Chapter 1

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 **??**.) 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.

1.1 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.

1.1.1 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.

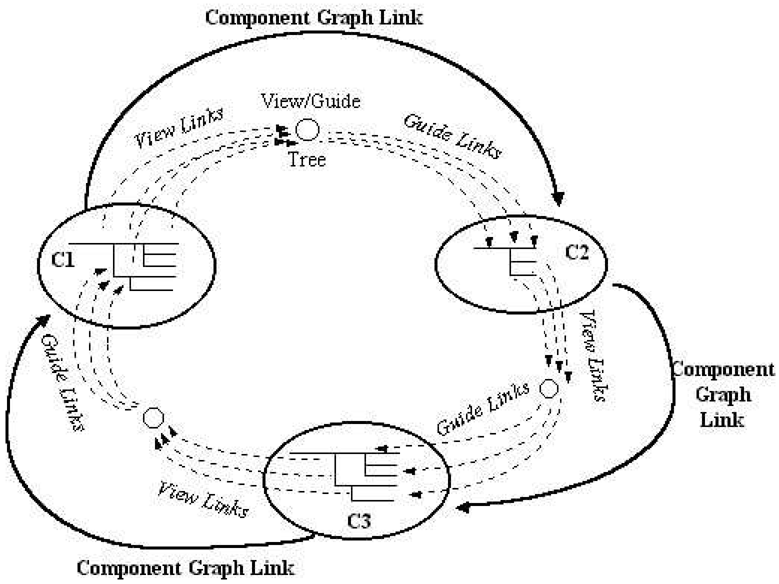
Boundary variables for a component are identified using a causal analysis (Iwasaki, 1988; Nayak, 1994) of the QDE. A variable v is a boundary variable for a component C if and only if v is directly upstream with respect to the causal ordering and related to a within–partition variable in C via a constraint 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 1.1: View/guide links and the component graph

describing the behavior of the shared variables (see figure 1.1). 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 1.2 presents an example component graph and demonstrates how structure and causality are applied for this graph.

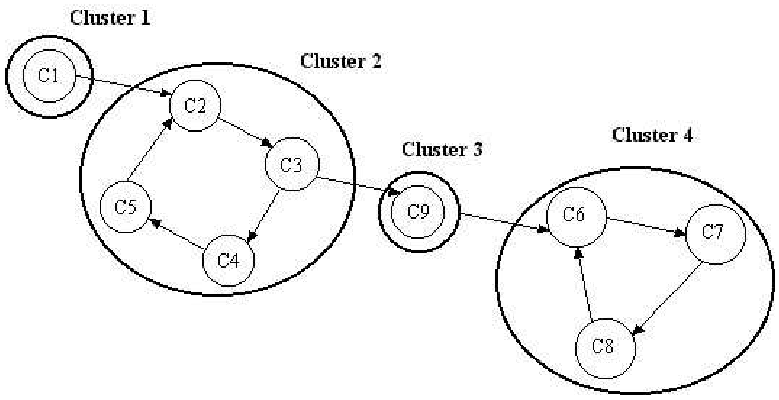
1.2 Two tank cascade example

Figure 1.3(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 1.3(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

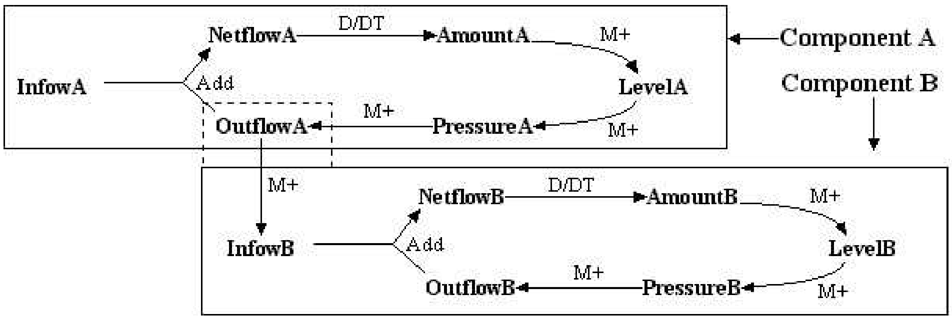
[2] The ordering of the simulation is slightly different since the description for component A could be infinite. For this reason, the simulation of component A is extended as additional information is required to drive the simulation of component B.



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 1.2: 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 1.3: 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 1.4 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 1.5(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 1.5(b) shows how a cycle exists within the corresponding component graph.

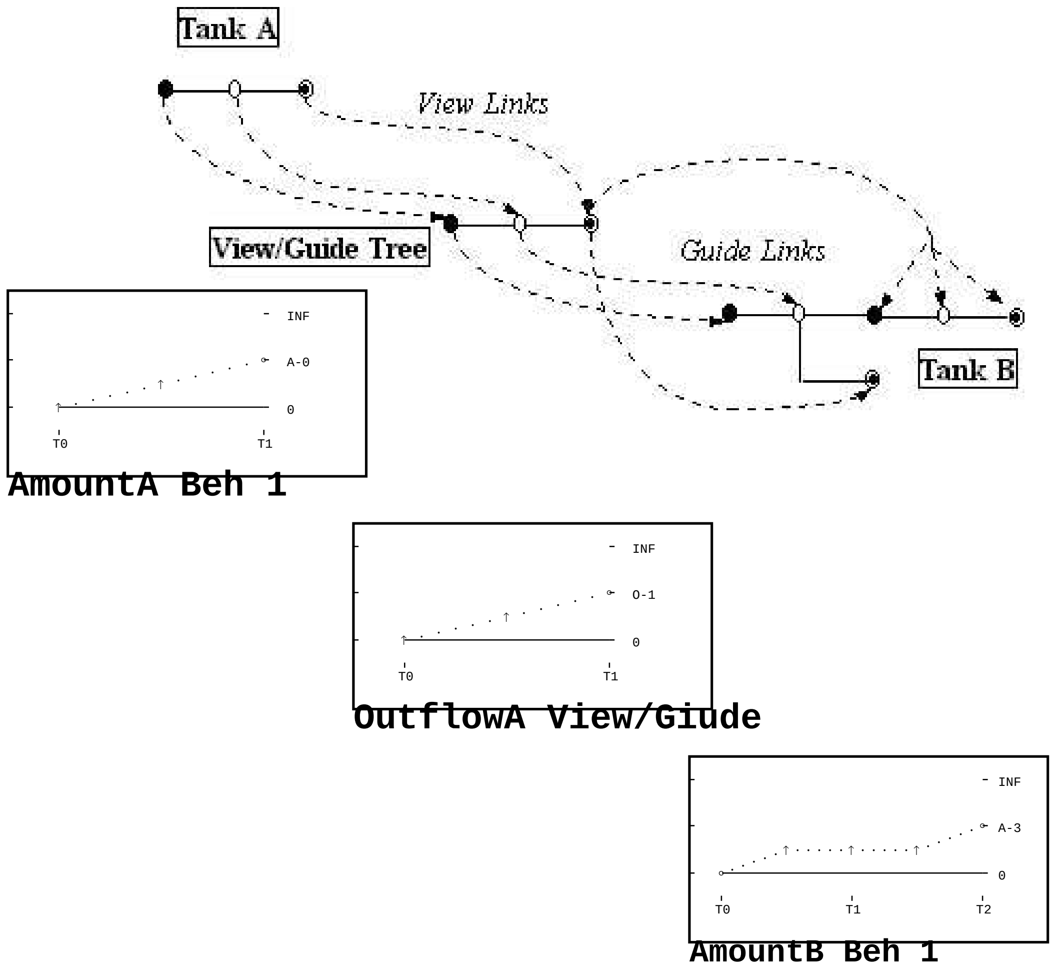
Simulation of this example generates three separate component trees with three view/guide trees providing a mapping from states in component A to states in the controller component, from the controller component to B and finally from B back to A (figure 1.4.) For a state to be globally consistent, a 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.

1.2.1 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 1.6 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 1.9 and 1.10 we discuss related and future work respectively.

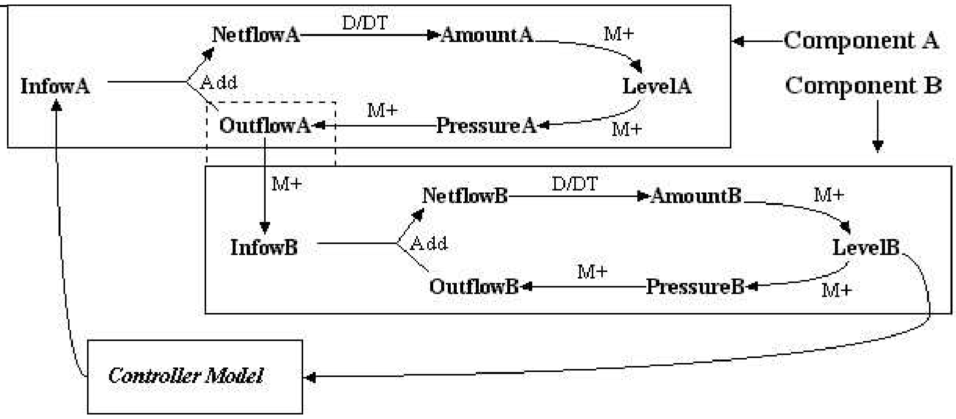
1.3 Components, the component graph and clusters

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

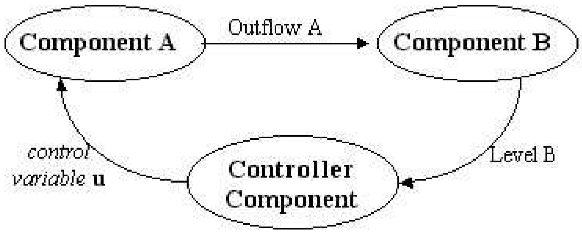


- DecSIM generates a *view/guide* tree when simulating component A. This tree is used to control the behavior of *OutflowA* in the simulation of component B.
- DecSIM maintains view links from the component tree for A to the view/guide tree and also asserts guide links from the view/guide tree to component 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 1.4: DecSIM simulation of the two tank cascade



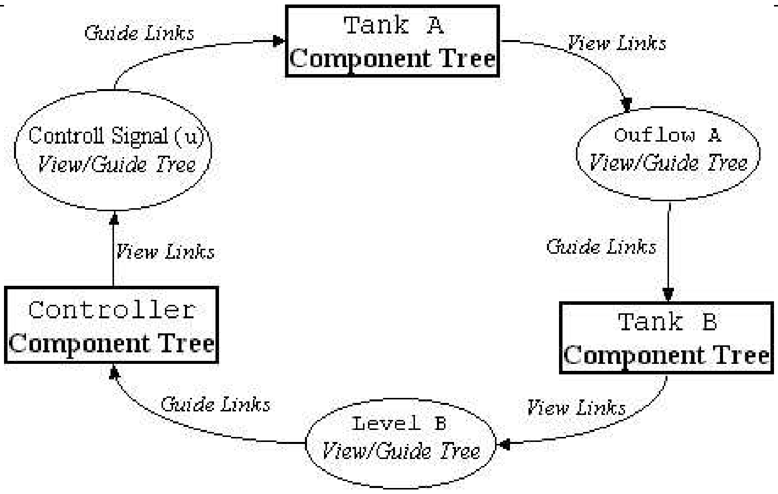
(a) Controlled two tank cascade



(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 1.5: 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 1.6: 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*(`Con`) represents the set of variables within `Con`, *Variables*(`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 1.6.4 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.

1.3.1 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 1.1 (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.

1.3.2 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'.

1.3.3 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 1.1 *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. □

1.3.4 Component graph

Definition 1.2 (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 1.3 (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 $A \overset{v_{ab}}{\to} B$ 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 1.4 (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]

1.3.5 Identifying clusters

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

Definition 1.5 (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 : $C \to 2^C$ *maps a component to the cluster (i.e. set of components) containing it.*

Lemma 1.2 (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 1.6 (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 1.3 *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. □

1.4 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 $A \overset{v_{ab}}{\rightarrow} B$, 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 1.7 (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.

1.4.1 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 1.7 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 1.8 (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 1.9 (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 $P_{vars}(S_T) = S_{T_{vars}}$.

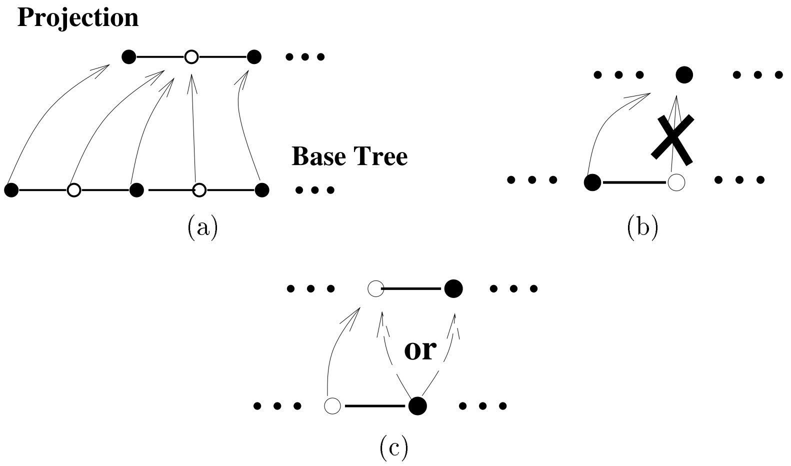
Each state added to the upstream component tree is either mapped to an existing state within the view/guide tree or the view/guide tree is extended and a new state is introduced. Formally, the mapping defined by the view/guide link defines a function $\mathit{ViewedBy}_{v_{ab}} : S_u \rightarrow S_{v/g}$ where S_u and $S_{v/g}$ correspond to states in the upstream component tree and the view/guide tree respectively. $\mathit{ViewedBy}_{v_{ab}}$ 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 1.1, generates the view/guide tree and defines the function $\mathit{ViewedBy}_{v_{ab}}$.

1.4.2 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 $\mathit{GuideOf} : S_{v/g} \rightarrow 2^{S_d}$ 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 $\mathit{GuideOf}(s)$ is equivalent to the inverse mapping function $\mathcal{P}^{-1}$ defined by the projection operator. Figure 1.2 describes the *guide link mapping algorithm* used to maintain the guide mapping links.

The mapping function $\mathit{GuideOf}$ 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 $s_d \in \mathit{GuideOf}(s_{v/g})$ 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 1.7: 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 $\mathit{ViewedBy}_{v_{ab}}$. 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 $\mathit{ViewedBy}_{v_{ab}}(s) = s_{v_{ab}}$ and return.
2. If $s =_{v_{ab}} \mathit{Pred}(s)$ and either s is a time–point state or $\mathit{ViewedBy}_{v_{ab}}(\mathit{Pred}(s))$ is a time–interval state, then set $\mathit{ViewedBy}_{v_{ab}}(s)$ to $\mathit{ViewedBy}_{v_{ab}}(\mathit{Pred}(s))$ and return.
3. If there exists an s' such that $s' \in \mathit{Succ}(\mathit{ViewedBy}_{v_{ab}}(\mathit{Pred}(s)))$ and $s' =_{v_{ab}} s$, then set $\mathit{ViewedBy}_{v_{ab}}$ 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 $\mathit{ViewedBy}_{v_{ab}}(\mathit{Pred}(s))$, set $\mathit{ViewedBy}_{v_{ab}}(s)$ to $s_{v_{ab}}$ and return.

Table 1.1: 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, $\mathit{pred}(s_d) \in \mathit{GuideOf}(\mathit{pred}(s_{v/g}))$, or $\mathit{pred}(s_d) \in \mathit{GuideOf}(s_{v/g})$.

1.4.3 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 1.10 (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 1.4 (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 **??**.

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 1.2: 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 1.11 (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.*

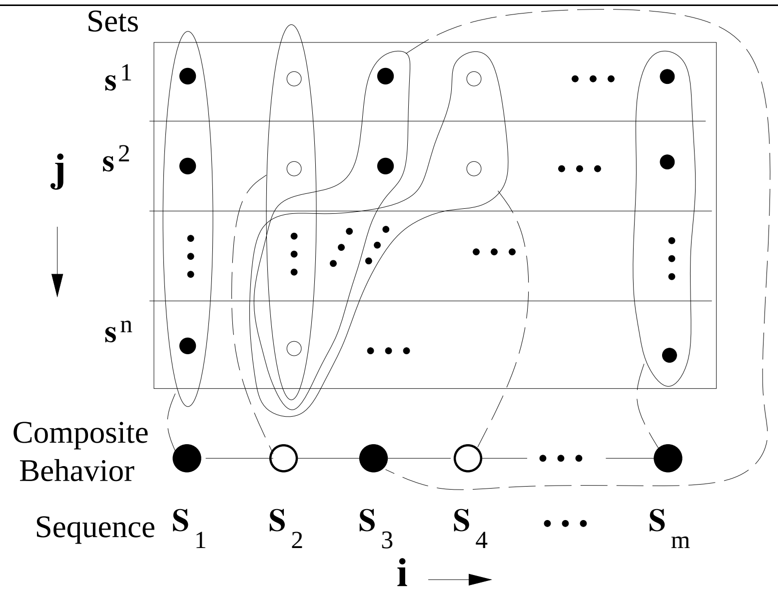
Definition 1.12 (Behavior composition) *A set of component behaviors $\{b_1, b_2, \ldots, b_n\}$ can be* composed *if and only if there exists a* composite behavior *B such that for all $1 \leq i \leq n$, $\Pi_{v_i}(B) = b_i$ where v_i corresponds to the set of variables described by b_i. Furthermore, 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 1.8 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 1.8: 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 1.1 (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* $C_i \stackrel{v_{ij}}{\to} C_j$ *is an edge in the component graph then* $M_{C_i \stackrel{v_{ij}}{\to} C_j}(s^i_n, s^j_m)$ *is true, and*
- *if* $C_j \stackrel{v_{ji}}{\to} C_i$ *is an edge in the component graph then* $M_{C_j \stackrel{v_{ji}}{\to} C_i}(s^j_m, s^i_n)$ *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 **??**.

Theorem 1.2 (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 **??**.

1.4.4 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 1.13 (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 1.14 (Global consistency) *A component state s is* globally consistent *if and only if there exists a solution to the component graph containing s.*

Theorem 1.3 (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 1.1 each pair of behaviors b_i and b_j are compatible. By theorem 1.2 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 1.2 if a set of behaviors is compatible then each pair must be compatible. By theorem 1.1, 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. $\square$

1.4.5 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 1.15 (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 $C_i \stackrel{v_{ij}}{\rightarrow} C_j$*

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

Definition 1.16 (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 1.2 in section 1.3.5, 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 1.17 (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 1.5 *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 $C_i \overset{v_{ij}}{\to} C_j$ and $C_i \notin Cluster(C_j)$, there exists a cluster supported state s_{c_i} such that $M_{C_i \overset{v_{ij}}{\to} C_j}(s_{c_i}, s_{c_j})$ 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. $\square$

Lemma 1.6 *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 $\mathcal{U}(C)$. $\square$

Theorem 1.4 *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 1.5 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 1.6, 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. □

1.4.5.1 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 1.3 and 1.4. 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.

1.4.6 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 $C' \overset{v_{ab}}{\rightarrow} C$ 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 1.3: 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 1.4: 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 1.5 regarding the soundness and completeness of the DecSIM algorithm with respect to behavioral description generated.

Lemma 1.7 (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 1.4. The global consistency test algorithm marks a state as globally consistent if and only if it is fully supported. Thus, by theorem 1.4 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 1.8 (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 1.6 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. $\square$

Theorem 1.5 (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 1.7, s is labeled globally consistent if and only if there exists a solution to the component graph. Lemma 1.8 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. $\square$

Lemma 1.9 *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 1.10 *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 1.3, 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.

1.5 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 1.6 (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 1.4, there exists a solution $sol = \{s_1, s_2, \ldots s_n\}$ to the component graph containing s_i. By theorem 1.3, 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 $i : 1 \leq i \leq n :: \Pi_{V_i}(B) = b_i$. Finally, by the QSIM soundness guarantee we know that QSIM will produce B. □

Theorem 1.7 (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 $i : 1 \leq i \leq m :: \Pi_{V_i}(B) = b_i$.*

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 1.5 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 1.2, 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 1.9 and 1.10), we know that DecSIM will generate all of the states in this set.

(4) By theorem 1.5, 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 1.8 *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 1.6 and 1.6 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. □

1.6 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.

1.6.1 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.

1.6.2 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 1.18 (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.

1.6.3 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 1.19 (Locally quiescent) *A component state s is labeled* locally quiescent *if and only if all of the variables within s are steady.*

Definition 1.20 (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.

1.6.4 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).

1.6.5 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.

1.7 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.

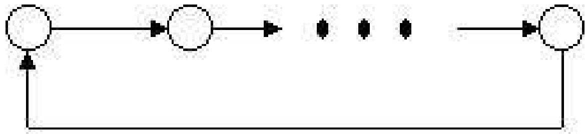
1.7.1 Simulation complexity

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:

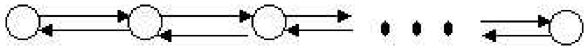
(1) **Cascade topology:** A sequence of N cascaded tanks provides a causally sequenced chain of components. This topology clearly demonstrates the benefits of a component–based simulation. In addition, this topology tests the impact of causality along with the partitioning selected.

(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 1.9: 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 1.9 provides a graphical representation of the topology for each model.

1.7.1.1 DecSIM vs QSIM

In all of the models tested, DecSIM was both exponentially faster and used exponentially less memory than QSIM. Figures 1.10 through 1.12 display plots demonstrating these results. In each of the plots, the x-axis shows the number of components

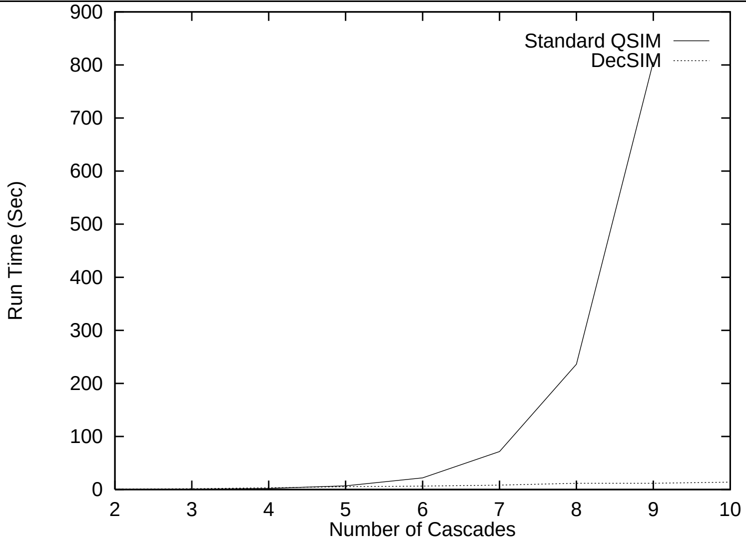
Number of Comp's	Cascade		Chained		Loop	
	QSIM	**DecSIM**	**QSIM**	**DecSIM**	**QSIM**	**DecSIM**
2	0.204	0.815	3.075	6.79	0.757	5.587
3	0.621	1.6	10.94	19.903	16.149	8.147
4	2.2	3.12	37.55	25.984	89.418	12.67
5	7.09	5.49	139.3	36.712	493.88	23.28
6	21.92	6.32	676	62.405	2758	48.73
7	71.59	8.39	1633	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 1.5: Results comparing DecSIM to QSIM for all three topologies

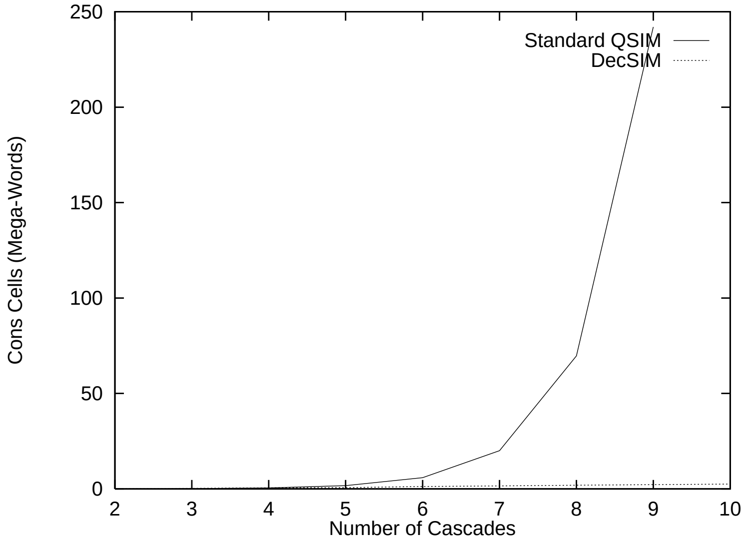
within the model while the y-axis shows either simulation time or space. Table 1.5 lists the data used to generate these plots. Each component is comprised of five variables, so the number of variables in each model is simply $5n$ where n is the number of components. All of the tests were run on a Sparc10 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 1.6 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

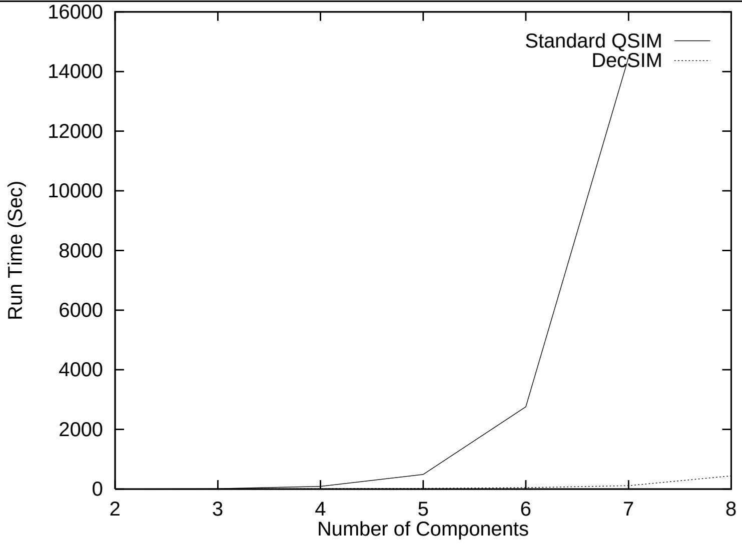


(a) Execution Time

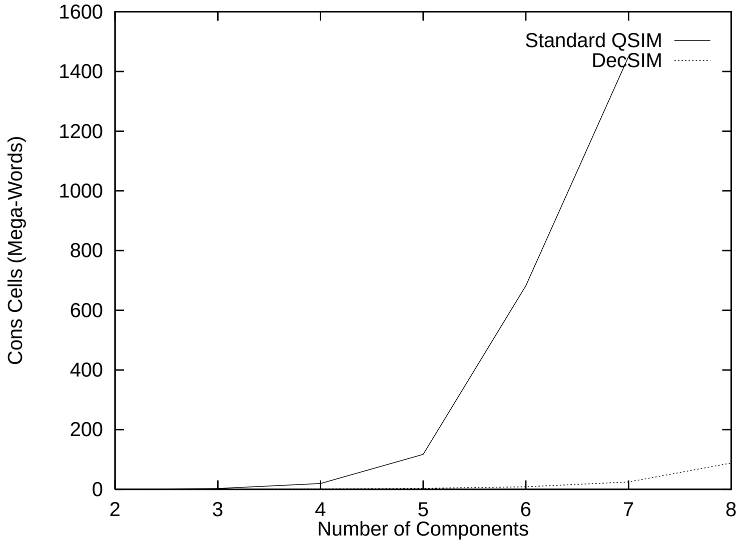


(b) Space

Figure 1.10: DecSIM vs QSIM: Cascade configuration

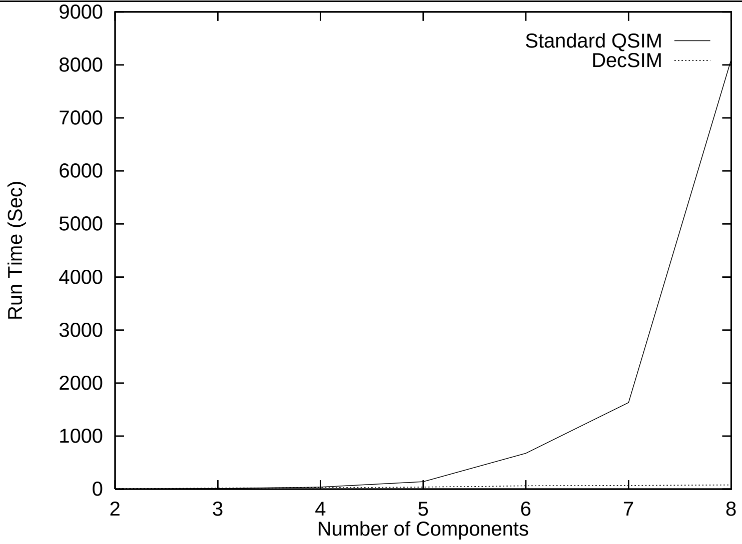


(a) Execution Time

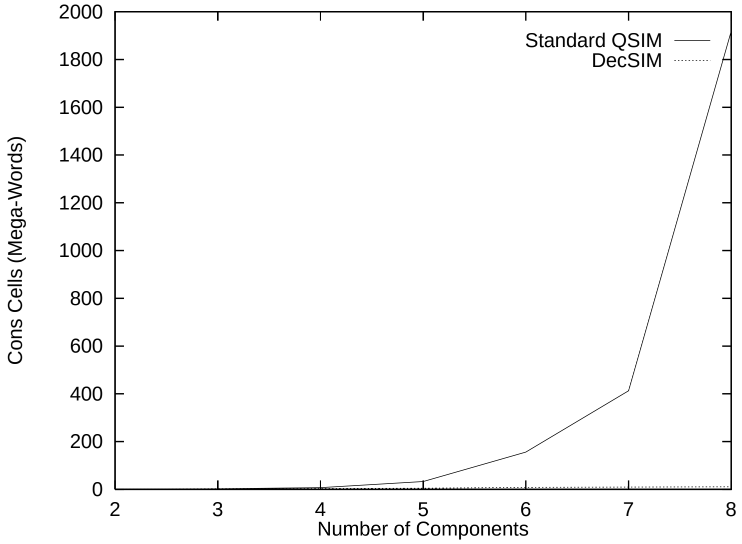


(b) Space

Figure 1.11: DecSIM vs QSIM: Loop configuration

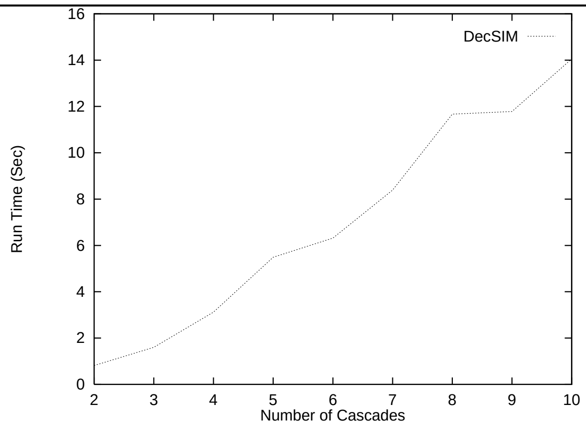


(a) Execution Time

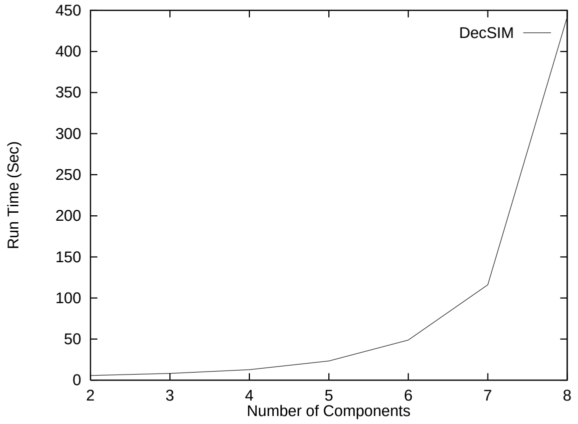


(b) Space

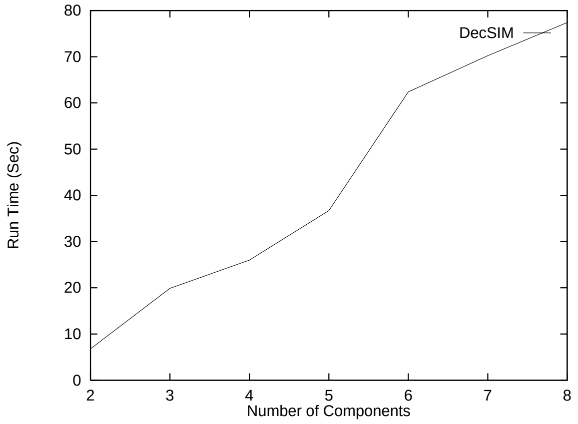
Figure 1.12: DecSIM vs QSIM: Chain configuration



(a) Cascade configuration



(b) Loop Configuration



(c) Chain configuration

Figure 1.13: 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 1.6: 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 1.13 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 1.11. Figure 1.14 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

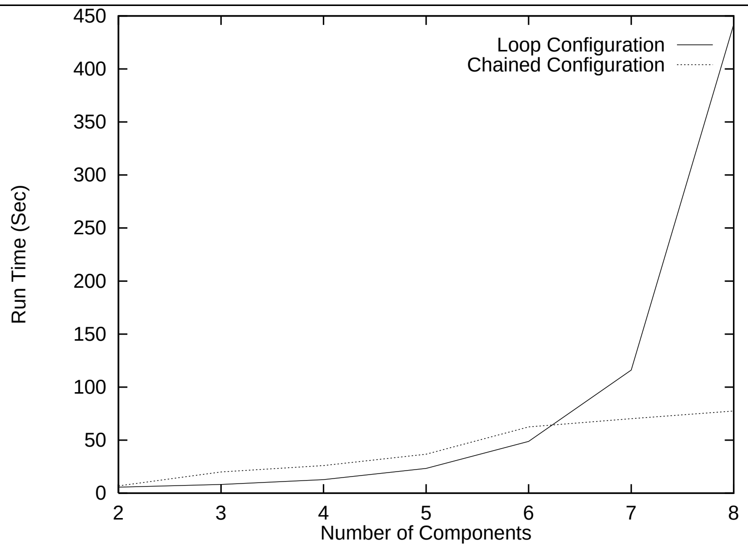


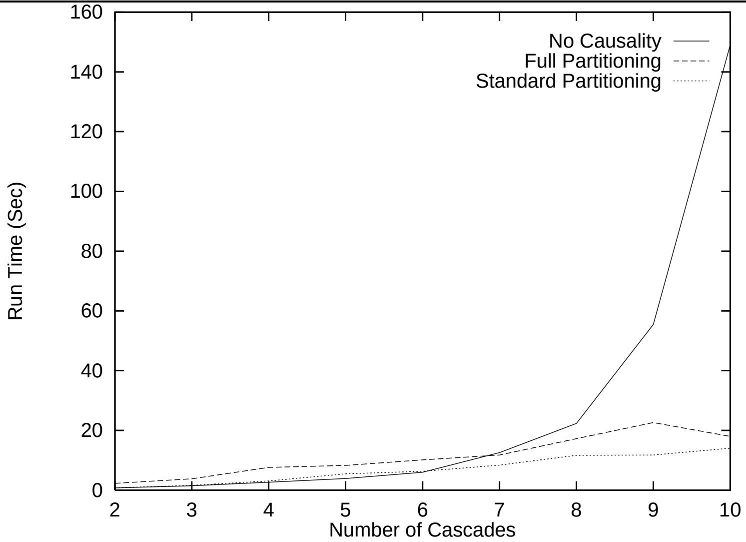
Figure 1.14: DecSIM: Chain configuration vs Loop configuration

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 1.15 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.

1.7.2 Soundness and completeness

To validate the theoretical results presented in section 1.5, we tested each of the models within the evaluation corpus (see table **??** in section **??**) 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 1.15: 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 1.5.

1.8 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 1.7 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.

1.8.1 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.

1.8.2 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:

- $\Rightarrow$ *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.*

- $\Rightarrow$ *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.

1.8.3 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 1.7 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.

1.9 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.

1.9.1 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.

1.9.2 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.

1.10 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.

1.10.1 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).

1.10.2 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 1.6.4, a number of alternative solutions exist. The solution selected will depend upon other extensions such as the integration of DecSIM with CC described above.

1.10.3 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.

1.10.4 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.

1.11 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.

Bibliography

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.

Brajnik, G. (1995). Introducing boundary conditions in semi–quantitative simulation. In *Ninth International Workshop on Qualitative Reasoning*, pp. 22–31 Amsterdam.

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.

Coiera, E. W. (1992). Qualitative superposition. *Artificial Intelligence*, *56*, 171–196.

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., & 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.

Even, S. (1979). *Graph Algorithms.* Computer Science Press, Rockville, Maryland.

Forbus, K. (1984). Qualitative process theory. *Artificial Intelligence*, *24*, 85–168.

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., & 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.

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.

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.

Rickel, J., & Porter, B. (1997). Automated modeling of complex systems to answer prediction questions. *Artificial Intelligence*, *93*, 201–260.

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press, San Diego, CA.

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.