, s^j_m)$ is true, and*
- *if $C_j \stackrel{v_{ji}}{\rightarrow} C_i$ is an edge in the component graph then $M_{C_j \stackrel{v_{ji}}{\rightarrow} C_i}(s^j_m, s^i_n)$ is true.*

Proof Sketch: Proof by induction on the number N of iterations through the simulation algorithm for each component. The initial states for the components are by definition compatible since they are each a projection of the initial state for the entire model. Then, we show that the mapping that is asserted by the component–edge predicate is the exact mapping that is required to generate a component behavior. The entire proof is contained in appendix A.

Theorem 3.2 (Compatible behavior set theorem) *A set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ each from a different component are compatible if and only if for all $i, j < n$ b_i and b_j are compatible.*

Proof Sketch: The proof of this theorem is simply an extension of the previous theorem. Once again the mapping maintained by the component–edge predicates ensure that a composite behavior can be generated. The proof for this theorem is contained in appendix A.

3.4.4 Global consistency

The component graph, coupled with the component–edge predicates specified via the view guide links, defines the *component graph constraint satisfaction problem.*

Definition 3.13 (Component graph CSP) *The* component graph CSP *is defined by the tuple $<V, D, C>$ such that*

- *the variables V correspond to components,*

- *the domain for each variable is the set of qualitative states defined in the component tree for the corresponding component (D is the set of domains for each variable),*
- *the constraints C are defined by the set of component–edge predicates defined for the edges within the component graph.*

A solution to the component graph corresponds to a set of component states $\{s_1, s_2, \ldots, s_n\}$ such that

(1) one state comes from each component, and

(2) for all i and j such that $i, j \leq n$, if there exists an edge $C_i \overset{e_{ij}}{\rightarrow} C_j$ then $M_{C_i \overset{e_{ij}}{\rightarrow} C_j}(s_i, s_j)$ is true.

Definition 3.14 (Global consistency) *A component state s is* globally consistent *if and only if there exists a solution to the component graph containing s.*

Theorem 3.3 (Global consistency $\Leftrightarrow$ compatibility) *The set of component states $\{s_1, s_2, \ldots, s_n\}$ is a solution to the component graph if and only if the set of behaviors $B = \{b_1, b_2, \ldots, b_n\}$ is compatible where b_i is the component behavior terminating in state s_i.*

Proof: [Global consistency $\Rightarrow$ compatibility] – By the definition, a set of states is a solution to the component graph if and only if the states satisfy all of the component–edge predicates (*i.e.* condition (2) in the definition of a solution). Thus, by theorem 3.1 each pair of behaviors b_i and b_j are compatible. By theorem 3.2 if each pair of behaviors is compatible, then the entire set is compatible.
[Compatibility $\Rightarrow$ global consistency] – Simply reverse the argument above. Thus, by theorem 3.2 if a set of behaviors is compatible then each pair must be compatible. By theorem 3.1, if each pair is compatible then each of the component–edge predicates must be satisfied. By the definition of a solution to the component, if each component–edge predicate is satisfied then a set of states is a solution. □

3.4.5 Global consistency algorithm

Determining whether a component behavior is globally consistent for a fully simulated set of component behaviors is a straight–forward constraint satisfaction problem given the characterization that has been provided. DecSIM, however, incrementally generates each component behavior. Thus, it is possible that as component behaviors are extended a solution to the component graph is generated containing a state already within a component tree that had previously not been globally consistent. Furthermore, the number of states that are not globally consistent is potentially quite large since the boundary variables are unconstrained within a component except with respect to the view/guide support links. Thus, extending behaviors that have not been determined to be globally consistent could be prohibitively expensive. To address these issues, we use the following techniques:

- successors of a component state s are only computed if s is determined to be globally consistent, and
- if a component state s is added to a component tree, the algorithm must find all solutions to the component graph containing s such that the solution contains at least one state whose status with respect to global consistency had previously been undetermined.

The algorithm used to test a state for global consistency exploits causality and structure within the component graph to reduce the complexity of finding a solution to the component graph in the following ways:

1. Transformation of the component graph into a tree–based representation by identifying clusters allows the algorithm to propagate the global consistency of a component state within regions of the component graph as opposed to computing a complete solution to the component graph.
2. Causality is used to assert the independence of a subgraph within the component graph. The global consistency of a component behavior is independent of the components that are strictly downstream with respect to causality. Thus, the algorithm only needs to identify a solution to the subgraph that is causally upstream from a component. For example, in the simple two tank cascade the behavior of the upstream tank is completely independent of the behavior of the downstream tank.

The global consistency algorithm is centered around the concept of *support*. Conceptually, a component state s is *supported* if for all components that are immediately upstream there exists a globally consistent state that is consistent with s with respect to the relevant component–edge predicate. The formal definition of support is divided into two types of support.

Definition 3.15 (Upstream support) *A component state s_{c_j} from component C_j is* upstream supported *if and only if for all components C_i such that $C_i \overset{v_{ij}}{\to} C_j$*

and C_i and C_j are not contained within a cluster, there exists a fully supported state s_{c_i} such that $M_{C_i \overset{v_{ij}}{\rightarrow} C_j}(s_{c_i}, s_{c_j})$ is true.

Definition 3.16 (Full support) *A component state s_{c_j} from component C_j is* fully supported *if and only if there exists a solution sol to the component subgraph defined by the cluster containing C_j such that*

- *the solution contains s_{c_j}, and*
- *all of the states in the solution are* upstream supported.

By lemma 3.2 in section 3.3.5, we know that each component is contained in one and only one cluster. Note that if the size of a cluster is one, then upstream support implies cluster support. Also, if there are no upstream components for a component C, then all of the states in the component tree for C are by definition fully supported.

To demonstrate that full support defines necessary and sufficient conditions for a state to be globally consistent, we define a partitioning of the component graph into two subgraphs around a given component C. The operator $\mathcal{U}$ and its complement $\bar{\mathcal{U}}$ are defined such that for a given component C, $\mathcal{U}$(C) returns the *causally upstream subgraph* for C while $\bar{\mathcal{U}}$(C) returns the remainder of the subgraph.

Definition 3.17 (Causally upstream subgraph) *A* causally upstream subgraph *for a component C is defined as the largest set of components U such that for all C_i where C_i is an element of U, either $C_i = C$ or there exists a path within the component graph from C_i to C.*

Thus, the causally upstream subgraph for C is the set of components and edges that causally influence C while its complement is the set of components and edges that are strictly downstream from C or causally unrelated.

Lemma 3.5 *For a given cluster C a component state s_c contained within the component tree for C is* fully supported *if and only if there exists a solution to the causally upstream subgraph for C that is consistent with s_c.*

Proof: Both directions of the if and only if will be proved by induction on the maximum length of the path from a root node of the cluster graph to the cluster containing C.
Base case: Length of the path is 0. A path length of zero means that there are no causally upstream clusters. Thus, $\mathcal{U}(C)$ is equivalent to *Cluster*(C). By the definition of cluster support, there exists a solution to this subgraph.

Inductive step: Assume the lemma is true for path of length n.

Fully supported $\Rightarrow$ solution to $\mathcal{U}(C)$ – For component C with a maximum path of length $n+1$ from a root of the cluster graph, assume that a state s_c within the component tree for C is fully supported.

(1) By assumption there exists a solution *sol* to the subgraph defined by *Cluster*(C). Furthermore, each state within this solution must be upstream supported.

(2) Since each of the states in the solution are upstream supported, then for all $s_{c_j} \in sol$ and for all components C_i such that $C_i \overset{v_{ij}}{\rightarrow} C_j$ and $C_i \notin Cluster(C_j)$, there exists a cluster supported state s_{c_i} such that $M_{C_i \overset{v_{ij}}{\rightarrow} C_j}(s_{c_i}, s_{c_j})$ is true.

(3) By the inductive hypothesis, there must exist a solution to the subgraph $\mathcal{U}(C_i)$ for each C_j referenced above. Since s_{c_j} is consistent with this solution (item 2) and by the definition of a cluster the rest of the components in the cluster containing C_j are unrelated to the components in $\mathcal{U}(C_i)$, the solution to $\mathcal{U}(C_i)$ is consistent to *sol*.

(4) Thus, the solution to *Cluster*(C) can be combined with each of the solutions to $\mathcal{U}(C_i)$ to form a complete solution to $\mathcal{U}(C_j)$ where C_i is a component that is not contained within the *Cluster*(C), but directly influences a component in *Cluster*(C).

Solution to $\mathcal{U}(C)$ $\Rightarrow$ fully supported – Let *sol* equal to the solution to the subgraph $\mathcal{U}(C)$. Thus, since *sol* also provides a solution to any subgraph of $\mathcal{U}(C)$, there exists a solution to *Cluster*(C) as well as all subgraphs $\mathcal{U}(C_i)$ such that C_i directly influences C but is not an element of *Cluster*(C). Thus, each of the states in *sol* are cluster supported. □

Lemma 3.6 *If the model is consistent, then for all solutions sol_u to the causally upstream subgraph for C there exists at least one solution sol_d to the remainder of the component graph such that sol_u and sol_d are consistent.*

Proof: By the definition of causally upstream subgraph, the variables contained in $\mathcal{U}(C)$ are connected to the variables contained in $\bar{\mathcal{U}}(C)$ via one of more causally directed constraints within the causal ordering. These variables are simply exogenous variables with respect to $\bar{\mathcal{U}}(C)$ whose behavior is unconstrained by $\bar{\mathcal{U}}(C)$. Thus, if there is a solution to $\mathcal{U}(C)$ and the model is consistent, then there will exist at least one solution to $\bar{\mathcal{U}}(C)$ that is consistent with $\mathcal{U}(C)$. □

Theorem 3.4 *If the model is consistent, then a state s is fully supported if and only if there exists a solution to the component graph containing s.*

Proof: By lemma 3.5 if s is cluster supported then there exists a solution *sol* to the subgraph for $\mathcal{U}(C)$ where C is the cluster containing s. By lemma 3.6, if the model is consistent, there must exist a solution to $\bar{\mathcal{U}}(C)$ that can be combined with *sol* to form a complete solution. This argument can be reversed to support the theorem in the opposite direction. □

3.4.5.1 The algorithm

The global consistency algorithm is divided into two parts. The *global-consistency-test* algorithm tests a state for global consistency while the *global-consistency-propagate* algorithm propagates globally consistency through the component graph. Both algorithms are simply straight–forward implementations of the definitions provided above. The algorithms are defined in figures 3.3 and 3.4. Note that if s is marked globally consistent then the propagate algorithm must find all solutions to the subgraph defined by the cluster containing s such that

- each solution contains s, and
- each solution contains at least one other state that previously had not been classified as globally consistent.

This condition ensures that the effects of s being marked globally consistent are propagated through the component graph without computing all possible solutions to the cluster subgraph.

3.4.6 Results of the algorithm

This section presents a sequence of lemmas and theorems that support the following statements:

(1) The global consistency test algorithm is sound and complete with respect to the identification of a state s as globally consistent.

(2) In combination, the test and propagation algorithms ensure that at any given point during a simulation all states that participate in a solution to the component graph are marked globally consistent.

Global Consistency Test Algorithm:

Given a component state s from a component C that is not globally consistent:

1. For each component C' such that $C' \stackrel{v_{ab}}{\rightarrow} C$ and C and C' are not contained within a cluster
   - if there exists a state $s' \in GuideMap_{v_{ab}}(s)$ that is marked globally consistent, then mark s' as upstream supported with respect to C',
   - otherwise return NIL.
2. Mark s *upstream supported.*
3. If s it is not contained within a cluster of size greater than 1, then mark s globally consistent and return.
4. Otherwise, attempt to find a solution to the subgraph defined by the cluster containing C that satisfies the following properties:
   - the solution contains s, and
   - all of the states within the solution are *upstream supported.*

   If a solution is found, mark s as *globally consistent.*
5. Return.

The function $GuideMap_{v_{ab}}$ traverses the view/guide links and returns the set of states that are mapped to s via the directed link labeled vab.

Table 3.3: The Global Consistency Test Algorithm

Global Consistency Propagate Algorithm:

Given a state s from component C that has just been marked globally consistent,

1. For each component C' such that $C \overset{v_{ab}}{\rightarrow} C'$ and C and C' are not contained within a cluster
   - for all s' such that $s' \in \mathit{ViewMap}_{v_{ab}}(s)$, if s' was previously not supported with respect to C, then mark s' as upstream supported with respect to C and *global-consistency-test* on s'.
2. If C is contained in a cluster of size greater than 1, then find an ordered list of solutions $sol = \{sol_1, sol_2, \ldots, sol_n\}$ such that each solution satisfies the following constraints:
   - each solution contains s,
   - all of the states within each solution are *upstream supported*, and
   - each solution sol_i contains at least one state that is not currently marked globally consistent and is not contained within a solution sol_j where $j < i$.
3. For all states s' such that s' is contained within one of the solutions and is not currently labeled globally consistent, mark s' globally consistent.
4. For all states marked globally consistent in step 3 propagate the results by calling this algorithm recursively. (All of the states must be marked before propagating to avoid recomputing the same solutions to the cluster subgraph.)

The function $\mathit{ViewMap}_{v_{ab}}$ traverses the view/guide links and returns the set of states that are s maps to via the directed link labeled v_{ab}. It is equivalent to the set identified by $\mathit{ViewedBy}_{v_{ab}}(\mathit{GuideOf}(s))$.

Table 3.4: The Global Consistency Propagation Algorithm

(3) The DecSIM algorithm does not preclude the generation of a state s that could potentially participate in a solution to the component graph.

These conclusions are used to support the claims in section 3.5 regarding the soundness and completeness of the DecSIM algorithm with respect to behavioral description generated.

Lemma 3.7 (Global consistency test is sound and complete) *Given a state s, the* global-consistency-test *algorithm marks s globally consistent if and only if there exists a solution to the component graph containing s.*

Proof: This lemma follows directly from theorem 3.4. The global consistency test algorithm marks a state as globally consistent if and only if it is fully supported. Thus, by theorem 3.4 there exists a solution to the component graph containing s. □

The algorithm also guarantees detection of any effects that s may have on the global consistency of other states within the component graph. Note that a state s can only affect the global consistency of another state s' if s participates in a solution to the component graph containing s'.

Lemma 3.8 (Test and propagate sound and complete) *For all component states s, s is contained within a solution to the component graph if and only if the* global-consistency-test *algorithm coupled with the* global-consistency-propagate algorithm *identifies the s as globally consistent.*

Proof: Lemma 3.6 guarantees the soundness and completeness of the global consistency test algorithm. Thus, all states marked participate in a solution to the component graph.

If a state s participates in a solution, and global consistency test is called, then it will be labeled globally consistent. Global consistency propagate ensures that any states that can potentially participate in a solution are tested. If state s' is contained within the same cluster as s, then propagate identifies all solutions to the cluster graph containing fully supported states that are not currently globally consistent. Thus, the solution containing s' will be identified and the state labeled. If s' is strictly downstream from the component containing s, then either the component containing s' is directly related to a component in the cluster or it is related by a sequence of components. In either case, if it participates in a new solution, the state must be supported by a sequence of globally consistent states relating all the way back to the component containing s. Propagation will propagate the effects of the new solution containing s through this sequence and identify s' as globally consistent. □

Theorem 3.5 (Global consistency invariance theorem) *At any given point during a DecSIM simulation, a component state s is labeled globally consistent if and only if there exists a solution to the component graph containing s.*

Proof: This condition is initially established by the decomposition of the initial state. By definition, the initial states within each component are all globally consistent. The only way that the status of the set of component states can change is by the generation of a new component state s. By lemma 3.7, s is labeled globally consistent if and only if there exists a solution to the component graph. Lemma 3.8 states that the addition of s is propagated through the component graph. Thus, if the condition being proved held before the introduction of s, then these algorithms ensure that it holds after the introduction of s. Since we know that this condition holds at the initial state, we know that the algorithms maintain this condition throughout the simulation. □

Lemma 3.9 *For a component state s, the set of states generated by the successor generation algorithm in a DecSIM simulation is a superset of the corresponding successor states within a standard QSIM simulation.*

Before proving this lemma, we will first provide a more formal characterization of the statement that is being made by establishing a relation between component states and complete qualitative states.

Suppose that M is a non–decomposed QSIM model and s is a component state corresponding to a decomposition of M. Furthermore, assume that s describes the set of variables V_i, which is a subset of the variables defined in M. Thus, s is a projection of a set of complete qualitative states describing all of the variables within M.

- Let, $\Pi^{-1}(s)$ return the set $\mathcal{S} = \{s_1, s_2, \ldots s_m\}$ of all consistent extensions of s such that each extension contains a qualitative value for all of the variables within the model.

- Let $\mathcal{SS}$ be the set of successor states for the states within $\mathcal{S}$ *i.e.* $\mathcal{SS} = \{ss | ss \in Succ(s)\}$ where $s \in \mathcal{S}$.

- Finally, let $\mathcal{CS}$ correspond to the set of component states defined by the projection of the states in $\mathcal{SS}$ back onto the subset of variables V_i *i.e.* $\mathcal{CS} = \{s | s = \Pi_{V_i}(ss)\}$ where $ss \in \mathcal{SS}$.

Formally, the lemma states that $succ(s) \subseteq \mathcal{CS}$.
Proof: The QSIM successor–generation algorithm uses continuity to define the space of possible successor states and then eliminates states due to constraints within the model. Since the set of component constraints is a subset of the constraints within the model and since DecSIM uses the same successor generation algorithm as QSIM, the set of successor states for s will a subset of those generated for the corresponding complete qualitative states. □

The only manner in which DecSIM restricts the generation of a component state is by limiting the states that are simulated to globally consistent states.

Lemma 3.10 *For a given component behavior $b = \{s_1, s_2, \ldots, s_n\}$, if s_n is globally consistent, then for all i such that $1 \leq i < n :: s_i$ is globally consistent.*

Proof: Since s_n is globally consistent, then by definition there exists a solution to the component graph containing s_n. Let *sol* represent this solution. By theorem 3.3, we know that there exists a corresponding set of compatible component behaviors. By the definition of compatible, this means that these behaviors can be composed to generate a set of consistent full behaviors. Therefore, any prefix of b terminating in state s_i, where $i < n$, also corresponds to a consistent full behavior. Therefore, by definition s_i is globally consistent. □

Therefore, if in the current state of the simulation a state s does not participate in a solution to the component graph, a successor of s cannot participate in a solution to the component graph. Thus, restricting the simulation to globally consistent states does not preclude the generation of a valid component behavior.

3.5 Theoretical Results

The DecSIM algorithm supports the following general claims:

(1) The behavioral description generated by DecSIM is equivalent to the representation generated by QSIM except for the temporal ordering of events for variables in separate components.

(2) The behavioral description generated by DecSIM, coupled with the constraints on the temporal ordering of events specified by the component–edge predicates represented as view/guide links, is equivalent to the representation generated by QSIM.

(3) A real–valued trajectory is described by the set of DecSIM component behaviors, coupled with the component graph predicates, if and only if it is also described by the behavioral description generated by a standard QSIM simulation.

The following theorems support claims (1) and (2). The third claim follows directly from claim (2) since the set of qualitative behaviors is identical.

Theorem 3.6 (DecSIM Completeness Guarantee) *Given a consistent model M, if DecSIM generates a component behavior b_i describing the subset of variables V_i based upon a decomposition of M, then there exists a corresponding behavior B within the behavior tree generated by QSIM such that $\Pi_{V_i}(B) = b_i$.*

Proof: Let s_i correspond to the terminal state in b_i. By definition, s_i is fully supported. Therefore, by theorem 3.4, there exists a solution $sol = \{s_1, s_2, \ldots s_n\}$ to the component graph containing s_i. By theorem 3.3, the set of behavior segments $\{b_1, b_2, \ldots b_n\}$ such that s_i is the final state in b_i are compatible. By the definition of compatible, there must exist a composite behavior B such that B is in the set of behaviors defined by the muti-way join of $\{b_1, b_2, \ldots b_n\}$ and for $i : 1 \le i \le n :: \Pi_{V_i}(B) = b_i$. Finally, by the QSIM soundness guarantee we know that QSIM will produce B. □

Theorem 3.7 (DecSIM Soundness Guarantee) *Given a consistent model M and a decomposition of the model into components $\{C_1, C_2, \ldots C_m\}$ where C_i describes variables V_i, if QSIM generates a behavior B, then DecSIM will generate the set of behaviors $\{b_1, b_2, \ldots b_m\}$ such that for $i : 1 \le i \le m :: \Pi_{V_i}(B) = b_i$.*

Proof: By induction on the length of B. *Base case: Length of B equals 1.*
B simply corresponds to the initial state. By definition the initial state for component C_i is $\Pi^s_{V_i}(S)$.
Inductive step: Assume for $|B| = n$ that DecSIM generates a set of component behaviors.
For a behavior B of length $n+1$, let the prefix of B of length n be represented by B^n and the set of behaviors generated by DecSIM be $b^n = \{b^n_1, b^n_2, \ldots, b^n_m\}$. Furthermore, let S^{n+1} represent the state that extends B^n to B.

(1) By the inductive hypothesis, DecSIM generates the behavior set b^n. By theorem 3.5 the set of component states $s^n = \{s_1^n, s_2^n, \ldots, s_m^n\}$ such that s_i is the terminal state in b_i^n is a solution to the component graph. Each one of the states is marked globally consistent and will be placed on the agenda for simulation.

(2) Since QSIM generates a behavior B^{n+1}, by definition there exists a set of compatible behaviors $b^n = \{b_1^{n+1}, b_2^{n+1}, \ldots, b_m^{n+1}\}$ where b_i^{n+1} describes the set of variables in V_i. By theorem 3.2, the set of terminal states $s^{n+1} = \{s_1^{n+1}, s_2^{n+1}, \ldots, s_m^{n+1}\}$ is a solution to the component graph.

(3) For all components C_i, we know that $\Pi_{V_i}^s(S^{n+1}) \in \{s_i^n \cup succ(s_i^n)\}$ by continuity. Since DecSIM does not restrict the generation of states except by limiting the simulation to states marked globally consistent (lemmas 3.9 and 3.10), we know that DecSIM will generate all of the states in this set.

(4) By theorem 3.5, if a solution exists within the component trees, DecSIM identifies the states within the solution as globally consistent. By (2) we know that a solution exists and by (3) we know that all of the states that comprise this solution will be generated.

By induction, DecSIM will generate the set of component behaviors b^n. □

Theorem 3.8 *Given a consistent model M, the set of qualitative behaviors defined by the composition of all of the component behaviors generated by DecSIM for a decomposition of M is equivalent to the set of qualitative behaviors generated by a standard qualitative simulation for M.*

Proof: By theorems 3.6 and 3.6 we know that each component behavior generated by DecSIM corresponds to a behavior in the tree generated by QSIM and conversely that each behavior generated by QSIM corresponds to a set of component behaviors. Thus, all that is left is to show that the set of behaviors defined by the composition of the component behavior trees does not include behaviors that are not generated by QSIM.

(1) By the definition of composition all of the behaviors that are generated are "valid" qualitative behaviors (*i.e.* successive states adhere to continuity constraints and each qualitative state assigns a valid qualitative value to each variable).

(2) Furthermore, each component state ensures that the constraints restricting the variables within the component are satisfied. Constraints between component behaviors are represented by shared variables. Thus, if a set of component states can be combined (*i.e.* there is not a conflict in the value for a variable), then the state will satisfy the constraints within M.

Thus, all of the behaviors defined by the composition of the component behaviors are consistent behaviors for the model M. If one of the composite behaviors is not generated by QSIM, this would violate the QSIM soundness guarantee. □

3.6 Other Issues

The presentation of the DecSIM algorithm has focused on the core simulation algorithm that is used to generate a behavioral description for each component. This section discusses various topics relevant to the detailed implementation of the DecSIM algorithm.

3.6.1 Acausally related components

Currently, DecSIM handles an acausal relation as a pair of causal relations. Thus, if there exists a constraint relating variables in components C_i and C_j whose causal direction is ambiguous, then an edge from C_i to C_j will be created in the component graph, as well as an edge from C_j to C_i. Thus, two separate view/guide trees will be maintained and C_i and C_j will be contained within the same cluster.

While this solution simplifies the algorithm, it introduces some inefficiencies with respect to both space and time. The view/guide trees generated for the two links in the component graph will be roughly equivalent at any point during the simulation. Certain behaviors may be slightly more extended within one than the other as each component tree is incrementally extended. Conceptually, however, the two view/guide trees describe the behavior of the same subset of variables. Combining these trees into a single view/guide tree would provide a more compact representation and would also combine the two component–edge predicates into one. Representing a causally ambiguous relation in this way, however, would complicate the characterization of the algorithm, since certain edges within the component graph would be non-directed. In addition, it would complicate the code, since the extension of a view/guide tree would depend upon two separate component trees.

3.6.2 Global inconsistency

Due to the incremental generation of the component trees, failure to identify a state as globally consistent does not imply that the state is *globally inconsistent*.

Definition 3.18 (Global inconsistency) *A component state s is* globally inconsistent *if and only if it can be shown that for all possible extensions to the component trees a solution to the component graph containing s does not exist.*

If a solution containing a state s does not exist to the component graph, it is still possible for a solution to be created if the predecessor of s participates in a solution containing a state that has not been simulated. This is because after simulation of the state, it is possible that the solution might be "extended" so that it or an alternative solution contains s. Thus, to identify a state s as globally inconsistent, it is necessary to evaluate its successor with respect to the solutions in which it participates.

Initially, we believed that it was necessary to develop an algorithm that could determine when a state became globally inconsistent. Later, it became clear that classifying a state with respect to global inconsistency was not important. Instead, DecSIM simply maintains on the component trees states that have not been identified as globally consistent. States that are not classified as globally consistent, however, are not displayed with the rest of the component tree when viewing the results. Thus, the user only sees the behaviors that have been identified as globally consistent. If a DecSIM simulation runs to completion, then any state that is not globally consistent is by definition globally inconsistent; failure to display these states is an appropriate representation of the system behaviors. On the other hand, if the simulation is infinite or is terminated before completion, failure to display these states is equivalent to the partial description that would have been provided by a QSIM simulation terminated at the same point in time with respect to the extension of the behaviors.

3.6.3 Detecting quiescence

QSIM identifies a state as quiescent if all of the variables within the system are steady. While a quiescent state is represented as a time–point state within the behavioral description, it actually corresponds to an open time–interval in which the dynamical system is able to remain for an infinite amount of time. Thus, quiescent states are not required to have successors. In fact, a stable system will only leave a quiescent state if it is perturbed by an exogenous factor.

A direct extension of the quiescence–detection algorithm is insufficient since in general component states are influenced by external factors. DecSIM handles quiescence in the same manner as consistency: a component state can be classified as *locally quiescent* and/or *globally quiescent*.

Definition 3.19 (Locally quiescent) *A component state s is labeled* locally quiescent *if and only if all of the variables within s are steady.*

Definition 3.20 (Globally quiescent) *A component state s from component C_s is* globally quiescent *if and only if s is locally quiescent and there exists a solution to the causally upstream component graph sol containing s such that for each state $s' \in sol$ s' is locally consistent.*

The algorithms used to detect and propagate global quiescence are almost identical to the algorithms used for global consistency.

3.6.4 Transitions

QSIM uses transitions to specify discontinuities in a variable or changes within the model. In general, DecSIM introduces an entirely new paradigm for the development of qualitative models that may fundamentally change the manner in which transitions are specified and handled. At this point in time, DecSIM does not handle transitions. Extending the algorithm to handle a restricted set of transitions is fairly straight–forward. Other types of transitions require additional information to be specified within the model.

A transition is specified by a *condition/effect* pair. The *condition* specifies a region of the trajectory space while the *effect* is a function. If a time-point state s is created that satisfies the *condition*, then the *effect* function is called to generate the set of successor states for *s*. The effect function will inherit certain values, assert certain values and infer other values.

Four different types of transitions must be considered depending upon whether the transition specification extends beyond the component boundaries and whether or not it specifies a change in the model.

1. The model does not change (i.e. the transition specifies a discontinuity in the value of a variable) and both the transition condition and the effect function refer only to variables that are contained within the same component.

2. The model does not change, but either the transition condition or the effect function refer to variables within multiple components.

3. The model changes, but the change is limited to the constraints within a component and the transition condition and effect both refer only to variables within the same component.

4. The model changes and either the transition condition or the effect function refer to variables within multiple components.

Changes in the model In general, transitions that specify a change in the model require an extension to the manner in which a model is specified. A partitioning of the variables would be required for both models, along with a mapping between the different variable partitions following a transition. Specifying a model in this manner could be quite cumbersome. Alternatively, the modeler could specify the model using an extension of the *component–connection* modeling paradigm provided by QSIM (Franke & Dvorak, 1990). This extension would allow the modeler to specify model transitions on a component basis. Additional capabilities would be required to allow him to specify the new qualitative values for each transition state and the conditions under which a transition would occur.

Variable discontinuities extending across component boundaries In general, if the condition leading to a discontinuity extends across component boundaries, then detecting the transition could significantly increase the complexity of the DecSIM algorithm. To detect a transition condition that extends across component boundaries, the algorithm would need to detect solutions to the component graph that satisfied the transition condition. An algorithm similar to the global consistency algorithm would be required to test for such a solution except that the algorithm would need to detect *all* solutions that satisfy the transition conditions not just one. Computing all solutions to the component graph could effectively eliminate the benefits provided by a component–based simulation algorithm since this computation would be equivalent to generating all temporal orderings of the component behaviors. A similar problem occurs if the effect function extends across component boundaries.

Discontinuities within a component Handling discontinuous changes that are restricted to the variables within a component allows transition detection to be performed using "local" information within each component. In addition, the *effect* of the transition can simply be propagated through the component graph. If a variable that is contained within another component changes discontinuously, then this change must be propagated to the other components. Determining the effect of a discontinuous change could be computed using a variation on Brajnik's *continuity relaxation* algorithm (Brajnik, 1995).

3.6.5 Landmark Introduction

DecSIM fully supports QSIM's ability to introduce landmarks during a simulation. For each variable, landmarks are initially introduced in the component tree that contains the variable as a within partition variable. Landmarks, in turn, are then generated within any view/guide trees describing this variable and finally within the downstream component trees. A mapping is maintained between landmarks within the different trees so that the equivalence of two qualitative values in different trees can be compared.

3.7 Empirical Evaluation

Empirical evaluation of the DecSIM algorithm demonstrates two results:

1. DecSIM performs exponentially better than QSIM on a range of component graph topologies.

2. DecSIM generates a behavioral description that is equivalent to the description generated by QSIM except for the temporal ordering of events for variables in different components.

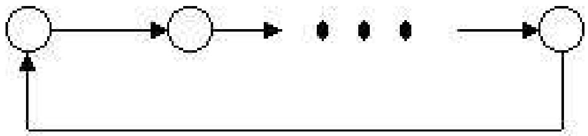
3.7.1 Simulation complexity

Three different models were used to evaluate DecSIM's run–time performance. Each of the models was designed to be easily extendible by adding identical components to the model. The models were selected to vary the topology of the component graph, and the results were evaluated both with respect to execution time and space. The following models were used:

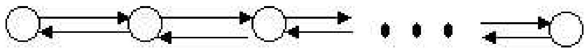
(1) **Cascade topology:** A sequence of N cascaded tanks provides a causally sequenced chain of components. This topology clearly demonstrates the benefits of a component–based simulation. In addition, this topology tests the impact of causality along with the partitioning selected.

(2) **Loop topology:** A feedback loop with N components, each with its own internal feedback loop, demonstrates the performance of the algorithm when all of the components are contained within a single cluster. The model was

(a) **Cascade topology**



(b) **Loop topology**



(c) **Chain topology**

Three models, with very different component–graph topologies, were used to evaluate how the structure of the model impacts the complexity of the simulation. The circles correspond to components within the decomposed model. Each component describes a simple feedback loop. The arrows indicate the direction of causality within the model.

Figure 3.9: Component graph topology for models used to evaluate DecSIM

motivated by one developed by Ironi and Stephanelli (1994) that models the glucose/insulin regulator system. Its structure is equivalent to a controlled N tank cascade in which the inflow of the top tank is controlled by the level of the last tank.

(3) **Chain topology:** A sequence of N chained components where each component causally influences the components on both sides demonstrates the performance of the algorithm on a set of tightly coupled components. The structure of this model to a sequence of cascaded tanks in which the outflow of tank i is controlled by the level of tank $i+1$. The actual model is a variation of the glucose-insulin model referenced above.

Figure 3.9 provides a graphical representation of the topology for each model.

3.7.1.1 DecSIM vs QSIM

In all of the models tested, DecSIM was both exponentially faster and used exponentially less memory than QSIM. Figures 3.10 through 3.12 display plots demonstrating these results. In each of the plots, the x-axis shows the number of components

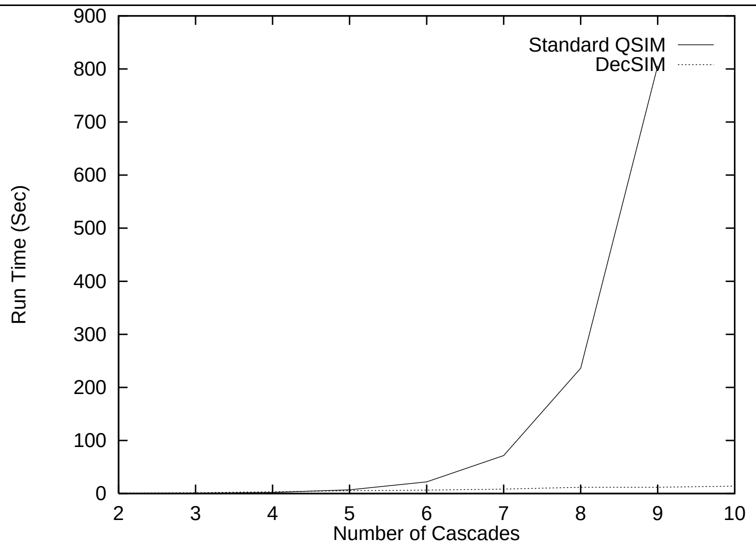
Number of Comp's	**Cascade** QSIM	**Cascade** DecSIM	**Chained** QSIM	**Chained** DecSIM	**Loop** QSIM	**Loop** DecSIM
2	0.204	0.815	3.075	6.79	0.757	5.587
3	0.621	1.6	10.94	19.903	16.149	8.147
4	2.2	3.12	37.55	25.984	89.418	12.67
5	7.09	5.49	139.3	36.712	493.88	23.28
6	21.92	6.32	676	62.405	2758	48.73
7	71.59	8.39	1633	70	14474	116.1
8	236	11.67	8101	77	nc	442.4
9	806	11.75	nt	nt	nt	nt
10	nc	14.05	nt	nt	nt	nt

Table 3.5: Results comparing DecSIM to QSIM for all three topologies

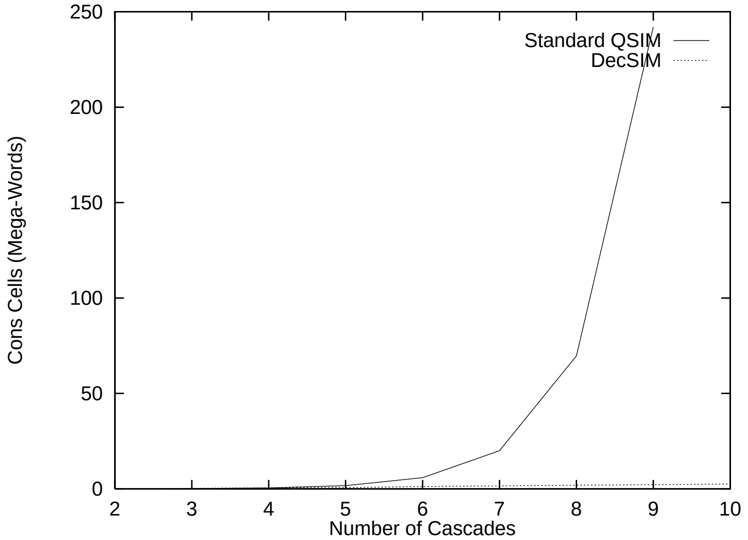
within the model while the y-axis shows either simulation time or space. Table 3.5 lists the data used to generate these plots. Each component is comprised of five variables, so the number of variables in each model is simply $5n$ where n is the number of components. All of the tests were run on a Sparc10 using Lucid Common Lisp.

For the cascade topology, the behavioral description is finite and thus the simulation was run to completion. For the chain and loop topologies, however, the feedback loop in the model causes an infinite simulation. In the results reported, a state-limit of 40 was used for each component tree. For the standard QSIM simulation, the behavior tree was extended to roughly the same extent as the component behaviors generated by DecSIM. This was accomplished by generating view trees on the fly while performing a QSIM simulation and comparing the view tree against the corresponding component tree. When all of the view trees had the same number of behaviors as the corresponding component tree, the QSIM simulation was terminated. As the number of tanks approached 7 or 8, the QSIM simulation often failed to terminate due to resource limitations. Table 3.6 summarizes the number of behaviors generated for each of the topologies tested.

For all three topologies, DecSIM clearly outperforms a standard QSIM simulation demonstrating the benefits provided by DecSIM as the size of the model increases. For example, a QSIM simulation for seven tanks configured in a loop topology took over 200 minutes while DecSIM took less than 2 minutes. The other topologies show similar differences for both execution time and space required for

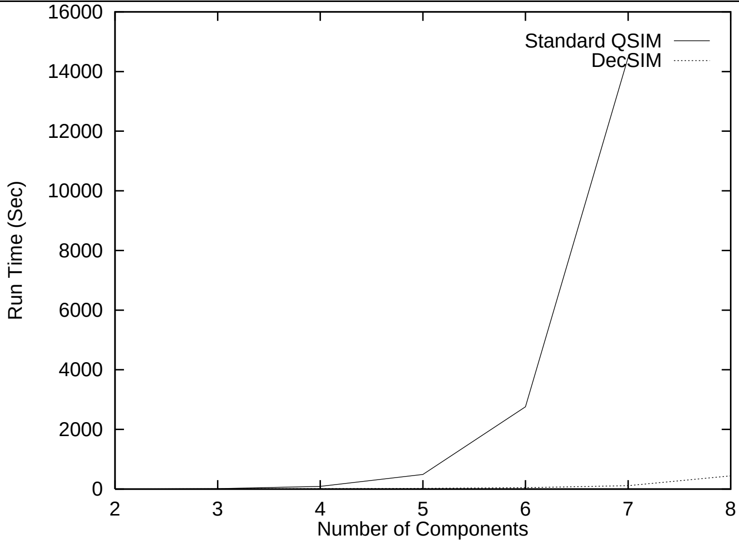


(a) Execution Time

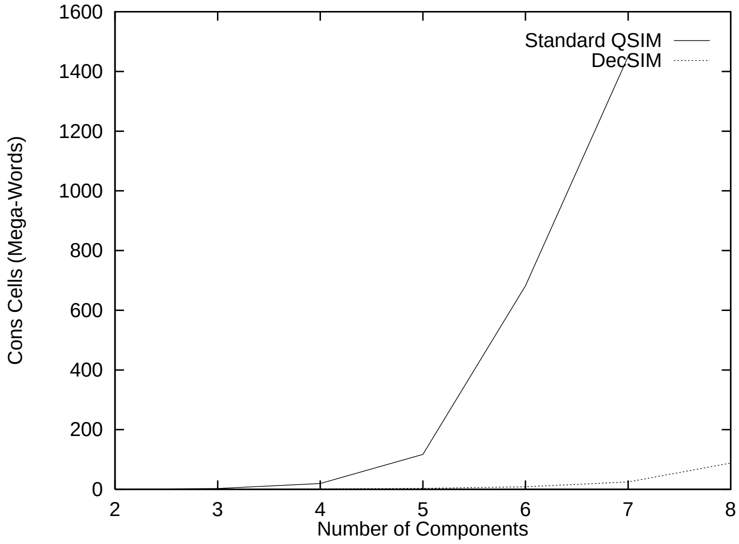


(b) Space

Figure 3.10: DecSIM vs QSIM: Cascade configuration

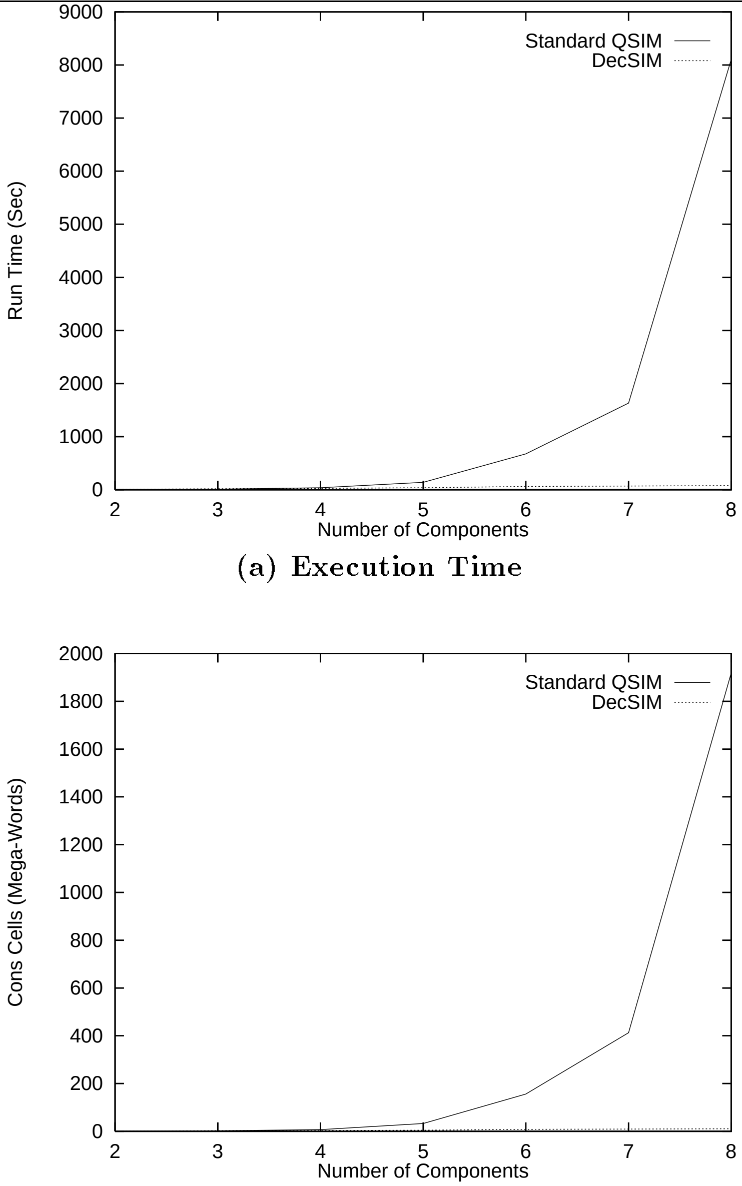


(a) Execution Time



(b) Space

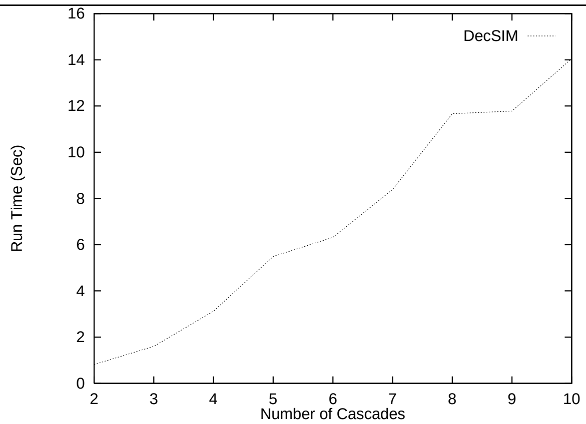
Figure 3.11: DecSIM vs QSIM: Loop configuration



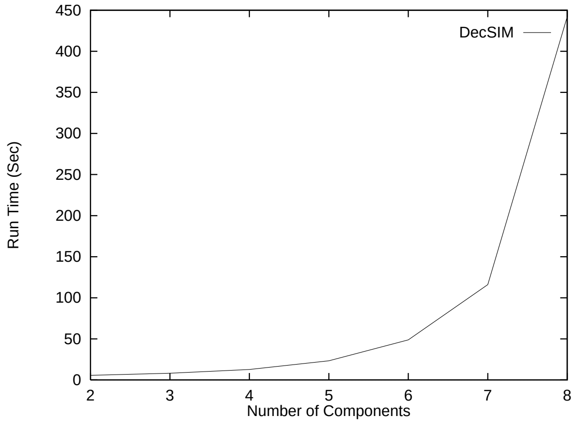
(a) Execution Time

(b) Space

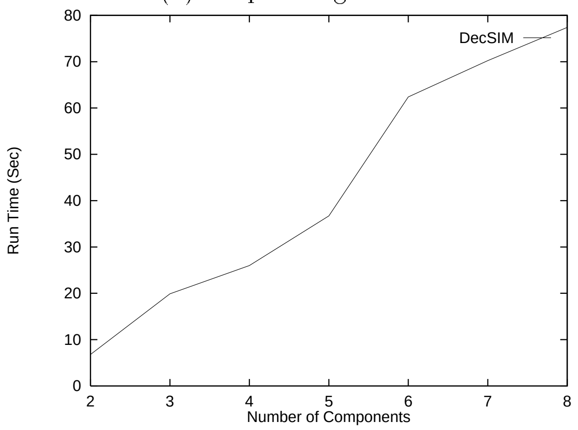
Figure 3.12: DecSIM vs QSIM: Chain configuration



(a) Cascade configuration



(b) Loop Configuration



(c) Chain configuration

Figure 3.13: DecSIM Only: Cascade, loop and chain configurations – Execution time

Number of Comp's	Cascade			Chained			Loop		
	DecSIM		**QSIM**	**DecSIM**		**QSIM**	**DecSIM**		**QSIM**
	Total Behs	Avg Behs	Total Behs	Total Behs	Avg Behs	Total Behs	Total Behs	Avg Behs	Total Behs
2	3	2	2	16	8	12	42	18	26
3	5	2	4	60	20	35	53	18	60
4	9	2	8	101	25	61	68	17	127
5	13	3	16	128	25	179	85	17	288
6	17	3	32	180	30	677	104	17	544
7	21	3	64	212	30	644	121	17	917
8	25	3	128	228	28	1071	138	17	nc
9	29	3	256	nt	nt	nt	nt	nt	nt
10	31	3	nc	nt	nt	nt	nt	nt	nt

nt – Not Tested

nc – Resource limitation prevented completion

Table 3.6: Number of Behaviors Generated for Different Topologies

the simulation.

These results demonstrate QSIM's inability to scale to larger models. DecSIM, on the other hand, performs quite well as the size of the model increases. Figure 3.13 displays the run–time results just for the DecSIM simulation. Note that both the cascade topology and the chain topology appear to be linear in the number of components. The loop topology, on the other hand, is clearly exponential in the number of components. Despite this exponential complexity, it is still significantly faster than a straight QSIM simulation, as shown in figure 3.11. Figure 3.14 plots the loop topology against the chain topology to highlight the differences. As discussed in the previous section, this distinction is due to the backtracking that occurs when finding a solution to the component graph. In the loop topology, the algorithm often extends to the very bottom of the search space before backtracking. For the chain topology, the depth of the search prior to backtrack is usually only two or three components. This distinction highlights the importance of the connectivity of the components within each cluster. A number of existing constraint satisfaction optimization techniques (Tsang, 1993) exist such as dependency–directed backtracking could be used to address this problem, but none of them would completely eliminate the need to backtrack.

To evaluate the impact of the partitioning algorithm and causality two additional versions of the cascade topology were tested: one in which the variables were completely partitioned (*i.e.* each variable was assigned a separate partition

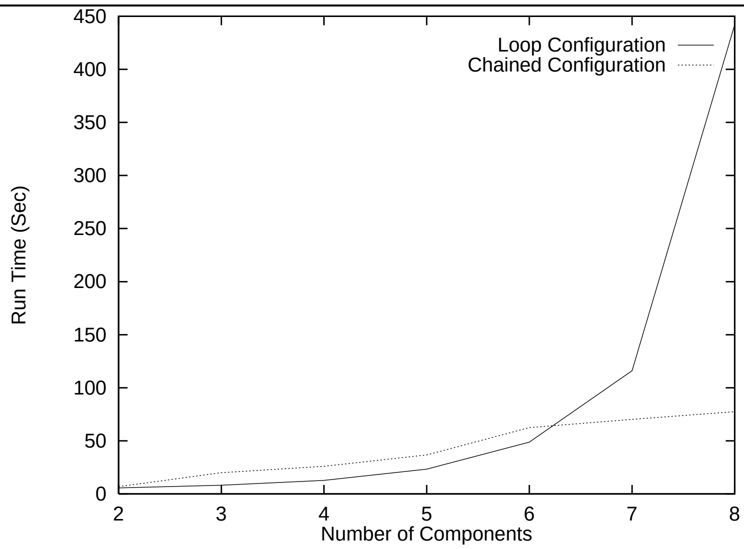


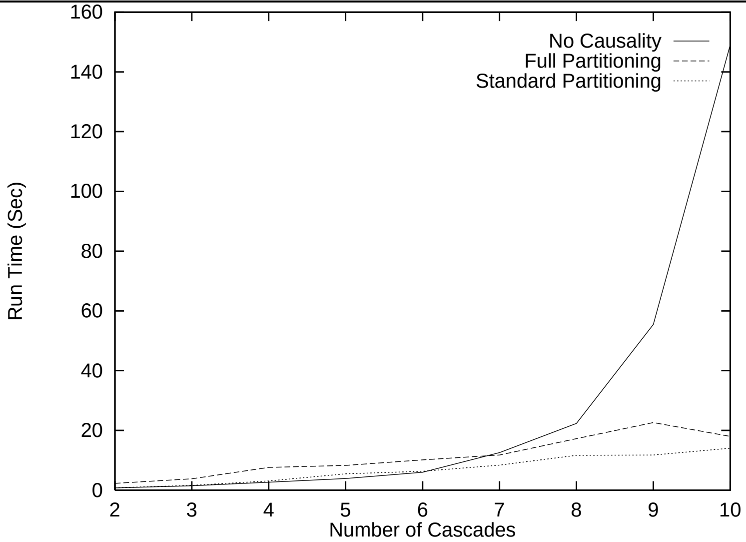
Figure 3.14: DecSIM: Chain configuration vs Loop configuration

generating a complete history–based representation) and another in which the standard partitioning (*i.e.* each tank being a separate partition) was used, but without causality. Eliminating causality corresponds to removing the directionality of the dependence between two connected tanks so that the topology of the component graph is actually similar to the chain topology. Figure 3.15 describes the results from the three different versions of the cascade topology. As expected, the standard partitioning that exploits causality performs the best; surprisingly, however, the no–causality version exhibits a sharp exponential up–turn as the number of tanks increases. Conversely, the full partitioning only performs slightly worse than the standard partitioning. In all three, however, DecSIM still performs significantly better than QSIM. These results demonstrate the benefits provided by causality within the component graph.

3.7.2 Soundness and completeness

To validate the theoretical results presented in section 3.5, we tested each of the models within the evaluation corpus (see table 1.1 in section 1.3) using a variety of variable partitionings. The results were automatically compared against a standard QSIM simulation as follows:

- a standard QSIM simulation was performed,



Three different partitionings of the cascade configuration are compared:

Standard – the components correspond to the physical tanks and causality is used when relating two connected tanks.

Full Partitioning – a full partitioning of the model where each component corresponds to a single variable, and

No Causality – a standard partitioning that does not use causality to order the components (*i.e.* the relationship between components is considered acausal and thus view/guide trees are maintained in both directions).

Figure 3.15: DecSIM: Different partitionings of the cascade configuration

- the variable–view code that generates a view/guide tree was used to generate a projection of the standard QSIM simulation onto the subsets of variables contained within the components

- each view/guide tree was compared to the component tree to ensure that the trees were identical[7], and

- for an infinite simulation, we controlled the extension of the QSIM tree to ensure that the two simulations explored similar regions of the trajectory space. (*i.e.* The simulations were terminated at roughly equivalent points.)

For all of the models tested, DecSIM and QSIM generated identical results. These results serve to reinforce the theoretical claims made in section 3.5.

3.8 Complexity Analysis

DecSIM is a qualitative simulation algorithm that exploits inherent structure within the model to eliminate irrelevant distinctions due to unrelated events. The empirical results reported in section 3.7 demonstrate that, for a variety of topological configurations of the constraint graph, DecSIM yields an exponential improvement in simulation time compared to a standard QSIM simulation. In fact, for certain topological structures within the constraint graph, DecSIM is polynomial in the number of variables within the model while QSIM is highly exponential.

DecSIM's decomposition of the model into components is a partitioning of the constraint satisfaction problem into smaller, more–tractable problems using an implicit representation of all possible solutions rather than an explicit enumeration. DecSIM reduces the complexity of a simulation in two ways. First, by partitioning the model, DecSIM reduces the solution space for the CSP. This feature alone can provide an exponential reduction in the complexity of a simulation. More importantly, however, since the branching factor is dominated by unconstrained events, separating loosely constrained variables into separate components eliminates the primary source of branching. Thus, the simulation of each component may become highly constrained resulting in a small number of solutions for each component.

The benefits provided by the DecSIM algorithm depend highly upon the topology of the constraint network and the degree to which it lends itself to decomposition. Furthermore, the variable partitioning clearly affects the complexity of the simulation. While DecSIM cannot guarantee a tractable simulation for any model,

[7]Two trees are compared by matching qualitative states within the trees.

it does provide a simulation algorithm whose complexity is a function of the problem specification rather than an artifact of the simulation algorithm. The advantages of a component simulation become more pronounced as components become more tightly constrained and the interaction between components decreases.

Providing a detailed complexity analysis of a structure–based constraint satisfaction algorithm is difficult since the efficiency of the algorithm is highly dependent upon the structure of the problem. Furthermore, the complexity of a qualitative simulation is also highly dependent upon the degree to which the constraints restrict the space of possible solutions to the CSP.

Four different factors affect the complexity of a DecSIM simulation:

- the degree of overlap between components,
- the size of the clusters within the component graph,
- the connectivity of the component graph within the clusters, and
- the degree to which a complete causal ordering can be inferred.

To demonstrate how each of these factors impact the complexity of the simulation, we first provide a characterization of the solution space, comparing DecSIM to QSIM. Then we will discuss the worst–case execution time, assuming that all of the components are contained within a single cluster. This analysis does not consider the degree to which the constraints restrict the solution space. Since DecSIM is guaranteed to generate an equivalent behavioral description, the degree to which the model constrains the simulation is the same for DecSIM and QSIM. Our analysis demonstrates that in the worst case the DecSIM run–time is within a factor k of QSIM, where k is the number of components; moreover, n the best case, DecSIM is exponentially faster than QSIM. Finally, we provide a more informal discussion of these factors and their impact on DecSIM simulations.

3.8.1 Solution space

A qualitative model defines a language that is used to generate a symbolic description of the potential behaviors of a dynamical system. This language defines a space of distinct qualitative behaviors that are consistent with continuity. Our discussion makes the following assumptions:

- there are n non-constant variables within the model,
- each variable can have v different possible values,

- landmarks are not introduced,[8] and
- a complete assignment of qualitative magnitudes completely defines the direction of change for each of the variables within the model.

The size of the qualitative trajectory space is determined by two factors: the branching factor within the tree and the depth of the tree.

Branching factor – If we assume that chatter does not occur, then all of the branches within the qualitative trajectory space occur following a time–interval state. Each variable that is changing can take on one of two possible values – it can remain the same or it can reach the landmark that it is approaching. Thus, at each branching point there are 2^n different successor states since there are a total of n different variables in the model.

Tree depth – Since there are n different variables, each with v different values, there are a total of v^n different unique time–interval states. Since non-chatter branches only occur following a time–interval state, time–point states need not be considered when evaluating the overall size of the tree.

Thus, the total size of the qualitative trajectory space (TS), given the assumptions stated above, is

$$\left(2^{n} \right)^{v^n}$$

.

If DecSIM partitions the variables into k evenly sized components and each component has b boundary variables, then the total number of variables within each partition is $(b+n/k)$, where b is at most $(n-n/k)$. Note that if the constraint graph is fully connected and no causal ordering can be inferred, then there are a total of n variables within each component. Thus, the size of the trajectory space for each component, represented CS, is

$$\left(2^{(b+n/k)} \right)^{v^{(b+n/k)}}$$

Since there are k components, the total size of the trajectory space for all of the components combined is $k \cdot CS$.

[8]The introduction of landmarks can result in an infinite behavioral description making it impossible to provide an upper–bound on the size of the behavior tree. The effect of landmark introduction should have the same impact on both QSIM and DecSIM.

While DecSIM also maintains view/guide trees and links relating components that share variables, the number of links is linear in the size of the component trees so it will not affect the overall complexity.

The following two conclusions can be drawn from this analysis:

- as the degree of overlap between components approaches zero, the size of the total solution space is reduced by an exponential factor k (*i.e.* for $b = 0$ $CS = (2^{n/k})^{n/k}$ so the total size for all components is $k \cdot (2^{n/k})^{n/k}$).

- as the degree of overlap approaches a fully connected constraint graph, the size of the DecSIM solution space is within a factor of k (*i.e.* for $b = (n - n/k)$ which is its max value, $CS = TS$ so the total size for all components is simply $k \cdot TS$.)

Note that this analysis is worst case. It does not take into account that the unconstrained relations of the original solution space are ideally separated into distinct components. Due to the types of constraints allowed by QSIM, a fully connected qualitative model would be over constrained resulting in a simulation with no consistent behaviors. While the worst–case scenario is unrealistic, it provides an upper bound on the size of the behavioral descriptions.

3.8.2 Efficiency

The overall complexity of a standard QSIM simulation is determined by the size of the representation that is being computed. As demonstrated in the previous section, DecSIM can significantly reduce the overall size of this representation. However, to ensure that each component behavior is globally consistent, DecSIM must find a solution to the component graph for each component state in the simulation.

The complexity of this task is determined by the size of the maximum cluster. If the maximum cluster size is 1, then finding a solution to the component graph is linear in the number of components. To analyze a worst–case scenario we will assume that all of the components are contained within a single cluster.

The CSP defined by the component graph has a total of k variables[9] with each variable having a potential domain size of CS. Thus, the worst–case complexity

[9]Note that a *variable* within the component CSP corresponds to the number of components and is not the same as a variable within the model.

of this CSP is

$$\left(\left(2^{(b+n/k)} \right)^{v^{(b+n/k)}} \right)^{k} .$$

.

At first glance, this figure suggests that potentially DecSIM is exponentially more expensive than QSIM. However, a closer inspection of the extreme case in which the constraint graph is fully connected with no causal ordering reveals that this is not the case. For a fully connected constraint graph, the size of each component trajectory space is simply TS. However, in this case all of the component graphs are identical and the mapping between any two graphs is one–to–one. Thus, a solution to the constraint graph CSP only needs to test one state within each of the other components, thus completely eliminating the exponent k in the overall complexity of this problem.

DecSIM is only required to find a *single* solution to the component graph, while QSIM computes all solutions. The worst–case analysis of these two tasks is the same; if there is only one solution, then they are equivalent. However, if the space of consistent solutions is large, then finding a single solution can be significantly faster than finding all solutions. Finding a solution to a CSP is linear in the number of variables if a solution can be found without backtracking. As the space of consistent solutions increases, the likelihood of finding a solution without backtracking increases, once again eliminating the exponential factor.
This analysis highlights two important features of the DecSIM algorithm:

⇒ *As the overlap between components increases, the mapping between the states within the components becomes tighter, thus reducing the overall complexity of the constraint graph CSP.*

⇒ *As the overlap between components decreases, the space of solutions to the constraint graph CSP increases. As the space of solutions to the constraint graph CSP increases, the benefits of a DecSIM algorithm also increases, since DecSIM is only required to find a single solution while QSIM finds all solutions.*

In the limit as the overlap between components increases, the complexity of a DecSIM simulation is within a constant factor k of a QSIM simulation. Conversely, as the overlap between components decreases DecSIM can be exponentially faster than QSIM.

3.8.3 Factors affecting complexity

As mentioned above, four factors affect the overall complexity of a DecSIM simulation.

Overlap between components – Each component tree branches on distinctions in both boundary variables and within partition variables. DecSIM's benefits depend upon the degree to which details of different components can be hidden from one another. If two components overlap so that an event in one component is always manifested in the shared variables, then the behavioral description for the individual component will be equivalent to the description generated if both components were combined. Thus, the benefits provided by DecSIM are inversely proportional to the overlap between component.

Cluster size – Partitioning the model into components divides the problem into smaller pieces. Determining whether a component behavior is globally consistent, however, is exponential in the cluster size. Thus, the overall complexity of the simulation decreases as the average cluster size decreases.

Connectivity of the component graph – Within a given cluster, the connectivity of the component graph can also affect the overall complexity of the simulation. The cost of finding a solution to the component graph is determined by the amount of backtracking that occurs within the constraint satisfaction algorithm. As the components within a cluster become more tightly related, the depth of the search tree prior to backtracking decreases. For example, if the components are connected in a large loop, then the algorithm may not backtrack until it has assigned a value to each component. Conversely, if the components are "chained" together in a more–sequential fashion, then conflicts are often found sooner. (See section 3.7 for empirical results supporting these claims.)

Causal ordering – Causality can be used to significantly reduce the complexity of a DecSIM simulation since it allows an independence relation to be asserted between sets of components. In addition, causality simplifies the representation of constraints relating two components since a directionality can be inferred for the constraint.

3.9 Related Work

DecSIM uses a structure–based constraint satisfaction algorithm to as an alternative to a standard qualitative simulation. There is related work within the fields of both qualitative reasoning and constraint satisfaction.

3.9.1 Qualitative reasoning

The problem of irrelevant distinctions resulting from event occurrence branching was first addressed by Williams (1986) with the Temporal Constraint Propagator (TCP). TCP forsakes the traditional state-based approach and independently describes the behavior of each variable over time as a set of variable *histories*. The relevant temporal relations between events are described separately. A temporal ordering of events is only provided when their interaction affects the value of other quantities.

TCP provides a number of compelling concepts and many of the intuitions behind the DecSIM algorithm are similar to the ideas proposed by Williams. In particular, TCP is the only qualitative simulation algorithm that uses a history–based representation. DecSIM, however, does not restrict itself to a history–based representation. A state–based representation is very effective for a system of closely interacting variables. Thus, DecSIM exploits the benefits of both representations. In addition, using TCP to reason about physical systems requires a significant amount of work by the modeler, and it is unclear how this system can be extended to incorporate advances in other qualitative reasoning paradigms. While TCP introduces a number of important ideas, many of them were not adequately addressed and the research has not been extended to explore how a history–based simulation algorithm can be used to reason about larger qualitative models.

Coiera (1992) presents a component simulation technique that handles models that can be divided into causally related components. Thus, it is similar to DecSIM in that it exploits inherent structure to decompose a model into components. Temporal ordering distinctions are eliminated by superimposing qualitative predictions from two causally unrelated processes on a single downstream variable. Coiera, however, only addresses the simulation of causally chained components. He does not provide a technique for reasoning about feedback loops within a system. Feedback loops are quite common in the devices that are analyzed using qualitative simulation techniques and thus this restriction significantly limits the impact that these techniques can have when simulating larger models.

DeCoste (1994) addresses the problem of intractable branching within qualitative simulation using the Qualitative Process Theory (Forbus, 1984) framework and the Qualitative Process Engine (QPE) simulation algorithm. As opposed to computing the entire set of potential system behaviors, DeCoste uses a goal–directed simulation method that focuses on distinctions relevant to the current task. Given a description of the desired *goal state*, he computes an abstract plan that is gradually refined to determine whether a path exists from the initial state to the goal state. Thus, if a path exists he avoids the need to explore the entire trajectory space. His approach is very effective if the information that is desired can be succinctly expressed as a goal state. It is our experience, however, that the modeler is often concerned with the overall behavior of the system. In particular, when developing a model the modeler needs to understand the range of behaviors described to evaluate the accuracy of the model. Thus, simply restricting the simulation to the set of trajectories leading to given goal state does not address the overall complexity problem encountered when simulating larger models.

Clancy and Kuipers(1993) and Fouche and Kuipers (1990) both address irrelevant temporal correlations via post-processing abstraction techniques that combine behaviors and states into a more–abstract representation following completion of the simulation. Their work is actually similar to the functionality that is provided by the generation of a view/guide tree. While both of these techniques provide a more–compact representation for the modeler to analyze, neither of them address the broader problem of simulation complexity.

3.9.2 Constraint satisfaction

A variety of structure–based constraint satisfaction techniques have been explored in the constraint satisfaction literature. The techniques used here are most closely related to two different concepts/algorithms: directed arc consistency and tree-clustering. These techniques rest upon two definitions:

1. A CSP is *node consistent* if and only if for all variables all values in the variable's domain satisfy the local constraints on the variable.[10]

2. A constraint satisfaction problem is *arc consistent* if and only if every arc in the constraint graph is arc consistent. An arc relating nodes x and y in the constraint graph is arc consistent if and only if, for every value a in the domain of x, there exists a value in the domain of y such that these two values are compatible with the constraint specified by the arc. Freuder (1982) demonstrates that a backtrack–free algorithm can be used to solve a tree–structured CSP that is node and arc consistent.

[10] In DecSIM, this is equivalent to saying that all component states are locally consistent.

Dechter and Pearl (Dechter & Pearl, 1988a) observe that node and arc consistency are actually a stronger conditions than what is required for a backtrack–free search algorithm. They introduce the slightly weaker concept of *directed arc consistency.* A CSP is directed arc–consistent under an ordering of the variables if and only if, for every value for variable x, there exists a compatible value for the variable y which is after x in the ordering.

If this property is satisfied by a tree–structured CSP, then to determine whether a solution exists for a given value a for a variable x, it is sufficient to ensure that a is consistent with a partial solution containing values only for the variables prior to x in the ordering. In DecSIM, the ordering of the nodes is defined by causality. Thus, to determine whether a state s is globally consistent, we are only required to find a solution to the subgraph that is causally upstream.

DecSIM converts the component graph into a tree-structured representation by identifying clusters. This is conceptually equivalent to the tree–clustering method (Dechter & Pearl, 1988b, 1989) for converting a CSP. Partitioning the component graph into clusters limits backtracking to the regions of the constraint graph contained within a cluster.

3.10 Future Work

DecSIM introduces an entirely new paradigm for qualitative simulation that requires many existing techniques to be rethought. In addition, DecSIM significantly extends the range of models that can be studied using qualitative simulation, since it allows larger more loosely–coupled systems to be simulated. In the past, a modeler needed to impose simplifying assumptions on the behavior of certain components within the model in order to reduce the overall complexity. These assumptions need no longer be applied, unless they are required to restrict the space of potential system behaviors.

The results presented here suggest four broad areas for future research that are covered in the following subsections.

1. refinements to the DecSIM algorithm
2. integrating and extending DecSIM so that it works in conjunction with other qualitative simulation techniques
3. applying DecSIM to the simulation of larger models,

4. extending the core DecSIM algorithm to other constraint satisfaction problems.

3.10.1 Refining the DecSIM Algorithm

Partitioning algorithms – My dissertation research focuses on specifying a component–based simulation algorithm. Additional research is required to identify the characteristics of an optimal partitioning and to develop an algorithm to automate model decomposition. Currently, DecSIM requires the modeler to identify a partitioning of the variables. Providing such a partitioning can simply be viewed as part of the model building task. Automating this process, however, will simplify the model building process and be beneficial when used in conjunction with automated model building systems (Rickel & Porter, 1997). The problem of partitioning a graph into closely related components has already been studied extensively in fields such as graph theory (Even, 1979) and constraint satisfaction (Tsang, 1993). Developing a partitioning algorithm for DecSIM primarily requires a characterization of the task so that existing research within these fields can be applied.

Optimizing the DecSIM algorithm – The development of the DecSIM algorithm has primarily focused on the specification of a sound and complete solution that is efficient with respect to the overall order of magnitude complexity of the algorithm. In general, very little attention has been paid to specific optimization techniques that could be applied. In particular, the current constraint satisfaction algorithm used to find solutions to the component graph uses a straight–forward backtracking algorithm. For certain topological configurations of the component graph, this algorithm could be significantly improved using existing techniques from the constraint satisfaction field (Tsang, 1993).

3.10.2 Integrating DecSIM with other techniques

Component–based modeling – One reason that we have not focused on developing an automated partitioning algorithm is that we feel that DecSIM challenges the manner in which models are currently developed. As the size of a model grows, it becomes significantly more difficult to maintain the model as a single composite unit. In particular, it becomes difficult to experiment with alternative representations of different components within the model since the components are not explicitly represented.

The component–connection (CC) model building paradigm (Franke & Dvorak, 1990; Kuipers, 1994) provides a representation language that allows the modeler to define independent component models that are connected to form a larger, composite model. Currently, CC compiles the model into a standard QSIM model prior to simulation. Integration of CC with DecSIM would allow the modeler to directly simulate a CC model. This model–building paradigm lends itself to the incremental development of a model and to the controlled application of simplifying assumptions as required to constrain the resulting set of component behaviors.

Semi–quantitative reasoning – Qualitative simulation is designed to allow the modeler to represent imprecise qualitative information about a dynamical system. Often, however, some *quantitative* information is available. Furthermore, precision within the model may be required to restrict the simulation to the region of the trajectory space that is relevant to the current task.

Semi–quantitative reasoning techniques are essential if qualitative simulation is to address a wide range of applications since strictly qualitative information often does not provide the precision required to address certain tasks. A number of semi–quantitative techniques (Kay, 1991; Kay & Kuipers, 1993; Berleant & Kuipers, 1988; Kuipers & Berleant, 1992) have been incorporated into the QSIM framework. These extensions allow the modeler to specify functional envelopes and quantitative bounds on landmark values along with other quantitative information. Currently, none of these techniques work with DecSIM, since they assume that the model is represented as a single component. Additional research is required to determine how DecSIM can be extended to incorporate these techniques.

Transitions – DecSIM needs to be extended to handle region transitions. As discussed in section 3.6.4, a number of alternative solutions exist. The solution selected will depend upon other extensions such as the integration of DecSIM with CC described above.

3.10.3 Applying DecSIM to larger models

The overall goal of the research described by this dissertation is to facilitate the application of qualitative reasoning techniques to real–world problem solving applications. DecSIM extends the range of models that can be tractably simulated. To further evaluate the benefits provided by DecSIM and the extensions that are required, larger more–complex models must be developed and simulated.

Initially, we had hoped to evaluate DecSIM by testing it on a variety of existing models. When evaluating larger models that had been developed by various researchers, we discovered that all of the working models were comprised of a fairly evenly connected set of variables. In other words, the models did not lend themselves to decomposition. After further inspection, it became clear that this was due to the limitations of the available simulation techniques. To obtain a working model, the modeler often had to make simplifying assumptions with respect to the representation of various components within the model (Kay, 1992). The larger models that did not fit this description were usually highly unconstrained since the modeler was unable to complete the model building process due to the complexity of the simulation.

3.10.4 Extending DecSIM to other CSP problems

As discussed throughout the dissertation, qualitative simulation can be characterized as a constraint satisfaction problem. The interesting characteristic of this problem is the extension of the constraint satisfaction through time. DecSIM partitions the CSP into components utilizing existing techniques within the CSP literature to exploit structure and simplify the process of identifying the existence of a solution. What is distinct is DecSIM's ability to incrementally reason through time about the consistent solutions to the CSP. Extending this technique to a broader class of CSP's requires a better characterization of the CSP along with an identification of other similar CSP problems.

One potential area for exploration is situation calculus model checking and planning. In both cases, reasoning through time is required. Often, a description of the entire space of solutions is not required although it is possible that it would be beneficial for certain applications. Computing a representation for the consistent solution space requires techniques similar to DecSIM to ensure that the overall description does not grow exponentially as the number of unrelated variables increase within the problem specification.

3.11 Conclusions

Intractable branching due to irrelevant distinctions is one of the major factors hindering the application of qualitative reasoning techniques to large, real–world problems. Many of these distinctions result from inherent limitations of a global, state–based representation. DecSIM eliminates the need to explicitly enumerate all possible solutions and instead provides a more compact representation that exploits the existing structure within a model. Both theoretical and empirical evidence has been presented to support the claim that DecSIM significantly reduces the overall complexity of a qualitative simulation thus facilitating the application of these techniques to larger, more complex problems.

Chapter 4

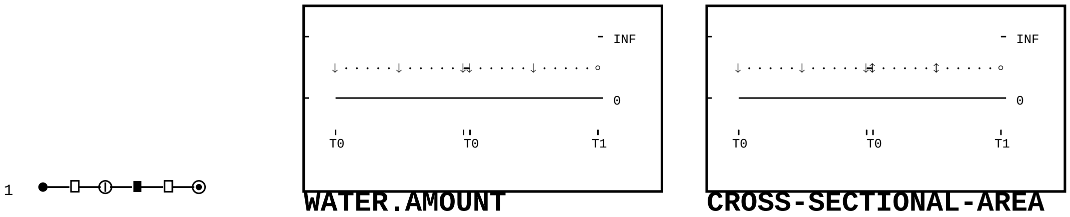
Eliminating Chatter through Abstraction

The phenomenon of chatter is a major source of complexity when reasoning about the behavior of a dynamical system using qualitative simulation. Eliminating this source of irrelevant distinctions increases the range of models that can be simulated and simplifies the process of developing a qualitative model. Figure 4.1 demonstrates how this problem can preclude the application of qualitative simulation techniques to a specific task.

4.1 Chatter Branching

Chatter occurs due to ambiguity in the qualitative description and the inability of some constraints to restrict the direction of change for a variable within certain regions of the state space. For example, suppose variables x, y and z are related by the constraint (ADD x y z) and x is constrained to be increasing while y is constrained to be decreasing. Using just the ADD constraint, the simulator is unable to infer a unique value for $[\dot{z}]$. If other constraints do not restrict $[\dot{z}]$, then z will be free to chatter. In another region of the state space, however (e.g. when both x and y are increasing), the ADD constraint can be used to infer that z must also be increasing. This example demonstrates an inherent limitation of the qualitative algebra. If two opposing influences are affecting the same variable, then simply knowing the sign of these influences does not provide enough information to determine which influence dominates. A chattering region of the state space describes the behaviors in which the dominant influence alternates back and forth.

Figure 4.2 demonstrates how chatter affects the simulation results for a model



(a) Behavior tree. **(b) Behavior plots.**

Rickel and Porter (Rickel & Porter, 1997) use qualitative simulation to answer prediction questions within the domain of plant physiology. The *TRIPEL* algorithm automatically generates a qualitative model from a large–scale botany knowledge base (Porter, Lester, Murray, Pittman, Souther, Acker, & Jones, 1988) in response to a user query. Answers are generated based upon the results of the simulation. Many of the models produced, however, exhibit a great deal of chatter. Thus, TRIPEL demands an efficient technique to automatically eliminate chatter.

- In this example, the following question is presented to the system: "What happens to the stomates when the leaves lose water?"
- Simulation of the simplest model generated by TRIPEL yields a single behavior (a) in which the cross sectional area of the stomates decreases as the leaves lose water (b).
- Dynamic chatter abstraction identifies a total of seven chattering variables. States that exhibit chatter are represented by a box in the behavior tree and a double arrow in the behavior plots. Note that following the second time–point, the cross sectional area of the stomates begins to chatter. This is because the model is unable to represent relevant order–of–magnitude information after this point. The information provided up to this point, however, is sufficient to answer the question in the proposed scenario.
- Without some form of automated chatter elimination, simulation of this model is intractable, generating hundreds of behaviors, and the query cannot be answered. While chatter–box abstraction can be used to simulate this model, dynamic chatter abstraction offers a factor of 10 speed–up in simulation time. For some of the models generated by TRIPEL, chatter–box abstraction is simply unable to complete the simulation due to resource limitations.

Figure 4.1: Using qualitative simulation to answer prediction questions.

of three connected tanks. Chatter dominates the behavioral description and obscures other relevant distinctions. When landmarks are introduced, chatter leads to an infinite simulation since the system can remain within the chattering region for an arbitrary number of qualitative states. If landmark introduction is not used, chatter still leads to intractable branching and an infinite number of valid paths in the behavioral description.

To eliminate chatter, the unconstrained region of the state space can be identified and abstracted into a single state within the behavioral description. Figure 4.3 describes the behavior of a single variable in the chattering region of the state space. While one variable chatters in this region, relevant distinctions may occur in the qualitative value of other variables in the model. We would like to eliminate the irrelevant distinctions resulting from chatter branching while retaining the distinctions in other non–chattering variables. A *chatter–box* defines the chattering region of the state space.

Definition 4.1 (Chatter–box) *For a model containing a set of variables V, a region of the qualitative state space R_{V_c}*[1] *is defined as a* chatter–box *if there exists a partitioning of the variables into a set of chattering variables C and a set of non-chattering variables NC such that*

(1) for all v such that $v \in NC$ the qualitative value for v not abstracted,

(2) for all v such that $v \in C$, there exists a region of the state space R_c, such that $R_c \subseteq R_{V_c}$ and $Qval(v, R_c) = Qval(v, R_{V_c})$, in which $[\dot{v}]$ is unconstrained with respect to the non–chattering variables, and

(3) R_{V_c} cannot be partitioned into two regions R_a and R_b such that the two regions are non-adjacent.

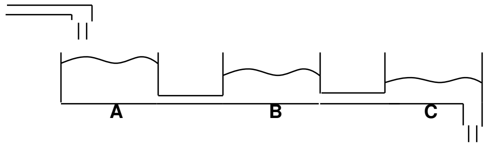
where $Qval(v, R)$ is the qualitative value of a variable v for the entire region R.

Definition 4.2 (Adjacent regions) *Two regions of the qualitative state space R_a and R_b are* adjacent *if and only if either*

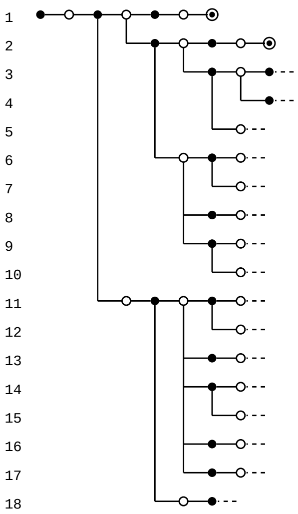
(1) there exists two concise qualitative states S_a and S_b such that $S_a \in R_a$ and $S_b \in R_b$ where either S_a is a consistent successor[2] *of S_b, or S_b is a* consistent successor *of S_a, or*

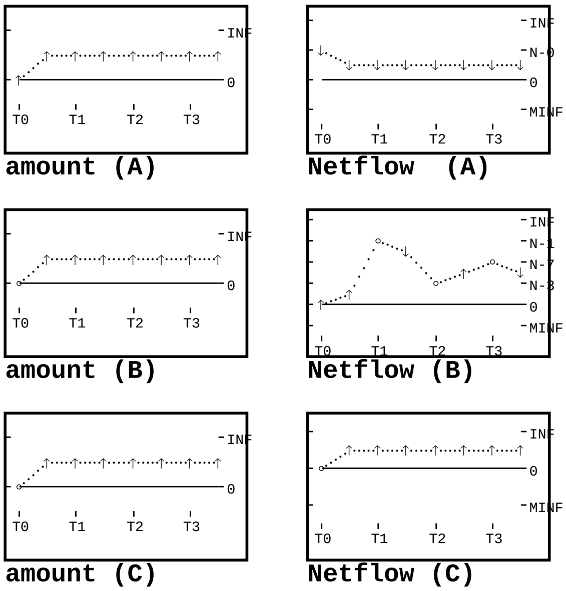
[1]The definition of a *region of the qualitative state space* is equivalent to the definition of an abstract qualitative state.

[2]State A is a *consistent successor* of state B if both A and B are consistent and if there exists the transition from B to A that is consistent with the QSIM continuity constraints.



(a)

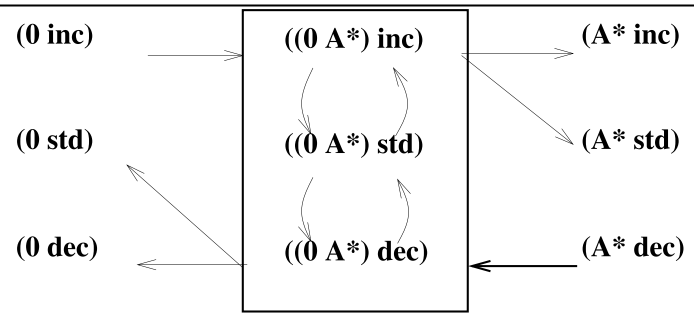




(b) (c)

- In a qualitative model of three tanks arranged in sequence connected by tubes (a), *NetflowB(t)* = *InflowB(t)* - *OutflowB(t)* is constrained only by continuity in the interval $(0, \infty)$.
- The simulation branches on all possible trajectories of *NetflowB(t)* while all other variables are completely uniform. Chatter results in an inifinte simulation that must be halted at an arbitrary state limit if it is to terminate. A single behavior (c) from the behavior tree (b) demonstrates the unconstrained movement of *NetflowB(t)*.
- Other techniques such as higher order derivative (HOD) information and ignore–qdirs (both discussed in section 4.2) are unable to eliminate chatter in *NetflowB(t)*. In this example, chatter box abstraction uses HOD information to eliminate chatter in other variables.

Figure 4.2: Intractable branching due to chatter in the simulation of a W tube.



The text in the figure represents qualitative values while the arrows indicate qualitative value transitions consistent with continuity.

- Qualitative value transitions consistent with continuity in the closed interval `(0 A*)` are identified by arcs within the figure.
- The boxed area denotes the chattering region of the state space. Once the system enters this region, the chattering variable can continue to cycle within the box.
- If landmarks are introduced, the simulation can remain within the boxed region for an infinite number of states since additional qualitative distinctions are introduced whenever the variable becomes steady.

Figure 4.3: Possible qualitative value transitions for a QSIM variable.

(2) there exists three qualitative states S_a, S_b, and S_{ab} such that $S_a \in R_a$, $S_b \in R_b$, and S_{ab} is on the boundary *between R_a and R_b, where either S_{ab} is a* consistent successor *of S_a and S_b is a* consistent successor *of S_{ab}, or S_{ab} is a* consistent successor *of S_b and S_a is a* consistent successor *of S_{ab}.*

Definition 4.3 (Unconstrained qvalue) *For a given region of the qualitative state space R the qvalue of a variable v is* unconstrained *within an abstract region of its description space A_v with respect to a set of variables V if and only if*

(1) for all concise qvalues in A_v there exists a concise qualitative state within R containing this qvalue,

$$\forall qv : qv \in A_v :: (\exists S : S \in R :: Qval(v, S) = qv)$$

(2) for all concise qvalues in A_v and for all concise qvalues for the variables in V, there exists a consistent completion of R containing these values.

$$\forall qv, v_a, qv_a : q \in A_v, v_a \in V, qv_a \in Qval(v_a, R) :: \\ (\exists S : S \in R :: Qval(v, S) = qv \ \wedge Qval(v_a, S) = qv_a)$$

When A_v is not specified it is assumed that it is the entire description space for the variable referenced.

Similarly, we can refer to simply the qmag or the qdir of a variable being unconstrained, in which case the statements above only apply to the appropriate portion of a qvalue.

A state exits the chatter–box when a non–chattering variable changes value or when a chattering variable exits the chattering region. Two chatter–boxes are adjacent if a non–chattering variable changes value while another variable continues to chatter. Note also that one chatter–box may be a proper subset of a larger chatter–box if the set of chattering variables are also related by a subset relation. A *maximal* chatter–box is defined as a chatter–box that is not contained in another chatter–box.

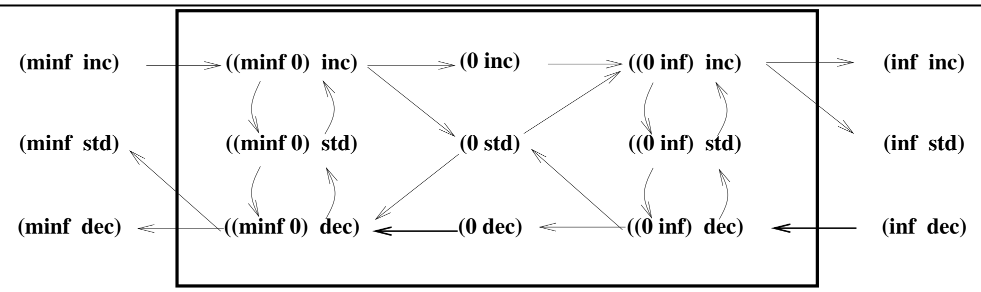
The phenomenon of chatter becomes more complicated if a landmark exists in the unconstrained region of the state space and the magnitude of the variable is unconstrained around this landmark. This phenomenon is called *landmark chatter* and it results in changes in both the magnitude and the direction of change for the chattering variable. A special case of landmark chatter, called *chatter around zero*, occurs when the landmark around which the chattering occurs is zero. In general, this occurs when a variable and its derivative are represented as variables in a model and both of them exhibit chatter. In this case, the derivative variable will chatter around zero as its integral chatters. Chatter around zero is particularly difficult to handle since one variable changes sign. Figure 4.4 demonstrates this phenomenon from the perspective of the transitions that are consistent with continuity.

4.1.1 Chatter versus oscillation

Chatter, as defined here, is a property of a *region* in the behavioral description (*i.e.* a set of partial qualitative behaviors) and not of a single behavior. This is what distinguishes chatter from oscillation; oscillation is a property of a single behavior. Within a chattering region, certain behaviors will be oscillatory in nature. Some, however, will increase, become steady and then begin increasing again while others will exhibit a combination of these two behaviors.

This distinction helps to answer the question: "Is chatter an actual behavior of the modeled system?" While in certain instances, chatter may be the result of spurious behaviors[3], in general chatter reflects potential behaviors of the modeled system (Kuipers, Chiu, Molle, & Throop, 1991). Since chatter is a property of a

[3] A qualitative behavior is *spurious* if there does not exist a corresponding real–valued trajectory.



- If the qualitative magnitude and direction of change of a variable v are unconstrained around a landmark and its direction of change is also unconstrained, then v can move freely within this region of the state space.
- The figure above shows the valid QSIM transitions consistent with continuity for a variable with the quantity space `(minf 0 inf)`. The boxed region identifies the chattering region of the state space for chatter around zero.
- Chatter around zero is particularly difficult to handle since the sign of the variable changes.

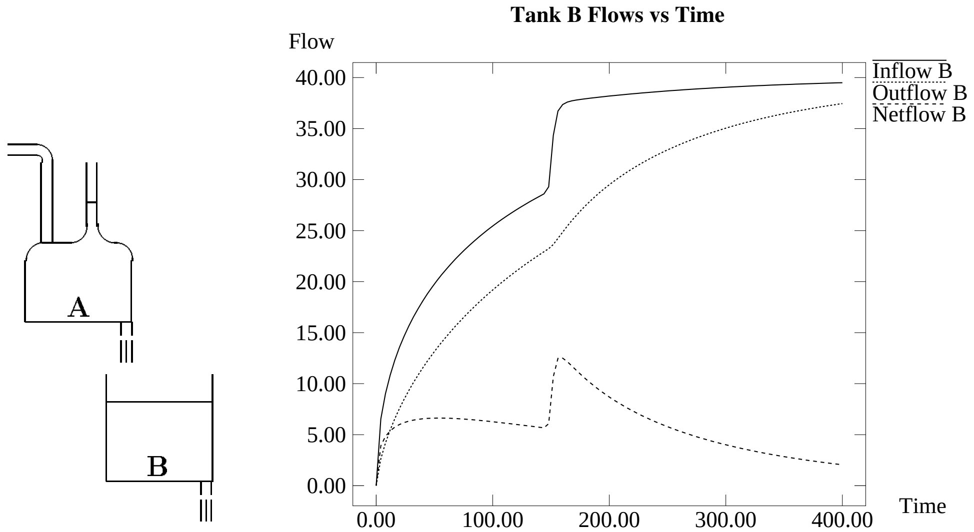
Figure 4.4: Chatter around zero: Possible QSIM transitions.

class of behaviors, a single real–valued trajectory will not exhibit chatter. However, for any given path in a chattering region, the class of dynamical systems described by the model may include an instance exhibiting this behavior. Figure 4.5 demonstrates this for a two–tank cascade using a numerical simulation.

4.1.2 Eliminating chatter

While chatter may correspond to potential behaviors of the system, a disjunctive enumeration of all possible combinations of values provides no useful information. Furthermore, this enumeration obscures other distinctions within the description and can result in an infinite simulation. Eliminating this source of distinctions is essential if qualitative simulation is to be used to reason about complex dynamical systems.

Two general abstraction techniques have been developed to eliminate chatter: chatter–box abstraction and dynamic chatter abstraction. While both techniques can be used to eliminate chatter, in combination they provide the modeler with a greater degree of flexibility. The next section provides a description of existing techniques for eliminating chatter. This is followed by a detailed description of both new algorithms. At the end of this chapter, both techniques are evaluated and a



(a) (b)

- In a standard QSIM simulation of a two tank cascade, *netflowB* exhibits chatter. At first glance, it might appear that the chattering behaviors are spurious and that the actual behavior of the system *netflowB* simply rises monotonically to its maximum value and then returns monotonically to zero.
- However, consider an actual pair of tanks in which the upper tank has a tall thin stack (a) thus causing the monotonic relationship between *outflowA* and *amountA* to have a sharp bend.[a]
- A numerical simulation of this example (b) shows that *netflowB* reaches a critical point, begins to decrease and then rises sharply before beginning to decrease once again. This is one of the behaviors described in the chattering region of the qualitative behavior.
- By extending this example, so that the upper tank has an arbitrary number of decreases in the diameter of the upper tank, we can create an arbitrary number of "dips" within the behavior of *netflowB*. This example demonstrates how chatter can reflect actual behaviors of a physical system.

[a]This example is taken from (Kuipers, 1994) page 256.

Figure 4.5: Numerical Simulation of a Two Tank Cascade

comparison of the two techniques is provided.

4.2 Previous Solutions

Two previous methods have been developed for eliminating chatter. The *higher–order derivative* (HOD) (Kuipers et al., 1991) technique uses the second and third–order derivatives of the chattering variables to determine the direction of change; the *ignore qdirs* approach simply ignores distinctions in the derivative of variables identified by the modeler as chattering. Neither of these techniques provide a general–purpose solution to the problem of chatter. In particular, neither can be used to eliminate chatter around a landmark.

4.2.1 Higher–Order Derivatives

Qualitative simulation uses continuity to restrict the potential successor values of a variable. Information about the sign of the first derivative (*i.e.* direction of change) is used to restrict the potential successor values for the magnitude of a variable. In a similar manner, the HOD technique uses information about the sign of the second and third–order derivatives of a variable to constrain the value of the first derivative. Expressions for the higher–order derivatives of a variable can either be provided by the modeler or, when possible, automatically derived through algebraic manipulation of the constraints within the model.

The HOD technique eliminates a chatter branch for a variable v by deriving a value for the sign of the second or third–order derivative either at the time–point t_i or in the interval $(t_{i-1}\ t_i)$ preceding t_i. This information is used to filter spurious behaviors that are inconsistent with continuity. If in the interval $(t_{i-1}\ t_i)$ the sign of the second derivative is the same as the sign of the first derivative, then the first derivative must be moving away from zero. Thus, v cannot become steady at t_i; therefore the corresponding behavior is spurious and can be filtered.

In a similar manner, higher–order derivative information can be used to filter behaviors in which a v becomes steady at t_i and then begins to change in the same direction as it was in $(t_{i-1}\ t_i)$. These behaviors are filtered when the sign of the second derivative of v at t_i is opposite the sign of its derivative in $(t_{i-1}\ t_i)$. When this condition holds, the second derivative continues to change in the same direction as it was in $(t_{i-1}\ t_i)$ and thus the variable begins to change in the opposite direction following t_i. If the sign of the second derivative is zero, then information about the the third derivative is used in a similar manner.

Higher–order derivative information is effective at filtering spurious behaviors resulting in chatter in certain situations. The following problems, however, may be encountered:

(1) Deriving expressions for the higher–order derivatives using variables represented within the model is not always possible and is often difficult to do automatically.

(2) The HOD constraint cannot be used when the expression evaluates to an ambiguous sign.

(3) The automatic derivation of the HOD constraint applies additional assumptions about the functions described by M^+ and M^- constraints. If two variables, x and y, are related by a monotonically increasing function, the *sign equality assumption* assumes that

$$[y''(t)] = [x''(t)].$$

Recent results (Say, 1997) suggest that this restriction could significantly restrict the class of dynamical systems modeled by the QDE.

While the HOD constraint can be quite useful, there are many instances in which it is unable to eliminate chatter. The three tank W-tube described in figure 4.2 provides a simple example in which HOD constraints are inadequate.

4.2.2 Ignore Qdirs

Ignore qdirs (Fouché & Kuipers, 1990) eliminates chatter by ignoring distinctions in a variable's direction of change throughout the simulation, thus eliminating the source of the ambiguity by providing a more–abstract description. This abstraction technique is applied to variables identified by the modeler prior to the simulation.

For some simpler models, simply ignoring the direction of change is an effective way to eliminate chatter without losing constraining power. For many models, however, a variable's direction of change is ambiguous only within certain regions of the state space. In other regions, this information may be relevant in constraining the behavior of the system. In addition, ignoring the direction of change requires the user to identify the variables prior to the simulation; furthermore, it may prevent the application HOD constraints. In the W-tube example, ignoring the direction of change of NetflowB prevents the application of the HOD constraint for other variables causing a number of them to exhibit chatter.

4.3 Chatter–Box Abstraction

The chatter–box abstraction algorithm eliminates chatter by abstracting each chattering region of the state space into a single qualitative state in the behavioral description. For each time–interval state, the following steps are performed:

Step 1: Define an upper bound on the potentially chattering region (*i.e.* identify potentially chattering variables),

Step 2: Explore the chattering region via a recursive call to the simulation algorithm to determine which variables actually exhibit chatter,

Step 3: Analyze the result of the recursive call to the simulation algorithm to identify the exact boundaries of the chattering region, and

Step 4: Create and insert into the behavior tree abstract states describing the chatter–box and its successors.

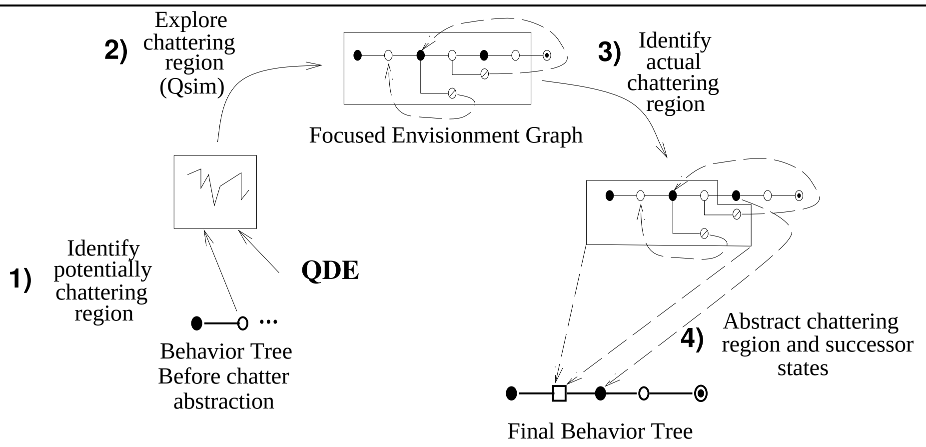
Potentially chattering variables are identified via an analysis of the QDE constraints to identify variables whose direction of change is potentially unconstrained. The set of potentially chattering variables is used to identify an upper bound on the chattering region of the state space. A *focused envisionment* is then used to explore this region of the state space and determine which variables, if any, exhibit chatter. A focused envisionment is a recursive call to the simulation algorithm defined as follows:

Definition 4.4 (Focused envisionment) *A* focused envisionment *is an attainable envisionment that is restricted to a specified region of the state space. States created outside the defined region are suspended from simulation (*i.e. *their successors are not computed).*

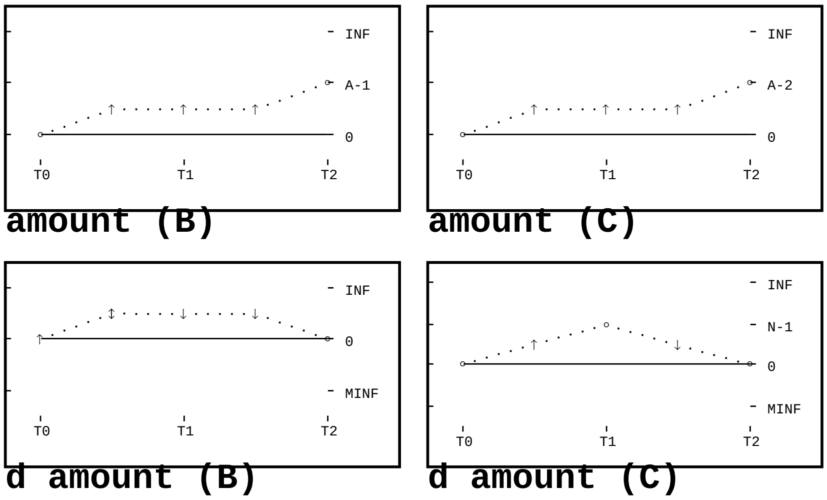
Chatter box abstraction uses the results from this simulation to identify the boundaries of the chatter–box, abstract this region into a single qualitative state, and generate the successors for this state. A more detailed discussion of each of these steps appears below. Figure 4.6 demonstrates the application of the algorithm to the W-tube example from figure 4.2.

4.3.1 Identifying potentially chattering region

Chatter box abstraction uses the identification of the potentially chattering region of the state space to bound the region to be explored during the focused envisionment. Potentially chattering variables are identified via an analysis of the QDE and the



(a) chatter–box algorithm



(b) W-tube behavior plots

- In this example, the potentially chattering variables include `Netflow-AC`, `Crossflow-BC`, `Delta-BC`, `Netflow-B`, `Netflow-A`, `Crossflow-AB`, and `Delta-AB` (see figure 4.7). Due to the application of HOD constraints during the focused envisionment, `NetflowB` is the only variable that actually exhibits chatter.
- In the focused envisionment graph, cycles are represented by dotted lines. The chattering region is abstracted into a single state in the main simulation, represented by a □. The qdir in the chattering region is represented by a bi-directional arrow in the behavior plot.

Figure 4.6: chatter–box abstraction algorithm applied to the W-tube.

current state. To handle chatter around landmarks, the algorithm expands the potentially chattering region to include QSIM introduced landmarks and landmarks identified as chatter landmarks within the QDE.

Analysis of the QDE Kuipers *et al.* (Kuipers et al., 1991) present an algorithm for identifying potentially chattering variables based strictly upon the constraints within the model. This technique is used in the automatic derivation of HOD constraints. We have extended the algorithm to include qualitative state information when appropriate and to reason about the full set of constraints provided by QSIM. Previously, the algorithm only handled a sub–set of QSIM's constraints.
There are two main steps to the algorithm:

Step 1 Partition the variables into chatter equivalence classes.

Step 2 Identify equivalence classes that cannot chatter.

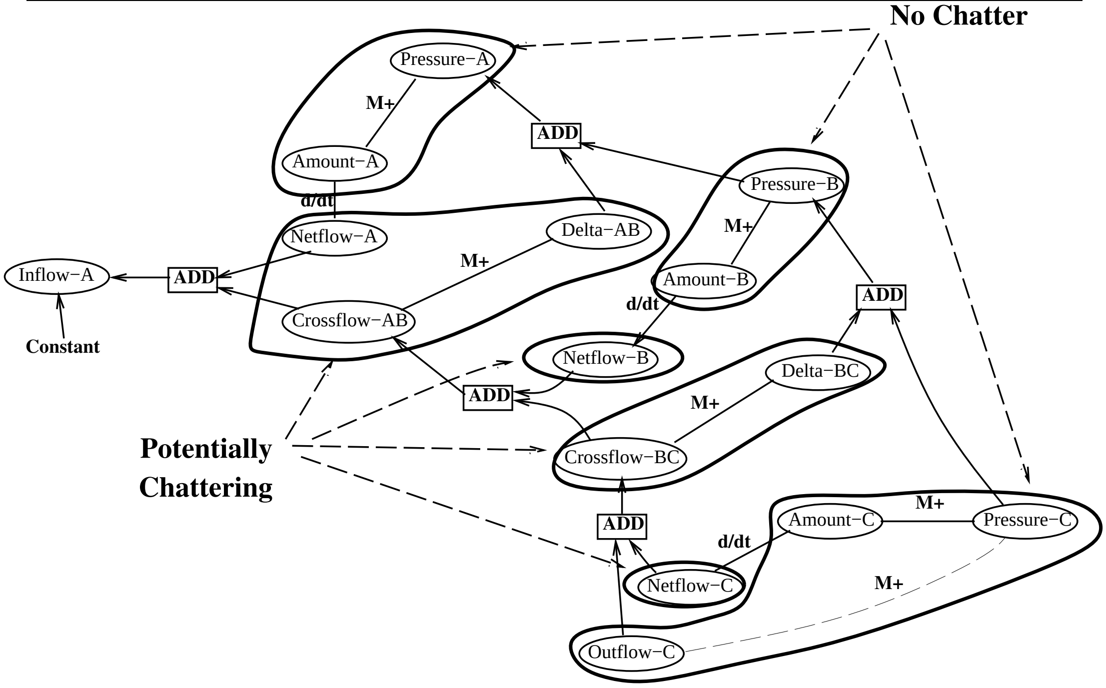
Variables that are not included within non-chattering equivalence classes are considered to be potentially chattering.

Definition 4.5 (Chatter equivalence) *Variables V_1 and V_2 are considered* chatter equivalent *within a region of the state space if and only if the sign of V_1' is uniquely determined by the sign of V_2' and vice-a-versa.*

The identification of chatter equivalence classes is based upon the observation that two variables are chatter equivalent if they are related by a monotonic function constraint (e.g. M^+ or M^-) or a refinement of a monotonic function constraint (e.g. $\text{ADD}(x, y, z)$ if either x, y, or z are constant). Note that the S^+ and S^- constraints[4] only apply the monotonicity restriction within a particular region of the state space. Thus, the current qualitative state must be used to determine if two variables related by an S^+ or an S^- constraint are chatter equivalent. Kuipers *et al.* (1991) gives a complete listing of the conditions under which two variables can be identified as chatter equivalent.

A chatter equivalence class is identified as non-chattering if either a variable in the class is constant or its derivative is explicitly represented in the model. The latter restriction, as discussed below, is relaxed if a variable's derivative is identified as potentially chattering around zero. Figure 4.7 demonstrates the detection of potentially chattering variables for the W-tube example.

[4]The S constraints describe a saturation function in which two variables, x and y, are monotonically related whenever x is within a region identified by two landmarks specified in the constraint. When x is outside this region, y is constant.



- The constraint graph identifies the relationships between the variables in the QDE. Binary constraints are represented as labeled arcs while the tertiary ADD constraint is represented by an additional node in the graph. The derivative constraint is represented by a directed edge pointing toward the derivative variable.
- Chatter equivalency classes are grouped by the bold lines and labeled with respect to whether or not the variables can potentially chatter. Variables related by an M^+ constraint grouped in the same equivalence class. In addition, since `Inflow-A` is constant, it is grouped into `Crossflow-AB`'s equivalence class due to the `ADD` constraint.
- The equivalence classes containing `Amount-A`, `Amount-B` and `Amount-C` are identified as non–chattering since their derivatives are represented in the model.

Figure 4.7: W-tube Chatter Equivalence Classes

Landmark chatter Once the algorithm identifies the set of potentially chattering variables, it is necessary to determine if any of these variables can potentially chatter around neighboring landmarks. Two types of landmarks exist in the quantity space of a variable: landmarks identified in the model and those created during the simulation. The landmarks in the model have been identified as relevant by the modeler. The chatter–box abstraction algorithm will not eliminate chatter around such a landmark unless the modeler identifies the variable and the landmark as a potential chatter landmark in the QDE. In particular, chatter around zero is only eliminated if the modeler specifies this in the model. This restriction is applied to increase the efficiency of the algorithm since chatter around a user–defined landmark is an unusual event that can be easily detected by the modeler.[5] If a variable is potentially chattering within an interval bounded by a landmark introduced during the simulation or identified as a chatter landmark in the model, then the potentially chattering region of the state space is expanded to include both this landmark and the interval on the other side of the landmark. The process continues until a user–specified, non–chatter landmark is encountered.

When chatter around zero occurs, it means the derivative constraint is unable to prevent chatter in the derivative variable. Thus, chatter equivalence classes marked as non–chattering due to the existence of a derivative constraint must be marked as potentially chattering.

Lemma 4.1 *The region of the state space identified as* potentially chattering *is a superset of the actual chattering region of the state space.*

Proof: Two variables x and y are identified as *chatter equivalent* if and only if $[\dot{x}] = [\dot{y}]$ or $[\dot{x}] = -[\dot{y}]$. Thus, all of the variables in an equivalence class chatter in unison. If the derivative of one of the variables in the class is constrained, then the derivatives of all of the variables within the class are constrained. A chatter equivalency class is identified as non–chattering only if the derivative of a variable within the class is constrained throughout the simulation by a single constraint. The chattering region is extended to include all QSIM introduced landmarks and any landmark within the model identified as a potential chatter landmark. Kuipers *et al.* (1991) contains a more detailed proof of this algorithm. □

[5] For automated model building applications, an alternative operating mode detects chatter around a landmark when it occurs in the main simulation and then backtracks to the preceding focused envisionment and expands the potentially chattering region to include this landmark.

4.3.2 Focused envisionment

A focused envisionment is used to explore the potentially chattering region of the state space to determine which variables chatter and whether chatter around a landmark occurs. Performing the envisionment is a simple matter of creating an initial time–point state from the time–interval state within the main simulation and recursively calling the QSIM algorithm. An additional filter detects when a state exits the potentially chattering region. Such states are suspended from simulation.

4.3.3 Identifying the boundaries of the chattering region

The chatter box abstraction algorithm identifies the actual chattering region of the state space through an analysis of the focused envisionment graph. A variable v is identified as chattering if and only if there exists a branch following a time–point state due to distinctions in the qualitative value of v. If this branch is due to a change in the qualitative magnitude, then the variable is considered to chatter around the distinguishing landmark. This information defines the actual chattering region of the state space. States that exit this region are labeled as *exit states*. Since the chatter–box defines a time–interval region of the state space, exit states are required to be time–point states.

4.3.4 Abstract state and successor state creation

The chattering region of the envisionment graph is abstracted to a single state by conjunctively combining the qualitative values of the variables in the graph. For example, if a variable v is chattering around a landmark l_j, then its qualitative magnitude is simply represented as (l_{j-1}, l_{j+1}). This representation allows for the description of a variable that simply alternates between increasing and steady or decreasing and steady without changing in the opposite direction. Such phenomena are described by the abstract qdirs `(inc std)` and `(std dec)` which correspond to the representation of a closed interval at zero.

Exit states are used to create the successors of the abstract chatter state. A chattering variable, however, may continue to chatter as a non–chattering variable changes its qualitative value. This results in an exit state for each consistent value assignment for the qdirs of the chattering variables. Exit states that are equivalent with respect to the non–chattering variables are combined to create a single abstract exit state. Since exit states are time–point states, the qualitative direction of change is represented as a *disjunctive* list of potential values.

$QV(v, t_i)$	$\Rightarrow$	$QV(v, t_i, t_{i+1})$
$\langle l_j, (inc\ std) \rangle$		$\langle l_j, std \rangle$
$\langle l_j, (inc\ std) \rangle$		$\langle (l_j, l_{j+1}), inc \rangle$
$\langle l_j, (std\ dec) \rangle$		$\langle l_j, std \rangle$
$\langle l_j, (std\ dec) \rangle$		$\langle (l_j, l_{j-1}), dec \rangle$
$\langle (l_j, l_k), (inc\ std\ dec) \rangle$		$\langle (l_j, l_k), (inc\ std\ dec) \rangle$
$\langle (l_j, l_k), (inc\ std) \rangle$		$\langle (l_j, l_k), (inc\ std) \rangle$
$\langle (l_j, l_k), (std\ dec) \rangle$		$\langle (l_j, l_k), (std\ dec) \rangle$

Table 4.1: Extensions to the QSIM I-Successor transition table

4.3.5 Time–point Successor Generation

Chatter–box abstraction extends the qualitative behavior language to include abstracted qvalues. This extension requires a modification to the part of the QSIM successor–generation algorithm that computes the successors for a time–point state with an abstract qvalue. For time–interval states, this extension to the representation language does not pose a problem since the chatter–box abstraction algorithm computes the successors via the focused envisionment.

The QSIM successor–generation algorithm uses continuity to identify the possible successor values for each variable. Extending the successor–generation algorithm simply requires specification of the successor values that are consistent with continuity for an abstracted qdir. Table 4.3.5 contains the extensions required to handle abstracted qdirs.

4.3.6 Results

The chatter–box abstraction algorithm eliminates chatter by transforming the behavioral description generated by QSIM into an alternative representation. Evaluation of the algorithm requires an analysis of the extent to which this transformation eliminates chatter without over–abstracting.

Theorem 4.1 *Chatter–box abstraction eliminates all chatter branches from the abstract behavioral description except chatter around a user–defined landmark not identified as a chatter landmark.*

Proof: The focused envisionment used to explore the chattering region of the state space is simply an extension of the regular simulation. Thus, if a chatter branch occurs in the main simulation following state S_i, then a corresponding chatter branch will occur in the focused envisionment for the time-interval preceding the chatter branch since identification of the potentially chattering variables provides an upper bound on the chattering region of the state space (lemma 4.1). Chatter–box abstraction analyzes the results of the focused envisionment and abstracts the entire chattering region into a single abstract qualitative state. Thus, S_i would be included within this abstract state and thus would not be generated as a successor.

Since the potentially chattering region only includes user defined–landmarks identified as potential chatter landmarks, then chatter can still occur around a user–defined landmark not identified as a chatter landmark. □

Theorem 4.2 (Chatter elimination completeness) *All variables detected as chattering by the chatter–box abstraction algorithm exhibit chatter within the identified region of the state space in the unabstracted behavior tree.*

Proof: Chatter–box abstraction only identifies a variable as chattering if there exists a chatter branch within the focused envisionment. Since a focused envisionment is simply an extension of a standard simulation, any state within the focused envisionment has a corresponding state within the non-abstracted behavior tree. Thus, if a chatter branch occurs within the focused envisionment then a corresponding chatter branch will occur in the unabstracted tree. □

4.3.6.1 Real valued trajectories

It is also important to evaluate the set of real–valued trajectories described by the abstracted behavioral description. Chatter box abstraction retains the focused envisionment graphs used by the algorithm. These graphs describe the abstracted region of the state space at the same level of detail as the unabstracted tree, except that landmarks are not introduced. Augmenting the behavioral description with information from the focused envisionment graphs results in a description that is equivalent to the non–abstracted behavioral description and thus describes the same set of real–valued trajectories.

Definition 4.6 (Chatter extension) *The* chatter extension *of an abstract behavior tree is the behavior tree that results when each abstract state and its successors are replaced by an "unfolding" of the corresponding focused envisionment graph.*

To generate an extension of an abstract behavior tree we will use the following mappings:

$\Pi_{entry}(S)$ takes an abstracted time–interval state from the main simulation and returns the time–interval state that follows the initial state of the focused envisionment corresponding to S. (This state is identical to the state that existed before the abstract state was generated.)

$\Pi_{fe}(S)$ takes a state S within the chattering region of a focused envisionment graph and returns the corresponding state within the main behavior tree. Thus, for the non-exit states within the focused envisionment graph, it returns the abstract time–interval state, and for an exit states, it returns the appropriate successor of the abstract time–interval state.

A chatter extension of an abstract behavior tree can be generated as follows:

(1) For each abstract time–interval state S_i, replace it with S_i' such that $S_i' = \Pi_{entry}(S_i)$,

(2) For each finite path within the chattering region of the focused envisionment that terminates in an exit state S_{exit}, generate a corresponding behavior segment that follows from S_i'. Call the terminating state in this behavior segment S_{exit}', and

(3) Generate a behavior segment that follows from S_{exit}' for each extended behavior that follows from $\Pi fe(S_{exit}')$.

Lemma 4.2 *A chatter extension of an abstract behavior tree is equivalent to a non–abstracted behavior tree generated by a standard QSIM simulation except for the introduction of landmarks.*

Proof: A focused envisionment graph uses the same algorithm to generate the successors of each state within the graph with two exceptions:

- landmarks are not introduced, and
- the focused envisionment generates a graph instead of a tree.

If landmarks are not introduced, an envisionment graph is simply a more–concise description of the behaviors in a behavior tree. Thus, each path within an envisionment graph corresponds to a behavior in a behavior tree. Thus, the two representations are identical modulo the introduction of landmarks. □

Lemma 4.3 *The introduction of landmarks within a chattering region does not constrain the set of real–valued trajectories described by the behavioral description.*

Proof: The introduction of a landmark $l_{j'}$ within the interval (l_j, l_{j+1}) for variable v partitions the interval into two regions. This distinction only serves to refine the space of real–valued trajectories if the constraints within the model can eliminate a behavior that passes through one of these segments. Following a chatter branch, however, behaviors are generated in both segments of the partitioned interval. If the simulation were able to eliminate the behavior of the system within one of these intervals, then chatter would not occur. The introduction of a landmark within a chattering region adds a qualitative distinction that increases the number of behaviors; however, this distinction fails to restrict the set of real–valued trajectories described by the behavioral description. □

Theorem 4.3 *The set of real–valued trajectories described by the abstracted behavioral description, coupled with the focused envisionment graphs, is the same as the set of trajectories described by the unabstracted tree.*

Proof: Since the set of qualitative behaviors described by the chatter extended abstract tree is identical to the set of qualitative behaviors within an unabstracted tree (lemma 4.2), and the introduction of landmarks within a chattering region fails to refine the set of real–valued trajectories (lemma 4.3), then the set of real–valued trajectories described by the two representations are identical. □

Theorem 4.4 *The set of real–valued trajectories described by the abstracted behavioral description is a super-set of the real–valued trajectories described by the unabstracted tree.*

Proof: Suppose S_a is an abstract state in the abstract behavior tree and B_{fe} is the chattering region of the focused envisionment corresponding to S_a. Then, by definition of the abstraction operation used to generate S_a,

$$\forall S : S \in B_{fe} :: S \subset S_a$$

Since S_a does not constrain the transitions in the abstracted region of the state space, any real–valued trajectory described by B_{fe} is also possible within S_a. Therefore, the theorem follows from this fact, coupled with theorem 4.3. □

Thus, chatter–box abstraction preserves the QSIM soundness guarantee. The space of real–valued trajectories described by the abstracted tree, however, may be larger than the space described by the unabstracted tree. This is due to the fact that a single abstract state is used to describe the chattering region of the state space. This state does not contain information correlating the direction of change for chattering variables. Certain correlations may be precluded by the constraints within the model. For example, if both `A` and `B` are chattering then their qdirs are represented as a the conjunctive list (`inc std dec`). The representation allows any combination for the qdir of these two variables. If these two variables are related by an M^+ constraint, then the model requires the qdir of the variables to be equal. The abstracted behavioral description, therefore, describes real–valued trajectories that are not described by the unabstracted tree.

Increasing the space of real–valued trajectories, however, has little effect on the results generated by the simulation when performing a specific task since the additional trajectories are limited to distinctions in the direction of change for the chattering variables. In general, chatter only appears in intermediate variables or explicit derivative variables represented in the model.

Theorem 4.5 *The set of real–valued trajectories for non–chattering variables described by the abstracted behavioral description is equivalent to the set of trajectories described by the unabstracted tree.*

Proof: By definition, non–chattering variables do not change qualitative value within a chatter–box. Thus, the qualitative value of a non–chattering variable is the same in the abstracted state as it is in the corresponding states of the non–abstracted tree. Thus, the abstraction operation does not eliminate any distinctions in a non–chattering variable. □

4.3.7 Discussion

Chatter–box abstraction introduces the concept of a recursive call to the simulation algorithm to explore a restricted region of the state space. One of the major advantages of this approach is that it allows for the seamless integration of extensions to the QSIM algorithm into the abstraction process. No modifications are required to the chatter box abstraction algorithm as QSIM's inference ability is extended via new constraints or extensions to the core algorithm.

The simulation of the W-tube model described in figure 4.2 demonstrates the benefits of this approach. HOD derivative constraints can eliminate chatter in all of the variables except for `NetflowB`. The focused envisionment incorporates the HOD reasoning without any special extensions. This feature is also exploited when TeQSIM restricts the simulation via trajectory constraints. The focused envisionment incorporates these constraints when it explores the potentially chattering region of the state space without any special processing.

Unfortunately, a draw back of this approach is that it still encounters some of the computational complexity problems due to intractable branching. Since the focused envisionment explores the chattering region of the state space via simulation, the size of the envisionment graph is exponential in the number of chattering variables. The worst–case complexity of the graph is $O(2^{(v_t - v_c)} 3^{eq})$ where v_t is the total number of non–constant variables, v_c is the number of chattering variables and eq is the number of chatter equivalency classes.[6]

Chatter–box abstraction works effectively for most of the qualitative models currently being developed; however, this technique encounters problems as the number of chattering variables increases. Using a Sparc10, the size of the focused envisionment graph becomes a problem when about 6 or 7 chatter equivalency classes simultaneously chatter within the same region of the state space.

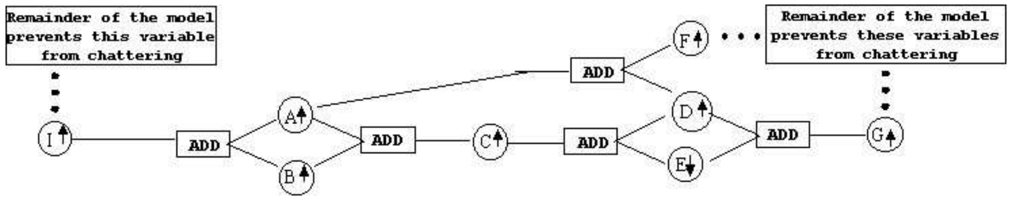
4.4 Dynamic Chatter Abstraction

Dynamic chatter abstraction uses an abstraction technique similar to chatter–box abstraction; however, as opposed to exploring the chattering region via simulation, it uses an understanding of the restrictions that are asserted by each constraint in the model, along with the current qualitative state, to determine if the derivative of a variable is constrained. For example, in the W-tube model, the direction of change for `NetflowB` is restricted only by the constraint `NetflowB` + `Flow-BC` = `Flow-AB`. In the time–interval following the initial state, all three variables are increasing. Thus, in this state the qdir of `NetflowB` is unconstrained and `NetflowB` is free to chatter.

Dynamic chatter abstraction, however, reasons not only about the qualitative values contained within the current state, but also about how these values change as variables begin to chatter. Once `NetflowB` chatters, its derivative can change sign and the addition constraint above no longer restricts the derivative of `Flow-AB`. If `Flow-AB` is not prevented from chattering by other constraints, it is free to chatter and must be identified accordingly in the current chatter–box. Figure 4.8 shows a more–complex example demonstrating how chatter can propagate through the constraint graph.

Once the chattering variables are identified, an abstract, time–interval state describing the chattering region of the state space is created and inserted into the behavior tree. Successors of this abstract state are computed through an extension

[6] There are three different values for the qdir of each chattering variable resulting in 3^{eq} different distinctions since all of the variables within an equivalency class chatter in unison. Each non–chattering variable can in turn have up to 2 values (*i.e.* it can remain the same or reach the approaching landmark). Thus, there can be a total of $(2^{(v_t - v_c)} 3^{eq})$ states within the envisionment graph.



In addition to reasoning about the current qualitative state, dynamic chatter abstraction also reasons about how the state can change as variables begin to chatter.

- The arrow next to each variable corresponds to the variable's current direction of change. Variables `I`, `F`, and `G` are all prevented from chattering by other constraints in the model. For each `ADD` constraint, the operands are connected to one side, and the result is connected to the other with the arrow head pointing toward the result.
- The derivative `B` is currently unconstrained and thus free to chatter. For each of the other variables, however, there is at least one constraint preventing it from chattering. In particular, note that `A` is only constrained by the `ADD` constraint relating `A`, `F` and `D`.
- Once `B` begins to decrease following a chatter branch, the constraints on the derivative of `C` are released, allowing `C` to chatter. Similarly, once `C` begins to decrease, `D` becomes free to chatter. This process repeats itself, causing the following sequence of derivatives to change sign:

  $\mathtt{B} \rightarrow \mathtt{C} \rightarrow \mathtt{D} \rightarrow \mathtt{E} \rightarrow \mathtt{C} \rightarrow \mathtt{B} \rightarrow \mathtt{A} \rightarrow$

  Note that `E`'s derivative must change sign before `C` and `B` can reverse sign and allow `A` to chatter. If `E` were prevented from chattering, then a state in which `A` were free to chatter would not be chatter reachable.
- All five of the variables listed above are included in the maximal chatter–box and thus must be identified as chattering.

Figure 4.8: Chatter Propagation

of the QSIM successor generation algorithm.

4.4.1 Detecting Chatter

For each time–interval state S, dynamic chatter abstraction determines if S is contained within a chatter–box, and if so, identifies the boundaries of the chatter–box (*i.e.* the set of chattering variables). To determine if a variable v can chatter following a given state S before the system exits the chatter–box, two questions must be answered:

Consistency - Is there a consistent state in which v is free to chatter?

Reachability - Can this state be reached from S only through changes occuring in other chattering variables?

Definition 4.7 (Chatter–reachable) *A qualitative state S' is considered* chatter–reachable *with respect to state S if and only if there exists a consistent path $b = {<}s_1, s_2, \ldots, s_n{>}$ where $n \geq 1$ such that*

- $S = s_1$, $S' = s_n$,
- *each qualitative change occuring in the path results from a chatter branch.*

For given time–interval state S, a three–step process is used to answer these questions:

Step 1 Partition the variables into chatter equivalency classes as in the chatter–box abstraction algorithm (see section 4.3.1 and figure 4.7).

Step 2 For each equivalency class EQ, define a *chattering region predicate* C_{eq} describing the conditions under which the variables in EQ are free to chatter. The predicate $C_{eq}(S')$ where S' is an abstract qualitative state, returns `true` if and only if the derivatives of the variables in EQ are unconstrained in S' with respect to the other variables in the model.

Step 3 For each equivalency class EQ, if a chatter–reachable state that satisfies C_{eq} exists, then classify the variables in EQ as chattering for state S.

4.4.2 Chattering region predicate: Syntax and semantics

C_{eq} is a boolean predicate specifying necessary and sufficient conditions for the variables in EQ to chatter. The predicate is a structured, boolean combination of atomic propositions called *assertions*. Each assertion specifies a constraint on the qualitative value of the variables in an equivalency class. Assertions can specify two types of constraints: an *outer bound* specifies a set of qualitative values for the variables within an equivalency class while an *inner bound* identifies a region in which S' must be unconstrained. Some of the assertions specify conditions relative to the qualitative value of the variable within the qualitative state, S, being evaluated by the dynamic chatter abstraction algorithm. Note that this state is different from the state S' that is the target of the predicate $C_{eq}(S')$. The constraints currently used by QSIM require six different assertion types. S and S' are the states being evaluated by the dynamic chatter abstraction algorithm and C_{eq} respectively.

(`:same-qdir` eq S) is true if and only if for all v such that $v \in V_{eq}$, $Qdir(v, S') = Qdir(v, S)$, where V_{eq} is the set of variables within equivalency class eq.

(`:change-qdir` eq) S is true if and only if for all v such that $v \in V_{eq}$, $Qdir(v, S')$ is the opposite of $Qdir(v, S)$).

(`:qdir` v $qdir$) , where v is a variable, is true if and only if $Qdir(v, S') = qdir$.

(`:chatter` eq) is used to help determine if a state satisfying C_{eq} is chatter–reachable. It is true if and only if the variables in eq have already been identified as chattering. While this assertion is included in C_{eq}, its semantics are not limited to the state S', and thus it is not formally included within the definition of C_{eq}.

(`:uncon-qdir` v *qdir-list*) , where *qdir-list* is a list of qdirs, is true if and only if *qdir-list* $\subseteq Qdir(v, S')$ and for all qd such that $qd \in$ *qdir-list*, there exists a consistent completion S_{qd} of S' where the $Qdir(v, S_{qd}) = Qdir(v, S')$.

(`:uncon-qmag` v $(lm_l\ lm_u)$) , where $(lm_l\ lm_u)$ is an interval in the quantity space for v, is true if and only if $(lm_l\ lm_u) \subseteq Qmag(v, S')$, and for all qm such that $qm \in (lm_l\ lm_u)$ there exists a consistent completion S_{qm} of S' where $Qmag(v, S_{qm}) = Qmag(v, S')$.

The last two assertion types specify inner bounds on S' while the first three specify outer bounds. In general, these assertion types only make minimal use of the concept of inner and outer bounds. The semantics of C_{eq}, however, have been defined in a general manner so that additional assertion types can be specified without extending the semantics.

Assertions are combined using standard boolean semantics for the boolean connectives `AND` and `OR` according to the following grammar.

Chattering region predicate	$\rightarrow$	(`AND` *dependency*$^+$)
Dependency	$\rightarrow$	(`OR` *condition*$^+$)
Condition	$\rightarrow$	(`AND` *assertion*$^+$)

Each dependency corresponds to a constraint in the model that restricts the derivatives of the variables in EQ with respect to other variables in the model; each condition specifies a different condition under which the constraint fails to restrict the derivatives of the variables in EQ. For $C_{eq}(S')$ to be satisfied, S' must satisfy at least one condition from each dependency. Figure 4.9 shows an example chattering region predicate for the W-tube model.

4.4.3 Defining a chattering region predicate

The assertions in a chattering region predicate depend upon the constraints in the model. The following QSIM constraint types can restrict a variable's derivative with respect to other equivalency classes: `MULT`, `ADD`, `D/DT`, and refinements of the multivariate `M` constraint.[7] While constraints such as $\mathtt{M}^{+/-}$, $\mathtt{S}^{+/-}$, and $\mathtt{U}^{+/-}$ all restrict a variable's direction of change, the variables within these constraints are in the same equivalency class and thus need not be considered in the specification of chattering region predicate.

For an equivalency class EQ, a dependency is added to C_{eq} for each constraint C and for each variable v such that

- the constraint type of C matches one of the constraints listed above, and
- v is contained within both C and EQ.

The structure of the dependency is determined by the constraint type, the role of v within the constraint (e.g. is it an operand or the result), and the qualitative values of the variables in C in the current state:

(`ADD` x y z) –

The `ADD` constraint requires the following relationship to hold between the derivatives of the variables:

$$[\dot{x}] + [\dot{y}] = [\dot{z}]$$

[7]Refinements of this constraint include the signed–sum constraint (`SSUM`), the `SUM` constraint, and the `SUM-ZERO` constraint. Kuipers (1994) defines the syntax and semantics for these constraints.

The following example demonstrates the chattering region predicate for equivalency node EQ_1 for the time–interval state S_1 from the W-tube example. The following equivalency classes (see figure 4.7 for the derivation of these classes) are relevant to this example.

Class Name	Variables	Direction of change in state $S1$
EQ_1	`NetflowA`	*dec*
	`Flow-AB`	*inc*
	`Delta-AB`	*inc*
EQ_2	`PressureB`	*inc*
	`AmountB`	*inc*
EQ_3	`PressureA`	*inc*
	`AmountA`	*inc*
EQ_4	`NetflowB`	*inc*
EQ_5	`Flow-BC`	*inc*
	`Delta-BC`	*inc*

Two constraints in the model restrict the derivatives of the variables in EQ_1 with respect to variables not included within EQ_1.

C_1: `Delta-AB` + `PressureB` = $\hat{\text{P}}$ressureA
C_2: `NetflowB` + `FlowBC` = `Flow-AB`

The chattering region predicate C_{eq_1} defines the conditions under which the variables in EQ_1 are free to chatter.

C_{eq_1}	Description
(AND (OR (AND (:change-qdir EQ_2 S_1) (:change-qdir EQ_3 S_1)) (AND (:same-qdir EQ_2 S_1) (:same-qdir EQ_3 S_1)))	Since `Delta-AB` is an operand in `ADD` constraint C_1 and all of the variables are *inc*, the derivatives of the variables in EQ_2 and EQ_3 must either both stay the same or both change to the opposite direction.
(OR (AND (:change-qdir EQ_4 S_1) (:same-qdir EQ_5 S_1)) (AND (:same-qdir EQ_4 S_1) (:change-qdir EQ_5 S_1))))	Since `Flow-AB` is the result in `ADD` constraint C_2 and all of the variables are *inc*, the derivatives for the variables in either EQ_4 or EQ_5 must remain the same while the derivatives in the other class change to the opposite direction.

Figure 4.9: Example Chattering Region Predicate for the W-Tube

If v is an operand, then the other two variables must change in the same direction. If v corresponds to x and that y and z are both changing in opposite directions, then the following dependency is added:

```
(OR  (AND  (:change-qdir EQ_y S)
            (:same-qdir EQ_z S))
     (AND  (:same-qdir EQ_y S)
            (:change-qdir EQ_z S)))
```

where EQ_y and EQ_z refer to the equivalency classes containing variables y and z respectively and S is the current qualitative state. If they are both changing in the same direction, then the dependency would state that either both equivalency classes change or both remain the same.

On the other hand, if v is the result, then the other two variables must change in the same direction.

This example demonstrates why assertions may include a reference to the current qualitative state. While the derivatives of the variables change in unison, they are not necessarily equal (*i.e.* they could be of opposite signs). The syntax allows us to specify the qdirs for all of the variables in the equivalency class with a single statement. In addition, as we will see in the next section, the `:change-qdir` assertion is useful when the algorithm determines if a state is chatter–reachable.

`(MULT` x y z`)` –

Differentiating both sides of the `MULT` constraint yields an equation describing the manner in which the derivatives of the variables are constrained.

$$[x][\dot{y}] + [\dot{x}][y] = [\dot{z}] \tag{4.1}$$

The conditions under which v is unconstrained depend upon the value of $[v_1] * [v_2]$, where v_1 and v_2 are the other two variables in the constraint. For example, if v is the result and $[x] * [y] = [+]$, then $[x]_0 = [y]_0$. Thus, for $\dot{z}$ to be unconstrained in equation 4.1, $[\dot{x}]_0$ must equal $-[\dot{y}]_0$ to ensure that one operand is positive and the other negative. Therefore, the qdirs of the other two variables must change in opposite directions. The following table defines whether the other two variables change in the same direction or in opposite directions. The vertical axis corresponds to the sign of $[v_1]*[v_2]$; the horizontal axis corresponds to the role of v.

	result	operand
+	opposite	same
-	same	opposite

((M s_1 s_2 ... s_n) (x_1 x_2 ... x_n y)) –

For a multivariate M constraint to be satisfied for the derivatives of the variables in the constraint, one of the following conditions must be satisfied for i, j in $0..n$

- for all i, $[s_i][\dot{x}_i] = [0]$;
- for some i, $[s_i][\dot{x}_i] = [?]$;
- for some i and j, $[s_i][\dot{x}_i] = [+]$ and $[s_j][\dot{x}_j] = [-]$.

Thus, for the derivative of a variable x_i to be unconstrained, there must exist j, k in $0..n$ such that $[s_j][x_j] = -[s_k][x_k]$ with neither value equal to [0].

Therefore, for each pair of variables, excluding v, in the constraint, a condition is defined asserting this constraint. The assertions used in the condition depend upon the current direction of change of the variables as well as the sign of the direction of influence.

(D/DT x y) –

The derivative constraint asserts that $[\dot{x}] = [y]_0$. Thus, as x chatters, y chatters around zero. If the variable being considered, v, corresponds to x, the following three assertions are included in the dependency:

(:uncon-qmag x (lm_l lm_u)) where (lm_l lm_u) is the interval around zero for x. This assertion simply states that the qmag of x must be unconstrained around zero.

(:chatter eq_x) – For y to chatter, x must also be free to chatter since after crossing zero x must be able to "turn around" and return to zero. Otherwise, the variable is implicitly constrained. Note that this assertion is different from the :change-qdir assertion since it does not place any constraints on the sign of the derivative of x. It simply states that it must be identified as chattering.

(:qdir x qd) where $qd = -[x]_0$. This assertion requires y to approach zero. This constraint is asserted to determine if there is a relationship between $\dot{y}$ and $\dot{x}$ via other constraints in the model. If $\dot{y}$ is constrained as x approaches zero, then the variable cannot chatter.

4.4.4 Identifying the set of chattering variables

To determine if a given state S is contained in a chatter–box and, if so, to identify the boundaries of the chatter–box, each chatter equivalency class must be evaluated. The boolean predicate $Chatter(S, EQ)$ is defined as follows:

$$Chatter(S, EQ) \iff \exists S' : C_{eq}(S') \wedge \\ S' \text{ is } \textit{chatter–reachable} \text{ from } S.$$

where S and S' are qualitative states. $Chatter(S, EQ)$ means that S is contained in a chatter box in which the variables in EQ chatter. If this predicate is `TRUE`, then the variables in EQ must be abstracted within the current state.

The *Chatter-test* algorithm (table 4.2) identifies the set of chattering variables for a given time–interval state. Three labels are used to classify the equivalency classes: `:chatter`, `:nochatter`, and `:undetermined`. Initially, *Chatter-test* labels all of the equivalency classes `:undetermined` unless the equivalency class contains a variable that is constrained from chattering by a single constraint.[8] These classes are marked `:nochatter`. Then, for each equivalency class whose status is still `:undetermined`, the *EQ-Chatter-Test* algorthim (table 4.3) is used to evaluate the predicate $Chatter(S, EQ)$.

EQ-Chatter-Test uses a backtracking algorithm to incrementally generate an abstract state that satisfies C_{eq} ensuring that the state is chatter reachable from the current state S. The algorithm initializes a partial state S' with information from S that cannot change as a result of chatter and then gradually refines the state with information from the assertions in C_{eq}. If an assertion of the form (`:change-qdir` EQ') or (`:chatter` EQ') is encountered, then the algorithm checks the status of EQ'. If it is still `:undetermined`, then *EQ-Chatter-Test* algorthm is called recursively to compute a status for EQ'. If it is determined that the variables in EQ' can chatter, then the information in the assertion is added to S' and the algorithm continues. The algorithm backtracks in the following three cases:

1. if the qualitative value information specified in the assertion conflicts with information contained within S',
2. if an assertion requires an equivalency class with the status `:nochatter` to chatter, or
3. if a cycle is encountered in the recursive calling sequence.

[8] Constraints meeting this criterion include the `CONSTANT`, `INCREASING`, and `DECREASING` constraints, as well as the $S^{+/-}$ constraints if the range is outside of the bend points.

Chatter-test:

Given a time–interval state S, a partitioning of the variables into equivalency classes, and a chattering region predicate for each equivalency class,

1. For all EQ_i such that EQ_i is an equivalency class, if EQ_i contains a variable constrained by a constraint of type `CONSTANT`, `INCREASING`, or `DECREASING` then mark EQ_i `:nochatter`. Otherwise, mark EQ_i undetermined.
2. For i from 1 to n where n is the number of equivalency classes
   - if $Status(EQ_i)$ = `:undetermined` then
     Ret-val = *EQ-Chatter-Test* (EQ_i,C_{eq_i}, *nil*)
   - if *Ret-val* = `:chatter`
     then set the status of EQ_i to `:chatter`,
     else set the status of EQ_i to `:nochatter`.

Table 4.2: The Chatter-test Algorithm.

Figure 4.10 shows an example that demonstrates how cycles within the a segment of the model can prevent a set of variables from chattering.

Once an abstract state satisfying all of the assertions is generated, the QSIM state–completion algorithm is used to test the consistency of this state for all of the constraints in the model. In addition, the algorithm must check to see if inner bounds specified by `:uncon-qmag` and `:uncon-qdir` assertions are satisfied by this state. Inner–bound assertions are processed by annotating the abstract state with the inner bound. An inner bound is satisfied if there is a consistent completion of the state for each qualitative value contained within the inner bound.

EQ-Chatter-Test is divided into four separate agorithms: *EQ-Chatter-Test*, *Check-dependencies*, *Check-assertions*, and *Check-state-completions*. Each of these algorithms return one of the following three values: `:chatter`, `:nochatter`, and `:cycle`. The status of an equivalency class EQ is determined as follows:

- If *EQ-Chatter-Test* ever returns either `:chatter` or `:nochatter` then the status of EQ is updated accordingly; future calls to *EQ-Chatter-Test* will simply return this value.

- If a non-recursive call to *EQ-Chatter-Test* returns the value `:cycle`, then the status of EQ is set to `:nochatter`. If a recursive call to *EQ-Chatter-Test* returns `:cycle`, then the status of EQ remains `:undetermined`.

$$\begin{array}{cccccc} A & + & B & = & D \\ [+] & & [-] & = & [+] \\ \\ B & + & C & = & E \\ [-] & & [+] & = & [-] \\ \\ C & + & D & = & F \\ [+] & & [-] & = & [+] \\ \\ D & + & A & = & G \\ [-] & & [+] & = & [-] \end{array}$$

Cycles in the constraint network can prevent a variable from chattering. While this problem is unusual in standard chatter examples, it is more common when eliminating chatter around zero or when dealing with HOD constriants.

- In the model segment above, the sign of the derivative of each variable is displayed below the variable. In addition, all four result variables (D, E, F, G) are assumed to be constrained from chattering by other variables in the model.
- In the current state, none of the variables are free to chatter. For A to chatter $\dot{B}$ has to change sign first (eq. 1). Thus, A depends upon B. Similarly, B depends upon C, C upon D, and D upon A.
- Dynamic chatter abstraction must detect the cycle and determine that none of the variables can chatter when evaluating the chattering region predicates. If a propagation algorithm is used, then this condition cannot be detected.
- Once one of the result variables constrained by the rest of the model changes sign, then all of the variables start chattering.

Figure 4.10: Cycle can prevent chatter

EQ-Chatter-Test:

Given an equivalency node EQ, a corresponding chattering region predicate C_{eq} and a stack of previously visited equivalency nodes *call-stack*,

1. If $status(EQ) \neq$ `:undetermined` then return $status(EQ)$.
2. If $EQ \in$ *call-stack*, then return `:cycle`.
3. Generate a qualitative state S' from S as follows:
   For all v such that v is a variable in the model:
   - if *equivalency-class*(v) has the status `:nochatter` then $Qval(v, S') := Qval(v, S)$.
   - if the integral of v is not explicitly represented within the model then $Qmag(v, S') := Qmag(v, S)$.[a]

   The rest of the values are left undefined.
4. Push EQ onto the *call-stack*,
   call *Check-dependencies*$(D, S', call - stack)$ and return the value returned by the call to *Check-dependencies*. $D = D_1, \ldots, D_n$ is a list of the dependencies in C_{eq}.

[a] This condition assumes that if a variable's derivative is not explicitly represented within the model, then it will not chatter around zero. This assumption is discussed in more detail in section 4.4.6 on landmark chatter.

Table 4.3: The EQ-Chatter-test Algorithm.

A brief description of each algorithm is provided below, while tables 4.3, 4.4, 4.5 and 4.6 contain a more detailed description.

EQ-Chatter-Test *EQ-Chatter-Test* (table 4.3) is the root algorithm that determines whether the variables within an equivalency class EQ are free to chatter. If the status of the equivalency class is still `:undetermined`, the algorithm initializes the partial state S' and calls *Check-dependencies* with the dependencies contained within EQ's chattering region predicate. This call returns the value provided by *Check-dependencies*. If the algorithm has been called recursively and EQ is already in the calling stack, then the algorithm returns `:cycle`.

Check-dependencies *Check-dependencies* (table 4.4) receives a list of dependencies and a partial state S'. It performs a recursive depth–first search of the dependencies, attempting to find a refinement of S' that satisfies at least one condition

Check-dependencies:

Given a set of dependencies $\{D_1, \ldots, D_n\}$, a partial state S' and a stack of previously visited equivalency nodes *call-stack*,

1. If there are no dependencies left, then call *Check-state-completions*(S') and return the value returned.
2. Else, select a dependency D_i and for each condition Sc_{i_j} within D_i,
   (a) Call *Check-assertions*(A, S',*call-stack*) where $A = \{A_1, \ldots, A_n\}$ is the set of assertions within C_{i_j}. Set *Result* to the value returned.
   (b) If *Result* = `:cycle` then set *Cycle-Found* to be `TRUE`.
   (c) If *Result* = `:chatter` then,
      - Create a copy of S' called S'' and update S'' with the assertions in C_{i_j}.
      - Call *Check-dependencies*($\{D_2, \ldots, D_n\}, S''$,*call-stack*). If this returns
        `:chatter` then return `:chatter`,
        `:cycle` then set *Cycle-Found* to be `TRUE` and continue.
        `:nochatter` then continue.
3. If a value has not been returned yet, then if *Cycle-Found* is `TRUE` then return `:cycle`. Otherwise, return `:nochatter`.

Table 4.4: The Check-dependencies Algorithm.

within each dependency. It selects a dependency from the list and iterates through the conditions. *Check-assertions* is called with each condition to see if S' can be appropriately refined. If it can, then the algorithm is called recursively with the refinement of S' and the remainder of the dependecies.

If *Check-dependencies* is called with a null dependency list, then the partial state S' has satisfied all of the dependencies. *Check-state-completions* is called with S' to see if there is a consistent completion and to determine if the inner bound annotations are satisfied. The value returned by *Check-dependencies* is determined by the following rule:

- If a recursive call to *Check-dependencies* or *Check-state-completions* returns `:chatter`, then return this value.

- Else, if a call to *Check-dependencies*, *Check-state-completions*, or

Check-assertions:

Given a set of assertions $\{A_1, \ldots, A_n\}$, a partial state S' and the *call-stack*,

1. For each assertion A_i in $\{A_1, \ldots, A_n\}$,
   (a) If A_i is a `:same-qdir` or `:qdir` assertion, then test to see if the values can be asserted in S'. If they cannot be asserted, return `:nochatter`.
   (b) If A_i is a `:uncon-qmag` or `:uncon-qdir` assertion, then test to see if the specified inner bound can be asserted. An inner bound cannot be asserted if S' already has a more–refined description for the variable in A_i.
   (c) If A_i is a `:change-qdir` or `:chatter` assertion referring to equivalency class EQ_i, then
      - If $EQ_i \in$ *call-stack*, return `:cycle`.
      - Check to see if the values can be asserted in S'. If they cannot, return `:nochatter`.
      - Call *EQ-Chatter-Test*(EQ_i,C_{eq_i},*call-stack*). If it returns either `:nochatter` or `:cycle`, then return this value. Otherwise continue.
2. If a value has not been returned, return `:chatter`.

Table 4.5: The Check-assertions Algorithm.

Check-assertions returns `:cycle`, then return this value.

- Otherwise, return `:nochatter`.

Check-assertions *Check-assertions* (table 4.5) receives a conjunctive list of assertions, a partial state S', and the call stack. It tests each assertion to determine whether it is consistent with S'. If an assertion requires an equivalency class EQ to chatter and EQ is in the call stack, *Check-assertions* returns `:cycle` . If EQ is not in the call stack, it calls *EQ-Chatter-Test* recursively to see if EQ can chatter.

If all of the assertions can be satisfied, then `:chatter` is returned. However, if one of the assertions cannot be satisfied, then return `:nochatter`.

Check-state-completions *Check-state-completions* (table 4.6) determines whether there is a consistent completion to the partial state S'. Such a result

Check-state-completions:

Given an annotated partial state S',

1. If there are no inner bounds, attempt to find one consistent completion S'_c such that

$$Qdir(v, S) = Qdir(v, S'_c) \vee Status(EQ_v) = \texttt{:chatter}.$$

for all v where v is a variable within the model whose value is unspecified in S'. If a completion is found, return `:chatter`. If a completion satisfying the first condition is found, but a recursive call to *EQ-Chatter-Test* returns the value of `:cycle` when attempting to evaluate the status of EQ_v, then return `:cycle`. Otherwise, return `:nochatter`.

2. If there are inner bounds, then a completion satisfying the condition above must be found for each qualitative value in the inner bounds. This ensures that S' is unconstrained in the region specified by the inner bounds.

Table 4.6: The Check-state-completions Algorithm.

ensures that the values asserted within S' are consistent with all of the constraints in the model. The QSIM state–completion algorithm has been refined to generate completions in an incremental manner. The algorithm "prefers" completions that are closest to the current qualitative state. If a completion requires a variable v to chatter (*i.e.* its qdir is different than in the current state) and v is not specified within S', then *EQ-Chatter-Test* must be called recursively to ensure that the variable can chatter. If the only consistent completions result in *EQ-Chatter-Test* returning `:cycle`, then *Check-state-completions* returns `:cycle`.

Check-state-completions must also ensure that the inner bound annotations are consistent with S'. For an inner bound to be consistent, S' must be unconstrained in the region specified by the inner bound. Thus, a completion must be found for each qualitative value contained within the inner bounds. Once again, if the completion requires a variable to chatter *EQ-Chatter-Test* must be called recursively.

4.4.5 W-tube example

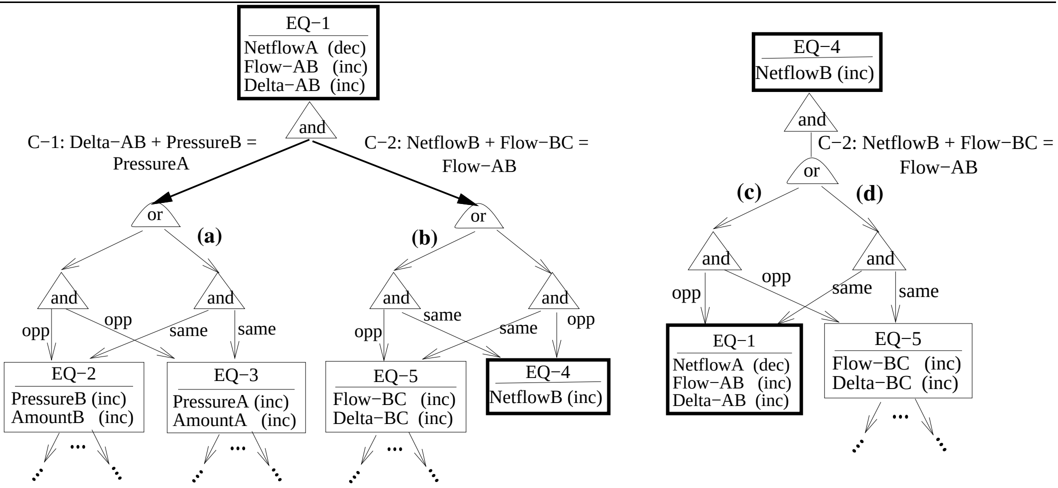
Figure 4.11 describes the chatter–detection algorithm for a pair of equivalency classes in the W-tube model. $C_{eq_i}$is represented as an AND–OR graph, providing an effective graphical representation for demonstrating the sequence of steps in the algorithm. Without using HOD information, evaluation of the entire dependency graph yields a total of seven chattering variables. This is consistent with the results of a standard QSIM simulation without any form of chatter elimination. The variables that do not chatter (the variables within the chatter equivalency classes for `Amount-A`, `Amount-B` and `Amount-C`) are constrained from chattering by a sequence of cycles in the dependency graph. For these variables to chatter, their derivatives must chatter around zero. The cycles are such that `NetflowA` depends upon `NetflowB` to chatter first, `NetflowB` requires `NetflowC` to chatter first and `NetflowC` in turn requires `NetflowA` to chatter first. Due to this cycle, none of the variables can chatter. This is an example of the situation where constraint propagation is too weak an inference method to determine that a variable will not chatter.

4.4.6 Landmark chatter

4.4.6.1 Chatter around zero

The most common and problematic form of landmark chatter is chatter around zero. One of the reasons that it is problematic is that chatter around zero involves the magnitude of a variable being unconstrained as opposed to just the direction of change. Zero, however, is often a relevant landmark that separates a chattering region from a non–chattering region. Thus, it is important to selectively identify and eliminate chatter around zero only when it occurs.

Dynamic chatter abstraction handles chatter around zero without any additional processing. The algorithm described above does not assume that the derivative constraint prevents the integral variable from chattering. Instead, information about the constraining power of the derivative constraint is represented via assertions in C_{eq}. For example, if a variable's derivative is explicitly represented in the model, then both the `qmag` and the `qdir` of the derivative variable must be unconstrained and the `qdir` of the derivative variable must head toward zero. If both a variable and its derivative are identified as chattering, then the derivative variable will exhibit chatter around zero and its qualitative magnitude is also abstracted.



Chatter equivalency class EQ_1 is evaluated as follows for the time–interval state S_1 following the initial state. To simplify the structure of the display, nodes EQ_4 and EQ_1 are displayed in two places in the above graph. (The order of the traversal of the AND–OR subgraphs has been slightly modified for the sake of this presentation.)

Evaluating EQ_1

Step-1 Instantiate a partial state description S_p with the qualitative value for `InflowA` since it is constant and thus cannot chatter. All of the other variables can potentially chatter. Push EQ_1 onto the call stack CS.

Step-2 To satisfy constraint C_1 traverse link (a). This link requires the variables in EQ_2 and EQ_3 to remain the same. Add this information to S_p.

Step-3 To satisfy constraint C_2, traverse link (b). This link requires the variables in EQ_4 to `change` their direction of change, so the variables in EQ_4 must be free to chatter. Since EQ_4 is classified as `:undetermined`, call the algorithm recursively to determine its status.

Evaluating EQ_4

Step-4 Traverse link (c) in the above graph. To satisfy this link, both EQ_1 and EQ_5 must be free to chatter. A cycle, however, occurs in the calling sequence since $EQ_1 \in CS$. Therefore, the algorithm backtracks.

Step-5 Traverse link (d). Since this link is satisfied by the current state, classify EQ_4 as chattering and return (*i.e.* `NetflowB` will chatter.)

Step-6 EQ_4 is therefore free to chatter. Augment S_p with the qualitative value information to satisfy link (b).

Step-7 The QSIM state completion algorithm is called to ensure that there exists a consistent completion of S_c. Variables that have not been identified as chattering must retain their current qualitative values within this completion. A state is identified and EQ_1 is classified as chattering.

Figure 4.11: Chatter detection for the W-tube.

4.4.6.2 Non-zero landmark chatter

Conceptually, there is no reason chatter around a non–zero landmark could not be detected in a manner similar to the detection of normal chatter. Standard chatter occurs when a variable's derivative is unconstrained around zero. To comprehensively handle chatter around a non–zero landmark, our analysis would need to be generalized and extended to reason about the manner in which the qualitative magnitude of a variable can be constrained. Constraints, however, can restrict the magnitude of a variable in more–complex fashions because of corresponding values. This causes an increase in both the algorithmic complexity and the computational complexity of the solution. Moreover, chatter around a non–zero landmark is not a major problem since most variables that chatter often only contain the landmarks `minf`, `0` and `inf`.

Instead, we use a more straight–forward approach in which chatter around non–zero landmarks is detected by a conservative algorithm after the set of chattering variables is identified. The algorithm is augmented by the ability to identify a landmark as a *chatter landmark* within the QDE, thus causing the algorithm to identify it as a chatter landmark when it neighbors a chattering region.

Landmark chatter algorithm A variable chatters because its derivative is unconstrained. If a variable is chattering and the chattering region is bordered by a non-zero landmark, then the variable will chatter around this landmark unless

(1) attaining the landmark causes the qdir of the variable to be constrained, or

(2) the model prevents the variable from reaching the landmark while remaining within the chattering region.

Condition (1) can only occur if there is a constraint that in essence "propagates" this new qmag in such a manner that the qdir is now constrained. Note, however, that for most of the constraints, the qmag of a variable does not affect the qdir. Since it is the qdir that must be constrained if the variable is to stop chattering, in most cases condition (1) will not prevent a variable from chattering around a landmark. Table 4.7 describes when this condition is satisfied for each constraint.

Condition (2), on the other hand, can prevent a variable from chattering because corresponding values may prevent the variable from reaching the landmark. For example, suppose

- `(ADD A B C)` is a constraint and the quantity spaces and values for the variables are

Var	*Qspace*	*Qvalue*
`A`	`(minf 0 A* inf)`	`((0 A*) inc)`
`B`	`(minf 0 B* inf)`	`((0 B*) dec)`
`C`	`(minf 0 C* inf)`	`((0 C*) inc)`

- the corresponding value `(A* B* C*)` exists, and
- `A` and `B` are prevented from chattering by other constraints while `C` is not prevented from chattering by other constraints.

In this example, `C` is free to chatter, but not around `C*`, because the corresponding value prevents `C` from crossing `C*` unless `A` or `B` change qualitative magnitude as well. This example demonstrates how a constraint can prevent a variable from chattering around a landmark.

The following algorithm identifies non-zero chatter landmarks by gradually extending the region identified as chattering around neighboring landmarks that are identified as chatter landmarks.

Step 1: Perform a static analysis of the QDE to identify non-chatter landmarks. A landmark is a non-chatter landmark if it is a threshold landmark for an S constraint or a U constraint or if it is a non-terminal landmark that is unreachable. This analysis is repeated each time a new QDE is used and the results are stored on the QDE.

Step 2: Dynamically identify non-chatter landmark equivalency classes. For each chatter equivalence class, analyze the corresponding values (cvalues) on each constraint relating variables in the equivalency class. Two landmarks are chatter equivalent if their respective variables are the only two non-constant variables in a constraint and the landmarks are contained in the same corresponding value. Label each class `:undetermined`.

Step 3: Label a non-chatter landmark equivalency class `:nochatter` if the class contains a non-chatter landmark or if it contains 0 and the variable does not chatter around zero.

Constraint	Affect qmag change has on qdir
M, $M^{+/-}$, $SSUM$, *SUM-ZERO*, ADD	Change in the qmag does affect the qdir of a variable.
$MULT$	Sign changes in a variable can affect the constraints on the qdir of a variable within the model. Note, however, that this only occurs when a variable crosses zero. Chatter around zero is a special case that is handled separately, so we do not need to worry about it here.
D/DT	Once again, if a variable crosses zero, this affects the qdir of the integral variable. Currently, we are only concerned with non-zero landmarks.
$S+/-$	A change in qmag can affect the qdir of the other variable. If the variable on the X axis crosses either of the threshold landmarks, then the variable on the Y axis is constrained to be steady. This means that if two variables are related by an S+/- constraint, both variables are chattering, and X crosses a threshold, then the Y variable will stop chattering. While it is possible for X to chatter around this landmark, such a scenario only results from a bug in the model since the threshold landmark is supposed to be significant. Therefore, if a landmark LM is a threshold landmark in an S+/- constraint, chatter around this landmark will not be asserted.
$U+/-$	A change in qmag can affect the qdir of the other variable. When the maximum/minimum point of the U is reached, the relationships between the qdirs of the variables change and thus the sign of the derivative of one of the variables will change. This can affect whether or not other variables will chatter, which can in turn affect whether or not the variables related by a U+ constraint will chatter. It may be possible to determine if the system will chatter around the landmark by determining which variables chatter once the relation changes. Similar to S+/-, however, this threshold was identified as "significant" by the modeler. Thus, at this point we will identify the threshold landmarks as non-chatter landmarks until an example is produced for which this is unsatisfactory.

Table 4.7: How a change in magnitude can affect the direction of change.

Step 4: For each landmark contained within an `:undetermined` equivalency class, attempt to find a consistent state S' in which the qualitative magnitude for the corresponding variable is on the "other side" of the landmark. S' should satisfy the following conditions:

- For all v where v is a non-chattering variable, $Qval(v, S') = Qval(v, S)$.
- For all v where v is a chattering variable that does not chatter around zero, $Qmag(v, S') = Qmag(v, S)_{(-\infty\ 0\ \infty)}$ where $Qmag(v, S)_{(-\infty\ 0\ \infty)}$ is the qmag of v in S projected onto the quantity space $(-\infty\ 0\ \infty)$.

If a completion cannot be found, then mark the landmark chatter equivalency class `:nochatter`.

Step 5: Mark all `:undetermined` equivalency classes `:chatter`.

Step 6: For each chattering variable, recursively extend the chattering interval by testing the bounding landmarks to determine if they are chatter landmarks. Given a landmark lm, a direction `:up` or `:down`, and the labeled landmark equivalency classes, use the following algorithm to identify an upper and a lower bound for the chattering interval.

- If lm is a terminating landmark or 0, return lm.
- If lm is unreachable, return lm.
- If lm is in an equivalency class labeled `:nochatter`, return lm.
- Otherwise, call the algorithm recursively with the next landmark in the quantity space in the direction identified by the input argument.

Open Issues As mentioned above, the algorithm described here is not intended to be comprehensive. Instead, it provides an efficient means to eliminate most instances of non-zero landmark chatter. Furthermore, the modeler can identify a landmark as either a non–chatter landmark or a chatter landmark in the QDE. The label for the landmark equivalency class containing a landmark identified in the QDE is labeled as specified by the modeler.

Some of the existing open issues with the algorithm include:

- The assumption that the threshold landmarks in the U and S constraints are non-chatter landmarks may be too conservative.

- The algorithm treats zero as a special landmark. It is possible that zero will not be a special landmark and that chatter around it may need to be classified and detected in a manner similar to chatter around other landmarks.
- Currently, the algorithm identifies "non-chatter landmark equivalency classes" in a very conservative manner. In particular, if three variable are related via an ADD constraint and all are non-constant, then no relation is asserted even though one may exist.

4.4.7 Abstract state creation and successor generation

Once the set of chattering variables is identified, the algorithm creates an abstract state with an abstracted qdir of (`inc std dec`) for each of the chattering variables. We have extended the QSIM state–successor algorithm to handle such abstract states.

If the qualitative value of a variable over a time–interval is $\langle (l_j, l_k)\ (inc\ std\ dec)\rangle$, where $k > j$, then at the ensuing time–point state the following values are consistent with continuity:

$$\langle (l_j, l_k)(inc\ std\ dec)\rangle$$
$$\langle l_j, (std\ dec)\rangle$$
$$\langle l_k(inc\ std)\rangle$$

For each time–point state, the successor table is extended in the same manner as in chatter–box abstraction (see table 4.3.5). Since a time–point state can also have an abstracted qdir, each time–point must be tested to determine when a variable stops chattering. A variable stops chattering when a related variable changes and begins to constrain the chattering variable. To detect chatter termination, the state–completion algorithm attempts to find a consistent state for each possible qdir. If a qdir does not participate in a consistent completion, then it is filtered out of the list of possible qdirs.

Lemma 4.4 (Qualitative successor extension) *The extensions to the QSIM qualitative value successor tables for abstract qualitative states defined above and in table 4.3.5 from section 4.3.5 preserve the QSIM guarantee that all possible successor values that are consistent with continuity are generated.*

Proof: Both extensions define the successor values for an abstract qualitative value qv_a as the union of successor values for each non–abstracted value in qv_a adjusted to eliminate distinctions that have been eliminated in qv_a. Thus, each value in a successor to the non–abstracted values in qv_a is also in the successors of qv_a. □

4.4.8 Results

Dynamic chatter abstraction has been evaluated both theoretically and empirically. This section presents theorems that demonstrate its effectiveness at eliminating chatter. Dynamic chatter abstraction is evaluated both with respect to the qualitative behaviors generated as well as the set of real–valued trajectories described by the behavioral description.

In this section, we summarize the discussion and omit the lengthier proofs. Please refer to appendix B for a more complete presentation and detailed proofs of all theorems.

4.4.8.1 C_{eq}: Sound and complete

Lemma 4.5 *For a given equivalency class EQ and time–interval state S describing the interval* (t_j, t_k)*, if the qdir of v is unconstrained except with respect to other variables in EQ for all v such that* $v \in EQ$*, then the variables in EQ will chatter following time point state* t_k*.*

The proof appears in appendix B.

Lemma 4.6 *The conditions specified in* C_{eq} *for each QSIM constraint type T define necessary and sufficient conditions for the variables in EQ to be unconstrained by a constraint of type T.*

The proof appears in appendix B.

Lemma 4.7 *If an abstract state S satisfies* C_{eq}*, then the derivatives of the variables in EQ are unconstrained in S except with respect to other variables in EQ. (*i.e. *A variable* v_1 *may be related to a variable* v_2 *by an* M^+ *constraint. Thus,* $\dot{v}_1$ *is constrained with respect to* $\dot{v}_2$*; however, they are contained in the same equivalency class and thus the exception applies.)*

The proof appears in appendix B.

Theorem 4.6 (**C_{eq} soundness**) *For a given abstract time–interval state S describing the interval (t_j, t_k) and an equivalency class EQ, if S satisfies C_{eq} then the variables in EQ will chatter following time point t_k.*

Proof: Since S satisfies C_{eq}, the variables in EQ are unconstrained except with respect to each other (lemma 4.5). Therefore, by lemma 4.7 the variables will chatter following time point t_k. □

Theorem 4.7 (**C_{eq} completeness**) *Assuming that the chatter equivalency class partitioning is complete, the predicate C_{eq} defines necessary conditions on an abstract qualitative state for the variables in EQ to chatter.*

The proof appears in appendix B. While this theorem requires the partitioning algorithm to be complete, it is not the case that C_{eq} fails to detect chatter in all instances where C_{eq} is incomplete. On the contrary, since the dependencies record the conditions under which a variable is unconstrained, it may be possible to use the information within the predicate to detect implicit chatter equivalency relationships between variables that arises because of the interaction of multiple constraints. Moreover, in all of the models tested, dynamic chatter abstraction detected all instances of chatter encountered.

4.4.8.2 *Chatter-test*: Sound and complete

Theorem 4.8 (*Chatter-test* **soundness**) *For a given qualitative state S and an equivalency class EQ, if* Chatter-test*(S, EQ) identifies the variables in EQ as chattering, then there exists an abstract state S' such that S' both satisfies C_{eq} and is chatter–reachable from S, assuming that the paths supporting each change in S' from S can be combined into a consistent sequence of states.*

The proof appears in appendix B.

Theorem 4.9 (*Chatter-test* **completeness**) *For a given qualitative state S and an equivalency class EQ if there exists a state S' satisfying C_{eq} that is chatter–reachable from S, then* Chatter-test*(S, EQ) identifies the variables in EQ as chattering.*

The proof appears in appendix B.

4.4.8.3 Real valued trajectories

Theorem 4.10 (QSIM soundness retained) *The set of real–valued trajectories described by an abstracted behavior tree that is generated using dynamic chatter abstraction includes all real–valued trajectories described by that corresponding an unabstracted behavior tree that exhibits chatter.*

Proof: Dynamic chatter abstraction performs a strict abstraction operation. Thus, dynamic chatter abstraction replaces a time–interval state S with a state S_a such that $S \subseteq S_a$. Thus, the set of precise numerical values consistent with S is a subset of the set consistent with S_a. Furthermore, by the qualitative successor–extension lemma (lemma 4.4), the successors of S_a contain all possible successor values for the precise numerical values in S_a that are consistent with continuity. Therefore, since each state and each transition are a superset of the values described by a non–abstracted tree, the set of real–valued trajectories described by an abstracted tree is a superset of those described by the corresponding non–abstracted tree exhibiting chatter. □

Note that the abstraction operation performed by dynamic chatter abstraction does introduce additional real–valued trajectories that do not appear in the non–abstracted behavior. There are two potential sources for these trajectories:

Abstraction operation – As with chatter–box abstraction, the representation used to describe an abstract state is not sufficiently refined to describe the correlations between chattering variables in the same equivalency class. As discussed in section 4.3.6, this loss of precision is simply a cost of abstraction; it does not, in general, affect the usefulness of the results.

Abstract state successor computation – Dynamic chatter abstraction purposely does not explore all paths through the potentially chattering region. Rather, it determines which variables will chatter, generates an abstract state, and then uses the extension to the QSIM successor tables to compute the successors to this abstract state. The algorithm uses the model constraints to ensure that each successor state is consistent. However, there is no guarantee that there exists a path through the chattering region that ends in each successor of the abstract state. Thus, it is conceptually possible that a successor of an abstract state would not have a corresponding state within the unabstracted tree.

In practice, we have not encountered this situation. Chatter–box abstraction does not suffer from this limitation since it computes all paths through the chattering region. Thus, we have tested this condition by comparing the results from dynamic chatter and chatter–box abstraction. In all of the cases examined, the behavioral descriptions for these two approaches were identical.

The reason that this condition has not proved to be a problem is because of the unconstrained nature of chatter. Since chattering variables are loosely constrained within the chattering region, it is very likely that a path exists through the chattering region for each potential exit state.

4.4.9 Complexity

Dynamic chatter abstraction uses a backtracking algorithm that in effect solves a constraint satisfaction problem. Thus, the worst–case complexity of the algorithm is exponential in the number of potentially chattering equivalency classes; however, in empirical evaluations of the algorithm we have not encountered an exponential time factor.

The algorithm complexity is determined by the manner in which the variables are constrained in the model and the number of chattering variables. In general, the complexity is driven by two factors:

(1) the ratio of the number of potentially chattering equivalency classes to the number of equivalency classes that exhibit chatter, and

(2) the density of the constraints that may or may not prevent an equivalency class from chattering (e.g. `ADD` and `MULT`).

The ratio of potentially chattering classes to chattering classes is important because of the recursive nature of the algorithm and frequency of cycles within the execution sequence. If a cycle occurs in a recursive calling sequence, then only the class on which the algorithm was initially called can be labeled `:nochatter` since the other classes may not encounter the cycle when called as the root class.[9] If a class is identified as chattering, however, this is not affected by the occurrence of a cycle. Furthermore, once a class is labeled the algorithm does not need to be called recursively on it. Thus, if the number of potentially chattering classes is high while the number of chattering classes is low, then the algorithm may have to backtrack a large number of times. Even when this occurs, however, the algorithm is still able to perform well since conflicts are often quickly detected.

The density of constraints that lead to assertions within the chattering region predicates also affect the complexity of the algorithm especially when there are a large number of potentially chattering classes that do not exhibit chatter. When evaluating a chattering region predicate, all of the different combinations of the different conditions from each dependency must be considered. For example, if a given equivalency class EQ participates in 4 `ADD` constraints, then for each `ADD` constraint the corresponding dependency will have two disjunctive conditions that need to be tested. All of the different combinations results in 2^4 or 16 different possibilities. Of course, having a single equivalency class involved in four `ADD` constraints is unlikely. If the variables in EQ end up chattering then all 16 options do not need to be tested. However, if the class does not chatter, then they will all be tested. This is why the ratio is important in this example.

The analysis above provides a brief evaluation of some of the factors that affect the complexity of the algorithm. As larger problems are encountered a more thorough analysis may be required if the algorithm begins to slow down. At this point, however, the benefits of performing a more detailed complexity analysis in conjunction with algorithmic optimizations is limited due to the performance of the algorithm on the models tested. The performance numbers are discussed in detail in section 4.5.

4.5 Empirical Evaluation

Both the chatter–box and dynamic chatter algorithms have been empirically evaluated using a corpus of over 20 models obtained from various researchers within the field of qualitative reasoning. Many of these models have been used to evaluate other techniques presented within this dissertation as well. Table 1.1 in section 2 provides a brief description of some of the models tested.

The empirical evaluation has been used to establish two results:

[9] This is one area where the algorithm could be significantly improved by recording information about cycles and leverage the work that has been done previously.

(1) to reinforce theoretical results and demonstrate that both algorithms eliminate chatter without over–abstracting, and

(2) to compare and contrast execution times of alternative solutions for eliminating chatter.

In all models within table 1.1, both abstraction techniques eliminated all instances of chatter without over–abstracting. This result was established by performing a standard simulation and querying the results to determine which variables chattered and when they chattered. Moreover, as expected, the resulting behavioral description was far more concise in all of the examples tested, since chatter causes intractable branching. Table 4.8 gives a detailed analysis of the results for several of these models, comparing execution time for chatter–box and dynamic chatter abstraction. Note that dynamic chatter performed significantly better.

We tested the asymptotic behavior of three chatter elimination techniques (ignore-qdirs, chatter–box abstraction, and dynamic chatter abstraction) using an extendible cascading tank model. In this model, the $netflow$ variables for each tank exhibit chatter. The model is extendible since the number of tanks can be increased. The results from these tests are listed in table 4.9 and plotted in figure 4.12.[10] Note that execution time increases exponentially for both chatter–box abstraction and ignore qdirs. For ignore qdirs this is caused by the *inconsistent-ignore-filter* (see section 4.6). For chatter–box abstraction, we were unable to complete the simulation for models with six or more tanks because of resource limitations.

Figure 4.13 shows only the dynamic chatter abstraction simulation results so that the reader can get a better sense of the asymptotic behavior of this abstraction technique. In this example, execution time appears to be polynomial in the number of tanks. Of course, this is not a general result, although in our experience the execution time of the dynamic chatter abstraction algorithm is extremely low. The results in table 4.8 reinforce this conclusion. The numbers in this table, however, measure the total execution time for the model and therefore do not isolate the cost per state. The execution time increases due to event branching.

4.6 Related Work

Both the ignore qdirs and higher–order derivative techniques eliminate chatter within certain models (see section 4.2). Neither technique, however, provides a general so-

[10] In the plot, the dynamic chatter abstraction line cannot be seen because it is so close to the x-axis.

Model	**Vars**	**Chat Vars**	**Number of Behaviors**			**Simulation time (sec)**	
			No chatter abstraction		**Abstract chatter**		
			Env	**Beh tree** (no lms)	**Beh tree** (w/ lms)	**Dyn Chat**	**Chat Box**
W–Tube	16	1	3	> 1845	1	0.9	3.4
Glucose-insulin Interaction	11	2	155	> 2807	41	14	52
Van der Pol Equation[a]	10	4	43	> 1788	12[b]	2.3	15
Controlled Hot/Cold tank[a]	14	5	24	> 601[c]	14	5.0	48.2
Turgor Stomates	19	7	509	> 2598	1	2.4	20.5
Cooling Plant[a]	15	10	659	> 4095	1	7.7	418
Heart Model	42	28	689	> 2784	200	100	[c]

- Each model was tested both with and without chatter abstraction. Both an envisionment and a behavior tree simulation (without landmark introduction) were performed without chatter abstraction using a state–limit of 5000. The number of behaviors are listed above. Note that while an envisionment often yields fewer behaviors, the total number of qualitatively distinct behaviors is the same, as there are an infinite number of paths within the envisionment graph.
- Both types of abstraction generated the same number of behaviors; dynamic chatter abstraction, however, was significantly faster than chatter–box abstraction.
- HOD constraints were used whenever applicable. Ignore qdirs does not work on models exhibiting chatter around zero. On other models, ignore qdirs can be used, however, it requires a significant amount of work by the modeler to identify which variables are chattering.

[a] Exhibits chatter around zero

[b] Oscillatory behavior results in infinite behavior tree. Simulation terminated once the structure within the tree could be determined.

[c] Could not be simulated to completion due to resource limitations.

Table 4.8: Evaluation of Dynamic Chatter and Chatter–Box Abstraction

# of Tanks	# of Chattering Vars	Total # of Vars	Chatter Box Env Size	Execution Time (sec)		
				Chatter Box	Ignore Qdirs	Dynamic Chatter
2 Cascade	1	9	9	0.369	0.107	0.067
3 Cascade	2	13	46	1.232	0.252	0.543
4 Cascade	3	17	249	6.586	0.805	0.523
5 Cascade	4	21	1312	39.67	2.609	0.98
6 Cascade	5	25	6759	287.6	8.768	1.552
7 Cascade	6	29			29.52	2.377
8 Cascade	7	33			102.9	3.443
9 Cascade	8	37			363.5	4.966
10 Cascade	9	41			1284	6.905

- Tests were run on a Sparc 10 using Lucid Common Lisp. Execution time measures the processing time for a single state.
- Blank entries correspond to simulations that could not be completed due to resource limitations.

Table 4.9: Comparison of chatter abstraction techniques for the N–tank cascade

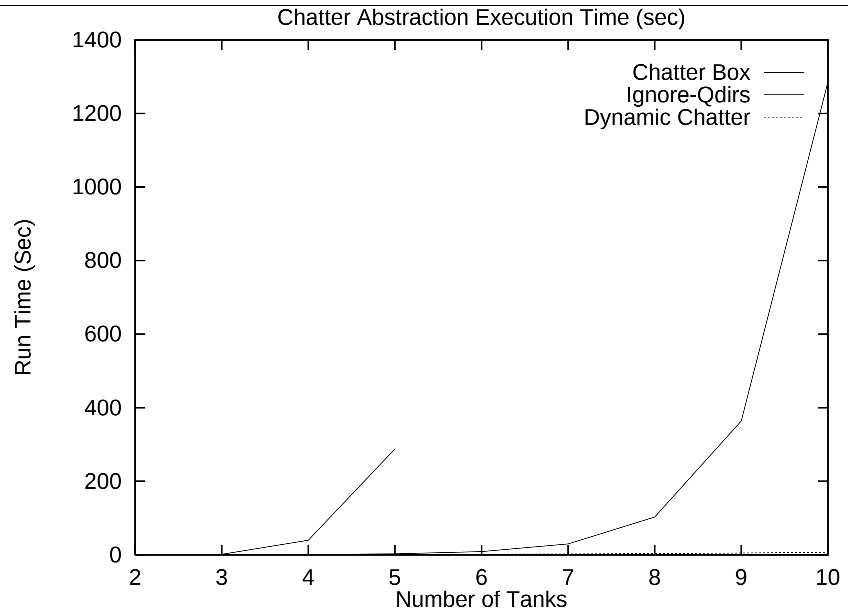


Figure 4.12: Comparison of chatter abstraction techniques for the N–tank cascade

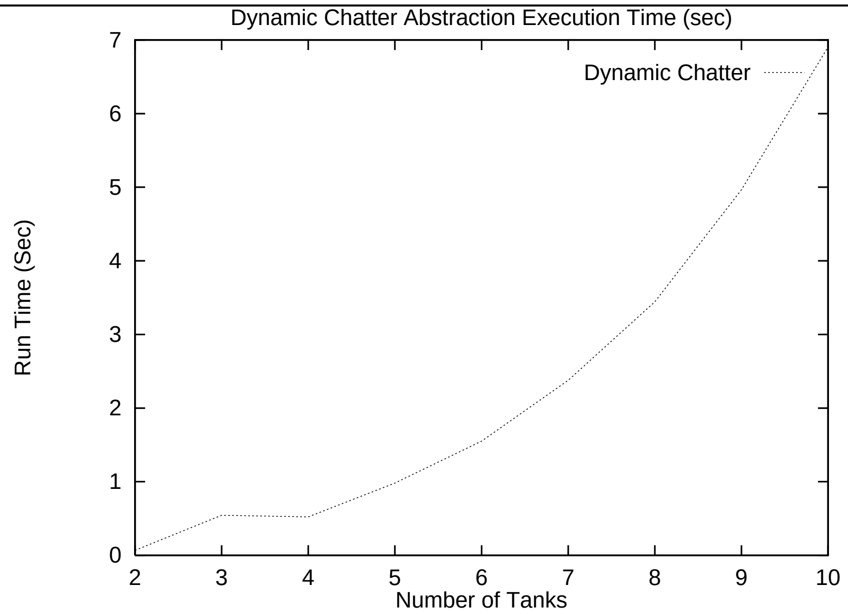


Figure 4.13: Dynamic chatter abstraction results for the N–tank cascade

lution that eliminates chatter in all cases.

HOD Constraints – The abstraction techniques proposed in this thesis are not intended to replace the use of higher–order derivatives to eliminate chatter. Instead, they complement HOD constraints, allowing them to be used when applicable. When available, HOD constraints can eliminate spurious behaviors and refine the behavioral description. However, in many cases the HOD technique is unable to eliminate all of the chatter in the model.

Ignore-qdirs – Ignore qdirs is similar to the abstraction techniques presented here in that it eliminates distinctions in a variable's direction of change. It differs from our abstraction techniques in several ways:

- Ignore-qdirs applies the abstraction operation throughout the simulation. Often, however, derivative information can be useful in constraining the simulation within certain regions of the state space. The abstraction techniques described here are selectively applied only to the regions in which a variable's derivative is unconstrained.
- Ignore qdirs requires the operator to identify the set of chattering variables prior to the simulation. Often this can involve a cumbersome, iterative process involving multiple simulations to identify the set of chattering

variables. Then as additional information is added to the model, these assertions must be withdrawn if the derivatives become constrained. Ignoring the direction of change in a variable that does not exhibit chatter may introduce additional spurious behaviors into the simulation.

- Ignore qdirs cannot eliminate landmark chatter – more specifically, chatter around zero. This problem becomes more common as larger models are simulated.

In addition, ignore-qdirs can be expensive in certain models. When ignoring a variable's direction of change, the *inconsistent-ignore* filter is used to ensure that there is at least one completion that is consistent with continuity for the ignored derivatives. Ignore-qdirs calls the QSIM state–completion algorithm to identify these completions. This process can become expensive as the number of chattering variables increases. Section 4.5 demonstrates that ignore-qdirs can require an exponential amount of time.

The problem of intractable branching due to chatter was also addressed by DeCoste (DeCoste, 1994) in the context of Qualitative Process Theory models (Forbus, 1984). DeCoste suggests that distinctions in the direction of change can simply be ignored when a unique value cannot be identified. Conceptually, this approach is similar to the one taken here. However, DeCoste does not discuss how to determine when a unique value can or cannot be determined. Furthermore, he suggests that it is a simple operation. While for simpler models, this may be the case, more–complicated models often do not lend themselves to straight propagation to determine if there is a unique value for a variable's direction of change (see figure 4.10). Finally, DeCoste does not consider the problem of chatter around zero or other landmarks.

4.7 Discussion

The abstraction algorithms proposed here provide distinct benefits. Dynamic chatter abstraction provides a scalable solution that to efficiently solves the problem of chatter. However, it is less modular than chatter–box abstraction. Since the algorithm incorporates information about how the simulation algorithm processes the constraints in the model, extensions to the simulation algorithm may require modifications to the dynamic chatter abstraction algorithm. Furthermore, there is no guarantee that an arbitrary extension will lend itself to the type of reasoning that is performed by the dynamic chatter abstraction algorithm. On the other hand, chatter–box abstraction uses the basic QSIM simulation algorithm as its main inference engine. As a result, extensions to the simulation algorithm can be incorporated without modifications to the chatter abstraction algorithm.

Currently, for example, dynamic chatter abstraction cannot reason about trajectory constraints specified by TeQSIM. Thus, if a trajectory constraint restricts a variable from chattering, dynamic chatter abstraction may still abstract the behavior of the variable, which in turn may affect other variables in the model. For chatter–box abstraction, however, trajectory constraints are applied as in a non–abstracted simulation thus supporting the integration of these two techniques.

While at first glance the execution time for chatter–box abstraction may cause concern about the utility of this technique, chatter–box abstraction provides a very effective and efficient technique for eliminating chatter for the majority of the models currently being developed, In general, the number of chattering variables in any single region of the state space is small, and thus the exponential nature of the algorithm does not play a role. In cases where a larger number of variables chatter, the modeler can often reduce the complexity of the focused envisionment by using ignore-qdirs for a subset of the variables. Often, for larger models, a number of variables chatter throughout the simulation. Thus, ignore qdirs does not cause over–abstraction. The component–based simulation technique provided by DecSIM may provide a solution for larger models, since the number of chattering variables within any one component would be reduced.

4.8 Future Work

Chatter–box and dynamic chatter abstraction are two comprehensive and practical solutions to chatter. Both, of course, could be refined and extended to improve performance or extend the range of applicability. Some of the potential extensions are discussed below:

1. The main advantage of chatter–box abstraction is its ability to be seamlessly integrated with extensions to the QSIM baseline. However, due to complexity problems it does not scale as the size of the model increases. Various extensions could reduce the impact of the exponential branching in the focused envisionments:

    - A focused envisionment identifies chattering variables from a set of potentially chattering variables. Once it has been determined that all of the variables in the set of potentially chattering variables exhibit chatter, then it is not necessary to explore the rest of the chattering region of the state space.[11] Instead, a technique similar to dynamic chatter abstraction could be used to generate an abstract state and compute its successors.

- To avoid repeatedly exploring the same region of the state space, the algorithm could maintain and use an index of the regions already explored. If a system re-enters an indexed region, then the results from the previous focused envisionment would be used to generate the abstract state and its successors.

2. Dynamic chatter abstraction uses constraint satisfaction to identify the set of chattering variables for a given state. Currently, it uses a fairly straight–forward, backtracking algorithm. The efficiency of the algorithm could be significantly improved in one of two ways:

   - Dynamic chatter abstraction uses a sound and complete constraint satisfaction algorithm to determine if a variable chatters. The worst–case complexity of this algorithm, however, is exponential. Alternatively, *constraint propagation* (Tsang, 1993) provides a polynomial–time algorithm that is potentially incomplete (*i.e.* it may identify a non–chattering variable as chattering). A propagation algorithm detects chattering using the following steps:

     (a) Identify variables that are restricted from chattering due to a single constraint independent of the other variables in the model. (e.g. If a variable is constant then it cannot chatter). These variables are labeled as `:nochatter` and the rest of the variables are labeled `:undetermined`.

     (b) Propagate this status through the constraint graph by selecting constraints with only a single potentially chattering variable v and determining if the constraint restricts v from chattering. This process continues no more information can be inferred.

     (c) Once step 2 completes, iff a variable's status is still `:undetermined` then it is identified as a chattering variable.

     While propagation works in many cases, for certain models it may identify a non–chattering variable as chattering due to cycles in the constraint graph. (See figure 4.10 for an example where propagation is too weak.)

[11] Additional exploration simply delineates the different combinations of values and identifies the exit states.

Propagation, however, could be used improve the efficiency of the current constraint satisfaction algorithm by quickly classifying certain variables as non–chattering. This could significantly reduce the exponential complexity of the constraint satisfaction portion of the algorithm by eliminating the need to test certain equivalency classes. While a constraint propagation algorithm has been developed and tested, it is not currently integrated into the dynamic chatter abstraction algorithm.

- Ordering algorithms could be developed for the equivalency nodes, dependencies, and conditions. Currently, the ordering of these elements is fairly close to random. An intelligent heuristic for ordering could greatly reduce the portion of the search space that is explored.

4.9 Conclusions

Chatter branching is a major source of irrelevant distinctions that has often hindered the application of qualitative simulation techniques to larger, more realistic problems. In particular, it can be quite frustrating when encountered by inexperienced modelers from other disciplines trying to use qualitative simulation to solve a particular problem. The techniques presented here provide two fully automated techniques that can be used to eliminate all instances of chatter, thus allowing the model building to focus on issues relevant to the problem that he is trying to address.

Chapter 5

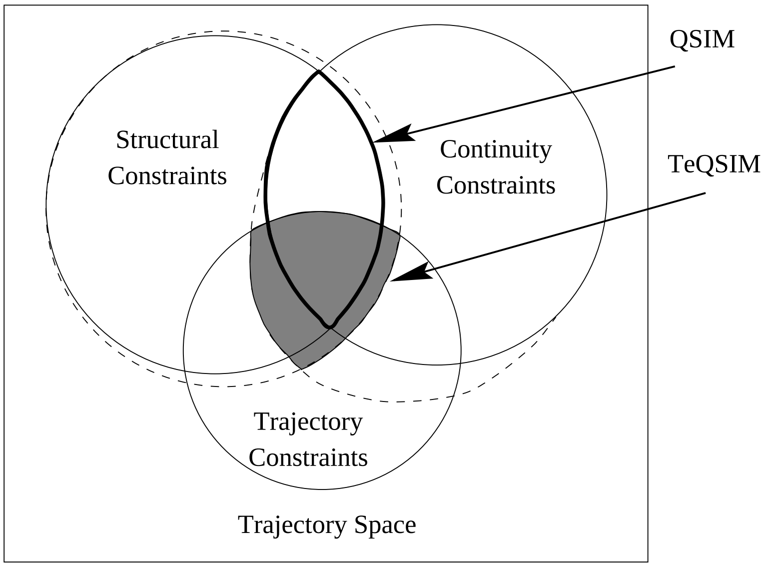
Temporally Constrained QSIM (TeQSIM)

Qualitative simulation uses a structural, equation–based model to describe the time–invariant relationships among the variables of a system. Simulation generates a description of the potential time–varying behaviors of the class of systems described by the model. The modeler, however, may have additional information about the behavior of the system that he would like to incorporate into the model. Alternatively, he may only be interested in the behavior of the system within certain restricted regions of the trajectory space. Currently, however, the modeler is unable to incorporate this type of information into the simulation process because of the restricted nature of the modeling language.

Temporally Constrained QSIM (TeQSIM) extends the expressiveness of the modeling language, allowing the modeler to express behavioral information via *trajectory constraints*. Trajectory constraints are expressed primarily through temporal logic expressions, containing both qualitative and quantitative information, which provide a declarative language for specifying both time–variant and time–invariant constraints. In addition, *discontinuous change expressions* are used to inject discontinuities into the simulation to control the behavior of exogenous variables.

Figure 5.1 shows how the different sources of constraining power contribute to TeQSIM's results. TeQSIM processes trajectory constraints by integrating temporal logic model checking into the qualitative simulation process resulting in three main benefits:

1. *Behavior filtering* tests each partial behavior segment against the trajectory constraints as the behaviors are incrementally generated. A behavior is elim-



TeQSIM uses three sources of information to constrain a simulation:

Structural constraints are specified as equations relating variables within the model; implicit

Continuity constraints restrict the relationship between variable values across time to ensure the continuity of each variable; and

Trajectory constraints are specified via temporal logic expressions restricting the behavior of individual variables and the interactions between the behaviors of related variables.

Each point in the diagram above represents a real valued trajectory. A qualitative behavior corresponds to a region within this space of trajectories.

- Discontinuous changes specified by the user cause a relaxation of the continuity constraints applied during simulation (dotted line surrounding the continuity constraints).
- Incorporating external events into the simulation extends the set of trajectories that are consistent with the structural constraints (dotted line surrounding the structural constraints).
- The qualitative behaviors generated by QSIM correspond to the trajectories consistent with both the unextended structural constraints and the unrelaxed continuity constraints (thick boundary region) while the set of behaviors generated by TeQSIM corresponds to those trajectories consistent with all three constraint types (shaded region).

Figure 5.1: TeQSIM constraint interaction

inated from the simulation when it can be shown that all of its possible completions fail to model the set of temporal logic expressions. Thus, the space of the behavioral description is restricted to include only behaviors that can satisfy the temporal logic expressions.

2. *Behavior refinement* integrates the numeric information in the temporal logic expressions into the qualitative simulation in order to provide a more–precise numerical description. This process restricts an individual behavior to include only those real–valued interpretations that model the set of temporal logic expressions thus improving the precision of the prediction.

3. *Injecting discontinuous changes* into the simulation allows the modeler to control the behavior of exogenous variables by either temporally–bound external events or qualitative changes occuring in the simulation.

Trajectory constraints can be used to specify the behavior of time–varying input variables for non–autonomous systems; to reduce the complexity of a simulation by focusing the simulation on a region of the state space; to incorporate observations into the simulation; and to reason about boundary condition problems by specifying available information about final or intermediate states.

This chapter describes both the syntax and the semantics used to specify trajectory constraints in TeQSIM and presents the model–checking algorithm that processes these constraints. This is followed by an example that demonstrates how trajectory constraints can be used to extract additional information from a qualitative simulation. Finally, results are presented that prove the model–checking algorithm sound and complete with respect to the trajectory constraints.

5.1 TeQSIM Overview

TeQSIM accepts as input a model (i.e. a QDE), an initial state, and a *trajectory specification*. The trajectory specification is comprised of three types of expressions: temporal logic expressions, discontinuous change expressions and external event declarations. An external event declaration defines a named, quantitatively bounded event that is not represented in the QDE. The declaration of external events introduces new relevant time–points that are inserted during the simulation. For example, the declaration `(event open :time (2 4))` defines a time–point called `open` that is quantitatively constrained to occur at time $t \in [2, 4]$. References to external events included in both temporal logic and discontinuous change expressions allow the modeler to temporally constrain the information contained within these expressions and specify correlations between external events and events generated during the simulation. Currently, the specification of external events is limited to a totally ordered sequence. [1]

The term *trajectory constraint* refers to both temporal logic and discontinuous change expressions. Temporal logic expressions (also called temporal logic constraints) are the primary method in TeQSIM to specify trajectory constraints and provide a unifying framework for filtering behaviors since discontinuous change expressions can be translated into an equivalent temporal logic expression.

5.1.1 Temporal logic constraint language

Temporal logic constraints are specified in TeQSIM using a variation of a propositional linear-time temporal logic (PLTL) (Emerson, 1990) called the *temporal logic constraint language* (TLCL). TLCL combines state formulæ, that express information about an individual state, with temporal operators such as *until*, *always*, and *eventually*. These operators extend the state formulæ across time.

The atomic state formulæ used within TLCL describe qualitative and quantitative information about individual states. For example, `(qvalue X ((0 X*) inc))` states that variable `X` is between `0` and `X*` and increasing. Propositions such as `value-<=` and `value-in` are used to specify numeric bounds and numeric ranges, respectively. Boolean combinations of these atomic propositions are also allowed.

Temporal operators are applied to create *path formulæ* to extend the information within the state formulæ across time. A path formula is defined recursively as either a state formula or a composition of two or more path formulæ via a temporal operator. Path formulæ are evaluated against sequences of states (*i.e.* behaviors). A path formula comprised of a single state formula is true of a behavior if it is true of the first state in the behavior.

Following Shults and Kuipers (1997), we have adopted as basic temporal operators `until`, `next`, and `strong-next` have defined additional ones using boolean operators. The path formula (`until` p q), where both p and q are path formulæ, is true for a behavior if p holds for all suffixes of the behavior preceding the first one where q holds (*i.e.* p is true until q becomes true for the first time), while (`strong-next` p) is true for a behavior if it contains at least two states and p holds in the behavior starting at the second state. `Next` is similar to `strong-next` except

[1] We have considered removing this restriction and allowing a partially ordered set of external events; however, at this point the applications that we have considered do not require this additional expressiveness. Furthermore, partially ordered external events increase the complexity of the simulation since all possible orderings must be considered.

that it is also true if the behavior consists of a single state. Other temporal operators can be defined as abbreviations from these two. The following abbreviations are used in the example in the next section. Their expansions, along with a complete description of the syntax and the semantics for both state and path formulæ is provided in section 5.3.

`(always` p`)`	p is always true.
`(between` p q r`)`	between the first occurrence of p and the following occurrence of q, r is true.
`(occurs-at` p q`)`	q is true at the first occurrence of p.
`(starts` p q`)`	q is true from the first occurrence of p.
`(follows` p q`)`	q is true from just after the first occurrence of p.

5.1.2 Discontinuous change expressions

Discontinuous change expressions define when a particular discontinuity can occur and specify its immediate effects (*i.e.* new values for the variables that change discontinuously). This information is propagated through the model to determine the variables affected by the discontinuous change. Thus, the expression

```
(disc-change (qvalue X (X* std)) ((inflow (if* inf) :range (400 440))))
```

states that when X reaches $X*$ and is `std`, `inflow` will instantaneously move into the interval `(if* inf)` and that the value of X will be within the range `(400 440)`.

These expressions both expand (since discontinuous changes are not normally generated) and constrain (since behaviors that do not exhibit the change are filtered) the state–space explored during the simulation. The discontinuous change processor injects the changes into the simulation by relaxing the continuity constraints imposed by QSIM. To restrict the simulation to behaviors that exhibit the specified discontinuity, discontinuous change expressions are translated into temporal logic constraints.

5.2 Problem solving with TeQSIM

The TeQSIM algorithm has been tested on a range of examples demonstrating a variety of tasks. The following example demonstrates how TeQSIM can be used to derive numeric bounds for a parameterized proportional controller. In addition, this example demonstrates the specification of a time–varying exogenous input to allow simulation of a non–autonomous system.

5.2.1 Parameter identification example

The model consists of a simple tank with a regulated output flow rate governed by the equations $\dot{V} = I - f(L, v)$ and $L = g(V)$. V denotes the volume of liquid in the tank, L the level, I the input flow rate, v the valve opening and $f(\cdot,\cdot)$ the output flow rate. f and g are partially known monotonic functions bounded by certain numeric functions.

TeQSIM can be used to help design a controller that maintains a constant water level ($L = L_s$) in the tank despite variations in the input flow rate. A proportional controller is added to the model to open or close the valve by an amount proportional to the error: $E = L - L_s$ and $v = v_s + KE$.

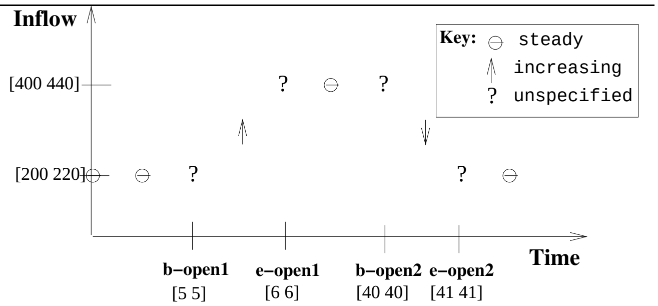
TeQSIM derives bounds on the constant K that ensure that the closed-loop behavior of the system satisfies a set of performance criteria specified via trajectory constraints: a perturbation to the input flow rate, bounds on the overshoot of the controller, and bounds on the time required to return to the nominal state following the termination of the perturbation. The specification of such information via trajectory constraints is straightforward, and the computation of the solution is relatively inexpensive. These bounds are used by the quantitative reasoning component of QSIM to infer bounds on `K`. Figure 5.2 shows the syntax of these trajectory constraints and figure 5.3 gives the results of the simulation.

Both components of TeQSIM– qualitative simulation and trajectory specification – are crucial to the solution of a problem of this nature. Qualitative simulation provides a discretization of trajectories essential for supporting a search mechanism. In addition, each qualitative behavior is refined via forward and backward propagation of quantitative information in the constraints. Trajectory constraints drive the system by specifying the behavior of the exogenous variable (*i.e.* input flow rate) and the controller's performance. This information cannot be represented within the QDE.

These results require careful interpretation. TeQSIM does not generate a proof that all real–valued systems satisfying the QDE behave in the specified manner when `K` is within the predicted range. QSIMś incompleteness means that the resulting behavior may be spurious. To address this, TeQSIM adds a *relative* guarantee that *if* a solution exists, it satisfies the predicted quantitative bounds. Other methods like Monte Carlo simulations (Brajnik, 1997) can be applied to ensure that a solution exists.

This example demonstrates how trajectory constraints can be used to simulate non-autonomous systems and perform parameter identification. TeQSIM can also help meet other design goals for the controlled tank system. Some examples

```
(event b-open1 :time 5)
(event e-open1 :time 6)
(event b-open2 :time 40)
(event e-open2 :time 41)
```



(a) External event declaration (b) Desired input trajectory for `Inflow`

	Temporal logic expression	Description

1	`(and (occurs-at (qvalue error (e* dec))`
	`(value-<= time 60))`
	`(follows (qvalue error (e* dec))`
	`(qvalue error ((0 e*))))`
		The system must settle (*i.e.* the error must go below `e*` and remain there) within 19 seconds of the end of the perturbation (*i.e.* at t=60s).

2	`(always (value-<= outflow 320))`	The outflow must remain below 320 cm^3/s.

3	`(until (and (value-in inflow (200 220))`
	`(qvalue valve (NIL std)))`
	`(event b-open1))`
		The inflow is constant in the range (200 220) until the beginning of the first opening action.

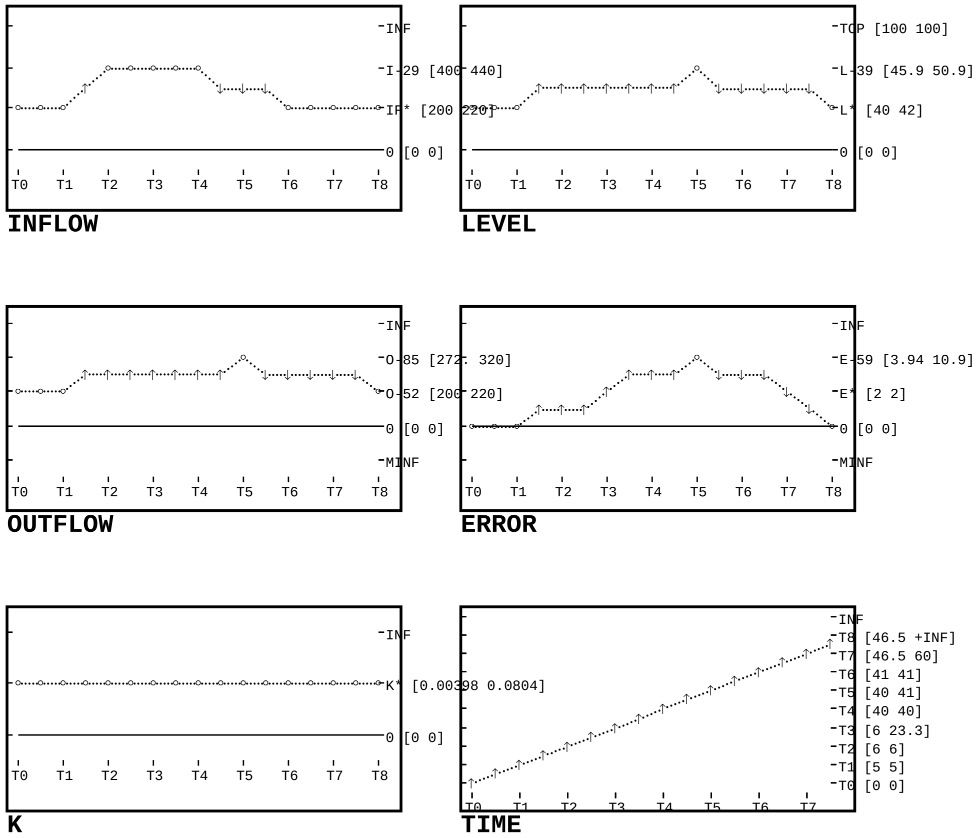
4	`(between (event b-open1)`
	`(event e-open1)`
	`(qvalue inflow (NIL inc)))`
		Between events `b-open1` and `e-open1` (*i.e.* the duration of the first opening action) the inflow is increasing.

5	`(occurs-at (event e-open1)` `(value-in inflow (400 440)))`	Inflow is steady and in the range (400 440) after the first opening action.
...		

(c) Trajectory constraints (closing action not described)

- The external event declaration (a) defines four external events that correspond to the beginning and end of an opening and a closing action. The description of the time–varying input (b) uses temporal logic constraints that refer to these four external events (c–3,4,5).
- Trajectory constraints are also used to place an upper bound on the settling time along with a bound on the outflow rate (c–1,2).

Figure 5.2: Trajectory specification for parameter identification example of a regulated tank–flow controller



- TeQSIM generates a single behavior using the trajectory constraints specified in figure 5.2. Notice that the behavior of `Inflow` matches the specification and that the bounds on `outflow` and `time` have been applied.

- Numerical information specified within the trajectory constraints is used to derive an upper and lower bound ([0.00398 0.0804]) for the constant **K**. The bounds can be seen in the time–plot for `K`.

Figure 5.3: Time plots generated by TeQSIM

are presented in table 5.1.

5.3 Trajectory Specification Language: syntax and semantics

As mentioned above, TLCL is the primary method for specifying trajectory constraints. First, we will present the syntax and semantics for TLCL then we will describe the syntax and the semantics for the external event declaration and the discontinuous change expressions.

5.3.1 Temporal logic constraint language: syntax and semantics

TeQSIM uses temporal logic constraints to incrementally guide and refine the qualitative simulation. Two extensions are required to traditional propositional linear–time temporal logics:

- A three–value logic is used that allows an expression to be *conditionally entailed* when quantitative information in the expression can be applied to a behavior to refine the description. A *refinement condition* specifies numerical bounds extracted from the TL expressions. Application of these conditions to the behavior eliminates the region of the state space that extended beyond the quantitative information specified in the TL expression.
- To apply the model–checking algorithm incrementally during the simulation, an *undetermined* result may occur when the behavior is insufficiently determined to evaluate the truth of a TL expression.

The following subsection gives a formal description of the syntax and semantics of TLCL. The syntax and semantics are derived from work done by Emerson (Emerson, 1990) and Kuipers and Shults (Kuipers & Shults, 1994).

5.3.2 Syntax

The propositional part of the language includes the following set $\mathcal{SF}$ of *state formulæ* (where v denotes a QSIM variable, $\mathcal{R}[v, s]$ the range of potential values for v in state s, v_s the unknown value of v in s, and n, n_i denote extended real numbers (*i.e.* $n, n_i \in \Re \cup \{\pm\infty\}$):

Continuous feedback control	A control law can be expressed in terms of a set of formulæ relating the value of the monitored variable (say `Level`) to the required of the control variable (say `Valve`). The resulting closed–loop behaviors can then be analyzed with respect to the controller's goal. The following trajectory constraint gives a partial specification of a control law that avoids the tank overflowing. It states that the valve opening must be increasing whenever the magnitude of `Level` is greater than `high` and the valve hasn't yet reached its maximum opening `max`. (always (between (qvalue Level ((high nil) nil)) (or (qvalue Level (high dec)) (qvalue Valve (max nil))) (qvalue Valve (nil inc))))
Continuous feed–forward control	A control law can be expressed in terms of a set of formulæ relating a predicted value of the monitored variable to the current value or trend of the control variable. The following trajectory constraint specifies that if the tank can potentially overflow then the valve opening should be increased until either it reaches its maximum value or `level` becomes smaller than `high`. (always (implies (eventually (qvalue Level (top nil))) (until (qvalue Valve (nil inc)) (or (qvalue Level (high dec)) (qvalue Valve (max nil))))))
Goal Oriented Simulation	The statements reported below can be used to check whether the tank will overflow within a specified time frame. Since TeQSIM is sound, if no behaviors are produced then the modeled system can not violate these constraints (assuming that the QDE model is valid). The following trajectory constraint limits the simulation to behaviors in which the tank `Level` reaches `high` within 150 seconds. (and (event horizon :time 150) (before (qvalue Level (high nil)) (event horizon)))

Table 5.1: Applying TeQSIM to other tasks using the regulated tank model.

(qvalue v ($qmag$ $qdir$)) , where $qmag$ is a landmark or open interval defined by a pair of landmarks in the quantity space associated with v and $qdir$ is one of {inc, std, dec}. NIL can be used anywhere to match anything. Such a proposition is true in state s exactly when the qualitative value of v in s matches the description ($qmag$ $qdir$).

(value-<= v n) is true in state s if and only if $\forall x \in \mathcal{R}[v, s] : x \leq n$; it is false if and only if $\forall x \in \mathcal{R}[v, s] : n < x$; it is unknown otherwise. In such a case the refinement condition is that the least upper bound of the possible real values of v is equal to n (*i.e.* $v_s \leq n$). (value->= v n) is similar.

(value-in v (n_1 n_2)) is true in state s if and only if $\mathcal{R}[v, s] \subseteq [n_1, n_2]$, and it is false if and only if $\mathcal{R}[v, s] \cap [n_1, n_2] = \emptyset$. It is unknown otherwise, and the refinement condition is that the greatest lower bound is equal to n_1 and the least upper bound is equal to n_2 (*i.e.* $n_1 \leq v_s \wedge v_s \leq n_2$).

Non–atomic propositions are defined using standard boolean operators (and, not); standard propositional abbreviations (true, false, or, implies, iff) are also allowed. As in (Kuipers & Shults, 1994), other proposition schema are defined that allow TeQSIM to reference attributes of states computed by QSIM (like whether a state is quiescent, is stable or occurs at infinite time).

Path formulæ are constructed from state formulæ by combining them using temporal operators. Apart from the boolean operators, the temporal operators next, strong-next and until are primitive, while the others can be derived from these primitives using syntactic translation rules. strong-next is included in addition to next because we deal also with finite paths and must provide sufficient expressiveness.

The set of *path formulæ* $\mathcal{PF}$ is defined by the following rule (where $p \in \mathcal{SF}$ and $\varphi \in \mathcal{PF}$):

$$\varphi ::= p | (\varphi \texttt{ and } \varphi) | (\texttt{not } \varphi) | (\texttt{strong-next } \varphi) | (\texttt{next } \varphi) | (\texttt{until } \varphi\ \varphi).$$

The following abbreviations are used:

```
      (or p q)  ≡  (not (and (not p) (not q)))
(releases p q)  ≡  (not (until (not p) (not q)))
  (before p q)  ≡  (not (until (not p) q))
(eventually p)  ≡  (until true p)
  (always p)    ≡  (releases false p)
   (never p)    ≡  (always (not p))
  (starts p q)  ≡  (releases p (implies p (always q)))
 (follows p q)  ≡  (releases p
                      (implies p (strong-next (always q))))
(occurs-at p q) ≡  (releases p (implies p q))
(between p q r) ≡  (releases p
                      (implies p (strong-next (until r q))))
```

These formulæ are translated into a *positive normal form* that is defined as follows: (i) `until`, `releases`, `next` and `strong-next` are the only temporal operators in the formula, (ii) the scope of every `not` in the formula is an atomic proposition, and (iii) such a scope does not include any proposition constructed using `value-<=`, `value->=` or `value-in`. The first two requirements do not restrict the expressiveness of the language since the abbreviations shown above can be used to transform a formula into a form that satisfies these conditions. The last requirement is due to the specific representation of numeric information in QSIM, which allows neither open numeric intervals nor disjunction of intervals.

5.3.3 Semantics

The semantics of a temporal logic path formulæ are defined using an interpretation structure $M = <S, \sigma, \Gamma, \mathcal{I}, \mathcal{C}, \mathcal{M}>$ where[2]:

- S is a set of states;
- $\sigma: S \to S$ is a partial function that maps states to their successors (we are defining a linear–time logic on finite and infinite paths, hence each state has at most one successor).
- $\mathcal{I}: \mathcal{SF} \times S \to \{t, f, ?\}$ is an assignment of truth values to propositions and states (? denotes the "ambiguous" truth value).
- Γ is a set of *refinement conditions*. Γ is closed with respect to the standard boolean operators $\{\wedge, \neg\}$ and contains the distinguished item C_{true}; a refine-

[2]These structures are extended form their standard definition (*e.g.* (Emerson, 1990)) to accommodate the refinement process.

ment condition is an entity that specifies how a state has to be refined (see below).

- $\mathcal{C}: \mathcal{SF} \times S \to \Gamma$ is a function (*condition generator*) that maps state formulæ and states into refinement conditions that determine how the state should be modified when the formula, interpreted on that state, has an ambiguous truth value. We require that (i) $\mathcal{C}(\varphi, s) = C_{\text{true}}$ if and only if $\mathcal{I}(\varphi, s) = t$, (ii) $\mathcal{C}(\varphi, s) = \neg C_{\text{true}}$ if and only if $\mathcal{I}(\varphi, s) = f$, and (iii) $\mathcal{C}(\varphi, s)$ is defined when $\mathcal{I}(\varphi, s) =?$.
- $\mathcal{M}: \Gamma \times S \to S$ is a function (*state modifier*) that maps a condition and a state into a refined state. For any state s, $\mathcal{M}(C_{\text{true}}, s) = s$ and $\mathcal{M}(\neg C_{\text{true}}, s) = \perp$.

We require that if φ is an atomic proposition then refinement conditions are *necessary and sufficient* for resolving the ambiguity, *i.e.* if $C = \mathcal{C}(\varphi, s)$ then $\mathcal{I}(\varphi, \mathcal{M}(C, s)) = t$ and $\mathcal{I}(\varphi, \mathcal{M}(\neg C, s)) = f$ (unless $C = C_{\text{true}}$, in which case $\mathcal{M}(\neg C, s) = \perp$, or $C = \neg C_{\text{true}}$, when we have $\mathcal{M}(C, s) = \perp$,).

Given an interpretation M, a *path* is defined as a sequence of states $x = <s_0, s_1, \ldots>$ such that for any pair of consecutive states (s_i, s_{i+1}), $\sigma(s_i) = s_{i+1}$. The length of a path is denoted by $|x|$, which can be infinite. For all non–negative integers $i < |x|$, x^i denotes the sub–path $<s_i, \ldots>$, $x^{(i,j)}$ denotes $<s_i, \ldots, s_j>$, and $x(i)$ denotes s_i. A *full–path extension* of a finite path x, denoted by $\widehat{x}$, is an infinite path that has x as a prefix. Finally, $\mathcal{M}$ is naturally extended to paths: if $x = <s_0, \ldots>$ then $x' = \mathcal{M}(C, x) = <s'_0, \ldots>$ where $s'_i = \mathcal{M}(C, s_i)$ for all i. If for some j $\mathcal{M}(C, s_j) = \perp$, then $\mathcal{M}(C, x) = \perp$.

An interpretation M *is subsumed by* M' (written $M \preceq M'$) if and only if M' contains all the states and conditions of M and the four functions of M are restrictions of those of M').

QSIM computes, in finite time, a set of behaviors, each representing a class of trajectories of the system being simulated. Although a QSIM behavior is a finite structure, it may represent infinite trajectories of the simulated system, as quiescent states are finite descriptions of fixed–point trajectories. A behavior $b = (s_0, \ldots, s_n)$ implicitly identifies a minimal (with respect to $\preceq$) interpretation structure M_b such that:

1. $S = \{s_0, \ldots, s_n\}$.
2. (s_i, s_{i+1}) belongs to the QSIM relations `successor` or `transition` and $\sigma(s_i) = s_{i+1}$ for all $i = 0, \ldots, n-1$
3. Γ consists of the set of all possible numerical bounds on QDE variables. These bounds can be represented as (boolean combinations of) inequalities between

the unknown value of the variable on a state and a real number; for example, the condition that the QDE variable X in state s has to be less than 5 is "$X_s < 5$".

4. $\mathcal{I}$ is determined by the qualitative values, numerical bindings and statuses of states; $\mathcal{I}$ may give ? only for propositions including `value-<=`, `value->=` and `value-in` as specified in section 4.1;
5. $\mathcal{C}$ is determined by numerical bindings in states;
6. $\mathcal{M}$ is determined by QSIM's the numerical inference capabilities.

A behavior b is *closed* if and only if QSIM detected that s_n is a quiescent state or that s_n is a transition state that has no possible successors (signaling that the trajectory of the dynamical system has reached a boundary of the operating region of the model). For a closed behavior b, the full–path extensions $\widehat{b} = b$. In the rest of the paper, when discussing a behavior b, we will implicitly assume to deal with interpretations M that subsume M_b.

Formulæ with quantitative information may lead to ambiguity when evaluated (*i.e.* the behavior only models a portion of the range specified). Ambiguity, however, is not purely a syntactic property, but rather depends on state information. For example, `(value-<= X .3)` will be (unconditionally) true on a state s where $\mathcal{R}[X, s] = [0, 0.25]$, but only conditionally true on s' where $\mathcal{R}[X, s'] = [0, 1.0]$. Because of ambiguity, to define the semantics of formulæ we need to introduce two entailment relations. The first one, called *models* ($\models$), is used to characterize non–ambiguous true formulæ; the second one, called *conditionally–models* ($\stackrel{?}{\models}$) characterizes formulæ that are ambiguous. The definition below gives the semantics of the language.

Definition 5.1 (Models and Conditionally–Models) *Given an interpretation M, the relations* models *($\models$) and* conditionally–models *($\stackrel{?}{\models}$) are defined as follows (we will write $x \stackrel{*}{\models} \varphi$ to mean ($x \models \varphi$ or $x \stackrel{?}{\models} \varphi$), and $x \stackrel{*}{\not\models} \varphi$ to mean that it is not the case that $x \stackrel{*}{\models} \varphi$):*

State formulæ (a ranges over atomic propositions, p and q over $\mathcal{SF}$, and s over S):

$$
\begin{array}{ll}
s \models a & \text{iff } \mathcal{I}(a,s) = t \\
s \stackrel{?}{\models} a & \text{iff } \mathcal{I}(a,s) = ? \\
s \models (p \texttt{ and } q) & \text{iff } s \models p \text{ and } s \models q \\
s \stackrel{?}{\models} (p \texttt{ and } q) & \text{iff } s \stackrel{*}{\models} p \text{ and } s \stackrel{*}{\models} q \text{ and } \stackrel{?}{\models} \text{ occurs at least once} \\
s \models (\texttt{not } p) & \text{iff } s \not\stackrel{*}{\models} p \\
s \stackrel{?}{\models} (\texttt{not } p) & \text{iff } s \stackrel{?}{\models} p.
\end{array}
$$

Path formulæ ($p \in \mathcal{SF}$ and $\varphi, \psi \in \mathcal{PF}$ and x is a non–empty path):

$$
\begin{array}{lll}
x \models p & \text{iff} & x(0) \models p \\
x \stackrel{?}{\models} p & \text{iff} & x(0) \stackrel{?}{\models} p \\
x \models (\texttt{strong-next } \varphi) & \text{iff} & |x| > 1 \text{ and } x^1 \models \varphi \\
x \stackrel{?}{\models} (\texttt{strong-next } \varphi) & \text{iff} & |x| > 1 \text{ and } x^1 \stackrel{?}{\models} \varphi \\
x \models (\texttt{next } \varphi) & \text{iff} & |x| > 1 \text{ implies } x^1 \models \varphi \\
x \stackrel{?}{\models} (\texttt{next } \varphi) & \text{iff} & |x| > 1 \text{ implies } x^1 \stackrel{?}{\models} \varphi \\
x \models (\texttt{until } \varphi\ \psi) & \text{iff} & \exists i \geq 0 : (x^i \models \psi \text{ and } \forall j < i : x^j \models \varphi) \\
x \stackrel{?}{\models} (\texttt{until } \varphi\ \psi) & \text{iff} & \exists i \geq 0 : (x^i \stackrel{*}{\models} \psi \text{ and} \\
 & & \forall j < i : x^j \stackrel{*}{\models} \varphi) \text{ and} \\
 & & \stackrel{?}{\models} \text{ occurs at least once} \\
x \models (\texttt{releases } \varphi\ \psi) & \text{iff} & \forall i \geq 0 : x^i \models \psi \text{ or} \\
 & & \exists j \geq 0 : x^j \models \varphi \text{ and } \forall k \leq j : x^k \models \psi \\
x \stackrel{?}{\models} (\texttt{releases } \varphi\ \psi) & \text{iff} & \forall i \geq 0 : x^i \stackrel{*}{\models} \psi \text{ or} \\
 & & \exists j \geq 0 : x^j \stackrel{*}{\models} \varphi \text{ and } \forall k \leq j : x^k \stackrel{*}{\models} \psi \text{ and} \\
 & & \stackrel{?}{\models} \text{ occurs at least once}
\end{array}
$$

The semantics of $(\varphi \texttt{ and } \psi) and (\texttt{not } \varphi)$ are similar to the propositional case. The definition of `releases` given here agrees with its syntactic equivalence.

To properly handle the refinement process, TeQSIM must restrict the usage of ambiguous formulæṪhis restriction is required because an arbitrary ambiguous formula may yield several alternative refinement conditions. A disjunction of refinement conditions cannot be applied to states without requiring a change in the successor function σ and without allowing disjunctive state information; this would be incompatible with the QSIM representation. Two different types of disjunction can result from certain ambiguous formulæ. A *state disjunction* stems from a disjunction of ambiguous state formulæ. For example, when interpreted against a particular state `(or (value-<= X 0.5), (value->= Y 15))` may yield the condition $(X_s \leq 0.5 \vee Y_s \geq 15)$. When applying such a condition to a state, $\mathcal{M}(C, s)$

yields two states – s' in which $X_{s'} \leq 0.5$ and s'' where $Y_{s''} \geq 15$. A *path disjunction*, on the other hand, occurs when an ambiguous formula is included in a path formula in such a manner that a sub–formula can be conditionally true for more than one sub–path. For example, in the path formula `(until p (value-<= X 0.5))`, a disjunction occurs across sub–paths regarding when the condition ($X \leq 0.5$) should be applied.

The following definitions restrict the syntax to formulæ that are well–behaved. A *potentially ambiguous formula* is: (i) any atomic proposition constructed using one of the following operators `value-<=`, `value->=` or `value-in`, or (ii) a path formula containing a potentially ambiguous sub–formula. *Admissible formulæ* are those formulæ φ that satisfy the following conditions:

1. φ is in positive normal form,
2. if $\varphi =$ (`until` p q) then q is not potentially ambiguous,
3. if $\varphi =$ (`releases` p q) then p is not potentially ambiguous, and
4. if $\varphi =$ (p `or` q) then at most one of p and q is potentially ambiguous.

The following lemma guarantees that checking a model against an admissible formula does not lead to disjunction of conditions.

Lemma 5.1 *For all admissible formulæ φ and for any interpretation M and path x, if $x \overset{?}{\models} \varphi$ then any necessary and sufficient condition C for making x a model for φ (*i.e. $\mathcal{M}(C, x) \models \varphi$ *and* $\mathcal{M}(\neg C, x) \not\models \varphi$*) is either a single condition or a conjunction of conditions.*
The proof follows from properties of M on propositions after a case–by–case inductive analysis of formulæ.

Even though the restriction to admissible formulæ reduces expressiveness, it does not hinder the practical applicability of TeQSIM. As long as important distinctions are qualitatively represented (using landmarks or events), most trajectory constraints can be cast into admissible formulæ. For example, the constraint that "until the level goes above 50, the input flow rate must be below 200" could be expressed with the following non–admissible formula `(until (value-<= InFlow 200) (value->= Level 50))`, where the two distinctions (200 and 50) do not correspond to qualitative landmarks. By adding a landmark to the quantity space of `Level` corresponding to the value 50, and by assigning the range $[50, 50]$ to such a landmark, we can rewrite the formula in an admissible form (*i.e.* `(until (value-<= InFlow 200) (qvalue Level (lm-50 nil)))`) without losing any information. However, introducing new landmark in a qualitative model increases the number of qualitative distinctions that the simulator detects, and this is why one often wants to use formulæ that provide numeric constraints using operators like `value-<=`.

5.3.4 External event declaration

TeQSIM specifies external events via a totally ordered sequence of external event declarations given in an *external event list*. Each declaration is of the form:

> `(event` *event-name* `:time (`*lb up*`))`

where *event-name* gives a symbolic name for the event, *lb* a lower bound on the temporal occurence of the event, and *ub* an upper bound. The temporal bounds can be omitted, in which case the bounds are assumed to be $(-\infty\ \infty)$. If *lb* and *ub* are equal, then a single value can be specified.

The semantics of the external event list are defined by translating the list into a set of TLCL expressions. The external event list $(E_1, E_2, \ldots, E_n)$, containing n event declarations, establishes the following set of TLCL expressions:

(1) For i from 1 to $n-1$, add the following expression:

> `(before` E_i E_{i+1}`)`

(2) For i from 1 to n, where E_i is of the form `(event` *event-name*$_i$ `:time (`*lb*$_i$ *ub*$_i$`))`, add the following expression:

> `(occurs-at (value-in time (`*lb*$_i$ *ub*$_i$`)) (event` *event-name*$_i$`))`

where `(event` *event-name*$_i$`)` refers to the qualitative state in which *event-name*$_i$ occurs.[3]

5.3.5 Discontinuous change expressions

Discontinuous change expressions are specified with the following syntax:

> `(disc-change` *preconds effects*`)`

where *preconds* is a boolean combination of `qvalue` propositions and *effects* is a list of expressions of the form `(`*variable qmag* `[:range` *range*`])`. This expression is translated into the temporal logic path formula

[3] External events are injected into the simulation by adding a *real–time* variable to the model. This expression is replaced by a reference to this real–time variable. This is discussed in more detail in section 5.4.

`(occurs-at` *preconds* `(strong-next` *effects'*`))`

where *effects'* is a conjunction of formulae `(qvalue` *variable* `(`*qmag NIL*`))` and `(value-in` *variable range*`)` derived from *effects*. This expression is true for a behavior if and only if *effects'* is true for the state immediately following the first state in which *preconds* is true. This formula is added to the list of temporal logic formulae used to guide and refine behaviors.

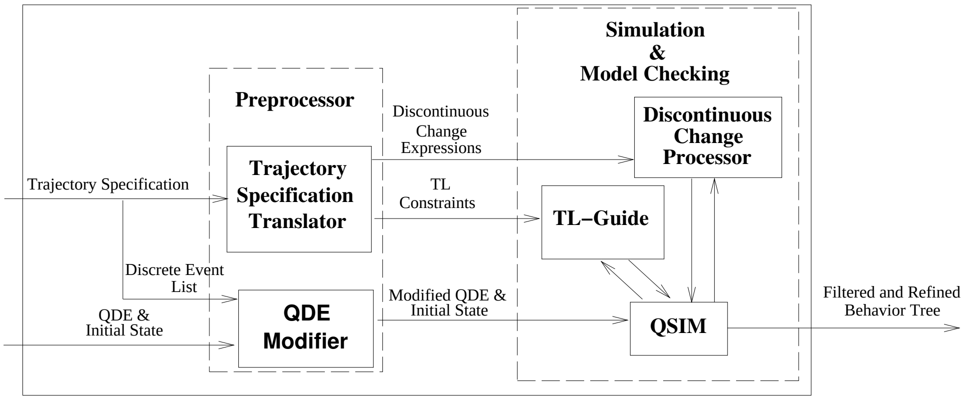
5.4 TeQSIM Theory and Architecture

TeQSIM is divided into two main components: the *preprocessor* modifies the QDE model and decomposes the trajectory specification into temporal logic and discontinuous change expressions, and the *simulation and model checking* component, which integrates temporal logic model checking into the simulation process by filtering and refining qualitative behaviors according to a set of temporal logic expressions and injects discontinuous changes into the simulation. Figure 5.4 gives an overview of the system architecture.

External events are incorporated into the simulation by the addition of an auxiliary variable that represents "real time" with a landmark corresponding to each event. The addition of this variable causes QSIM to branch on different orderings between external events and internal qualitative events identified during the simulation. The occurrence of the external events is restricted by the applicable quantitative bounds and trajectory constraints specified by the modeler.

Temporal logic and discontinuous change expressions are allowed to directly reference events in the event list using the form (`event` e_1), where e_1 is the name of an external event. These references are replaced by the appropriate formula containing a reference to the real–time variable and the landmark corresponding to the event. The addition of the real–time variable incorporates external events into the simulation in a seamless manner that does not require special handling during the simulation and model–checking component of the algorithm.

This section briefly describes how the *Discontinuous Change Processor* injects changes into the simulation and then gives a detailed discussion of how the *Temporal Logic Guide* algorithm (TL–Guide) incorporates temporal logic model checking into the simulation.



TeQSIM is an extension of the QSIM qualitative simulation algorithm. It contains four main components:

QDE Modifier – adds an auxiliary real–time variable to the QDE along with the appropriate constraints. The quantity space for the variable is derived from the *external event list*.

Trajectory Specification Translator – decomposes and translates the trajectory specifications into temporal logic and discontinuous change expressions. References to discrete events within both the temporal logic and discontinuous event expressions are replaced with the appropriate references to the real–time variable and the corresponding qualitative magnitude.

TL–Guide – performs temporal logic model checking during the simulation in order to filter and refine behaviors to satisfy the set of TL expressions.

Discontinuous Change Processor – injects discontinuous changes into the simulation by identifying states that satisfy the preconditions of a change and then propagating the effect to derive a new state.

Figure 5.4: TeQSIM architecture.

5.4.1 Discontinuous Change Processor

The Discontinuous Change Processor monitors states as they are created and tests them against the preconditions of applicable discontinuous change expressions. For a qualitative state s and a discontinuous change expression e, a new state is inserted into the simulation following state s if the preconditions in e are satisfied by s and a discontinuous change is required to assert the effects. A new, possibly incomplete, state s' is created by asserting the qualitative values specified in the effect and inheriting values from s for variables that are not affected by the discontinuous change via *continuity relaxation* (see below). All consistent completions of s' are computed and inserted as transition successors of s. Each discontinuous change expression can only be applied once within each behavior.

Qualitative reasoning uses continuity constraints to restrict the possible changes that can occur within a system. In order to predict the effects of discontinuous changes, however, continuity constraints must be relaxed, which leads to a combinatorial explosion of possible outcomes. TeQSIM uses Brajnik's *continuity relaxation* algorithm (Brajnik, 1995) to propagate the effects of a discontinuous change through the model by identifying variables that are necessarily continuous and variables that are potentially discontinuous. If only one variable in a non–differential constraint is not known to be continuous, then it is inferred to be necessarily continuous. Such a technique assumes that state variables (*i.e.* those that are integrals of model variables) and input variables are necessarily continuous unless mentioned in the discontinuous change expression.

5.4.2 TL–Guide model checking algorithm

Model checking and behavior refinement is performed by the *Temporal Logic Guide* algorithm. Each time QSIM extends a behavior by the addition of a new state, the behavior is passed to the TL–Guide. The behavior is filtered if there is sufficient information within the partially formed behavior to determine that all completions of the behavior fail to satisfy the set of TL expressions. If the behavior can potentially model the set of TL expressions, then it is refined by incorporating relevant quantitative information contained within the TL expressions. Otherwise the behavior is retained unchanged. The incremental nature of the algorithm allows behaviors to be filtered and refined as early as possible during the simulation.

Given a potentially partial behavior[4] b, TL–Guide computes (among other things) a truth value from the set $\{\mathtt{T}, \mathtt{F}, \mathtt{U}\}$. A definite answer (*i.e.* `T` or `F`) is provided

[4] A *partial behavior* is a behavior that has not been fully extended via simulation.

when b contains sufficient information to determine the truth value of the formula. For example, a non–closed behavior b will *not* be sufficiently determined with respect to the formula `(eventually` p`)` if p is not true for any suffix of b, since p may become true in the future.

A behavior is considered to be *sufficiently determined* with respect to a formula whenever there is enough information within the behavior to determine a single truth value of the formula for all completions of the behavior. If a behavior is not sufficiently determined for a formula, then `U` is returned and the behavior is not filtered out by TeQSIM. Notice that indeterminacy is a property independent from ambiguity: the former is related to incomplete paths, while the latter deals with ambiguous information present in states of a path[5].

Definition 5.2 (Sufficiently determined) *A behavior b is* sufficiently determined *for a positive normal formula φ (written $b \rhd \varphi$) iff $|b| > 0$ and one of the following conditions is met:*

1. b is a closed behavior, or φ is a proposition
2. $\varphi =$ `(`p_1 `and` p_2`)` and either $\forall i : b \rhd p_i$ or $\exists i : b \overset{*}{\not\models} p_i$ and $b \rhd p_i$
3. $\varphi =$ `(`p_1 `or` p_2`)` and either $\forall i : b \rhd p_i$ or $\exists i : b \overset{*}{\models} p_i$ and $b \rhd p_i$
4. $\varphi =$ `(strong-next` p`)` and $b^1 \rhd p$
5. $\varphi =$ `(next` p`)` and $b^1 \rhd p$
6. $\varphi =$ `(until` $p\ q$`)` and $\exists i < |b| : (b^i \overset{*}{\models} q$ and $b^i \rhd q$ and $\forall j < i : b^j \overset{*}{\models} p$ and $b^j \rhd p)$
7. $\varphi =$ `(until` $p\ q$`)` and $\exists i < |b| : (b^i \overset{*}{\not\models} p$ and $b^i \rhd p$ and $\forall j \leq i : b^j \overset{*}{\not\models} q$ and $b^j \rhd q)$
8. $\varphi =$ `(releases` $p\ q$`)` and $\exists i < |b| : (b^i \overset{*}{\models} p$ and $b^i \rhd p$ and $\forall j \leq i : b^j \overset{*}{\models} q$ and $b^j \rhd q)$
9. $\varphi =$ `(releases` $p\ q$`)` and $\exists i < |b| : (b^i \overset{*}{\not\models} q$ and $b^i \rhd q$ and $\forall j < i : b^j \overset{*}{\not\models} p$ and $b^j \rhd p)$.

Table 5.2 gives a graphical representation of conditions 6-9.

The relation $\rhd$ is important because it implies that the truth of a formula is invariant with respect to the extension of a behavior. More specifically:

Lemma 5.2 (On extensions) *For any finite path x of an interpretation and any positive normal formula φ such that $x \rhd \varphi$,*

[5] Inferences within the model–checking algorithm are limited to the information contained within the behavior. Thus, even though the constraint `(constant X)` is included within a model, the model–checking algorithm will not be able to determine that the formula `(eventually (qvalue X (nil inc)))` is false for all completions of a non–closed behavior.

case	**formula**	**example**	**result**

6	$\varphi = (\texttt{until}\ p\ q)$	`b:  ooooooioooooooo`
		`p:  ++++++`
		`q:        +`
			$b \models^{*} \varphi$

7	$\varphi = (\texttt{until}\ p\ q)$	`b:  ooooooooooioooo`
		`p:            -`
		`q:  -----------`
			$b \not\models^{*} \varphi$

8	$\varphi = (\texttt{releases}\ p\ q)$	`b:  oooooooiooooooo`
		`p:         +`
		`q:  ++++++++`
			$b \models^{*} \varphi$

9	$\varphi = (\texttt{releases}\ p\ q)$	`b:  oooooooiooooooo`
		`p:  -------`
		`q:         -`
			$b \not\models^{*} \varphi$

Assuming that b is not closed, conditions 6 through 9 in the definition of sufficiently determined define the conditions shown graphically above.

- Symbols `+` and `-` at position i means that the formula is true or false respectively when checked against the sub–behavior starting at i.

Table 5.2: Conditions for a behavior being sufficiently determined

$$
\begin{array}{rcl}
x \models \varphi & \Longleftrightarrow & \forall \widehat{x} : \widehat{x} \models \varphi, \vee \\
x \stackrel{?}{\models} \varphi & \Longleftrightarrow & \forall \widehat{x} : \widehat{x} \stackrel{?}{\models} \varphi, \vee \\
x \not\models \varphi & \Longleftrightarrow & \forall \widehat{x} : \widehat{x} \not\models \varphi
\end{array}
$$

The proof is by induction on formula length.

As QSIM extends a behavior with a new state s, the model–checking algorithm evaluates the truth of a temporal–logic path formula φ by evaluating a modified version of φ against s. (In other words, it does not have to re-evaluate the entire behavior with respect to φ.) Incremental evaluation of a temporal logic expression is achieved by progressively decomposing each path formula into two parts: a *present component* and a *future component.* A path formula φ is true if and only if the present component is true on the current state and the future component is true on the behavior starting from the successor state.

A *progressed* formula is a path formula in which these two components have been made explicit. Function Δ achieves this: $\Delta[\cdot]$ is applied to a positive normal formula φ and returns a new formula equivalent to φ which is a boolean combination of sub–formulæ that are either propositions or path formulæ scoped by `next` or `strong-next`. Δ is defined as follows:

$$
\begin{array}{rcl}
\Delta[p] & = & p, \text{ if } p \in \mathcal{SF} \\
\Delta[(\texttt{not } p)] & = & (\texttt{not } \Delta[p]) \\
\Delta[(p \texttt{ and } q)] & = & \Delta[p] \texttt{ and } \Delta[q] \\
\Delta[(p \texttt{ or } q)] & = & \Delta[p] \texttt{ or } \Delta[q] \\
\Delta[(\texttt{next } p)] & = & (\texttt{next } p) \\
\Delta[(\texttt{strong-next } p)] & = & (\texttt{strong-next } p) \\
\Delta[(\texttt{until } p\ q)] & = & \Delta[q] \texttt{ or } (\Delta[p] \texttt{ and } (\texttt{strong-next } (\texttt{until } p\ q))) \\
\Delta[(\texttt{releases } p\ q)] & = & \Delta[q] \texttt{ and } (\Delta[p] \texttt{ or } (\texttt{next } (\texttt{releases } p\ q)))
\end{array}
$$

We also ensure that a progressed formula is in disjunctive normal form (DNF) – *i.e.* it is a disjunction of terms, each of which is a conjunction of literals. A literal is either an atomic proposition, (`not` p) where p is a proposition, (`next` ϕ), or (`strong-next` ϕ) where ϕ is a path formula.

The following lemma gives an important characterization of progressed formulæ:

Lemma 5.3 (Equivalence) *For any path x and normalized formula φ:*

$$\begin{aligned} x \models \varphi &\iff x \models \Delta[\varphi] \\ x \models^{?} \varphi &\iff x \models^{?} \Delta[\varphi] \\ x \rhd \varphi &\iff x \rhd \Delta[\varphi] \end{aligned}$$

The proof is by induction on formula length.

The function $\mathcal{P}$ extracts a "present component" from a progressed formula by removing sub–formulæ starting with `next` operators. Notice how dependence on s plays a role only for the `strong-next` formula: in all other cases s is ignored. For the `strong-next` formula, the state is retained because checking the closedness of the behavior ending at the current state is the "present component" and this cannot be accomplished by relying solely on syntax. The *present formula extractor* $\mathcal{P}[\cdot,\cdot]$ maps a progressed formula $\varphi = \bigvee \tau_j$ and a state s into a proposition p such that p represents what must be true in s if φ has to be true on s and its successors. It is defined as follows:

$$\begin{aligned} \mathcal{P}[\varphi, s] &= \bigvee \mathcal{P}[\tau_j, s] \\ \mathcal{P}[\tau, s] &= \begin{cases} \texttt{FALSE} & \text{if } \langle s \rangle \text{ is closed and } \tau \text{ contains a } \texttt{strong-next} \text{ literal} \\ \tau' & \text{otherwise, where } \tau' \text{ results from } \tau \text{ after removing all } \texttt{next} \\ & \text{and } \texttt{strong-next} \text{ literals.} \end{cases} \end{aligned}$$

In addition, since the output of $\mathcal{P}$ is a proposition, we assume that a simplification step occurs, removing all redundant `TRUE`s and `FALSE`s from the resulting formula (for example (P `or` `FALSE`) is simplified into P).

$\mathcal{P}$ removes all temporal sub–formulæ from its input. For example, $\mathcal{P}[(P \texttt{ or } (\texttt{next } Q)), s] = P$ and $\mathcal{P}[(P \texttt{ and } (\texttt{next } Q)), s] = P$. In a closed behavior, terms containing a `strong-next` literal are "short–circuited" to `FALSE`.

The function $\mathcal{F}$, on the other hand, is responsible for characterizing the future component of a formula. The future component depends on which sub–formulæ of the present component are true or false. For example, if the formula is (Q `or` (P `and` (`next` φ))), where Q and P are propositions, then when Q is true the future component is the formula `TRUE`, whereas if Q is false and P true the future component is φ.

$\mathcal{F}[\cdot,\cdot]$ maps a progressed formula $\varphi = \bigvee \tau_j$ and a state s into another formula φ' that represents what must be true for the future of s if φ is true from s:

$$\mathcal{F}[\varphi, s] = \bigvee \mathcal{F}[\tau_j, s]$$

$$\mathcal{F}[\tau, s] \qquad \begin{array}{l}(\text{assume } \tau = (\bigwedge p_j) \wedge (\bigwedge(\texttt{next } \beta_j)) \wedge (\bigwedge(\texttt{strong-next } \gamma_j)) \\ \text{where } p_j \text{ are propositions})\end{array}$$

$$= \begin{cases} \texttt{FALSE} & \text{if exists } p_k : s \not\models \mathcal{P}[p_k, s] \\ (\bigwedge \beta_j) \wedge (\bigwedge \gamma_j) & \text{if } s \stackrel{*}{\models} \mathcal{P}[\tau, s] \text{ and } <s> \text{ is not closed} \\ \texttt{TRUE} & \text{if } s \stackrel{*}{\models} \mathcal{P}[\tau, s] \text{ and } <s> \text{ is closed} \end{cases}$$

In this case too we assume that a simplification step occurs, removing all redundant TRUE's and FALSE's from the resulting formula.

Notice that if a term contains a strong-next literal and the behavior is closed then the term is "dropped" from the result. For example,

$$\mathcal{F}[(P \texttt{ and } (\texttt{next } Q)), s] = \begin{cases} \texttt{TRUE} & \text{if } s \stackrel{*}{\models} P \text{ and } s \text{ is closed} \\ Q & \text{if } s \stackrel{*}{\models} P \text{ and } s \text{ is not closed} \\ \texttt{FALSE} & \text{otherwise} \end{cases}$$

$$\mathcal{F}[(P \texttt{ or } (\texttt{next } Q)), s] = \begin{cases} \texttt{TRUE} & \text{if } s \stackrel{*}{\models} P \text{ or } s \text{ is closed} \\ Q & \text{otherwise} \end{cases}$$

$$\mathcal{F}[(P \texttt{ or } (\texttt{strong-next } Q)), s] = \begin{cases} \texttt{TRUE} & \text{if } s \stackrel{*}{\models} P \\ Q & \text{if } s \stackrel{*}{\not\models} P \text{ and } s \text{ is not closed} \\ \texttt{FALSE} & \text{if } s \stackrel{*}{\not\models} P \text{ and } s \text{ is closed} \end{cases}$$

So far, we have defined formal tools that are used to provide a formal definition of the incremental model checking algorithm. Next, we describe the model–checking algorithm, beginning with the procedure that checks whether a state is a model of a state formula and computes refinement conditions.

The function $\Psi : (\varphi, s) \rightarrow (v, c)$ maps an admissible state formula φ and a state s into a pair (v, c) where $v \in \{\texttt{T}, \texttt{F}\}$ and c is a necessary and sufficient refinement condition (possibly C_{true} or $\neg C_{\text{true}}$) obtained via the condition generator (see the interpretation structure in section 5.3.3). The function Ψ is defined as follows:

$$\Psi[p, s] = (\texttt{T}, C) \iff s \stackrel{*}{\models} p \text{ and } \mathcal{M}(C, s) \models p \text{ and } \mathcal{M}(\neg C, s) \not\models p, \text{ and}$$
$$\Psi[p, s] = (\texttt{F}, \neg C_{\text{true}}) \iff s \stackrel{*}{\not\models} p$$

The function Π is used to evaluate a progressed path formulæ(*i.e.* Formulæ that are the output of Δ). Π is called an *extended propositional interpretation* since it "short–circuits" the temporal operators in the progressed formula (that is, the sub–formulæ whose top–level operator is next or strong-next). This function determines whether the present component of a formula is true, conditionally true, or false with respect to a state and whether the behavior consisting of that state only is sufficiently determined with respect to the formula. In addition, Π also computes the future component of the formula.

$\Pi : (\varphi, s) \rightarrow (v, c, \varphi')$ maps an admissible progressed formula φ and a state s into a triple $(v, c, \varphi') = (\Pi_v[\varphi, s], \Pi_c[\varphi, s], \Pi_f[\varphi, s])$ where $v \in \{\mathtt{T}, \mathtt{F}, \mathtt{U}\}$, c is a refinement condition and φ' is the future component of φ:

φ	$\Pi_v[\varphi, s]$	$\Pi_c[\varphi, s]$
proposition	$\Psi_v[\varphi, s]$	$\Psi_c[\varphi, s]$
(`not` p)	$\begin{cases} \mathtt{T} & \text{if } \Pi_v[p, s] = \mathtt{F} \\ \mathtt{F} & \text{otherwise} \end{cases}$	$\begin{cases} C_{\text{true}} & \text{when } \Pi_v[\varphi, s] = \mathtt{T}; \\ \neg C_{\text{true}} & \text{when } \Pi_v[\varphi, s] = \mathtt{F}; \end{cases}$
(`next` ϕ)	$\begin{cases} \mathtt{T} & \text{if } <s> \text{ is closed} \\ \mathtt{U} & \text{otherwise} \end{cases}$	C_{true}
(`strong-next` ϕ)	$\begin{cases} \mathtt{F} & \text{if } <s> \text{ is closed} \\ \mathtt{U} & \text{otherwise} \end{cases}$	$\begin{cases} \neg C_{\text{true}} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ C_{\text{true}} & \text{otherwise} \end{cases}$
$\bigwedge \phi_j$	$\begin{cases} \mathtt{F} & \text{if } \exists j : \Pi_v[\phi_j, s] = \mathtt{F} \\ \mathtt{T} & \text{if } \forall j : \Pi_v[\phi_j, s] = \mathtt{T} \\ \mathtt{U} & \text{otherwise} \end{cases}$	$\begin{cases} \neg C_{\text{true}} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \bigwedge \Pi_c[\phi_j, s] & \text{otherwise} \end{cases}$
$\bigvee \tau_j$	$\begin{cases} \mathtt{F} & \text{if } \forall j : \Pi_v[\tau_j, s] = \mathtt{F} \\ \mathtt{T} & \text{if } \exists j : \Pi_v[\phi_j, s] = \mathtt{T} \\ \mathtt{U} & \text{otherwise} \end{cases}$	$\begin{cases} \neg C_{\text{true}} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \bigvee \Pi_c[\tau_j, s] & \text{otherwise} \end{cases}$

φ	$\Pi_f[\varphi, s]$
proposition	$\begin{cases} \mathtt{TRUE} & \text{if } \Pi_v[\varphi, s] = \mathtt{T} \\ \mathtt{FALSE} & \text{otherwise} \end{cases}$
(`not` p)	$\begin{cases} \mathtt{TRUE} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \mathtt{FALSE} & \text{otherwise} \end{cases}$
(`next` ϕ)	$\begin{cases} \mathtt{TRUE} & \text{if } \Pi_v[\varphi, s] = \mathtt{T} \\ \phi & \text{otherwise} \end{cases}$
(`strong-next` ϕ)	$\begin{cases} \mathtt{FALSE} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \phi & \text{otherwise} \end{cases}$
$\bigwedge \phi_j$	$\begin{cases} \mathtt{TRUE} & \text{if } \Pi_v[\varphi, s] = \mathtt{T} \\ \mathtt{FALSE} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \bigwedge \Pi_f[\phi_j, s] & \text{otherwise} \end{cases}$
$\bigvee \tau_j$	$\begin{cases} \mathtt{TRUE} & \text{if } \Pi_v[\varphi, s] = \mathtt{T} \\ \mathtt{FALSE} & \text{if } \Pi_v[\varphi, s] = \mathtt{F} \\ \bigvee \Pi_f[\tau_j, s] & \text{otherwise} \end{cases}$

Notice that since φ is in positive normal form, when $\varphi =$ (`not` p), p cannot be potentially ambiguous, and therefore $\Pi_f[\varphi, s]$ is a trivial refinement condition.

Given a behavior $b = <s_0, \ldots, s_n>$ and an admissible formula φ, the *evalu-*

ation sequence is defined as:

$$\begin{aligned}
\varphi_0 &= \Delta[\varphi] \\
v_i &= \Pi_v[\varphi_i, s_i] \\
c_i &= \Pi_c[\varphi_i, s_i] \\
\varphi_{i+1} &= \Delta[\Pi_f[\varphi_i, s_i]].
\end{aligned}$$

The following lemma characterizes useful properties of the evaluation sequence.

Lemma 5.4 (Properties of the evaluation sequence)

$$b^i \models \varphi_i \quad \Longrightarrow \quad b^{i+1} \models \varphi_{i+1} \tag{5.1}$$

$$b^i \stackrel{*}{\models} \varphi_i \quad \Longleftrightarrow \quad b^{i+1} \stackrel{*}{\models} \varphi_{i+1} \tag{5.2}$$

$$b^{(i,j)} \triangleright \varphi_i \quad \Longleftrightarrow \quad b^{(i+1,j)} \triangleright \varphi_{i+1} \tag{5.3}$$

$$b^{(i,i)} \not\triangleright \varphi_i \quad \Longleftrightarrow \quad v_i = \mathtt{U} \tag{5.4}$$

$$b^{(i,i)} \triangleright \varphi_i \wedge s_i \stackrel{*}{\models} \varphi_i \quad \Longleftrightarrow \quad v_i = \mathtt{T} \tag{5.5}$$

$$b^{(i,i)} \triangleright \varphi_i \wedge s_i \not\stackrel{*}{\models} \varphi_i \quad \Longleftrightarrow \quad v_i = \mathtt{F} \tag{5.6}$$

$$s \models \mathcal{P}[\varphi, s] \quad \Longleftrightarrow \quad \Pi_c[\varphi, s] \equiv C_{\text{true}} \tag{5.7}$$

$$\mathcal{F}[\varphi, s] \quad \equiv \quad \Pi_f[\varphi, s] \tag{5.8}$$

The proof follows from the definitions of $\mathcal{P}$, $\mathcal{F}$, and Π.

5.4.3 Temporal Logic Guide algorithm

TL–Guide takes as input a non empty QSIM behavior b and an admissible formula φ representing the user–defined temporal logic constraints. TL–Guide incrementally steps through the behavior calling, TLG–1 as each new state is examined. The process terminates once it can be determined whether or not b models φ.

TL–Guide(b, φ) :
  $i := 0$;
  repeat
    $v :=$TLG–1$(b^{(0,i)}, \varphi)$;
    $i := i + 1$;
  until $i = |b|$ **or** v

The core of TL–Guide is the procedure TLG–1[6] TLG–1 takes as input a non–empty behavior and an admissible formula φ. It uses Π to evaluate the progressed version of φ on the last state of the behavior and applies conditions when appropriate.

TLG–1($<s_0, \ldots, s_i>, \varphi$) :
  done := false;
  if $i = 0$
    then $f := \Delta[\varphi]$
    else $f := s_{i-1}$.future;
  $(v, c, g) := \Pi[f, s_i]$;
  case v
    `T`: *apply–conditions*(c, s_i); done := true;
    `F`: *refute*$(<s_0, \ldots, s_i>)$; done := true;
    `U`: *apply–conditions*(c, s_i);
  endcase;
  if $v =$ `U` **then** s_i.future := $\Delta[g]$;
  return(done)

The procedure *apply–conditions* applies refinement conditions to a state while *refute* marks a behavior as inconsistent and removes it from the simulation agenda.

The TL–Guide algorithm invokes Π and Δ to compute the evaluation sequence of the behavior $b = <s_0, \ldots, s_i>$ with respect to φ. A progressed version of the formula is stored on the final state for use in future calls to the function. Only the last two states are used in a single call to the function.

A problem arises in this algorithm when a term in a progressed formula leads to refinement conditions whose application must be delayed until other terms in the formula are resolved into a definite truth value (*i.e.* `T` or `F`). The following example demonstrates this problem. Suppose $\varphi =$ (P `or` (`next` ϕ)) where $P =$`(value-<= V .3)`. Assume that P is conditionally true for a state s. Thus, $<s> \triangleright \varphi$ and $<s> \not\triangleright$ (`next` ϕ); notice that while $<s> \stackrel{?}{\models} \varphi$, this may not necessarily hold in the future. In fact, suppose the subsequent state is s':

- if $<s'> \models \phi$ then $<s, s'> \models \varphi$
- if $<s'> \not\models \phi$ then $<s, s'> \stackrel{?}{\models} \varphi$
- if $<s'> \not\triangleright \phi$ then the algorithm must continue.

[6]TeQSIM implements TL–Guide by interleaving simulation steps (*i.e.* extending a behavior with a new state) with calls to TLG–1 as each state is created.

In this case, extending the behavior with a new state did not change the status of $\rhd\varphi$, but it did change the status of $\overset{*}{\models}\varphi$, resolving it into a stricter entailment relation.

The cause of this problem is the interaction between the local nature of Π and the concept of conditional entailment. Application of refinement conditions when ambiguity exists between $\models$ and $\overset{?}{\models}$ results in the unnecessary elimination of real–valued trajectories that are potentially consistent. Since QSIM is unable to retract quantiative information once it is applied to a state, the application of the refinement conditions must be delayed until the sub–formula in the non–ambiguous branch of the disjunction becomes sufficiently determined.

The following necessary and sufficient conditions characterize the situation described above in the evaluation of a progressed formula $\varphi = \tau \vee \tau_1 \vee \ldots \tau_k$ for a behavior b:

(i) $b \overset{?}{\models} \mathcal{P}[\tau, b(0)]$
(ii) the set $\{\tau_j : b \not\rhd \tau_j\}$ is not empty
(iii) none of the other terms τ_h are such that $b \overset{*}{\models} \tau_h$ and $b \rhd \tau_h$.

When these conditions are satisfied, then:

$$b \overset{?}{\models} \mathcal{P}[\varphi, b(0)] \not\Rightarrow b \overset{?}{\models} \varphi$$

which says that refinement conditions that are sufficient and necessary for the present component may not be necessary for the entire formula. When this situation occurs, φ is called a *condition delaying expression*, the refinement condition generated is called a *delayed condition*, and the τ_j mentioned in condition (ii) are called the *triggers* for the delayed condition.

After extending b into b', the delayed condition may be resolved. Thus if a trigger becomes definitely true, then its delayed condition is no longer necessary any more (that is if $b' \rhd \tau_j \wedge b' \models \tau_j$ then condition (iii) is no longer true and we get $b' \rhd \varphi \wedge b' \models \varphi$). On the other hand, if a trigger becomes definitely false but (ii) still holds (that is, there are other triggers), nothing has changed for the delayed condition. If (ii) does not hold, however, then the delayed condition becomes necessary ($b' \rhd \tau_j \wedge b' \overset{*}{\not\models} \tau_j$ and not (ii) implies $b' \overset{?}{\models} \varphi \wedge b' \rhd \varphi$).

Once the model–checking algorithm detects the problem describe above, it performs the following steps:

1. if the formula φ is sufficiently determined, then model checking — which would normally cease — must continue evaluating the triggers;

2. delayed conditions must be stored until their triggers are resolved and either condition (ii) or (iii) does not hold any more;
3. if a trigger evaluates to true, then its delayed condition must be dropped;
4. if a trigger evaluates to false, its delayed condition must be applied unless there are other triggers;
5. multiple delayed conditions may need to be stored as the algorithm progresses through the behavior.

The complete solution is based on an intertwined implementation of Π and TLG–1, allowing the problem to be detected and handled efficiently.

5.5 Results

TeQSIM eliminates all behaviors that are inconsistent with either the structural constraints or the trajectory constraints. While QSIM can only be proven to be sound with respect to the structural constraints, TeQSIM can be proven to be both sound and complete with respect to the temporal logic expressions specified within the trajectory constraints. The following theorem characterizes the correctness and completeness of the TL–Guide algorithm. The proof of the theorem is contained in appendix C.

Theorem 5.1 (TL–Guide is sound and complete) *Given a QSIM behavior b and an admissible formula φ then TL–Guide:*

1. *refutes b if and only if b is* sufficiently determined *($b \rhd \varphi$) and for all full-path extensions $\widehat{b}$ of b, $\widehat{b}$ does not model φ ($\widehat{b} \not\models \varphi$).*
2. *retains b without modification if and only if*
   (a) *b is sufficiently determined ($b \rhd \varphi$) and b models φ ($b \models \varphi$); or*
   (b) *b is not sufficiently determined ($b \not\rhd \varphi$) and there is no necessary condition C for refining b into a model for φ (i.e. $\not\exists C \neq C_{\text{true}}$ such that if $b'' = \mathcal{M}(\neg C, b)$ then for all full-path extensions $\widehat{b''}$: $\widehat{b''} \not\models \varphi$).*
3. *replaces b with b' if and only if*
   (a) *b is sufficiently determined ($b \overset{?}{\models} \varphi$), b conditionally models φ ($b \rhd \varphi$), and there exists a necessary refinement condition C such that applying the refinement condition to b generates a new behavior b' such that b' models φ ($\exists C : C \neq C_{\text{true}}$ such that $b' = \mathcal{M}(C, b) \models \varphi$ and if there exists b'' such that $b'' = \mathcal{M}(\neg C, b)$ then for all full-path extensions $\widehat{b''}$: $\widehat{b''} \not\models \varphi$); or*

(b) *b is not sufficiently determined* ($b \not\models \varphi$), *but there exists a necessary refinement condition* C *(i.e.* $\exists C : C \neq C_{\text{true}}$ :: *if* $b'' = \mathcal{M}(\neg C, b)$ *then for all full–path extensions* $\widehat{b''}$: $\widehat{b''} \not\models \varphi$) *and* $b' = \mathcal{M}(C, b)$.

5.6 Problem Solving with Trajectory Information

TeQSIM's ability to specify trajectory information, discontinuous changes and external events is a powerful extension to qualitative simulation; these capabilities broaden the range of relevant dynamical system theory problems that can be simulated with qualitative simulation techniques. In particular, trajectory specifications may refer to dependent or independent variables of a piecewise–continuous dynamical system described by ordinary differential equations. In the following classes of dynamical system theory problems, the role that the constrained variables play within the model make them important test cases for TeQSIM. Both classes can be simulated effectively only by incorporating trajectory information into a qualitative simulation.

Simulation of non–autonomous systems: Trajectory specifications for input variables describe their time–varying behavior and drive the simulation. The specification of discontinuous changes in the input variables enables the user to model external actions that occur on faster time–scales than the simulation. This is useful when modeling

- situations where limited knowledge is available regarding the transient period during which the action occurs,
- situations where the transient is by itself uninteresting, or
- a discrete event system (e.g. a digital controller) acting upon an otherwise continuous plant.

Solution of boundary condition problems: Trajectory constraints can be used to specify information about the behavior of dependent variables at various time points, thereby restricting the space of solutions of the differential equation. Available boundary condition information may include:

- knowledge about intermediate or final states of the system, or
- knowledge about observed variables.

Trajectory information combined with qualitative simulation, unifies these two classes, allowing TeQSIM to solve problems that combine features of both classes. Table 1 outlines some of the tasks that we have examined that benefit from TeQSIM's incorporation oftrajectory information.

5.7 Related Work

TeQSIM smoothly integrates temporal logic model checking into the qualitative simulation process. The ideas and techniques involved are related to work in the fields of qualitative reasoning and temporal logic.

5.7.1 Qualitative reasoning

The incorporation of trajectory information into a qualitative simulation has not been extensively explored in the literature. DeCoste (1994) introduces *sufficient discriminatory envisionments* to determine whether a goal region is possible, impossible, or inevitable from each state of the space. This is accomplished by generating the simplest state description that is sufficient for inferring these discriminations. Though similar in spirit, our work is: (i) more general, because TeQSIM enables the user to address a wide category of problems, not limited to determining reachability of a state; (ii) semantically well–founded because of its foundation in temporal logic and (iii) formally proved to provide guaranteed results.

Washio and Kitamura (1995) present a technique that uses temporal logic to perform a *history–oriented envisionment* to filter predictions. TeQSIM, within a more rigorously formalized framework, provides a more–expressive language, it refines behaviors as opposed to just filtering them, and it incorporates discontinuous changes into behaviors.

The integration of temporal logic model checking and qualitative simulation was initially investigated by Kuipers and Shults (1994, 1997). These authors a branching–time temporal logic to prove properties about continuous systems by testing the entire behavioral description against a temporal logic expression. The appropriate truth value depends upon whether or not the description satisfies the expression. Our work focuses on *constraining* the simulation as opposed to *querying* it after the simulation is completed.

Forbus (1989) explicitly introduces the concept of discrete action, with pre and post–conditions, in the *action–augmented envisionment.* The purely qualitative total envisionment so produced includes all possible instantiations of known actions. Forbus allows only instantaneous actions and adopts heuristic criteria (based on minimality of the change in the description) to handle discontinuities. No provision is made to handle quantitative information, nor to prove correctness of the

Goal oriented simulation	Input:	A description of the desired (or undesired) behavior.
	Output:	Behaviors that are "consistent" with the goal specification.
	Relation:	By specifying the desired behavior and invoking TeQSIM, a modeler can gain an understanding of these behaviors. Specifying undesired behaviors allows the modeler to determine *whether* these behaviors exist and to decide how they can be avoided when they do exist.
Analysis of discrete actions	Input:	A list of discontinuous changes resulting from actions and trajectory constraints specifying when the actions can be performed.
	Output:	The behaviors resulting from the actions.
	Relation:	Multiple simulations can be performed to evaluate alternative courses of actions.
Analysis of control responses	Input:	Trajectory constraints specifying the desired closed–loop behavior of the system and an environmental perturbation.
	Output:	A description of a controller response to the perturbations that is necessary for achieving the closed–loop behavior specified as input.
	Relation:	All potential control responses satisfying the closed–loop behavior are included within the behavioral description. If there is no consistent behavior, then the desired closed–loop behavior cannot be achieved by any controller in response to the perturbation.
Parameter identification	Input:	A model containing a partial description of a controller and trajectory constraints describing the desired behavior of the controller and a perturbation.
	Output:	Behaviors containing quantitative bounds on the unknown controller parameters.
	Relation:	The range of potential values for unknown controller parameters that are necessary to achieve the desired behavior.
Analysis of feedback control laws	Input:	Specification of a control law via trajectory constraints.
	Output:	A description of the resulting closed–loop behaviors.
	Relation:	The control law is specified by relating the value of the monitored variable with the required value, or *trend*, of the control variable.

Table 5.3: Tasks to which TeQSIM has been applied.

discontinuity–handling mechanism.

Discontinuities have also been investigated by Nishida and Doshita (1987), who propose two methods for handling discontinuities caused by external agents or generated autonomously within a system (*e.g.* change in operating regime): (i) approximating a discontinuous change by a quick continuous change and (ii) introducing "mythical states" to describe how a system is supposed to evolve during a discontinuous change. The former requires complex machinery to compute the limit of the quick change, while the latter is based on heuristic criteria for selecting appropriate states. Both methods are interesting, but their effectiveness and formal properties are difficult to ascertain.

Iwasaki and colleagues (1995) discuss a semantics for discontinuous changes that is more appropriate when dealing with hybrid systems. Their work leads to the adoption of a complex non–standard analysis semantics for reals and the development of a mechanism similar to continuity relaxation but requiring user–supplied frame axioms.

5.7.2 Temporal logic

The trajectory–specification language described here is similar to other formalisms for specifying temporal constraints. Our language is strictly more expressive than both Allen's *interval algebra* (Allen, 1984) and Dechter, Meiri and Pearl's *temporal constraint networks* (Dechter, Meiri, & Pearl, 1991) due to its ability to arbitrarily compose path formulæ using temporal operators. All of the relations defined by Allen can be expressed using the composition of temporal operators in addition to more complex relations. The usage of the language in TeQSIM is also different: instead of asserting temporal constraints in a database of assertions and querying whether certain combinations of facts are consistent, TeQSIM checks that a database of temporally related facts generated by QSIM satisfy a set of temporal logic constraints.

Bhat *et al.* (1995) present an algorithm for CTL* model checking that uses a set of inference rules that compute formula progression in a similar to TeQSIM. Our algorithm differs from theirs in two ways. First, Bhat *et al.* do not deal with refinement, and this has the consequence that their algorithm does not have to handle (delayed or not) refinement conditions. Second, their algorithm drives the extension of paths, in the sense that the successor of a state is actually generated only when all formulæ have been progressed to `next` operators. TL–Guide, on the other hand, is embedded in the QSIM architecture which drives the model checking activity.

Bacchus and Kabanza utilize temporal logic within the domains of forward chaining planning (Baccus & Kabanza, 1995) and reactive planning (Bacchus & Kabanza, 1996). They use temporal logic to constrain the path selected to reach a goal state and to express search control information. One of the primary differences between their work and ours is that planning is generally concerned with finding a single solution that satisfies the constraints, whereas qualitative simulation must generate all solution paths. Bacchus and Kabanza also use a similar incremental model–checking algorithm that uses formula progression.

5.8 Discussion and Future Work

TeQSIM defines a general methodology for incorporating arbitrary trajectory information into the qualitative simulation process. The incremental nature of the model–checking algorithm aggressively filters and refines behaviors as early as possible in order to minimize the complexity of the simulation. In terms of model checking complexity, (assuming that the most expensive operation is checking a proposition, *i.e.* computing Ψ), TL–Guide applies Π at most once on each state of the behavior. Π evaluates the present component of the formula, which is a boolean formula in disjunctive normal form. With a caching mechanism, this operation can be performed by evaluating each proposition at most once. Therefore, TL–Guide performs essentially $O(Nn)$ propositional checks when the temporal constraints contain N different atomic propositions and the behavior contains n states. Furthermore, in all the examples tested for this project, the practical time–complexity of a TeQSIM simulation is dominated by the qualitative simulation phase, not model checking. In fact, TeQSIM often decreases the complexity of the simulation by filtering behaviors that are not relevant to the current task.

The following extensions would increase the expressiveness of the trajectory specification language described here.

Limited first–order expressiveness – TeQSIM's temporal logic is limited to propositional expressions and is thus unable to quantify over attributes of states. Certain trajectory constraints require references to values in other states within the behavior. For example, the description of a decreasing oscillation requires the ability to compare the magnitude of a variable across states. A limited form of first–order temporal logic may provide a language that is sufficiently expressive for these concepts without incuring excessive computational complexity costs.

Metric temporal logic – TeQSIM behaviors are potentially infinite structures because of the introduction of landmarks during simulation. Deriving a definite answer for formulae such as `(eventually` p`)` is not always possible when potentially infinite behaviors are involved since it is always possible for p to occur in the future. Metric temporal logic (Alur & Henzinger, 1993) allows a horizon for a temporal logic expression to be defined, allowing statements like "within 50 seconds, the tank level reaches 70 inches." These statements are only expressible within our logic using an externally defined event. Extending the logic would give the modeler more flexibility to express relevant constraints.

Discontinuous change specification – In the current version of TeQSIM we provide very simple means for representing and reasoning about discontinuous changes. While sufficient for certain kinds of problems (*e.g.* driving the simulation by controlling the behavior of an exogenous variable), these are insufficient for others (*e.g.* identifying repetitive actions performed by a controller). We have considered two possible extensions to TeQSIM to address these issues:

- supporting more–complex relationships between the precondition and the effect using additional temporal logic operators. For example, the modeler may want to identify a sequence of states over which the action can be performed or express information about the *possibility* of a discontinuous change.
- allowing preconditions to be specified using an arbitrary temporal logic expression. This would extend the range of addressable feed–forward control problems.

Functional envelopes — The semi–quantitative reasoning (Berleant & Kuipers, 1988) part of TeQSIM uses interval bounds and static functional envelopes for monotonic functions to derive quantitative information about a behavior. NSIM (Kay & Kuipers, 1993) derives dynamic envelopes describing a variable's behavior with respect to time. Currently, only interval information can be specified by TeQSIM trajectory constraints. Extending the language to include information about temporal bounding envelopes would increase the precision of the solutions computed by TeQSIM.

5.9 Conclusions

Qualitative simulation and temporal logic provide two alternative formalisms for reasoning about change across time. TeQSIM integrates these two paradigms by incorporating trajectory information specified via temporal logic into the qualitative simulation process. Behaviors that do not model the set of temporal logic expressions are filtered during simulation. Numeric information specified within the TL expressions can be integrated into the simulation to provide a more precise numerical description for the behaviors which model these expressions. Not only does TeQSIM fundamentally extend the range of problems that can be addressed via qualitative simulation, it also provides the modeler with a powerful tool that can be used to control the simulation to address some of the complexity problems discussed presented in the rest of this dissertation.

The correctness of the TL–guide algorithm along with the correctness of QSIM guarantee that all possible trajectories of the modeled system compatible with the model, the initial state and the trajectory constraints are included in the generated behaviors. In addition, the completeness of TL–guide ensures that all behaviors generated by TeQSIM are potential models of the trajectory constraints specified by the modeler.

Chapter 6

Future Directions

The overall goal of our research has been to identify and address problems that have hindered the application of qualitative reasoning techniques to large, real–world problems. Two primary problems have been identified: the complexity of the simulation algorithm and ambiguity within the behavioral description.

In the past, much of the concern has focused on the complexity of both the simulation algorithm and the resulting behavioral description. The techniques presented here provide a scalable simulation algorithm that avoids irrelevant distinctions. By decomposing the model, DecSIM eliminates combinatoric branching due to the complete temporal ordering of unrelated events while chatter abstraction eliminates irrelevant distinctions that result in an infinite simulation and intractable branching when a variable's derivative is unconstrained. These techniques, coupled with existing methods for querying and analyzing the results of a simulation,[1] provide the modeler with a flexible qualitative simulation algorithm that provides various methods to analyze the results of a simulation. While these techniques do not completely solve the problem of a complex simulation and/or behavioral description,[2] they greatly reduce the impact of these problems, thus focusing attention on the degree to which the information contained within the behavioral description can be used to solve relevant problems.

An ambiguous behavioral description describing both desired and undesired behaviors results when there is insufficient information within the model to suf-

[1]Temporal–logic (Shults & Kuipers, 1997) can be used to query the results of a simulation to extract information while post–processing abstraction techniques (Clancy & Kuipers, 1993; Clancy et al., 1997) can reduce the complexity of the behavioral description by focusing on specific distinctions.

[2]Since qualitative simulation is inherently intractable, simulation complexity will always be an issue that must be addressed.

ficiently constrain the simulation. TeQSIM addresses the problem of ambiguous behavioral descriptions by providing the modeler with a declarative language for specifying behavioral constraints. While TeQSIM extends the range of problems that can be addressed using qualitative simulation, it does not completely address the core question that still must be answered:

> "Is the type of information traditionally contained within a qualitative or semi–quantitative model sufficient to infer interesting and novel results that can be used to effectively solve relevant real–world problems?"

Thus, the main question to be answered focuses on information content as opposed to computational complexity. We feel that this is the core issue that must be addressed in future research within the field of qualitative simulation. Furthermore, we feel that the best way to address this question is to integrate qualitative simulation techniques into a larger problem solving context to determine the extent to which these techniques can be used to solve realistic problems.

This chapter describes future directions for research along two dimensions:

(1) how qualitative simulation can be integrated with other techniques to provide a more comprehensive solution to the problem of using imprecise information to reason about the physical world for both common–sense reasoning and engineering problem solving applications, and

(2) what specific advances are required within the field of qualitative simulation for these techniques to be applied to a broader range of problems.

The next two sections discuss these two topics. In general, we focus our discussion on how qualitative simulation can be applied to solve problems; thus we avoid references to the benefits provided by the specific techniques described in this dissertation.

6.1 Using qualitative simulation within a larger problem solving context

Before discussing how qualitative simulation can be used in a larger, problem–solving context, we provide a brief discussion on how artificial intelligence techniques in general have been used in real–world applications. This discussion demonstrates the importance of hybrid solutions that integrate multiple inference techniques.

6.1.1 Using artificial intelligence to solve real–world problems

In many sub–fields within computer science, such as operating systems, compilers, databases, etc, the boundaries of the field are often determined by a moderately well–defined problem. For example, the field of databases is concerned with techniques for maintaining, storing, and accessing large amounts of data in an efficient manner. Defining the field of artificial intelligence (AI), however, has proven to be quite difficult since the problem being addressed is rather broad and often poorly defined. In any ten introductory AI books, one is likely to find at least five or six (if not ten) different definitions for the term *artificial intelligence*. Furthermore, the field has been sub–divided into disciplines that are defined either by a very broad, general problem (*e.g. machine learning*, *knowledge representation*, *natural language understanding*, and *planning*) or by a set of techniques (*e.g. neural networks* and *uncertain reasoning*). In either case, it is often difficult to tie artificial intelligence to a set of specific applications that can be addressed with a single technique.

In the 1970's and 1980's, a number of advances were made in various disciplines within AI in the development of general techniques for solving computationally hard problems. Machine learning developed algorithms for learning a compact, structured representation for the information contained in a large data set (Shavlik & Dietterich, 1990), planning and scheduling developed efficient algorithms for identifying an optimal sequence of actions to achieve a specified goal (Allen, Hendler, & Tate, 1990), and qualitative reasoning developed techniques for deriving an abstract behavioral description for an imprecise structural model of a dynamical system (Weld & de Kleer, 1990). These are just a few of the advances that were made over this period.

Translating these advances into compelling applications, however, proved difficult since many of these techniques did not provide a complete solution to a specific, real–world problem. Recently, however, many of these techniques have been integrated with other methods both from within the field of AI and from other disciplines resulting in impressive applications and lucrative businesses.

- NASA researchers have integrated planning, scheduling and qualitative reasoning techniques with a real–time executive to monitor, diagnose and control a deep space probe for the Deep Space One mission (Pell, Bernard, Chien, Gat, Muscettola, Nayak, Wagner, & Williams, 1997).

- SRI's Multiagent Planning Architecture (MPA) has been used to perform battle field planning for the Joint Forces Command and Control (JFCC). MPA integrates planning, scheduling and temporal reasoning agents into a flexible agent architecture (Wilkins, Meyers, desJardins, & Berry, 1997).

- The new field of knowledge discovery in databases (KDD) uses machine learning along with various statistical and data warehousing techniques to identify patterns and "discover knowledge" within large sets of unstructured data. KDD systems have proved useful for identifying consumer shopping patterns, detecting credit card fraud, and processing credit card applications (Fayyad, Piatetsky-Shapiro, Smyth, & Uthurusamy, 1996).

- I2 Technologies, Red Pepper software, and Magnugistics are just a few of the companies offering advanced planning and scheduling products within the burgeoning field of *enterprise resource planning* (ERP). These companies integrate planning and scheduling algorithms within a large, integrated software system to perform tasks such factory floor planning, job–shop scheduling, and raw materials distribution.

- Trilogy Corporation, a highly successful company recently featured on the cover of *Forbes Magazine*, uses constraint satisfaction techniques to perform product configuration for sales force automation.

These are just a few of the many real–world applications that utilize artificial intelligence techniques. All of these examples, however, integrate multiple techniques when solving the specified problem.

6.1.2 Applying qualitative simulation to real–world problem solving tasks

Qualitative simulation provides a set of techniques for deriving a behavioral description for an imprecisely defined, structural model of a dynamical system. These techniques are relevant both for common sense reasoning and engineering problem solving applications. By themselves, however, these techniques do not provide a complete solution to the problem of reasoning about dynamic change in the physical world. Instead, they are just part of the overall solution. The next two subsections describe how qualitative simulation can be integrated with other techniques to provide an overall solution to the general tasks of common–sense reasoning and engineering problem solving. To address the problem of common–sense reasoning, we propose integrating qualitative simulation with other AI techniques while for engineering problem solving we propose integration with methods currently used by engineers to reason about dynamical systems.

6.1.2.1 Common–sense reasoning

Common–sense reasoning about the physical world requires an understanding of how objects and forces interact with each other to perform common tasks that humans perform on a regular basis. Traditionally, common–sense reasoning uses a logic or frame–based representation to describe the entities and relationships within the domain of interest along with both general and special purpose inference techniques (Davis, 1990). Using a general purpose inference engine to reasoning about autonomous change, however, can be difficult due to the myriad of influences that may affect a dynamical system.

Qualitative simulation provides a powerful, special–purpose inference engine for reasoning about dynamic change within the world. In the TRIPEL system, Rickel and Porter (Rickel & Porter, 1997) demonstrate how qualitative simulation can be integrated with a large–scale knowledge–base to answer queries within the domain of plant physiology. TRIPEL uses automated model building techniques to generate a model at the appropriate level of detail for the given query from *model fragments* within the knowledge–base. By using qualitative simulation, TRIPEL is able to derive a rich behavioral description that can be used to answer a variety of questions.

TRIPEL demonstrates how qualitative simulation can be part of an overall solution to a common–sense reasoning task. TRIPEL, however, focuses on the task of automated model building; thus, it limits the degree of interaction between the knowledge–base and the qualitative simulator is limited to extracting information from the knowledge–base to build the model. In addition, the queries are restricted to questions requiring inferences that can be made via qualitative simulation.

A more comprehensive solution requires

- a flexible interface to the inference capabilities provided by the qualitative simulator so that these capabilities can be used in a variety of ways (*e.g.* a query about the relationships between variables when the system is in a steady–state would not require a complete dynamic simulation),
- additional techniques for controlling the simulation (*e.g.* goal directed simulation techniques to focus the simulation on the information required to answer a specific query),
- a declarative, logic–based language for incorporating additional information into the simulation,[3]

[3]TeQSIM already provides this capability; however, a more expressive language would provide

- automated tools for extracting information from the behavioral description and the ability to incorporate the results from a simulation back into the knowledge–base so that additional inference techniques can process the information,
- an executive control component that determines when to perform a qualitative simulation to address a given task and in what manner,
- an explanation facility to support and explain the inferences made during the simulation, and
- more sophisticated automated modeling techniques that can be used to build a component–based model and to refine the model as needed based upon the results of a preliminary simulation.

In general, the overall goal is to fully integrate the qualitative simulation algorithm into a more general–purpose, knowledge–based system that uses the qualitative simulation algorithm as a special purpose inference engine to solve certain specialized tasks. A system of this nature would be particularly useful for developing an advanced tutoring systems for one of the natural sciences. Integrating qualitative simulation into a knowledge–based system allows the tutoring system to answer a wide range of questions. In particular, qualitative simulation can be used to identify behaviors that are precluded due to the structural properties of the modeled system and to explain how the structural properties of the system affect the resulting behavior.

6.1.2.2 Engineering problem solving

In recent years, a large portion of the qualitative reasoning community has focused on developing techniques that can be applied to various engineering problem solving tasks such as monitoring, diagnosis, design and control (Iwasaki & Farquhar, 1996; Ironi, 1997). Qualitative and semi–quantitative simulation can be used in various ways to address these tasks. For example, when developing a robust controller, semi–quantitative simulation can be used to measure the response of the system to various perturbations. Alternatively, qualitative simulation can be used during the initial design phase to evaluate alternative designs with respect to a set of desired properties.

As mentioned above, however, qualitative simulation can only be used to solve part of the overall problem. As more information is required, qualitative

more flexibility when specifying information.

simulation and even semi–quantitative simulation often fail to provide the desired precision. A wide variety of quantitative techniques are commonly used by engineers to reason about the behavior of a dynamical system. For qualitative simulation to be relevant within a more general problem solving context, it must be fully integrated with these quantitative techniques so that the user can use all of the available information to reason at multiple levels of detail.

While semi–quantitative simulation integrates qualitative and quantitative techniques, it does this from a "QSIM–centric" perspective in which the quantitative inferences are driven and controlled by the *qualitative* simulation.[4] It is our view that qualitative and quantitative inference techniques must be integrated as equal partners such that neither system depends upon the other.[5] Integrating qualitative information with existing numeric techniques requires a generalized model representation language that exhibits the following features:

- the syntax of the representation language should be driven by the information being specified and not a specific inference algorithm (*e.g.* the current syntax used to specify QSIM models must be replaced with a more–standard, equation–based syntax that is more intuitive for an engineer to manipulate),
- the language should be compositional in nature such that individual components can be described independently and then combined as need to form larger components,[6]
- the language should support multiple levels of description so that all of the available information can be specified (*e.g.* at a minimum, the language should support structural, qualitative, semi–quantitative, and ordinary differential equations.[7]),
- the language should support structural as well as behavioral information about individual components, and

[4]Recent advances such as NSIM (Kay & Kuipers, 1993) can operate as a stand alone system; however, additional research is still required.

[5]An appropriate analogy would be an integrated software package such as Microsoft Office which provides a set of independent software applications that are able to interact with the other applications as needed.

[6]We avoid using the term *model fragment* since it is commonly used within the compositional modeling literature. Our view of a *component* is less granular that what is commonly thought of as a model fragment. Thus, an individual component might correspond to detailed model of an engine or even an entire boiler assembly.

[7]A *structural differential equation* (SDE) describes the basic structure of the system. For example, the SDE for a bathtub might take the form $\dot{x} = q - u * f(x)$ where x is the amount of water in the tub, q the inflow and u the drain opening.

- the language should support the explicit specification of assumptions about the structure and behavior of individual components.

These requirements are very similar to the features currently provided within the various compositional modeling languages that have been developed within the field of qualitative reasoning (Falkenhainer & Forbus, 1991; Farquhar, 1993; Farquhar, Iwasaki, Fikes, & Bobrow, 1996). These languages, however, have been developed as general purpose languages that can be used for both common–sense reasoning applications as well as engineering problem solving. It is our opinion that these two tasks are often very different and that to address engineering problem solving tasks, the syntax and structure of the language should be driven more by the tools that are currently used by engineers as opposed to existing AI techniques. Furthermore, the existing compositional modeling languages were designed to support automated model building techniques. While automated techniques for combining components may be required, we are not interested in completely automating this process, as we view the engineer as an integral part of the model building process.

Specifically, we are interested in exploring how QSIM and other qualitative inference techniques can be integrated with existing tools for

- algebraic manipulation (*e.g.* Mathematica)
- numerical simulation (*e.g.* MATLAB), and
- data acquisition (*e.g.* LabView)

providing a multi-purpose tool box for reasoning about the behavior of a dynamical system.

The generalized representation language is intended to support model refinement and exploration through the incremental specification of more detailed or alternative descriptions. It can the thought of as a "knowledge–base" describing the modeled system. Furthermore, the system would support the integration of information derived using multiple inference techniques. For example, results from a Monte Carlo simulation might be used to generate a qualitative behavioral description which could be queried using temporal logic. Alternatively, semi–quantitative simulation could be used to identify critical regions of the trajectory space that would be explored in detail using more precise quantitative techniques.

The overall goal of this proposal is to integrate qualitative simulation with existing quantitative techniques to provide a general–purpose tool box that can be used to address a variety of engineering problem solving tasks. Thus, engineers would be able to exploit the benefits provided by qualitative inference techniques within a familiar environment while avoiding many of the shortcomings encountered when using qualitative simulation by itself.

6.2 Advances within the field of qualitative simulation

This section provides a brief description of specific advances that are required within the field of qualitative simulation.

Identifying "likely" behaviors – The soundness guarantee provided by qualitative simulation ensures that the behavioral description contains all possible real–valued trajectories consistent with the qualitative model. The behavioral description, however, does not provide any information regarding the likelihood of a given behavior. This limitation can be problematic since the behavioral description may include a number of undesirable behaviors that are highly unlikely to occur. While absolute guarantees may be necessary in certain limited applications such as designing a nuclear power plant, in most applications, it is sufficient to provide probabilistic guarantees. Furthermore, this information could be used to reduce the complexity of the simulation by pruning unlikely behaviors.

To generate a behavioral description containing information about the likelihood of a given behavior , additional information must be incorporated into the model. Probability information could be incorporated into the model in various ways.

1. The likelihood of a given event or a sequence of events could be specified by the modeler or derived through an analysis of historical information. This information could be used to determine the likelihood of a dynamic behavior.
2. If qualitative simulation is used for diagnosis, the likelihood of a given failure mode within specific components could be specified.
3. When using a semi–quantitative model, quantitative bounds and functional envelopes could be augmented with probability distributions.

Alternatively, probability information could be inferred using an auxiliary technique such as Monte Carlo simulation to determine the likelihood of a given behavior (Brajnik, 1997).

These ideas are very preliminary. One of the drawbacks of these ideas is that the probability information is incorporated directly into the qualitative simulation process. This is similar to how semi–quantitative simulation handles quantitative information. A more challenging, but potentially more promising approach would allow a qualitative simulation algorithm to communicate asynchronously with Bayesian network inference algorithm (Shafer & Pearl, 1990) such that each algorithm could run independent of the other. The two algorithms would exchange information, task each other as needed and refine their descriptions independently.

Incorporating additional information and specifying assumptions – The precision of the behavioral description is dependent upon the information contained within the model. TeQSIM extends the expressiveness of the modeling language; however, various types of information still cannot be represented within the QSIM framework. Ideally, you would like to provide the modeler with an expressive, high–level language for specifying additional information and explicitly representing assumptions to be applied during the simulation. For example, the modeler might be interested in specifying order–of-magnitude information or a default assumption such as: "assume that A is greater than B unless this leads to a contradiction." This language should also allow the modeler to eliminate certain distinctions and select specific behaviors of interest. For example, the modeler may be interested in eliminating behaviors in which two unrelated events occur simultaneously to reduce the complexity of the simulation. Note that this extension weakens the soundness guarantee; however, it does so in a controlled manner and provides the modeler with a more flexible tool.

Extracting information from a simulation – A qualitative behavioral description provides a rich source of information about the potential behavior of a dynamical system. As the size of the system grows, however, it becomes more difficult to understand and extract information from this description simply through a manual analysis. Instead, automated techniques are required for extracting information and answering questions. Shults and Kuipers (Shults & Kuipers, 1997) use temporal logic to query the results of the simulation; however, the expressiveness of the language is limited to propositional expressions. Thus, this language cannot be used to describe certain phenomenon such as an increasing or decreasing oscillation or to describe global properties of the entire behavioral description. A more expressive language is required both to query the results of a simulation and to represent the information extracted from the behavioral description in a logic–based language that can be processed by other inference techniques.

Simplifying the model building process – Building a qualitative model is a difficult process that often requires multiple iterations before the appropriate model is obtained. While a number of automated modeling techniques have been developed (Rickel & Porter, 1997; Nayak, 1992), these techniques do not provide the degree of control that may be required when developing a detailed model of a given system. The following extensions would simplify the model building process:

- More sophisticated explanation facilities are required to explain the results of the simulation and answer questions about why a given behavior may or may not be included. Clancy, Brajnik and Kay (Clancy et al., 1997) provides some preliminary explanation facilities for QSIM; however, additional research is still required.
- An equation–based modeling language that allows an engineer to specify relationships in a more intuitive manner is required.
- An automated model refinement procedure is required to analyze the results of the simulation and identify potential revisions to refine the behavioral description. This technique would help a modeler explore alternative modifications when refining a model.

Generating a finite behavioral description – The introduction of landmarks within the QSIM framework allows the representation of behaviors such as increasing and decreasing oscillations. However, the introduction of landmarks can also results in an infinite behavioral description. Ideally, QSIM would generate a closed–form behavioral description representing the asymptotic behavior of the system. Weld (Weld, 1986) uses aggregation to generate such a description for a single behavior. These ideas would need to be extended to identify patterns within the behavioral description and to develop a language for representing these patterns using a finite representation.

6.3 Conclusions

In this chapter, we described some preliminary ideas for the integration of qualitative simulation with other techniques to provide a more comprehensive problem–solving engine. It is our opinion, that research of this nature is required to determine the relevant contributions provided by qualitative simulation and to identify new directions for research within the field.

Chapter 7

Conclusions

Various research efforts (Williams & Nayak, 1996; Shimomura, Ogawa, Tanigawa, Umeda, & Tomiyama, 1996; Forbus, 1996) have recently begun to field applications that utilize qualitative reasoning techniques to perform tasks such as monitoring, diagnosis, control and tutoring. In general, however, these applications do not utilize sophisticated qualitative simulation techniques because of the complexity of the simulation process and the cost of building large–scale models. Instead, they use either quantitative techniques or causal dependency graphs to reason about the behavior of the system.

Qualitative simulation provides a powerful tool for reasoning about the behavior of an imprecisely defined dynamical system but its application to real–world problems has been severely limited by several practical limitations.

(1) Traditionally, simulation is performed at a single level of detail, highlighting a fixed set of distinctions. Often, irrelevant distinctions lead to an intractable simulation for larger, more realistic models.

(2) The complexity of the behavioral description often makes it difficult to extract information from the behavioral description and evaluate the results of the simulation.

(3) Developing qualitative models is often a difficult process. Phenomena such as chatter often complicate this process, thus discouraging an individual attempting to develop a model.

(4) The expressiveness of the qualitative modeling language of is restricted to a fixed set of structural constraints. Representation of information that does not fit into this paradigm requires extensions to the simulation algorithm.

My dissertation directly addresses all four of these problems by providing techniques that reduce the complexity of a qualitative simulation and extend the expressiveness of the modeling language. Eliminating exponential branching due to irrelevant distinctions enables the development of larger, more interesting models. Furthermore, as these distinctions are eliminated from the behavioral description, the process of developing a qualitative model is simplified because it is easier to evaluate the results of the simulation.

Each of the three techniques proposed here, independently address a specific problem that has hindered the application of qualitative simulation techniques.

7.0.1 DecSIM

The ability of a composite, state–based representation to provide a compact representation of a potentially exponential solution space is inherently limited. The abstract representation used by qualitative simulation to describe a dynamical system is inherently unconstrained. Thus, for larger, more interesting systems to be simulated, an alternative representation is required.

DecSIM solves this problem by providing an alternative representation that partitions the variables into components. By sub-dividing the problem, DecSIM can exponentially reduce the size of the solution space. DecSIM implicitly represents the set of all solutions by maintaining links between the solutions to the sub–problems in order to represent constraints on how the solutions can be combined. DecSIM uses these constraints to ensure that each solution to a sub–problem corresponds to a complete solution.

By decomposing a model, DecSIM can exponentially speed–up the simulation for many problems, while generating a behavioral representation that is guaranteed to be equivalent to the representation provided by a standard QSIM simulation. Moreover, the benefits of a DecSIM simulation increase with the size of the model and as the variables become more loosely constrained.

7.0.2 Chatter abstraction techniques

Chatter branching is a common and major source of irrelevant distinctions; it leads to an infinite behavioral description and makes the results of the simulation unusable. Chatter occurs when the system enters an unconstrained region of the trajectory space. The two automated techniques proposed in this thesis eliminate chatter through abstraction. Both identify the unconstrained regions of the trajectory space and abstract them into a single qualitative state in the behavioral description. Chatter box abstraction leverages the inference capabilities of the simulation algorithm while dynamic chatter abstraction provides a more–efficient solution by avoiding the need to explore all possible paths through the chattering region. Both techniques have been shown to eliminate all instances of chatter without over–abstraction. By solving the problem of chatter, my work eliminates one of the major road blocks that have hindered the application of qualitative simulation techniques to larger, more–realistic systems.

7.0.3 TeQSIM

Qualitative simulation and temporal logic are two alternative formalisms for reasoning about change across time. TeQSIM integrates these two paradigms by smoothly incorporating trajectory information, specified via temporal logic, into the qualitative simulation process. During simulation, TeQSIM filters behaviors that do not model the set of temporal logic expressions and integrates numeric information specified within the TL expressions into the simulation to provide a more precise numerical description for the behaviors which model these expressions. Not only does TeQSIM fundamentally extend the range of problems that can be addressed via qualitative simulation, but it also provides the modeler with a powerful tool that can be used to control the simulation, allowing him to address some of the inherent complexity problems of qualitative simulation.

7.0.4 Concluding Remarks

The three techniques presented here, DecSIM, chatter abstraction, and TeQSIM, address two of the primary problems that have hindered the application of qualitative simulation techniques within real–world problem solving scenarios: simulation complexity and ambiguity within the behavioral description. The resulting qualitative simulation algorithm provides a powerful tool for reasoning about the behavior of imprecisely defined dynamical system enabling other researchers to explore how these techniques can be applied to address novel and interesting problems.

Appendix A

DecSIM Theorems, Lemmas and Proofs

Lemma A.1 (Component edge lemma) *For each edge within the component graph such that* $A \overset{v_{ab}}{\to} B$, *the* component edge predicate $M_{A \overset{v_{ab}}{\to} B}(a_i, b_j)$ *is true if and only if the following statements are satisfied:*

(1) $a_i =_{v_{ab}} b_j$,

(2) a_i *is an initial state iff* b_j *is an initial state,*

(3) If a_i *and* b_j *are not initial states,* $(M_{A \overset{v_{ab}}{\to} B}(a_{i-1}, b_{j-1}) \vee (M_{A \overset{v_{ab}}{\to} B}(a_i, b_{j-1}) \vee (M_{A \overset{v_{ab}}{\to} B}(a_{i-1}, b_j)$

Proof: The view and guide links are defined such that the view mapping is many–to–one while the guide mapping is one–to–many. Thus, $M_{A \overset{v_{ab}}{\to} B}(a_i, b_j)$ is true if and only if both a_i and b_j map to the same state in the view/guide tree. Furthermore, this state describes the variables v_{ab}. This will be used throughout the proof.

If $M_{A \overset{v_{ab}}{\to} B}(a_i, b_j) \Rightarrow$ *conditions 1-3* – Since both a_i and b_j map to the same state describing v_{ab}, $a_i =_{v_{ab}} b_j$. Furthermore, both mapping functions only map initial states to initial states.

According to the definition of the $\mathit{ViewedBy}_{v_{ab}}$ function, if a_i maps to a state s then a_{i-1} must either map to s or to its predecessor. A similar condition exists for the *GuideOf* mapping function. By composing these definitions either the predecessors of a_i and b_j map to each other or one of them maps to the predecessor of the other.

If conditions 1-3 $\Rightarrow M_{A \overset{v_{ab}}{\to} B}(a_i, b_j)$ – Once again, demonstrating this is simply a composition of the definition of the $\mathit{ViewedBy}_{v_{ab}}$ and *GuideOf* functions.

□

Lemma A.2 *If $M_{AB}(s_i, s_j)$ and $M_{AB}(s_{i+1}, s_{j+1})$ and either s_i is a time–point state while s_j is a time–interval state or vice-a-versa then $M_{AB}(s_{i+1}, s_j)$ and $M_{AB}(s_i, s_{j+1})$ are also true.*

Proof: Two cases will be consisdered:

Case 1: Suppose s_i, s_j, s_{i+1} and s_{j+1} are all equivalent with respect to the shared variables. By lemma 3.4, $M_{AB}(s_{i+1}, s_j)$ and $M_{AB}(s_i, s_{j+1})$ are true.

Case 2: Suppose s_i, s_j, s_{i+1} and s_{j+1} are not equivalent with respect to the shared variables. Then a transition must occur in a shared variable from s_i to s_{i+1} and an identical transtion must occur from s_j to s_{j+1}. However, the QSIM transition table does not allow for a transition that is valid both from a time–point to a time–interval as well as from a time–interval to a time–point. Thus, this is a contradiction and therefore the assumption is false.

Therefore, $M_{AB}(s_{i+1}, s_j)$ and $M_{AB}(s_i, s_{j+1})$ are true. □

Theorem A.1 (Compatible behavior theorem) *A given pair of component behavior segments $\{a_1, a_2, \ldots a_n\}$ and $\{b_1, b_2, \ldots b_m\}$ from components A and B respectively are compatible if and only if*

- *if $A \overset{v_{ab}}{\to} B$ is an edge in the component graph then $M_{A \overset{v_{ab}}{\to} B}(a_n, b_m)$ is true, and*
- *if $B \overset{v_{ba}}{\to} A$ is an edge in the component graph then $M_{B \overset{v_{ba}}{\to} A}(b_m, a_n)$ is true.*

Proof: Proof by induction on the total number N of the globally consistent states in both component behaviors.
Throughout the proof, we will use to symbol $M_{AB}(a_i, b_j)$ as shorthand for the expression if defined then $M_{A \overset{v_{ab}}{\to} B}(a_i, b_j)$ and $M_{B \overset{v_{ba}}{\to} A}(b_j, a_i)$ are true.

Base case: $N = 2$. Both component behaviors have a single, initial state. If the states are compatible, then by definition any shared variables must be equivalent and the predicates must be satisfied. Conversely, if the predicates are satisfied, then the shared variables are equivalent and since both stats are time–point state the behaviors are compatible.

Inductive step: Assume the lemma is true for $N = n + m - 1$.

If compatible $\Rightarrow M_{AB}$ – It is given that $\{a_1, a_2, \ldots a_n\}$ and $\{b_1, b_2, \ldots b_m\}$ are compatible. Let $(S_1, S_2, \ldots S_p)$ correspond to a composite behavior for these two behaviors where each S_i corresponds to a pair of states a_j and b_k that are combined to form the composite state. By the definition of a composite behavior $S_p = <a_n, b_m>$ Therefore, $a_n =_{v_{ab}} b_m$. S_{p-1} can correspond to either:

Case 1: $<a_{n-1}, b_m>$,
Case 2: $<a_n, b_{m-1}>$, or
Case 3: $<a_{n-1}, b_{m-1}>$.

Case 1: $S_{p-1} = <a_{n-1}, b_m>$ The behaviors terminating in a_{n-1} and b_m are compatible. Thus, by the inductive hypothesis, $M_{AB}(a_{n-1}, b_m)$. By lemma 3.4, since $M_{AB}(a_{n-1}, b_m)$ and $a_n =_{v_{ab}} b_m$, then $M_{AB}(a_n, b_m)$.

Cases 2 and 3: The argument is identical to Case 1 since lemma 3.4 accounts for all three cases in the same lemma.

If $M_{AB} \Rightarrow$ *compatible* – It is given that $M_{AB}(a_n, b_m)$ is true. By lemma 3.4, $a_n =_{v_{ab}} b_m$ and either

Case 1: $(M_{AB}(a_{n-1}, b_{m-1})$,or
Case 2: $(M_{AB}(a_n, b_{m-1})$, or
Case 3: $(M_{AB}(a_{n-1}, b_m)$

Case 1: $M_{AB}(a_{n-1}, b_{m-1})$ – Since the combined length of the behaviors terminating in a_{n-1} and b_{m-1} is less than $n + m$, by the inductive hypothesis we know that the behaviors terminating in these two states are compatible. Thus, there exists a corresponding composite behavior $CB = (S_1, S_2, \ldots S_p)$ with $S_p = <a_{n-1}, b_{m-1}>$. Since $a_n =_{v_{ab}} b_m$,

(1) if a_{n-1} and b_{m-1} are both time–point states then S_p is a time–point state and a_n and b_m are time–interval states. Thus the composite behavior can be extended by a time–interval state S_{p+1} corresponding to the combination of a_n and b_m.

(2) Similarly, if a_{n-1} and b_{m-1} are both time–interval states, then their successors are both time–point states and the composite behavior can be extended by a time–point state S_{p+1} corresponding to the combination of a_n and b_m.

(3) If one of the states is a time–point state and one is a time interval state, the by lemma A.2 $M_{AB}(s_{i+1}, s_j)$ and $M_{AB}(s_i, s_{j+1})$ are true. Suppose that s_i is the time–point state and s_j is a time–interval state. By the definition of a composite behavior, S_p must correspond to a time–point state. Composite behavior CB' can be defined as an extension of CB by adding a composite state $S_{p+1} = <s_{i+1}, s_j>$ and $S_{p+2} = <s_{i+1}, s_{j+1}>$. Thus, CB' is a composite behavior corresponding to the component behaviors terminating in s_{i+1} and s_{j+1}.

□

Theorem A.2 (Compatible behavior set theorem) *A set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ each from a different component are compatible if and only if for all $i, j < n$ b_i and b_j are compatible.*

Proof: Both directions are proven simply through an extension of theorem 3.1.

If compatible $\Rightarrow$ each pair is compatible – If $\{b_1, b_2, \ldots, b_n\}$ are compatible then there exists a corresponding composite behavior $CB = (S_1, S_2, \ldots S_p)$. For any given pair of components C_i and C_j, the projection of CB onto the variables described by C_i and C_j corresponds to a composite behvaior for behaviors b_i and b_j. Thus, b_i and b_j are compatible.

If each pair compatible $\Rightarrow$ set is compatible – Select two behaviors b_i and b_j. Since they are compatible they can be combined into a composite behavior CB_{ij}. By theorem 3.1, we know that all of the comonent edge predicates are satisfied. Furthermore, the predicates have been satisfied starting from the initial state. Thus, the CB_{ij} is in turn compatible with each of the remaining comonent behaviors. Thus, combine CB_{ij} with another component behavior and form a new composite behavior. Continue combining the composite behavior with additional component behaviors until all of the component behaviors have been incorporated into a single composite behavior. Thus, the set of behaviors is compatible. □

Appendix B

Dynamic Chatter Abstraction: Theorems and Proofs

This appendix contains a complete listing of the proofs and theorems for dynamic chatter abstraction. Some of the results presented in section 4.4.8 are duplicated here to simplify the process of reading and evaluating the results. The theorems within the appendix are broken up into three sections: 1) theorems and proofs relating to C_{eq}, 2) theorems and proofs relating to *Chatter-test* and theorems and proofs relating to the real valued trajectories described by an abstracted tree.

B.1 C_{eq}: Sound and complete

Lemma B.1 *For a given equivalency class EQ and time–interval state S describing the interval* (t_j, t_k)*, if for all* v *such that* $v \in EQ$*, the qdir of* v *is unconstrained except with respect to other variables in EQ, then the variables in EQ will chatter following time point state* t_k*.*

Proof: By the definition of a chatter equivalency class, we know that for any two variables x and y within EQ they are constrained such that either $[\dot{x}] = [\dot{y}]$ is always true or that $[\dot{x}] = -[\dot{y}]$ is always true. Thus, the variables in EQ can all become steady at t_k since neither the constraints relating these variables nor the constraints in the rest of the model restrict this behavior. Similarly, following t_k all of the variables can resume changing in the same direction as before or all of them can begin changing in the opposite direction. The branch following t_k is a chatter branch and thus the variables are chattering. □

Lemma B.2 *The conditions specified within C_{eq} for each QSIM constraint type T define necessary and sufficient conditions for the variables in EQ to be unconstrained with respect to a constraint of type T.*

Lemma B.3 *If an abstract state S satisfies C_{eq}, then the derivatives of the variables in EQ are unconstrained within S except with respect to other variables in EQ.*

Proof: Since S satisfies C_{eq} we know that S must satisfy at least one condition within each dependency for C_{eq} . A dependency exists for each constraint C within the model that can constrain the direction of change for the variables in EQ. Thus, by lemma 4.6 we know that for all C such that C is a constraint within the model referring to a variable in EQ the variables in EQ are unconstrained with respect to C. Thus, the variables in EQ are unconstrained by the model with respect to variables not contained within EQ. □

Theorem B.1 (**C_{eq} soundness**) *For a given abstract time-interval state S describing the interval (t_j, t_k) and an equivalency class EQ, if S satisfies C_{eq} then the variables in EQ will chatter following time point t_k.*

Proof: Since S satisfies C_{eq} the variables in EQ are unconstrained except with respect to each other (lemma 4.5). Therefore, by lemma 4.7 the variables will chatter following time point t_k. □

Theorem B.2 (**C_{eq} completeness**) *Assuming that the chatter equivalency class partitioning is complete, the predicate C_{eq} defines necessary conditions on an abstract qualitative state for the variables in EQ to chatter.*

Proof: Assume that there exists a state S in which the variables in EQ are free to chatter. Furthermore, assume that C_{eq}(S) is false.

- There must be at least one dependency D in C_{eq} which is not satisfied. There must be a constraint within the model corresponding to D. Call this constraint C_D. Let the variables contained within C_D not contained within V_{EQ} be represented by V_{C_D}. Furthermore, if V_{EQ} corresponds to the set of variables in EQ, let $V_{EQ-C_D} = V_{EQ} \cap V_{C_D}$.
- By lemma 4.6 we know that the conditions in D specify necessary and sufficient conditions for the derivative of the variables in V_{EQ-C_D} to be unconstrained with respect to V_{C_D}. Therefore, the derivatives of the variables in V_{EQ-C_D} must be constrained with respect to V_{C_D}.

- Since the variables in V_{EQ-C_D} chatter following S, some of the variables in V_{C_D} must also chatter in unison.
- However, this is a contradiction with the assumption that the chatter equivalency partitioning algorithm is complete. Therefore, if a state exhibits chatter in the variables in EQ then it must satisfy the predicate C_{eq}.

□

While this conclusion requires the partitioning algorithm to be complete, it is not the case that C_{eq} fails to detect chatter in all instances where C_{eq} is incomplete. On the contrary, since the dependencies record the conditions under which a variable is unconstrained, it may be possible to use the information within the predicate to detect implicit chatter equivalency relationships between variables due to the interaction of multiple constraints. In addition, note that in all of the models tested dynamic chatter abstraction detected all instances of chatter encountered.

B.2 *Chatter-test*: Sound and complete

Theorem B.3 (*Chatter-test* **soundness)** *For a given qualitative state S and an equivalency class EQ if* Chatter-test*(S, EQ) identifies the variables in EQ as chattering then there exists an abstract state S' such that S' satisfies C_{eq} and is chatter reachable from S assuming that the paths supporting each change in S' from S can be combined into a consistent sequence of states.*

The main task in proving this theorem is explaining the assumption. First, we will introduce two new definitions to assist us in this task:

Definition B.1 (Sufficient chatter support) *A set of equivalency classes N provides* sufficient chatter support *for an equivalency node EQ to chatter if and only if the variables in EQ can chatter given that the only other classes allowed to chatter are contained within N. The set is* minimal *if and only if there does not exist a proper subset of N that also provides sufficient chatter support for EQ. Note that there may be more than one set that offers minimal sufficient chatter support for a given equivalency class.*

Definition B.2 (Chatter support structure) *For equivalency class EQ_m, a partially ordered, set of equivalency classes $N = \{(EQ_1, \ldots, EQ_m\}$ is a* chatter support structure *Sup_{EQ_m} for EQ_m ordered by the predicate $\prec_{sup}$ if and only if*

(1) N offers sufficient chatter support for EQ_m,

(2) the partial ordering predicate $\prec_{sup}$ satisfies the condition that

$$\textit{For all } EQ_i : EQ_i \in Sup_{EQ_m} :: Sup_{EQ_i} = \{EQ_j | EQ_j \preceq_{sup} EQ_i\}.$$

Thus, condition (2) provides a recursive definition for the partial order such that a chatter support structure for each class within N is also defined.

Once again, N is minimal if there does not exist a proper subset of N that is also a chatter support structure for EQ_m.

The difficult portion of the proof is demonstrating that S' is chatter reachable from S. To simplify the process of the proof, I will first provide an informal discussion of the algorithm along with an explanation of the assumption.

Let N be a minimal chatter support structure for EQ. Each path within N provides a different sequence of changes that need to be free to happen to satisfy one of the dependencies within C_{eq}. Each path can be thought of as an abstract plan. For example, the sequence A, B, C would mean that first A has to be free to chatter and then change signs, then B and then C. It is an abstract plan because it does not specify when something needs to change back. Thus, for example sometimes a variable A has to change signs to allow B to change signs, but then A has to change back before C can begin to chatter. (see figure 4.8). Note that dynamic chatter abstraction *does not compute all of the paths in the chattering region.* Instead, it simply *ensures that the path is possible due to variables chattering.*

If we think of the partial order defined upon N as a graph, then a *support path* for EQ_m is a path within N starting from a leaf node that ends in EQ_m. Dynamic chatter abstraction ensures that for each dependency within C_{eq} that there exists a set of support paths that ensure that all of the variables that need to change for S' to be reached are free to chatter. It does not, however, test to see if the support paths can be combined to form a consistent sequence of states leading for S to S'. Thus, we assume that the support paths can be combined in such a manner to generate such a sequence of states.

While at first, this assumption might seem unjustified, we have yet to encounter an example where it is not satisfied. This is due to the unconstrained nature of chatter. Since the direction of change for the variables in a chattering region are highly unconstrained it is very likely that there will be some sequence of changes that can occur that will lead to S' making it chatter reachable. Computing all of these paths can be exponentially expensive and thus we avoid performing this computation.

Proof:

Consistent state satisfying C_{eq} – *Chatter-test* only identifies an equivalency class EQ as chattering if it can identify a state S' that satisfies C_{eq}. Thus, by the completeness theorem for C_{eq}we know that the variables in EQ will be free to chatter in S'.

S' **is chatter reachable from** S – Let N be a minimal chatter support structure for EQ. *Chatter-test* implicitly computes such a structure in segments. The portion of the structure relating a class EQ to the classes immediately preceding it in the structure is computed the first time that *EQ-Chatter-Test* is called in which it returns `:chatter`. Since this must occur for each equivalency class identified as chattering, the entire structure will be computed by *Chatter-test*. Since *Chatter-test* backtracks on cycles the entire structure can be represented as support paths rooted at leaf nodes. The equivalency classes in the leaf nodes are free to chatter in S. Once they chatter, then the next set of classes will chatter assuming that the paths can be combined. This process repeats itself until S' is reached from S via a chatter reachable path.

Note that if cycles were allowed in N, then the assumption would not be sufficient to guarantee that S' was chatter reachable from S. (See figure 4.10.)

□

Theorem B.4 (*Chatter-test* **completeness)** *For a given qualitative state S and an equivalency class EQ if there exists a state S' satisfying C_{eq} that is chatter reachable from S, then* Chatter-test*(S, EQ) identifies the variables in EQ as chattering.*

Proof: *Chatter-test* restricts its search of the qualitative state space in two ways:

(1) it initializes a partial state with the qualitative value information for the variables identified as non–chattering due to static conditions, and

(2) it restricts its search when either

   (a) an equivalency class that must change is identified as non–chattering, or

   (b) a cycle occurs in the calling sequence.

Thus, to show that *Chatter-test* will detect S' and therefore identify the variables as chattering, we need to show that the restrictions listed above will not prevent *Chatter-test* from detecting S. To do this, we will use the fact that there exists a path from S to S' in which each change is due to a chattering variable changing the sign of its derivative as stated in the antecedent.

Restriction 1:

Since there exists a path from S to S' only containing changes in chattering variables, S and S' can only differ in distinctions due to chatter. Thus, S an S' are identical with respect to non–chatter distinctions and therefore the assertion in restriction 1 does not prevent the algorithm from identifying S'.

Restriction 2a:

This restriction will not prevent S' from being found by the same argument as presented for restriction 1.

Restriction 2b:

- Select the shortest chatter reachable path between S and S' such that variables do not change sign simultaneously unless they are constrained to do so by the model.
  - For example, suppose x and y are both free to chatter following a given state s_i and their derivatives are not constrained with respect to each other. Thus, the path in which they chatter simultaneously can be extended to the path where x chatters and then y chatters resulting in the same qualitative state.

  Call this path $b = (s_1, \ldots, s_n)$ such that $s_1 = S$ and $s_n = S'$.
- Thus, each pair of time–interval states s_i and s_{i+2} separated by a time–point state s_{i+1} differ only in the direction of change for the variables within a single equivalency class.
- Represent b as a sequence of equivalency classes $(C_1, C_2, \ldots, C_{n/2})$ where each C_i corresponds to an equivalency class that changed sign between states $s_{(n-1)/2}$ and $s_{(n+1)/2}$. Collapse this sequence to eliminate duplicates by removing equivalency classes that appear earlier within the list resulting in the sequence $b_{eq} = (EQ_1, EQ_2, \ldots, EQ_m)$ where $m \leq n/2$. This sequence is called the *equivalency dependency sequence.*
  - These duplicates can be removed since *Chatter-test* does not compute the exact path that leads to S'. It simply ensures that each of the equivalency classes that must change are free to chatter.

- Note that for a given EQ_i that the set $\{EQ_1, \ldots, EQ_{i-1}\}$ is the set of equivalency nodes that must be identified as chattering to satisfy C_{eq_i} for the given path. Therefore, it can be said that EQ_i depends upon $\{EQ_1, \ldots, EQ_{i-1}\}$ to chatter.
- While it is possible that there will be other equivalency dependency sequence for EQ, b_{eq} is the minimal set since b is the shortest path between S and S'.[1]

- By induction on the length of this sequence we will show that there exists a path in the calling sequence of *Chatter-test* where a cycle does not occur. Let m equal to the length of the sequence.

  Base case: $m = 0$

  A sequence of length zero occurs when the path from S to S' includes a single state. Thus, $S = S'$ and the conditions in C_{eq} can be satisfied without requiring any variable to `:change-qdir`.

  Inductive step:

  Assume: for a sequence of length $m-1$ or less a cycle does not occur.

  Assume that for a state S' to satisfy C_{eq} for equivalency class EQ that b_{eq} is of length m. *Chatter-test* is called recursively for each equivalency node EQ_j where $j \leq m$ when an assertion requiring EQ_j to chatter is encountered.

  (1) The equivalency dependency sequence for EQ_j is $b_{eq_j} = (EQ_1, \ldots, EQ_{j-1})$.

  (2) Thus, the length of b_{eq_j} is $j-1$ and $EQ \ni b_{eq_j}$.

  Thus, due to item 2 and the inductive hypothesis, a cycle will not occur in the recursive call to *Chatter-test*. $\square$

B.3 Real valued trajectories

Theorem B.5 (QSIM soundness retained) *The set of real–valued trajectories described by an abstracted behavior tree generated using dynamic chatter abstrac-*

[1]Note that there may be other minimal equivalency dependency sequences since there may be other paths of equal length to b that will result in a different set of dependent equivalency classes.

tion includes all real–valued trajectories described by an unabstracted behavior tree exhibiting chatter.

Proof: Dynamic chatter abstraction performs a strict abstraction operation. Thus, for a given time–interval state S, dynamic chatter abstraction replaces it with a state S_a such that $S \subseteq S_a$. Thus, the set of precise numerical values consistent with S is a subset of the set consistent with S_a. Furthermore, by the qualitative successor extension lemma (lemma 4.4), the successors of S_a contain all possible successor values for the precise numerical values in S_a consistent with continuity. Therefore, since each state and each transition is a superset of the values described by a non–abstracted tree, the set of real–valued trajectories described by an abstracted tree is a subset of those described by a non–abstracted tree exhibiting chatter. $\Box$

Note that the abstraction operation performed by dynamic chatter abstraction does introduce additional real–valued trajectories no included within the non–abstracted behavior. There are two potential sources for these trajectories:

Abstraction operation – As with chatter box abstraction, the representation used to describe an abstract state is not sufficiently refined to describe the correlations between chattering variables contained within the same equivalency class.

As discussed in section 4.3.6, this loss of precision is simply a cost of abstraction and in general does not impact the usefulness of the results generated.

Abstract state successor computation – Dynamic chatter abstraction purposely does not explore all paths through the potentially chattering region. Instead, it determines what variables will chatter, generates an abstract state and then uses the extension to the QSIM successor tables to compute the successors to this abstract state. The constraints within the model are used to ensure that each successor state is consistent. However, there is no guarantee that there exists a path through the chattering region that ends in each successor of the abstract state. Thus, conceptually it is possible that a successor of an abstract state would not have a corresponding state within the unabstracted tree.

In practice, we have not encountered this situation in any of the models tested. Chatter box abstraction does not suffer from this limitation since it computes all paths through the chattering region. Thus, we can test for this condition by comparing the results from dynamic chatter abstraction against the results using chatter box abstraction. In all of the models tested, the behavioral descriptions were identical.

The reason that this condition has not proved to be a problem is due to the unconstrained nature of chatter. Since chattering variables are loosely constrained within the chattering region, it is very likely that there will exist a path through the chattering region for each potential exit state.

Appendix C

TeQSIM Soundness and Completeness Theorem Proof

Theorem C.1 (TL–Guide is sound and complete) *Given a QSIM behavior $b = <s_0, \ldots, s_n>$ and an admissible formula φ then TL–Guide:*

1. *refutes b iff for all full–path extensions $\widehat{b}$: $\widehat{b} \not\models \varphi$ and $b \rhd \varphi$.*
2. *retains b without modifying it iff*
   (a) *$b \models \varphi$ and $b \rhd \varphi$; or*
   (b) *$b \not\rhd \varphi$ and there is no necessary condition C for refining b into a model for φ (i.e. $\nexists C \neq C_{\text{true}}$ such that if $b'' = \mathcal{M}(\neg C, b)$ then for all full–path extensions $\widehat{b''}$: $\widehat{b''} \not\models \varphi$).*
3. *replaces b with b' iff*
   (a) *$b \stackrel{?}{\models} \varphi$ and $b \rhd \varphi$ and exists $C \neq C_{\text{true}}$ such that $b' = \mathcal{M}(C, b) \models \varphi$ and C is necessary (i.e. if there exists b'' such that $b'' = \mathcal{M}(\neg C, b)$ then for all full–path extensions $\widehat{b''}$: $\widehat{b''} \not\models \varphi$); or*
   (b) *$b \not\rhd \varphi$ and there is a necessary condition $C \neq C_{\text{true}}$ (such that if $b'' = \mathcal{M}(\neg C, b)$ then for all full–path extensions $\widehat{b''}$: $\widehat{b''} \not\models \varphi$) and $b' = \mathcal{M}(C, b)$.*

Proof. Let (v_i, c_i, φ_i) be the evaluation sequence of b with respect to φ. Numbers in parentheses refer to statements of lemma 4.

(1,$\Longrightarrow$): If b is refuted then $v_n = \mathtt{F}$. Therefore (6) leads to $b^n \rhd \varphi_n \wedge b^n \not\models \varphi_n$. From $b^n \rhd \varphi_n$ it follows that (3) $b^{n-1} \rhd \varphi_{n-1}$ until $b^0 \rhd \varphi_0$. From $b^n \not\models \varphi_n$ it follows (2) that $b^n \not\models \varphi_{n-1}$. And by repeating this we get $b^0 \not\models \varphi_0$ from which the extensions lemma brings $\forall \widehat{b} : \widehat{b} \not\models \varphi$.

(1,$\Longleftarrow$): Assuming that for any arbitrary extension $\widehat{b}$ of b, $\widehat{b} \not\stackrel{*}{\models} \varphi$ and $b \rhd \varphi$ implies that (extensions lemma) $b \not\stackrel{*}{\models} \varphi$ and (equivalence lemma) $b^0 \not\stackrel{*}{\models} \varphi_0$ and $b^0 \rhd \varphi_0$. The latter implies (3) that $b^1 \rhd \varphi_1$ and so forth until $b^n \rhd \varphi_n$. Similarly we get $b1 \not\stackrel{*}{\models} \varphi_1$ and so forth until $b^n \not\stackrel{*}{\models} \varphi_n$. Now $b^n \rhd \varphi_n \wedge b^n \not\stackrel{*}{\models} \varphi_n$ leads (6) to $v_n = \mathtt{F}$ and to refutation of b.

(2, $\Longrightarrow$): If b is retained then $v_n = \mathtt{T}$ or $\mathtt{U}$. If $v_n = \mathtt{T}$ then (5) $b^n \stackrel{*}{\models} \varphi_n$ and $b^n \rhd \varphi_n$. Then this implies (1) $b^{n-1} \stackrel{*}{\models} \varphi_{n-1}$ and so forth until $b^0 \stackrel{*}{\models} \varphi_0$. Similarly for $b^0 \rhd \varphi_0$. At this point the equivalence lemma leads to $b \stackrel{*}{\models} \varphi$ and, as the behavior is not modified, then $b \rhd \varphi$. If $v_n = \mathtt{U}$ then (4) implies $b^n \ntriangleright \varphi_n$,and repeated application of (3) leads to $b^0 \ntriangleright \varphi_0$ and then to $b \ntriangleright \varphi$. To prove that no conditions needs to be applied, observe that if the behavior is not modified then all c_i have to be either C_{true} (which is trivial) or delayed conditions that have not been applied, that is they are not necessary.

(2.a, $\Longleftarrow$): From the hypothesis $b \models \varphi$ the equivalence lemma leads to $b^0 \models \varphi_0$; by repeatedly applying (1) we get that $b^1 \models \varphi_1, \ldots, b^n \models \varphi_n$ and since $b \rhd \varphi$ it follows that there are no delayed conditions that might be applied, hence $b^i \models \mathcal{P}[\varphi_i, s_i]$ and therefore $c_i = C_{\text{true}}$ and TLG–1 never refines any state of b.

(2.b, $\Longleftarrow$): Observe that from $b \ntriangleright \varphi$ the equivalence lemma yields $b^0 \ntriangleright \varphi_0$ and that by repeatedly applying (3) we get $b^i \ntriangleright \varphi_i$ for all i from 0 to n. Since $b^i \ntriangleright \phi$ implies $b^{(i,i)} \ntriangleright \phi$, then $v_i = \mathtt{U}$ holds and none of the $b^{(0,i)}$ is ever refuted. To prove that b is not refined if there is no necessary refinement condition, let's assume the contrary, *i.e.* that b is refined. Therefore there is at least one $c_j \not\equiv C_{\text{true}}$ that is applied to b. Hence $b^j \stackrel{?}{\models} \varphi_j$. c_j is also necessary for b, in the sense that any model of φ has to agree with c_j. Or, in other terms, if a behavior agrees with $\neg c_j$ then it cannot be a model for φ. To prove this observe that if $\neg c_j$ is applied to $b(j)$ we obtain a new behavior b'. But $b'^j \not\stackrel{*}{\models} \varphi_j$ and so forth until $b' \not\stackrel{*}{\models} \varphi$. In addition $b'^j \rhd \varphi_j$ and down to $b' \rhd \varphi$. At this point the extensions lemma yields that none of the possible extensions of b' is a model for φ, contrary to our hypothesis.

(3, $\Longrightarrow$): If b has been refined then at least one of the c_i is $\not\equiv C_{\text{true}}$, and $v_n = \mathtt{T}$ or $\mathtt{U}$. If $v_n = \mathtt{T}$ then $b^n \stackrel{*}{\models} \varphi_n$ and $b^n \rhd \varphi_n$. The former implies that $b \stackrel{*}{\models} \varphi$ and the latter $b \rhd \varphi$. In fact, $b \stackrel{?}{\models} \varphi$, because if $b \models \varphi$ were true, b would have been retained. Applying C to b gives a new behavior b' and $b' \models \varphi$. Therefore in the case $v_n = \mathtt{T}$ behavior b is a potential model of φ. In case $v_n = \mathtt{U}$ then $b^n \ntriangleright \varphi_n$ and then the conclusion $b \ntriangleright \varphi$ follows. To prove that condition C is necessary, observe that if b is refined with C then at least one of the c_i has to be $\not\equiv C_{\text{true}}$ and applied. Then $s_i \stackrel{?}{\models} \mathcal{P}[\varphi_i, s_i]$ and therefore C is necessary and sufficient.

(3, $\Longleftarrow$): We have to prove that b is refined using a condition C that is necessary and sufficient. Since C has to be a conjunction of elementary conditions, we have to prove that TL–Guide applies all of these conditions and nothing more. Take any element c' of C. Then c' refers to a state s_j and $s_j \stackrel{?}{\models} \mathcal{P}[\varphi_j, s_j]$ and, since Ψ is complete, $c_j \equiv c'$. c_j is either applied directly or it is delayed (in which case, since c' is necessary then the trigger of c' has to be definitely false and c' is applied). In neither case TL–Guide applies any additional conditions. $\square$

Appendix D

QSIM References

This appendix contains relevant information extracted from Kuipers(Kuipers, 1994) to assist the reader.

D.0.1 QSIM Transition Tables

The following table lists the valid transitions for a variable from either a time–point state or a time–interval state. The values specified in these tables are computed using the Intermediate Value and the Mean Value Theorems and have proven to be sound and complete with respect to the underlying numerical values described.

- **P-Successors**: point to interval.

$QV(v, t_i)$	$\Rightarrow$	$QV(v, t_i, t_{i+1})$
$<l_j, std>$		$<l_j, std>$
$<l_j, std>$		$<(l_j, l_{j+1}), inc>$
$<l_j, std>$		$<(l_{j-1}, l_j), dec>$
$<l_j, inc>$		$<(l_j, l_{j+1}), inc>$
$<l_j, dec>$		$<(l_{j-1}, l_j), dec>$
$<(l_j, l_{j+1}), inc>$		$<(l_j, l_{j+1}), inc>$
$<(l_j, l_{j+1}), dec>$		$<(l_j, l_{j+1}), dec>$
$<(l_j, l_{j+1}), std>$		$<(l_j, l_{j+1}), std>$
$<(l_j, l_{j+1}), std>$		$<(l_j, l_{j+1}), inc>$
$<(l_j, l_{j+1}), std>$		$<(l_j, l_{j+1}), dec>$

- **I-Successors**: interval to point.

$QV(v, t_i, t_{i+1})$	$\Rightarrow$	$QV(v, t_{i+1})$
$<l_j, std>$		$<l_j, std>$
$<(l_j, l_{j+1}), inc>$		$<l_{j+1}, std>$
$<(l_j, l_{j+1}), inc>$		$<l_{j+1}, inc>$
$<(l_j, l_{j+1}), inc>$		$<(l_j, l_{j+1}), inc>$
$-1z<(l_j, l_{j+1}), inc>$		$<(l_j, l_{j+1}), std>$
$<(l_j, l_{j+1}), dec>$		$<l_j, std>$
$<(l_j, l_{j+1}), dec>$		$<l_j, dec>$
$<(l_j, l_{j+1}), dec>$		$<(l_j, l_{j+1}), dec>$
$<(l_j, l_{j+1}), dec>$		$<(l_j, l_{j+1}), std>$
$<(l_j, l_{j+1}), std>$		$<(l_j, l_{j+1}), std>$

Bibliography

Allen, J., Hendler, J., & Tate, A. (Eds.). (1990). *Readings in Planning.* Morgan Kaufmann, San Mateo,CA.

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, *23*, 123–154.

Alur, R., & Henzinger, T. (1993). Real–time logics: complexity and expressiveness. *Information and Computation*, *104*(1), 35–77.

Bacchus, F., & Kabanza, F. (1996). Planning for temporally extended goals. In Clancey, B., & Weld, D. (Eds.), *Proc. of the Thirteenth National Conference on Artificial Intelligence.* AAAI Press.

Baccus, F., & Kabanza, F. (1995). Using temporal logic to control search in a forward channing planner. In *Proc. of the Third European Workshop on Planning.* AAAI Press.

Berleant, D., & Kuipers, B. (1988). Using incomplete quantitative knowledge in qualitative reasoning. In *Proc. of the Sixth National Conference on Artificial Intelligence*, pp. 324–329.

Bhat, G., Cleaveland, R., & Grumberg, O. (1995). Efficient on–the–fly model checking for CTL*. In *Proc. of Conference on Logic in Computer Science (LICS–95).*

Brajnik, G. (1995). Introducing boundary conditions in semi–quantitative simulation. In *Ninth International Workshop on Qualitative Reasoning*, pp. 22–31 Amsterdam.

Brajnik, G. (1997). Statistical properties of qualitative behaviors. In *Proc. of the Eleventh International Workshop on Qualitative Reasoning about Physical Systems*, pp. 233–240. Cortona, Italy.

Brajnik, G., & Clancy, D. J. (1996a). Guiding and refining simulation using temporal logic. In *Proc. of the Third International Workshop on Temporal Representation and Reasoning (TIME'96)* Key West, Florida. IEEE Computer Society Press. To appear.

Brajnik, G., & Clancy, D. J. (1996b). Temporal constraints on trajectoriesin qualitative simulation. In *Proc. of the Tenth International Workshop on Qualitative Reasoning about Physical Systems*, pp. 22–31. Fallen Leaf Lake, CA.

Brajnik, G., & Clancy, D. J. (1997). Focusing qualitative simulation using temporal logic: theoretical foundations. *Annals of Mathematics and Artificial Intelligence*. To appear.

Bredeweg, B. (1992). *Expertise in Qualitative Prediction of Behavior*. Ph.D. thesis, Department of Social Science and Informatics University of Amsterdam, Amsterdam, The Netherlands.

Clancy, D., & Kuipers, B. (1993). Behavior abstraction for tractable simulation. In *Proc. of the Seventh International Workshop on Qualitative Reasoning about Physical Systems*, pp. 57–64.

Clancy, D. J., Brajnik, G., & Kay, H. (1997). Model revision: Techniques and tools for analyzing simulation results and revising qualitative models. In *Proc. of the Eleventh International Workshop on Qualitative Reasoning about Physical Systems*, pp. 53–66. Cortona, Italy.

Clancy, D. J., & Kuipers, B. J. (1997a). Dynamic chatter abstraction: A scalable technique for avoiding irrelevant distinctions during qualitative simulation. In *Proc. of the Eleventh International Workshop on Qualitative Reasoning about Physical Systems*, pp. 67–76. Cortona, Italy.

Clancy, D. J., & Kuipers, B. J. (1997b). Model decomposition and simulation: a component based qualitative simulation algorithm. In Kuipers, B. J., & Webber, B. (Eds.), *Proc. of the Fourteenth National Conference on Artificial Intelligence*. AAAI Press.

Clancy, D. J., & Kuipers, B. J. (1997c). Static and dynamic abstraction solves the problem of chatter in qualitative simulation. In Kuipers, B. J., & Webber, B. (Eds.), *Proc. of the Fourteenth National Conference on Artificial Intelligence*. AAAI Press.

Coiera, E. W. (1992). Qualitative superposition. *Artificial Intelligence*, *56*, 171–196.

Davis, E. (1990). *Representations of Commonsense Knowledge.* Morgan Kaufmann Publishers, Inc.

de Kleer, J., & Brown, J. S. (1985). A qualitative physics based on confluences. In Hobbs, J. R., & Moore, R. C. (Eds.), *Formal Theories of the Commonsense World*, chap. 4, pp. 109–183. Ablex, Norwood, New Jersey.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, *49*, 61–95.

Dechter, R., & Pearl, J. (1988a). Network–based heurisitcs for constriant-satisfaction problems. *Artificial Intelligence*, *34*, 1–38.

Dechter, R., & Pearl, J. (1988b). Tree-clustering schemes for constraint processing. In *Proceedings of the Seventh National Conference on Artificial Intelligence* Los Altos, CA. Morgan Kaufman.

Dechter, R., & Pearl, J. (1989). Tree–clustering for constraint networks. *Artificial Intelligence*, *38*, 353–366.

DeCoste, D. (1994). Goal–directed qualitative reasoning with partial states. Tech. rep. 57, The Institute for the Learning Sciences, University of Illinois at Urbana–Champaign.

Doyle, J., & Sacks, E. (1992). Proglemena to any future qualitative physics. *Computational Intelligence*, *8*, 187–209.

Emerson, E. (1990). Temporal and modal logic. In van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science*, pp. 995–1072. Elsevier Science Publishers/MIT Press. Chap. 16.

Even, S. (1979). *Graph Algorithms.* Computer Science Press, Rockville, Maryland.

Falkenhainer, B., & Forbus, K. (1991). Compositional modeling: finding the right model for the job. *Artificial Intelligence*, *51*, 95–143.

Farquhar, A., Iwasaki, Y., Fikes, R., & Bobrow, D. (1996). A compositional modeling language. In *Proc. of the Tenth International Workshop on Qualitative Reasoning about Physical Systems.* Fallen Leaf Lake, CA.

Farquhar, A. (1993). *Automated Modeling of Physical Systems in the Presence of Incomplete Knowledge.* Ph.D. thesis, Department of Computer Sciences, the University of Texas at Austin. Available as technical report UT-AI-93-207.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., & Uthurusamy, R. (1996). *Advances in Knowledge Discovery and Data Mining.* AAAI Press / MIT Press, Cambridge, Massachusetts.

Forbus, K. (1984). Qualitative process theory. *Artificial Intelligence*, *24*, 85–168.

Forbus, K. (1989). Introducing actions into qualitative simulation. In *IJCAI–89*, pp. 1273–1278.

Forbus, K. D. (1996). Self–explanatory simulators for middle–school science education: A progress report. In *Proc. of the Tenth International Workshop on Qualitative Reasoning about Physical Systems*, pp. 52–56. Fallen Leaf Lake, CA.

Fouché, P., & Kuipers, B. (1990). An assessment of currrent qualitative simulation techniques. In *Proc. of Fourth International Worskhop on Qualitative Reasoning about Physical Systems*, pp. 195–205.

Franke, D., & Dvorak, D. (1990). CC: Component-connection models for qualitative simulation – a user's guide. Tech. rep. AI90–126, Artificial Intelligence Laboratory, Department of Computer Sciences, University of Texas at Austin.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Communications of the ACM*, *32*, 755–761.

Hayes, P. J. (1985). The second naive physics manifesto. In Hobbs, J., & Moore, B. (Eds.), *Formal Theories of the Commonsense World.* Ablex Publishing Corp.

Ironi, L. (Ed.). (1997). *Proceedings of the Eleventh International Workshop on Qualitative Reasoning about Physical Systems.* Cortona, Italy.

Ironi, L., & Stefanelli (1994). QCMF: a tool for generating qualitative models for compartmental structures. In *8th International Workshop on Qualitative Reasoning about Physical Systems* Nara, Japan.

Iwasaki, Y. (1988). Causal ordering in a mixed strcuture. In *Proc. of the Seventh National Conference on Artificial Intelligence*, pp. 313–318. AAAI Press / The MIT Press.

Iwasaki, Y., & Farquhar, A. (Eds.). (1996). *Proceedings of the Tenth International Workshop on Qualitative Reasoning about Physical Systems.* AAAI Press. Fallen Leaf Lake, CA.

Iwasaki, Y., Farquhar, A., Saraswat, V., Bobrow, D., & Gupta, V. (1995). Modeling time in hybrid systems: how fast is "instantaneous"?. In *IJCAI–95*, pp. 1773–1780 Montréal, Canada. Morgan Kaufmann Publishers, Inc.

Kay, H., & Kuipers, B. (1993). Numerical behavior envelopes for qualitative models. In *Proc. of the Eleventh National Conference on Artificial Intelligence.* AAAI Press/MIT Press.

Kay, H. (1991). Monitoring and diagnosis of multi-tank flows using qualitative reasoning. Master's thesis, Artificial Intelligence Laboratory, The University of Texas at Austin.

Kay, H. (1992). A qualitative model of the space shuttle reaction control system. Tech. rep. TR AI92-188, Artificial Intelligence Laboratory, Department of Computer Sciences, University of Texas at Austin.

Kuipers, B. (1994). *Qualitative Reasoning: modeling and simulation with incomplete knowledge.* MIT Press, Cambridge, Massachusetts.

Kuipers, B., & Berleant, D. (1992). Combined qualitative and numerical simulation with Q3. In Faltings, B., & Struss, P. (Eds.), *Recent advances in qualitative physics*, pp. 3–16. MIT Press.

Kuipers, B., Chiu, C., Molle, D. D., & Throop, D. (1991). Higher–order derivative constraint in qualitative simulation. *Artificial Intelligence*, *51*, 343–379.

Kuipers, B., & Shults, B. (1994). Reasoning in logic about continuous change. In *Principles of Knowledge Representation and Reasoning (KR–94).* Morgan Kaufmann Publishers, Inc.

Nayak, P. (1994). Causal approximations. *Artificial Intelligence*, *70.*

Nayak, P. (1992). *Automated Modeling of Physical Systems.* Ph.D. thesis, Department of Computer Science, Stanford University.

Nishida, T., & Doshita, S. (1987). Reasoning about discontinuous change. In *AAAI–87*, pp. 643–648.

Pell, B., Bernard, D., Chien, S., Gat, E., Muscettola, N., Nayak, P., Wagner, M., & Williams, B. (1997). An autonomous spacecraft agent prototype. In *Proceedings of the First International Conference on Autonomous Agents*. ACM Press.

Porter, B., Lester, J., Murray, K., Pittman, K., Souther, A., Acker, L., & Jones, T. (1988). Ai research in the context of a multifunctional knowledge base: The botany knowledge base project. Technical report AI88–88, Artificial Intelligence Laboratory, The University of Texas at Austin.

Rickel, J., & Porter, B. (1994). Automated modeling for answering prediction questions: selecting the time scale and system boundary. In *Proc. of the 12th National Conference on Artificial Intelligence*. AAAI Press / The MIT Press.

Rickel, J., & Porter, B. (1997). Automated modeling of complex systems to answer prediction questions. *Artificial Intelligence*, *93*, 201–260.

Say, A. C. C. (1997). Limitations imposed by the sign–equality assumption in qualitative simulation. In *Proceedings of the Eleventh International Workshop on Qualitative Reasoning about Physical Systems*, pp. 165–174. Cortona, Italy.

Shafer, G., & Pearl, J. (Eds.). (1990). *Readings in Uncertain Reasoning*. Morgan Kaufmann, San Mateo, CA.

Shavlik, J. W., & Dietterich, T. G. (Eds.). (1990). *Readings in Machine Learning*. Morgan Kaufmann, San Mateo,CA.

Shimomura, Y., Ogawa, K., Tanigawa, S., Umeda, Y., & Tomiyama, T. (1996). Development of self—maintenance photocopiers. In *Proc. of the Tenth International Workshop on Qualitative Reasoning about Physical Systems*, pp. 225–234. Fallen Leaf Lake, CA.

Shults, B., & Kuipers, B. J. (1997). Proving properties of continuous systems: qualitative simulation and temporal logic. *Artificial Intelligence*, *92*, 91–129.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press, San Diego, CA.

Washio, T., & Kitamura, M. (1995). A fast history–oriented envisioning method introducing temporal logic. In *Ninth International Workshop on Qualitative Reasoning (QR–95)*, pp. 279–288 Amsterdam, NL.

Weld, D. S. (1986). The use of aggregation in causal simulation. *Artificial Intelligence*, *30*, 1–17.

Weld, D. S., & de Kleer, J. (Eds.). (1990). *Readings in Qualitative Reasoning About Physical Systems*. William Kaufman.

Wilkins, D. E., Meyers, K., desJardins, M., & Berry, P. (1997). Summary of multiagent planning architecture.. Working Document. SRI Project 7150. http://www.ai.sri.com/ wilkins/mpa/mpa.ps.

Williams, B. C., & Nayak, P. P. (1996). A model–based approach to reactive self-configuring systems. In *Proc. of the Tenth International Workshop on Qualitative Reasoning about Physical Systems*, pp. 274–282. Fallen Leaf Lake, CA.

Williams, B. (1991). A theory of interactions: Unifying qualitative and quantitative algebraic reasoning. *Artificial Intelligence*, *51*, 39–94.

Williams, B. C. (1986). Doing time: Putting qualitative reasoning on firmer ground. In *Proc. of the Fourth National Conference on Artificial Intelligence*, pp. 105–113.

Vita

Daniel Joseph Clancy was born in New Orleans, Louisiana on January 11, 1964, the son of Joseph Ignatius Clancy and Mary Ethel Clancy. Dan's father died shortly before he was born in a plane crash, and his mother remarried two years later to Gerald E. Siefken who raised Dan. After graduating from Jesuit High School in New Orleans, Louisiana, he attended Duke University. He received a Bachelor of Arts with a double major in Drama and Computer Science. Upon graduation, he moved to Boston, Massachusetts and worked for MITRE Corporation for 4 years as a Software/Systems Engineer. In August of 1989, he entered the Masters Program in Computer Science at the University of Texas at Austin and later transfered to the Doctoral program. In the summer of 1993, he worked at the NASA and in the summer of 1994 he worked at Xerox Webster Research Center. He also has worked as a consultant at Trilogy Corporation in Austin, Texas, since October of 1996. Dan married Suzanne Marie Burke on May 28, 1995. They are currently expecting their first child.

Permanent Address: 1507 Oxford Ave.
Austin, TX 78704

This dissertation was typed by the author.