A THEORY OF TELEOLOGY

APPROVED BY

DISSERTATION COMMITTEE:

_____

_____

_____

_____

_____

To Leo, Elmer, Verna and Hazel

A THEORY OF TELEOLOGY

by

DAVID WAYNE FRANKE, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 1992

Acknowledgments

Although the completion of a doctoral program is recognized as an individual's achievement, such an accomplishment is only possible with the guidance and support faculty, friends, and family. I attempt to acknowledge here those who have provided guidance, support, and inspiration for this work.

My advisor, Professor Benjamin Kuipers, introduced me to this research area and provided the seeds for the ideas described in this dissertation. His patient guidance and constructive criticism have contributed greatly to the completion of this research and dissertation. Professor Allen Emerson, Professor Ray Mooney, Professor Bruce Porter, and Dr. Michael Huhns provided many insightful comments, questions, and suggestions in reviewing and evaluating this research. These committee members and the Computer Sciences faculty at the University of Texas at Austin have provided a stimulating and challenging environment for graduate work. Professor Kuipers is also responsible for building an excellent research group in Qualitative Reasoning at the university, and I have benefited greatly through my association with this group. In particular, I would like to thank Dan Berleant, David Bridgeland, Dan Clancy, Dan Dvorak, Adam Farquhar, Bert Kay, and Raman Rajagopalan for many interesting discussions and useful ideas.

The larger qualitative reasoning, functional reasoning, and model-based reasoning communities have been a wealth of ideas and support. I would like to acknowledge the fruitful interaction with: the Functional Reasoning

community at Ohio State University, particularly B. Chandrasekaran, Ashok Goel, and Jon Sticklen; the model-based reasoning group at Vanderbilt, particularly Gautam Biswas and Stefanos Maginaras, for their interest and help in building CC; and Rich Doyle of the Jet Propulsion Laboratory.

For their support and encouragement over the years and for their continuing friendship I thank David Burgess, Carroll Hall, Bill Read, and Bill Turpin.

Most important is the love and support of my family. My parents, JoAnn, Loren, Rose, and Leo have provided constant encouragement. Melissa, Megan, and Debbie have sacrificed their time and activities that I might complete this accomplishment. I love you all.

David Wayne Franke

The University of Texas at Austin
May 1992

A THEORY OF TELEOLOGY

Publication No._____

David Wayne Franke, Ph.D.
The University of Texas at Austin, 1995

Supervising Professors: Benjamin J. Kuipers

A representation language for teleological descriptions, or descriptions of purpose, is defined. The teleology language, TeD, expresses the descriptions of purpose in terms of design modifications that guarantee the satisfaction of design specifications. These specifications express potential behaviors the designed artifact should or should not exhibit. We define an abstraction relation on behavior and implement model checking and classification algorithms that compute this abstraction relation. The model checking algorithm determines whether or not a behavior satisfies a specification. The classification algorithm provides effective indexing of behaviors and teleological descriptions. We implement an acquisition technique for teleological descriptions and demonstrate how teleological descriptions can subsequently be used in diagnosis, explanation, case-based reasoning, design by analogy, and design reuse.

We demonstrate the behavior language, teleology language, acquisition of teleological descriptions, and application of teleological descriptions in explanation, diagnosis, and design reuse via examples in the thermal, hydraulic,

electrical, and mechanical domains. We define additional teleological operators that express purposes like *prevent*, *order*, *synchronize*, *maintain*, and *regulate*, demonstrating the ability to represent common human-generated descriptions of purpose in TeD. Expressing the purpose of *preventing* an undesirable behavior is unique to TeD, and is an example of TeD's ability to express purposes regarding missing behaviors and components removed from a design.

The teleology language developed in this work represents a significant advance over previous work by providing a formal language that 1) is independent of any particular domain of mechanisms or behavior language, 2) can be effectively acquired during the design process, and 3) provides an effective means of classifying and indexing teleological descriptions.

Table of Contents

List of Tables

List of Figures

Chapter 1

On Describing Purpose

1.1 Teleology

The representation of real-world systems and mechanisms initially was concerned with the structure of such systems, and subsequently evolved to issues of representation and derivation of the behaviors of these systems. Examination of real-world system behavior gave rise to derivation and understanding of causal relationships. Given this base of representation and derivation techniques for structure, behavior, and causality, the representation and derivation of descriptions of *teleology* or *purpose* is presented as the next ingredient of human understanding of real-world systems to be studied. We assume that real-world systems are designed to achieve specific behaviors, and that each component and subsystem has been included in the design to contribute in some way to these behaviors.[1] In teleology we intend to capture the manner in which a component, at any level of the structure hierarchy, contributes to the behaviors of its ancestors in the structure hierarchy.

One can understand both the utility of descriptions of structure, behavior, causality, and purpose and the differences among these descriptions by considering the questions that can be answered with this information. A

[1] As de Kleer points out [deK85], the goals of efficient design, manufacture, and maintenance of artifacts dictate that designers avoid superfluous components in the design, and hence each component should contribute to the ultimate purpose of the design in some way.

structure description addresses questions of the form "How is this mechanism constructed?" and "What are the physical (static) characteristics of this mechanism?" A behavior description addresses questions of the form "What does the mechanism do?" or "What are the dynamic characteristics of the mechanism?" Representations of structure and behavior provide the framework for a class of problem-solving techniques called *model-based* reasoning (*cf.* Davis and Hamscher's discussion of model-based troubleshooting [DH88] and the proceedings of the model-based reasoning workshops [MBR89, MBR90, MBR91]). Causal reasoning techniques build on this framework, providing analyses and descriptions that address questions of the form "How does the mechanism accomplish its behavior?" Finally, a teleological description addresses questions of the form "Why is this portion of the mechanism designed in this way?" or "What is the purpose of this piece of the mechanism?"

When examining human-generated descriptions of systems or mechanisms, one finds that they are rich with descriptions of purpose, as well as descriptions of structure, behavior and causality. In fact, descriptions of purpose are very valuable in communicating and understanding design descriptions, since they convey an aspect of the design process here-to-fore not (automatically) captured or derivable from a declarative design description, namely the designers' intent. Consider the following design description segment in which Herbert and Williams [HW87, p. 91] discuss modeling a pressurizer subsystem of a power plant:

> *Figure 3 shows a schematic of a pressurizer subsystem, the main purpose of which is to control the pressure of the primary circuit by maintaining a steam-water interface within the vessel through the controlled addition of heat and water spray.*

This brief excerpt contains a description of purpose, "*... to control the pressure of the primary circuit ...*" as well as a causal account, "*... by maintaining a steam-water interface within the vessel ....*" An examination of the substructure would show components whose purposes were "*to maintain a steam-water interface*," "*to control the addition of heat*," and "*to control the addition of water spray*."

The thesis of this work contends that the description of purpose of a component, mechanism, or activity can be expressed in terms of specifications or requirements for the system in which the component, mechanism, or activity is embedded. These specifications describe static (e.g., physical dimensions) and dynamic (behavior) characteristics of the system. Specifically, descriptions of purpose are expressed as *guarantees* that these specifications hold for the design. Further, these descriptions can be captured during the design process or by examining design histories, and can subsequently be used in explanation, diagnosis, and design.

As an example of this approach, consider an electrical engineer designing an input selection circuit.[2] The engineer begins with specifications describing the desired static (e.g., size) and dynamic (behavior) characteristics of the resulting circuit design. For example, one behavior specification for the input selection circuit is "invert the data signal when the control signal is high (logic true) and leave the output unchanged when the control signal is low (logic false)." The engineer also begins with specifications from the domain of CMOS circuit design, such as "the input value to a logic gate should not maintain a steady, intermediate value (voltage) between low and high, causing

[2]This design example is discussed in detail in Chapter 6.

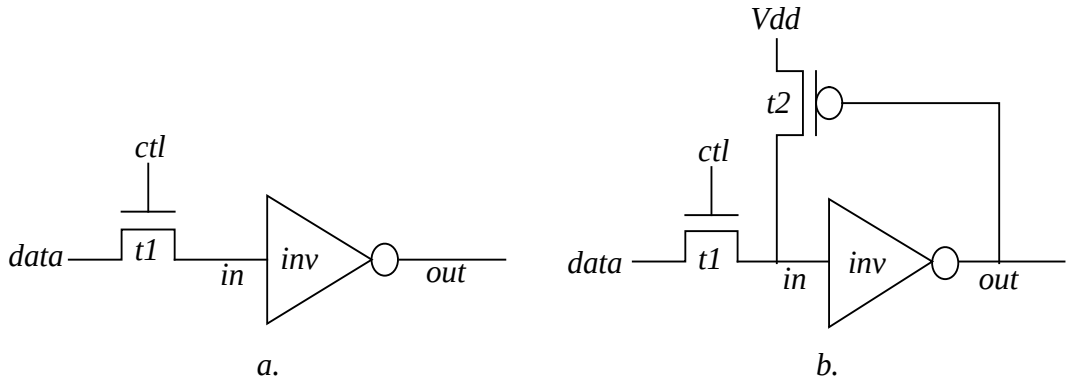


Figure 1.1: CMOS Input Selection Circuit - Schematics

the gate to consume power by allowing current to flow."

Interacting with a design system, the engineer proposes a design, evaluates the design with respect to the specifications, and makes modifications to the design in an attempt to make the design meet the specifications. In this interaction, teleological descriptions can be acquired and subsequently used to assist the engineer in completing the design, as demonstrated in the following example.

Engineer: Generate the possible behaviors of design 1 (*see schematic in Figure 1.1a.*) for signal `ctl` high and signal `in` transitioning from low to high.
Computer: The possible behaviors are: (*displays the behaviors*).
Engineer: Evaluate the behaviors with respect to the design specifications.
Computer: For signal *in*, the design does not meet specification 5: "the input value to a logic gate should not maintain a steady, intermediate value (voltage) between low and high, causing the gate to consume power by allowing current to flow." (*If the design system has a database of teleological descriptions and a teleological description referencing this specification exists in the database, a recommendation for modifying the design can be made.*)

Engineer: (*Modifies the design by adding feedback transistor t2, as shown in the schematic in Figure 1.1b.*) Generate the possible behaviors of design 2 for signal `ctl` high and signal `in` transitioning from low to high.
Computer: The possible behaviors are: (*displays the behaviors*).
Engineer: Evaluate the behaviors with respect to the design specifications.
Computer: The design meets all specifications. The purpose of the design modification transforming design 1 into design 2 is to guarantee specification 5.

We define representation languages for behavior descriptions, specifications, and teleological descriptions that allow acquisition, classification, and indexing of teleological descriptions. With these capabilities we can realize the design process described in this example.

1.2 Teleology, What's the Purpose?

We have characterized teleological descriptions as addressing questions of the form "Why is this portion of the mechanism designed in this way?" or "What is the purpose of this piece of the mechanism?" This identifies one important use of teleological descriptions, namely explanation. de Kleer's EQUAL system [deK85] generates teleological descriptions (of a form different than that defined herein) for the purpose of explanation of electrical circuits. The need for representing and expressing the purpose of a particular command or command sequence is identified for the LOX Expert System (LES) [SJD85], both for human consumption and for analysis by automated systems.

The ability to express such explanations implies their use not only by humans but also by systems that automate problem-solving tasks. The task

domains of diagnosis and design are two domains that can utilize teleological descriptions to extend the applicability and performance of automated problem-solving systems. Steinberg and Mitchell [SM84] point out that in the domain of VLSI circuits, the role of information regarding the purpose of a circuit element is very similar in the problem-solving tasks of design and diagnosis.

1.2.1 Diagnosis

Deriving and utilizing causal relationships is an approach currently used in explanation systems (e.g., Bylander and Chandrasekaran [BC85], Doyle [Doy86], and Kuipers [Kui87]) and diagnosis systems (e.g., Cantone et al. [CLMG85], Davis [Dav85], De Jong [DeJ85], Genesereth [Gen85], Milne [Mil85], Sembugamoorthy and Chandrasekaran [SC85], Steinberg and Mitchell [SM84], and White and Frederiksen [WF85]). However, in mechanisms with highly interconnected structure or feedback loops, causal relationships can exist between virtually every pair of components of the mechanism (and between variables of a mechanism model). As stated by Steinberg and Mitchell [SM84],

> *The resulting focus* (of causal reasoning) *is generally broader than that determined from [the representation of purpose] because out of the many places in the circuit that can impact any given output specification, only a small proportion of these involve circuitry whose main purpose is to implement that specification.*

In diagnosis, domain specific heuristics can be applied to select among potential causes, but are not applicable outside their particular domain. If an observed symptom of a mechanism is considered either as an unwanted behavior (or a missing behavior), then a teleological description that relates a component of the mechanism with the prevention (introduction or guarantee) of that

behavior provides a heuristic for selecting among potential causes. Certainly, teleological descriptions would not introduce any relationship not discovered via causal analysis, as purpose necessarily requires causality. Hence, teleological descriptions can provide a more productive initial focus of attention for diagnosis.

1.2.2 Design

Mostow [Mos85] discusses the potential for improving the design process through capture and representation of design rationale. Franck [Fra89] points out that "Design is a form of teleological reasoning, in that from the intended purpose or anticipated behavior one can select elements that have the adequate structure to do so." Teleological descriptions provide a means for representing design rationale. de Kleer [deK85] notes that disciplines such as electrical engineering have developed specialized vocabularies for denoting the purpose of mechanisms and components. Given the ability to capture and represent teleological descriptions (either generated by humans or programs), these descriptions can be used to classify mechanism descriptions. Further, such descriptions can be used to index other design information, whose relevance is determined by the current concern of the designer (e.g., component size). The ability to realize these descriptions in a formal language will facilitate their use as knowledge about the design. Assuming that such a language can be identified that is independent of any particular domain of mechanisms, the language will allow teleological descriptions for mechanisms that include components from several domains, such as electrical, mechanical, and hydraulic.

1.2.3 Design Reuse

Design reuse is an area of current research in both CAE (computer aided engineering) environments (see work reported by Mostow [Mos85], Mostow and Barley [MB87], and Huhns and Acosta [HA88]) and CASE (computer aided software engineering) environments (see discussions by Biggerstaff and Richter [BR86] and Prieto-Diaz and Freeman [PF87], and collections of papers examining reuse in software engineering [IEEE88, IEEE87, IEEE84]). In each, the reuse problem can be addressed by providing:

1. Techniques for capturing and representing information by which a design component should be classified for subsequent retrieval, and

2. A language for describing the characteristics of the design component the designer wishes to examine as a candidate for reuse.

As stated by Mostow and Barley [MB87] in discussing reuse of design plans,

> *For design by analogy, finding a suitable design to retrieve from a repository of previous designs requires knowing where to look. How can designers avoid a time-consuming search through such a repository when they've never seen the relevant entry or can't remember where to find it? As we develop a larger database of design plans, we expect the process of finding relevant ones to become a bottleneck*
> ...

Teleological descriptions add another dimension by which designs can be classified and retrieved. For example, a designer may wish to examine components that can control some variable (say tank fluid level) of a system

under design. In the absence of a description of purpose, designers must rely on their mental inventory of likely components, structural features of likely components, or specific behaviors of likely components in order to construct a query for the search. Further, designers will more likely miss innovative solutions that do not fit their current mental model of how to solve the problem (i.e., what kind of component to use), such as an analogous problem solution from another design domain (e.g., electrical vs hydraulic).

Current reuse approaches are based on structure classification and hierarchy or on classifications organized around keywords that represent behavioral categories. The problem with classification based on structure (and keywords that correspond to specific structure abstractions) is that potential reusers must know the specific design structure or abstraction in order to find that design. Further, the designer has to know that the structure being requested actually addresses the problem to be solved, such as realizing a specific behavior. The problem with a classification based on behavior, represented either by keywords or by explicit behaviors and their abstraction, is twofold. First, the candidate components that might solve the designer's current problem may have component level characteristics (e.g., behavior) that have no obvious correlation to the current design problem, such as eliminating some undesirable system level behavior. Second, indexing designs with respect to their behavior does not help in situations in which the designer wants a candidate design that does not exhibit certain undesirable behaviors. In summary, indexing designs and design components for reuse via teleological descriptions provides more semantic content for the reusing designer.

In reusing an existing design, if the design does not match the current requirements exactly, it will require some modification. As this design is being

modified, knowledge of the purpose of components will benefit the (resuing) designer in much the same way teleological descriptions aid the diagnosis task. The task is driven by a change in the requirements, additional required or prohibited behaviors in reuse and modified behavior on the part of the mechanism in diagnosis. When a requirement changes, a teleological description relating a particular design decision to that requirement gives the reusing designer an initial set of candidates for modification. Teleological descriptions also help the designer understand the original purpose of components in the mechanism design. For example, a particular component may have been selected for certain static properties like size, as well as the behavioral consequences of the component. Steinberg and Mitchell's REDESIGN system [SM84] applies both causal reasoning and reasoning about purpose to accomplish the redesign of a circuit based on changes in the specification for the circuit. As was claimed for diagnosis, REDESIGN uses teleological descriptions to provide tighter focus for selecting candidate components for modification.

1.3 A Life-Cycle Model

To organize these uses of teleological descriptions, we describe a life-cycle for engineered mechanisms and identify the role of teleological descriptions in the various life-cycle stages. The life-cycle has the following stages (Figure 1.2):

- Design stages
  - Product specification
  - Initial design
  - Design evaluation

  - Design modification

- Support and Reuse
  - Explanation
  - Diagnosis
  - Design reuse

The design stages capture teleological descriptions and can use teleological descriptions to achieve design reuse. Teleological descriptions captured during design can be used in later stages when diagnosing, explaining, or reusing the designed artifact.

The design process model used here is of the Propose-Critique-Modify family described by Chandrasekaran [Cha90]. Our design process model starts with a set of specifications for the design, including physical characteristics and descriptions of (required, prohibited, ...) behaviors. In addition to the specifications of a particular design, there is often a set of specifications, or design principles, that describe general engineering practice for the domain at hand. These specifications include characteristics of the design required for manufacturing, maintenance, standards conformance, and regulatory requirements.

Given specifications for the design, the design process proceeds as a series of structure modifications, starting from some initial structure. Accompanying this series of structure descriptions is a corresponding series of evaluations of the design with respect to the various design specifications. Evaluation of dynamic characteristics (e.g., functional correctness, performance, and thermal operating characteristics) and static characteristics may require complex computations like simulation, timing analysis, formal verification of function,

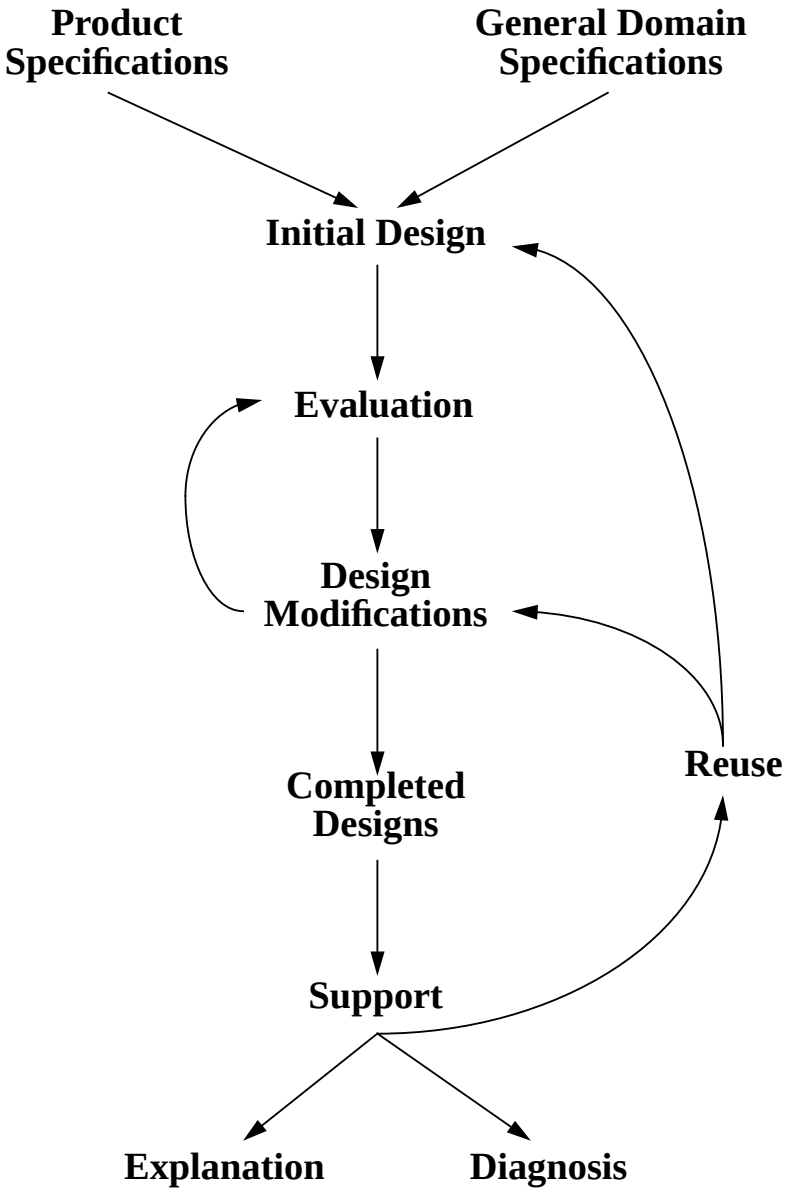


Figure 1.2: Life Cycle Model

and thermal modeling. Ideally, this process continues until evaluation shows that all specifications have been satisfied, or amended so as to be satisfied. By examining the changes to the structure in light of the specifications, descriptions as to the purpose of the design modifications can be inferred, as we will demonstrate with an example.

A hypothetical design system described by Abelson et al. [AEH*89] supports interaction between a designer and an intelligent computer assistant that aides in the analysis and evaluation of a proposed design. During interaction with the design assistant, the designer identifies an undesirable behavior of the design (specifically, an oscillation at a particular frequency). The designer then creates a design modification to correct this behavior ("an active stabilizer to damp the family B motions"). This is precisely the point at which a description of purpose can be acquired. The purpose of the addition of the "active stabilizer" is "to damp the family B motions." The work described herein defines a language for representing and indexing such purposes.

1.4 Previous and Related Work

Representation, acquisition, and application of descriptions of purpose have been addressed in previous research, which we summarize here. Detailed comparison of the work described in this dissertation (i.e., the Teleological Description (TeD) language) with previous research appears in Chapter 10.

The two most significant contributions discussed in the literature are de Kleer's EQUAL system [deK85] and the *Functional Representation* [SC85] work at Ohio State University. Representing purpose in design systems is addressed in Steinberg and Mitchell's REDESIGN system [SM84], and an approach to diagnosis called the *theory of responsibilities* has been developed by

Milne [Mil85]. More recent efforts in representing purpose have been undertaken by the Conservation of Design Knowledge (CDK) Project [BSZ89] at NASA Ames Research Center, in Gruber's ASK system [Gru91], and in medical reasoning research in Downing's BIOTIC system [Dow90].

de Kleer's EQUAL system [deK85] expresses teleological descriptions in terms of behaviors of a component. Each description is based on causal assumptions on the parameters of the component. EQUAL identifies a functional characterization (teleological description) by matching derived behavior with prescribed behavior prototypes that have been enumerated, named, and added as domain specific knowledge. Limitations of this approach are that teleological descriptions are prescribed, domain specific, and limited to describing relationships among variables of a single component.

Functional representation (FR) described by Sembugamoorthy, Chandrasekaran [SC85], Goel [Goe89], Sticklen, and Bond [SCB89] and *functional modeling* (FM) described by Sticklen et al. [ST90, SKB90] address 1) representing "how a device functions" and 2) applying this information to explanation, diagnosis, and design. Each function definition in FR contains a **ToMake** clause that references a particular behavior that the FR function is supposed to achieve. Keuneke [Keu91] extends the **ToMake** clause of FR to include the "function types" **ToMaintain**, **ToPrevent**, and **ToControl**. The TeD language can formally capture the semantics of these "purposes," providing an effective means of classifying and indexing FR descriptions.

Steinberg and Mitchell's REDESIGN system [SM84] utilizes representations of purpose to focus the selection of candidate components for the redesign task. REDESIGN expresses teleological descriptions as "rules that embody [...] general knowledge about circuit design tactics." These rules spec-

ify decomposition steps for realizing a design in available components, and are captured independently of the design process and added to the design system as domain specific knowledge.

Milne [Mil85] describes an approach to automated troubleshooting called the *theory of responsibilities*. Responsibilities relate a particular component of a design to a desired output (behavior). Responsibilities are assigned automatically based on *second principles* representing "the type of description that an electronics engineer uses to describe various building blocks of circuits." This domain specific knowledge must be elicited from designers and represented, and the thoroughness of the responsibility assignments depends on the depth of understanding provided in the second principles.

The Conservation of Design Knowledge (CDK) Project [BSZ89] at NASA Ames Research Center addresses the problems of representing and acquiring design rationale using a philosophy similar to TeD. TeD provides a formal language for representing design rationale descriptions captured in the CDK acquisition work, and the CDK work complements TeD by providing acquisition techniques.

Downing's BIOTIC system [Dow90] critiques natural (e.g., human and reptilian) circulatory models with respect to *teleologies*, desired global behaviors of a system. The teleologies of BIOTIC correspond to design specifications of the TeD language, and hence are prescribed.

Gruber's ASK system [Gru91] elicits justifications from experts via an interactive dialogue with the expert. TeD provides a formal language for representing ASK explanations (teleological descriptions), provides indexing capabilities for ASK explanations, and addresses acquisition of teleological descriptions during design.

1.5 Claims of This Work

The claims of this work are:

1. Descriptions of purpose can be represented formally in a language that is independent of a particular domain of mechanisms or behavior description language (specifically the Teleological Description (TeD) language), and these descriptions of purpose can be expressed in terms of the primitive operators **Guarantees** and **unGuarantees**,

2. Descriptions of purpose can be effectively acquired in the design process given information available in current design methodologies, and

3. The representation language facilitates the classification and retrieval of descriptions of purpose for use in design explanation, design reuse, design by analogy, case-based reasoning, and diagnosis.

1.6 Outline of This Dissertation

The stated claims are achieved in the TeD (Teleological Description) language described herein. The basic ideas that formed this work are presented to familiarize the reader with the concepts of the approach. The TeD language is described in detail around a simple example: an idealized steam boiler design. We then describe behavior abstraction and properties of the TeD language based on this abstraction. Two designs are investigated in detail, an electronic circuit and an electromechanical motor, to demonstrate the breadth of the TeD language and acquisition of teleological descriptions. Applications of teleological descriptions in design explanation, design reuse, design by analogy, and diagnosis are discussed, and the details of acquisition, classification and retrieval of teleological descriptions for these tasks are presented. This

work is compared to related work, and a summary of the contributions of this work and potential extensions of the research are presented.

Chapter 2 describes the basic ideas of the teleology language TeD and its relationship to structure and behavior languages, namely that teleological descriptions relate design modifications expressed in the structure language to design specifications expressed in the behavior language. Further, a teleological description makes the statement that a modification is made to *guarantee* one or more design specifications for the design.

Chapter 3 defines an ontology for teleology and how these elements are expressed in the TeD language. The ontological elements are design specifications (behaviors), design descriptions (structure), and design modifications (a history of structure modifications). Based on these elements we formally define the teleological description language TeD.

Chapter 4 defines behavior abstraction and its role in specifications and teleological descriptions. Behavior abstraction permits the expression of design specifications involving 1) a subset of mechanism variables and 2) qualitative behavior. Behavior abstraction is the basis for verification of behaviors with respect to specifications and for classification and retrieval of teleological descriptions. We also define composed teleological operators to demonstrate how descriptions better matching human generated, prose descriptions of purpose can be formally represented.

Chapter 5 presents theorems describing generalization and specialization properties between teleological descriptions given generalization and specialization properties between behaviors referenced by these descriptions. The language properties described in these theorems support design reuse based on teleological descriptions.

Chapter 6 explores detailed examples in the electronic circuit and electromechanical domains, demonstrating the various languages (structure, behavior, teleology) and acquisition of teleological descriptions.

Chapter 7 discusses the use of teleological descriptions in the task domains of design explanation, design reuse, design by analogy, and diagnosis.

Chapter 8 describes an index structure based on behavior abstraction. This index structure supports the resolution of queries for explanation, design reuse, and diagnosis tasks discussed in Chapter 7.

Chapter 9 describes techniques for acquiring teleological descriptions, with details of the acquisition technique implemented in this work.

Chapter 10 reviews previous and related work in teleology and demonstrates where the TeD language has extended other work and how the other approaches can be expressed in the TeD language.

Chapter 11 summarizes the claims and their support, citing the contributions, current limitations, and potential extensions of this work.

Chapter 2

The Basic Idea

2.1 Goal

In Chapter 1 we described the task of designing an input selection circuit (Figure 1.1) to meet design specifications. We address the problem of representing the purpose of adding the feedback transistor *t2* (Figure 1.1b), that purpose being to eliminate the behavior in which the voltage at the inverter input (*in*) is a steady value between low (logic value 0) and high (logic value 1) voltage. The ideas described in this chapter are the conceptual basis for the teleology and behavior languages described in Chapter 3 and Chapter 4. These languages express the purpose of adding the feedback transistor as

$$\delta \textbf{ Guarantees } \phi$$

where δ denotes the design modification that adds the feedback transistor and ϕ denotes the specification (`specification 5`).

2.2 Specifications as Drivers for Design

In examining human generated descriptions of purpose (*cf.* [HW87], [Kow85], [McC88], [Ray86], [SS88], and [WMK89]), a common theme emerges. These descriptions relate design features (i.e., particular design decisions) to design goals (specifications). In fact, the desire to achieve design goals is a primary motivation of the design activity, succinctly stated by Herbert Simon:

> *Synthetic or artificial objects – and more specifically prospective artificial objects having desired properties – are the central objective of engineering activity and skill. The engineer, and more generally the designer, is concerned with how things* ought *to be – how they ought to be in order to* attain goals*, and to* function. [Sim81, p. 7]

Design specifications are described in terms of behavioral (dynamic) characteristics, such as the desired range for the operating temperature of a steam boiler, and physical (static) characteristics, such as dimensions or weight of the desired artifact. At the evaluation step of the design process (see Figure 1.2) the designer determines whether the design meets the specifications set out for the design.[1] In general the goal of the design task is that all specifications hold for the design, or for design d and specifications $\{s_1, \ldots, s_n\}$

$$\forall\, i, \texttt{holds}(s_i, d).$$

Descriptions of purpose of design decisions are expressed in terms of *specification predicates* like `holds` that relate designs and design specifications. For static characteristics, a specification predicate might indicate whether the weight of the artifact is less than some limit, or whether the design artifact meets certain regulatory requirements such as coding standards or expression in a standard design representation language. Static characteristics can be easily expressed in terms of a specification predicate, although actual measurement or verification may be a nontrivial computation. Dynamic characteristics (e.g. behavior) have a rich vocabulary in prose descriptions, using such verbs

[1] This determination ranges from simple inspection by the designer to probabilistic measures gained by simulation under some percentage of system inputs to certainty obtained via rigorous proof methods.

as control, transfer, regulate, and prevent. Hence, we will investigate purposes relative to behavioral characteristics in greater depth, showing that these verbs can be expressed in terms of a simpler, underlying language.

To demonstrate the rich vocabulary used in prose descriptions of purpose, consider a steam power plant inside a petrochemical refinery. One purpose of this power plant is to generate power (i.e., translate chemical energy to thermal and/or mechanical energy) for other operations in the refinery. Further, the transmission and application of power via steam was chosen as opposed to electricity or chemicals (e.g., gas), since the manifestation of energy in the form of steam prevents the fire and petrochemical explosion hazard that exists when using either electricity or chemical combustion.

2.3 A Language for Purpose

Teleological descriptions should provide the knowledge required to answer questions of the form "What is the purpose of this structural entity (component or connection)?"[2] We represent such structural entities as design modifications. For a component or connection, it was either added as a new structural element of the design or it replaced a component of the design. Further, removal of a component or connection is easily represented as a design modification. If just the structural entity itself were referenced without the design modification, one could not express the purpose of removing a component or connection.

Expressing teleological descriptions in terms of design modifications

[2]Selection of a specific parameter value such as the size of a transistor can be viewed as selecting a specific component from a set of alternative components differing only in that parameter value.

has an intuitive justification in that teleological descriptions expressed in prose or verbally are often structured as follows: "Suppose this release valve weren't present in the design. Then the internal pressure of the steam boiler vessel might exceed the rated maximum for the vessel and result in an explosion." Explanation of the purpose of specific parameter values like the release point of the pressure valve or the size of a transistor is expressed in a similar manner. For example, "Suppose the release pressure of the valve were 2000 psi instead of 1500 psi. Since the rated maximum safe pressure for the steam boiler vessel is 1600 psi, it could possibly explode."

Teleological descriptions expressed in terms of design changes also have a pragmatic motivation. Those points in the design process when the designer evaluates a design with respect to the specifications naturally define a set of modifications made for one or more purposes. The evaluation prior to the modifications identifies one or more specifications that are not currently met by the design. The evaluation following the design modification sequence determines whether the specifications the designer was attempting to address have been met. Consequently, these design evaluation points are precisely the points at which a description of purpose should *and can* be captured. Although current CAD (computer aided design) and CAE (computer aided engineering) systems support design methodologies in which capture of teleological information can be included, these systems do not explicitly represent a methodology and hence do not reason about or with the methodology.[3] However, support of development and verification methodology has been researched (e.g., the Programmer's Apprentice [RS84, Wat84]), and Fiduk et al. [FKKP90] describe

[3]Systems currently provide a suite of tools, and the designers themselves are responsible for applying the appropriate tool to the appropriate data, in the correct sequence.

current research in explicitly representing methodologies and reasoning about design activity with respect to methodologies. This methodology management research will provide the methodological infrastructure for acquisition of teleological descriptions.

The following sections summarize the key insights for understanding our approach to representing purpose, the TeD language.

2.3.1 Modifications and Specifications

The first key insight is that a teleological description references:

1. Design modifications (e.g., addition of a component, modification of a specific parameter value) made during the design process, and
2. Effects these changes have with respect to the design meeting its specifications.

Consider the case of a pressure release valve on the steam boiler, described by Kuipers in [Kui85]. The steam boiler has a design specification stating that the internal pressure should not exceed some maximum value, beyond which the boiler vessel might explode. The design modification made to address this specification is the addition of a pressure activated release valve. The purpose of the addition of the pressure release valve is to prevent the internal pressure of the boiler from exceeding some critical value at or beyond which the boiler vessel might explode. We compare this description of purpose with behavioral and causal descriptions: a behavior description of the pressure release value states that the valve opens at some prescribed pressure and steam escapes the vessel via the pressure release valve; a causal description explains how this behavior is achieved (i.e., the internal workings of the valve). Neither the

behavioral description nor the causal description captures why this behavior was desired, and hence why the pressure release valve was added to the design.

2.3.2 Guarantees

The second key insight is that behavior changes of the design artifact can be expressed in terms of a teleological operator *Guarantees*. Informally, this teleological operator denotes a predicate function whose arguments are a design modification and a specification predicate and whose truth value is determined by examining the specification predicate in the context of the static and dynamic characteristics of the unmodified and modified designs.[4] This operator is justified on two accounts, empirical and theoretical. As an empirical justification, we make the observation that design requirements are often expressed in terms of behaviors required or prohibited for the mechanism being designed. This is most easily demonstrated in a stimulus-response specification of behavior: "When set of conditions (state) A occurs, the mechanism should bring about condition set (state) B, possibly within a specified time constraint." This is the essence of the **ToMake** clause of functional representation described by Sembugamoorthy and Chandrasekaran [SC85], namely "given an initial set of conditions, the purpose is to bring about (ToMake) some other set of conditions."

As a formal basis we have defined the teleological operator in terms of the operators of modal logic [Tur84, Che80]. If one formulates the possible static characterizations or dynamic behaviors of a mechanism as possible worlds, then specifications can be viewed as "possibly," "necessarily," or "not

[4]The teleological operator *Guarantees* is defined formally in Sections 3.7.1 and 3.7.2.

possibly" holding in those possible worlds. A guaranteed specification necessarily holds in all possible worlds (behaviors) of the mechanism.

2.3.3 The Need for Context

The final key insight is that while the structure and possible behaviors of a system component can be described and understood outside the context of the system in which the component is embedded, a description of purpose cannot be described independent of such context. Simon identifies the role of context for functional explanation and description as follows:

> *An important fact about this kind of explanation is that it demands an understanding mainly of the outer environment. ... In this manner, the properties with which the inner environment has been endowed are placed at the service of the goals in the context of the outer environment.* [Sim81, p. 11,15]

Clearly, structure descriptions make sense independent of the context of the enclosing system, since the structure of each component is described independent of any specific enclosing system. Similarly, potential component behaviors can be derived independent of any specific enclosing system.[5] Such component behavior is described via a state representation in which the state is comprised of variables and values for those variables. The purpose of a component, however, can only be described in the context of the static and dynamic characteristics of the system in which the component is embedded. Consider the steam boiler

[5]The actual behaviors a component exhibits when embedded in a larger system may be a subset of the possible behaviors if the enclosing system restricts values of variables defined at the component's interface.

example once again. The behavior of the release valve can be described in terms of the valve aperture and the pressure against the valve. The purpose of the release valve, however, requires reference to the behavior of the system in which the valve is embedded. For example, the purpose of the release valve would not be "to open and close," as this would require further interpretation in the context of the system in which the valve operated.

The need expressed here for languages for structure, behavior, causal, and teleology descriptions originates from Kuipers' identification of structural, behavioral, and functional descriptions [Kui85, p. 170], where he states:

> *The* structural description *consists of the individual variables that characterize the system and their interactions; it is derived from the components of the physical device and their physical connections. The* behavioral description *describes the potential behaviors of the system as a network of the possible qualitatively distinct states of the system. I reserve the term* functional description *for a description that reveals the purpose of a structural component or connection in producing the behavior of a system. Thus, the function of a steam-release valve in a boiler is to prevent an explosion; the behavior of the system is simply that the pressure remains below a certain limit. The existing literature frequently obscures this distinction by using the term 'function' to refer to behavior.*[6]

In conclusion, while structure, behavior, causal, and teleology languages describe unique aspects of a mechanism, they are closely related in that

[6]In the work described herein, the terms *teleology* or *purpose* will be used instead of function, to avoid this confusion of the terms *function* and *behavior*.

one language references another (e.g., a description in the teleology language references behavior descriptions) and hence relies on that language for its expression and derivation.

2.4 Goals of TeD

To these insights we add the following additional goals for TeD:

- To be independent of any particular structure or behavior language or specific predicates used to express specifications. This allows the techniques and capabilities developed here to be applied to multiple modeling approaches, including both qualitative and quantitative.
- To be independent of any particular domain of mechanisms. Examples given here include electrical, hydraulic, and thermal domains.
- To allow hierarchical descriptions referencing predicates or other teleological descriptions. This will allow the construction of arbitrarily complex descriptions of purpose.

Chapter 3

Ontology and Representation

3.1 An Ontology for Teleology

In preparation for a discussion of a representation for teleological descriptions, a discussion of an ontology[1] for teleology is in order. Simon characterizes an ontology for purpose as follows:

> *Fulfillment of purpose or adaptation to a goal involves a relation among three terms: the purpose or goal, the character of the artifact, and the environment in which the artifact performs.* [Sim81, p. 8]

The ontological elements of the teleology language TeD are:

- Design specifications (desired static and dynamic characteristics)
- Design descriptions (structure, behavior)
- Design modifications

The ontological elements identified by Simon are realized in TeD as follows:

- *Purpose or goal* - Design specifications

[1]The ontology identifies the entities that are available for representing problem situations [Gre83].

- *Character of the artifact* - Design descriptions (structure, behavior)

- *Environment* - Derived behaviors

We describe the ontological elements of TeD in this chapter. Structure, design modification, and design history languages are described first. We then describe behavior and specification languages, and finally the teleology language TeD. We discuss design specifications, providing intuitive definitions for the predicates, leaving the formal definitions for Chapter 4. Chapter 4 formally defines the specification predicate `occursIn` which is used in teleological descriptions concerning behaviors. In this chapter, we give the essential features of structure, design modification, design history, behavior, and teleology languages, with an example language for each.

To demonstrate the languages and concepts, we use an idealized model of a steam boiler (the double heat flow system defined by Kuipers [Kui85, p. 175], shown in Figure 3.1). The behaviors exhibited by the steam boiler demonstrate the qualitative description of a system reaching equilibrium. In one possible behavior, equilibrium occurs after the vessel's internal pressure has exceeded some maximum value. A modified design in which a pressure sensor is added eliminates this behavior, and the teleology language captures the intent of the design modification, namely to eliminate the possibility that the internal pressure exceeds some maximum value and explodes. This particular teleological description involves a behavior which is not exhibited by an implementation of the final design, a unique capability of TeD. We discuss this capability further when comparing the TeD language to other work in Chapter 10.

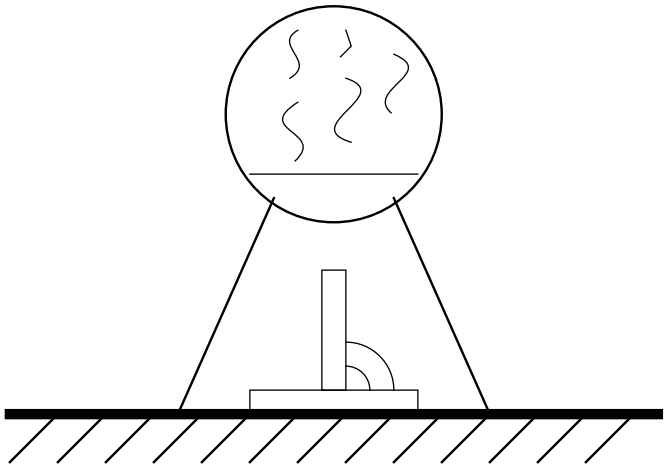


Figure 3.1: Idealized Steam Boiler

3.2 Structure

Many languages for representing structure have been put forth (see structure languages proposed by Abelson and Sussman [AS85], Davis [Dav85], de Kleer and Brown [dKB85], Forbus [For85], Franke and Dvorak [FD90], Kuipers [Kui85, Kui86], Williams [Wil85], and VHDL [VHDL87]) and each has associated behavior languages and envisioning semantics (discussed by de Kleer [deK77], de Kleer and Brown [dKB82], and Kuipers [Kui82, Kui86]). The structure and behavior languages described herein abstract these existing language definitions to avoid basing the teleology language on a specific structure or behavior language and associated envisioning semantics.

Davis and Hamscher [DH88] identify several common themes in representing structure:

- Structure representation should be hierarchical;

- Structure representation should be object centered and isomorphic to the organization of the device;

- Behavior can be represented by a set of expressions that capture the interrelationships among the values on the terminals of the device.

While the structure language described here supports these themes, it allows structure ontologies other than one component per physical object. For example, the ontological elements might be processes [For85] or generic mechanisms [Kui89b] with descriptions of purpose associated with these elements. At the appropriate level of abstraction, a design description may consist of processes or generic mechanisms as opposed to physical components.

Structure languages used here describe models (designs) hierarchically composed from simpler models called components [AS85, dKB85]. Hierarchy is achieved by applying this decomposition recursively to components. Each component has an interface, expressed in terms of terminals, that can be connected to terminals of other components. Interactions between two components are restricted to these connections. At some point in the hierarchy, components are described in terms of the primitives of the associated behavior language. We call these primitives *variables* and *behavior constraints*.[2] Formally, a structure D is a tuple

$$\langle V, Q, qs, con, C \rangle, \text{ with}$$

[2]This representation retains sufficient generality to describe complex systems such as a microprocessor. Variables represent the interface and internal state of the microprocessor and behavior constraints represent the modification the microprocessor makes on its internal state and outputs based on the current internal state and input values. Such behavior constraints may be expressed as arbitrarily complex procedural code.

V - a set of variables
Q - a set of quantity spaces
qs - a mapping from V to Q
con - a set of constraint types
C - a set of tuples $\langle c, v_1, \ldots, v_n, cv_1, \ldots, cv_m \rangle$
where $c \in con$, $v_i \in V$, and $cv_i \in qs(v_1) \times \ldots \times qs(v_n)$.

To demonstrate structure description, we describe the steam boiler example in the structure language CC [FD90] which uses Kuipers' QSIM [Kui85, Kui86] behavior language.[3] In this example we model temperatures, pressure and heat inside the boiler vessel, and the heat flow from the heat source to the boiler vessel to the surrounding air, a heat sink. The model is constructed from three components, a flame (heat source), a boiler vessel, and the surrounding air (heat sink). For the boiler vessel, variables expressing the difference between the internal temperature and the flame and air temperatures are also included. These temperature difference variables are referenced in constraints that relate heat flow into and out of the boiler vessel to those temperature differences. Specifically, the heat flow into the boiler vessel is a monotonically increasing function of the temperature difference between the flame and the boiler vessel. As this difference increases, the heat flow into the boiler vessel increases.

The component types used to construct the steam boiler example are `Boiler-Vessel`, `Heat-Source`, and `Heat-Sink`. The `Boiler-Vessel` component definition (Figure 3.2) defines terminals, variables, and behavior constraints introduced into a model when it is included. The steam boiler model (Figure 3.3) defines three component instances and connections among the terminals of these component instances. Using the hierarchical naming scheme of

[3]The CC structure language allows definition of hierarchical component models from which CC generates a qualitative differential equation (QDE) suitable for envisioning with QSIM.

```
(define-component-interface
  Boiler-Vessel
  "Boiler Vessel in thermal domain" thermal
  (terminals in out)
  (quantity-spaces
   (defaults (temperature temperature-qspace)
             (entropy heat-qspace))))

(define-component-implementation
  primitive Boiler-Vessel
  "Boiler Vessel for heat flow, in QSIM primitives"
  (terminal-variables (in  (inFlow  heat-flow (lm-symbol IF))
                           (Tin     temperature))
                      (out (outFlow heat-flow (lm-symbol OF))
                           (Tout    temperature)))
  (component-variables
    (netFlow  heat-flow    display (lm-symbol NF))
    (heat     entropy      display (lm-symbol H))
    (pressure (hydraulic pressure) display (lm-symbol P)
                           (quantity-space pressure-qspace))
    (T        temperature  display)
    (dTin     temperature  display
                           (quantity-space base-quantity-space))
    (dTout    temperature  display
                           (quantity-space base-quantity-space)))
  (constraints
    ((ADD T dTin Tin)   (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTin inFlow)   (0 0))
    ((ADD T dTout Tout) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTout outFlow) (0 0))
    ((ADD inFlow outFlow netFlow) (0 0 0))
    ((d/dt heat netFlow))
    ;; Assume constant fluid/gas mass, so heat follows temperature
    ((M+ heat T)     (0 0) (Ha* AT*) (Hf* FT*))
    ((M+ pressure T) (0 0) (Pa* AT*) (Pf* FT*))
    ))
```

Figure 3.2: Boiler-Vessel Component Definition in CC

```
(define-component-interface
  SB "Steam Boiler" thermal
  (quantity-spaces
    (defaults (temperature temperature-qspace)
              (entropy heat-qspace)
              (heat-flow base-quantity-space))))

(define-component-implementation
  1 SB
  "Simple steam boiler"
  (components
    (Vessel boiler-vessel (display netflow heat pressure T
                                   dTin dTout inFlow outFlow))
    (Flame  heat-source)
    (Air    heat-sink))
  (connections (p1 (Flame out) (Vessel in))
               (p2 (Vessel out) (Air in))))
```

Figure 3.3: Steam Boiler Model Definition in CC

CC, these instances are:

```
(SB vessel)
(SB flame)
(SB air)
```

Again using the hierarchical naming scheme of CC, the variables contributed by component `(SB vessel)` to the variable set V are:

```
(SB vessel inFlow)
(SB vessel Tin)
(SB vessel outFlow)
(SB vessel Tout)
(SB vessel netFlow)
(SB vessel heat)
(SB vessel pressure)
(SB vessel T)
(SB vessel dTin)
(SB vessel dTout)
```

Behavior constraints contributed by component `(SB vessel)` to the constraint set C are:

```
(ADD (SB vessel T) (SB vessel dTin) (SB vessel Tin))
(M+ (SB vessel dTin) (SB vessel inFlow))
(ADD (SB vessel T) (SB vessel dTout) (SB vessel Tout))
(M+ (SB vessel dTout) (SB vessel outFlow))
(ADD (SB vessel inFlow) (SB vessel outFlow) (SB vessel netFlow))
(d/dt (SB vessel heat) (SB vessel netFlow)))
(M+ (SB vessel heat) (SB vessel T))
(M+ (SB vessel pressure) (SB vessel T))
```

The complete steam boiler example appears in Appendix A.

3.3 Design Modification and History

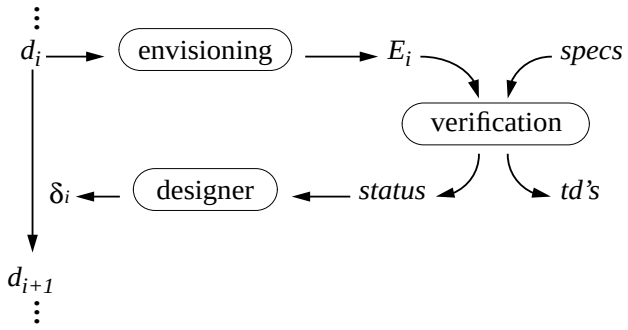
The structure language provides a means for describing a single point in the history or evolution of a design. A description of *design history* also requires a means for describing the transitions from one state of the design (distinct from a behavior state of an artifact or instance of the design) to another. The term *design modification* denotes such a transition, and is a relation between two structure descriptions. A design history is a pair comprised of an initial design (structure description) and a sequence of design modifications,[4] denoted

$$(d_0, \langle \delta_1, \delta_2, \ldots, \delta_n \rangle)$$

where d_0 is the initial design and δ_i are design modifications. This design history defines a sequence of designs (structure descriptions)

$$d_0, d_1, \ldots, d_{n-1}, d_n$$

[4]Representing a design history as an initial design and design modifications is common in commercial CAE and CAD systems, providing a record of design changes and the ability to undo design changes. For efficiency, new "initial" designs are created periodically in such systems by applying the modifications and explicitly representing the new "initial" design.



d_i - i^{th} version of the design
E_i - envisionment for design d_i (see Section 3.5.2)
$specs$ - design specifications
$td's$ - teleological descriptions captured in verification
$status$ - results of verification
δ_i - modifications generated by the designer

Figure 3.4: Design Process Flow (Single Step)

where d_i is the result of applying design modification δ_i to design d_{i-1}. The design history is captured during the initial design, evaluation, and design modification steps in the life-cycle model of Figure 1.2. Figure 3.4 gives a more detailed process description for the evaluation (envisioning and verification) and design modification (designer) steps.

To demonstrate design modifications, a simple language is defined here. The language elements correspond roughly to structure editing operations, although some editing operations will be a composition or sequence of the language primitives given here (e.g., "Replace component A with component B" is realized via "Delete component A" and "Add component B" with appropriate connections specified). The modification language primitives are:

- Add-x to a component, where x is a component, connection, terminal, variable, quantity space, or constraint

- Remove-x from a component, where x is a component, connection, terminal, variable, quantity space, or constraint
- Rename-x, where x is a component instance, variable, or terminal
- Change-x, where x is a parameter value (landmarks in a quantity space), a quantity space of a variable, a component implementation type, a variable type, or the default domain of a component.
- Create-x, where x is a new component type or quantity space. The new definition is a copy of an existing one, which will then be modified.

Primitives of the modification language given in Table 3.1 are interpreted in the context of a component definition, while those in Table 3.2 are interpreted in the editing environment, outside the context of any specific component.

3.4 Design Instantiation

Given design definitions as described above, one instantiates a design by actually constructing the artifact specified by the design description, by building a model of the design suitable for simulation or analysis, or by some combination of artifact and model. In any case, the design description is *instantiated* and variables are created. If a physical artifact is constructed, the variables are in the artifact itself. One can talk about the *state* of an instantiated design, namely a function mapping variables to values. In terms of a structure description,

$$s : V \rightarrow qs(v_1) \times \ldots \times qs(v_n), \text{ where } n = |V|.$$

Accordingly, the value of variable v in state s is denoted $s(v)$.

```
(for-component ⟨type descriptor⟩
  (add-component ⟨instance name⟩ ⟨type⟩ ⟨options⟩ . ⟨connections⟩)
  (add-connection ⟨connection spec⟩ ⟨connection spec⟩ ...)
  (add-terminal ⟨term-name⟩ ⟨term-name⟩ ...)
  (add-terminal-variable ⟨term-name⟩ ⟨name⟩ ⟨type⟩ . ⟨options⟩)
  (add-component-variable ⟨name⟩ ⟨type⟩ . ⟨options⟩)
  (add-constraint ⟨constraint⟩ ⟨constraint⟩ ...)
  (remove-component ⟨instance name⟩ ⟨instance name⟩ ...)
  (remove-connection ⟨connection spec⟩ ⟨connection spec⟩ ...)
  (remove-terminal ⟨name⟩ ⟨name⟩ ...)
  (remove-variable ⟨name⟩ ⟨name⟩ ...)
  (remove-constraint ⟨constraint⟩ ⟨constraint⟩ ...)
  (rename-component-instance ⟨current name⟩ ⟨new name⟩)
  (rename-variable ⟨current name⟩ ⟨new name⟩)
  (rename-terminal ⟨current name⟩ ⟨new name⟩)
  (change-quantity-space ⟨variable⟩ ⟨quantity space⟩)
  (change-component-implementation ⟨inst-name⟩ ⟨impl-name⟩)
  (change-variable-type ⟨name⟩ ⟨new type⟩)
  (change-domain ⟨domain⟩)
  )
```

Table 3.1: Modification Language Syntax - Component Relative

```
(create-new-component-type ⟨existing type⟩ ⟨new type name⟩)
(create-new-quantity-space ⟨name⟩ ⟨lm list⟩⟨parent⟩ ⟨cvalues⟩)
(change-parameter-value ⟨quantity space name⟩ ⟨new qspace⟩)
```

Table 3.2: Modification Language Syntax - Environment Relative

Although CC produces a flattened[5] representation expressed as a QSIM qualitative differential equation (QDE), no assumptions in the behavior or teleology languages are made based on whether a design instance is represented in a hierarchical form or a flattened form. In either case, representing the state requires unique instances for those variables of the design which describe the design at the current abstraction level.[6]

To complete the language for static characterization we require the definition of a *consistent state*, a state assigning variable values that are consistent with all behavior constraints imposed by the design.

3.5 Behavior

A discussion of behavior requires an instantiation of a design in the form of a model, a physical artifact, or some combination of these two forms. The process of envisioning produces a characterization of the (possibly infinite) set of possible behaviors of the mechanism. We describe a behavior language, adopting existing terminology of behavior descriptions (see behavior language descriptions of de Kleer and Brown [dKB85], Forbus [For85], and Kuipers [Kui85, Kui86]). For each pair of structure and behavior languages there is an envisioning semantics addressing issues such as valid transitions between variable values, valid transitions between states, and representations of time. We next describe the behavior language and envisionment properties required by the TeD language.

[5] A *flat* representation expresses a model at the lowest level of abstraction, and is produced from a hierarchical representation. See [FD90] for details of the CC flattening procedure.

[6] Modeling techniques such as time-scale abstraction [Kui87] may have lower abstraction levels which are effectively instantaneous from the point of view of the current model, and hence can be described by variables of the current abstraction level.

3.5.1 Single Behaviors

Two states s_1 and s_2 are said to be *adjacent states* if for each variable v, $s_2(v)$ is a legal next value of $s_1(v)$ (defined by the envisioning semantics of the behavior language and $qs(v)$). A *behavior* is a possibly infinite sequence of consistent states $\langle s_0, s_1, \ldots, s_n, \ldots \rangle$ where s_i, s_{i+1} are adjacent states. The first state of the sequence is called the *initial state*.

3.5.2 Envisionment

Given the structure description for a design and an instance of that design, the envisioning process produces a characterization of some or all of the possible behaviors of the instance. This characterization is called an *envisionment*[7] [deK77, dKB82, Kui82, Kui86]. The qualitative modeling approach attempts to derive all the possible behaviors from a given set of initial states, and hence provides the capability to generate such an envisionment. A *total envisionment* represents all possible behaviors of the design instance, and an *attainable envisionment* represents all possible behaviors from a specific set of initial states. In terms of structure description D, an attainable envisionment of D from initial state s is written as $\mathrm{E}(\{s\})$ and denotes the set of all behaviors b of D where $b = \langle s_1, s_2, \ldots \rangle$ and $s_1 = s$. Letting S denote the set of all states of D, a total envisionment of D is written as $\mathrm{E}(S)$ and abbreviated as E.

[7] The term envisionment is intended to include descriptions of behaviors generated from a design instance whose initial state is not an equilibrium, as well descriptions of behaviors resulting from perturbations to a system in equilibrium [dKB82].

```
(SB vessel T)  =  (AT* nil)
 (SB flame T)  =  (FT* std)
   (SB air T)  =  (AT* std)
```

Figure 3.5: Initial Variable Values - Steam Boiler

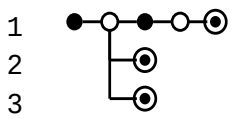


Figure 3.6: Behavior Tree - Steam Boiler

3.5.3 Example

To demonstrate behavior description and envisionment, we return to the steam boiler example and use the envisioning semantics and behavior language of QSIM. Our model of the steam boiler produces three qualitatively distinct behaviors, shown graphically in the QSIM behavior tree in Figure 3.6. In the initial state of the model, the contents of the boiler vessel are at the same temperature as the surrounding air, with the heat source (flame) having a temperature greater than that of the surrounding air. This initial state is determined by the variable values shown in Figure 3.5.

The mechanism reaches an equilibrium with the internal temperature of the boiler vessel at some point between the temperature of the air and the flame. The three possible behaviors are determined by the possible values for the internal pressure when equilibrium is reached. This equilibrium pressure can be either less than, equal to, or greater than the landmark value `Pmax*` in the quantity space

```
(0 Pa* Pmax* inf).
```

Landmark `Pa*` represents the pressure (within the vessel) at air temperature.

The qualitative plots (from QSIM) for variables `(SB vessel T)` and `(SB vessel pressure)` are shown in Figure 3.7. The initial state for this attainable envisionment is determined by the variable assignments given in Figure 3.5 and is the first state given in each behavior.

3.6 Design Specifications

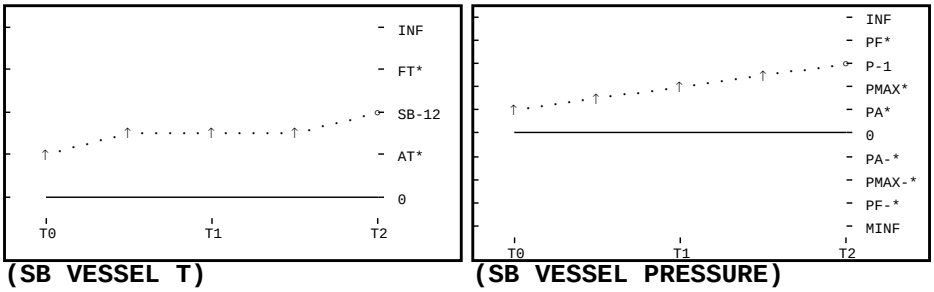
For the steam boiler example, the landmark `Pmax*` represents a maximum safe value for the pressure inside the boiler vessel. A design specification expresses the fact that to achieve a correct design, the internal pressure of the boiler vessel should never exceed the maximum safe value. Hence, a designer will modify the design to eliminate the undesirable behavior from the mechanism, namely the behavior in which this maximum is exceeded.

3.6.1 Scenarios

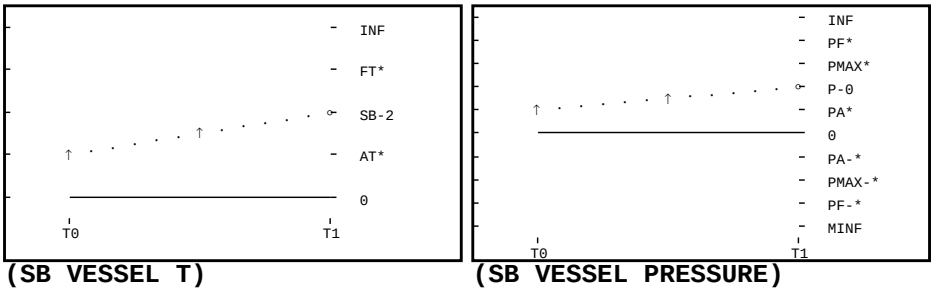
The design specification language identifies behaviors or behavior abstractions (called *scenarios*, and defined in Chapter 4) and whether they are required or prohibited. For the steam boiler example, the behavior abstraction is an internal boiler vessel pressure greater than the safe maximum, `Pmax*`. We express the behavior abstraction as a sequence of states (in this case the sequence contains only one state), written

$$\langle\{\texttt{(pressure ((Pmax* inf) ign))}\}\rangle.$$

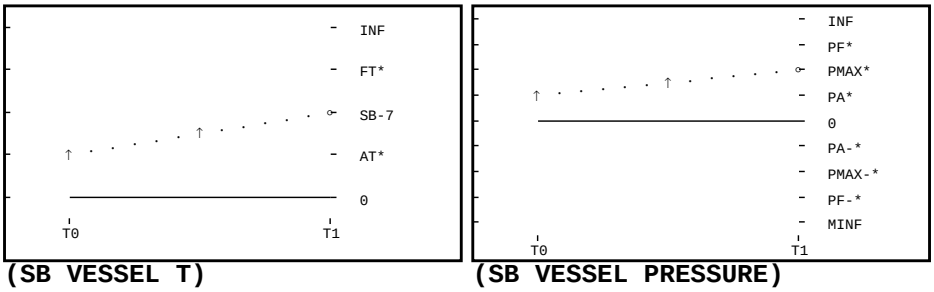
[8]Branching is according to whether pressure reaches landmark Pmax*.



Behavior 1



Behavior 2



Behavior 3

Figure 3.7: Qualitative Plots from QSIM[8]

3.6.2 Specification Predicates

To develop the teleology language, we introduce the *specification predicate*. A specification predicate is evaluated in the context of a behavior, or possible world, and its truth value indicates whether the specification holds in the behavior (possible world). We use the term *possible world*[9] because the design created to meet the specifications will exhibit many different *possible* behaviors, either because of different initial conditions or inherent nondeterminism in the design (e.g., a qualitative model). In expressing teleology, we want to state that a specification predicate holds in none, some, or all possible behaviors (worlds) of a design. In the steam boiler example, we want to state that in all possible behaviors of the design, the internal pressure does not exceed `Pmax*`. Modal logic provides an existing formalism for this purpose.

Design specifications involve required or prohibited scenarios, and to express whether these specifications hold in a behavior, we introduce the specification predicate `occursIn(`σ`,`b`)`, where σ is a behavior abstraction (scenario) and `occursIn(`σ`,`b`)` is true for behavior (possible world) b if σ abstracts b. The specification predicate `occursIn`, behavior abstraction, and a specification language are described in detail in Chapter 4.

3.7 Teleology

The teleology language relates design modifications (changes in structure) to design specifications (desired static and dynamic characteristics). With the teleology language, we can formally express the designer's intent in mod-

[9]The term "possible world" is taken from modal logic, which we will use in Section 3.7.2 as a formal basis for teleological descriptions.

ifying the steam boiler, namely to prevent the behavior in which the internal pressure exceeds `Pmax*`, possibly resulting in an explosion.

Since the teleological description of an entity is context dependent, it is not possible to enumerate all possible teleological descriptions of an entity. In this work, the structure hierarchy and associated design specifications of the system in which a component is embedded provide the context in which teleological descriptions of the component are developed or evaluated, as discussed in Section 2.3.3.

3.7.1 Primitive Teleological Operator

Teleological operators are the language primitives for teleological descriptions. In the context of a design modification, a single teleological operator relates the unmodified design to the modified design in terms of the specification predicates. In the following definitions, ϕ_i are specification predicates,[10] d and d' are designs (structure descriptions), δ a design modification such that d' is the design obtained by applying δ to d, and E and E$'$ are the envisionments[11]

[10] Recall that specification predicates are evaluated in the context of a specific behavior or possible world.

[11] We use the term envisionment to characterize not only the possible behaviors of a mechanism, but also to characterize the possible physical configurations of the mechanism. A design may be underconstrained and allow more than one physical configuration, particularly in the early stages of the design process. For example, alternative floorplans for a VLSI chip will have different dimensions giving different total area, as well as different timing characteristics.

of d and d', respectively. We define the operator **Guarantees** as:[12]

$$\delta \textbf{ Guarantees } \phi \Leftrightarrow \left\{ \begin{array}{l} \exists\ b \in \mathrm{E}, \quad \neg\phi, \\ \text{ and} \\ \forall\ b' \in \mathrm{E}', \quad \phi. \end{array} \right.$$

A teleological operator makes a statement about both the modified and unmodified designs. The statement made for the unmodified design is the negation of the statement made for the modified design (modulo the envisionment, or set of possible worlds). This point may seem trivial, but is crucial in that it attributes the newly attained truth of the specification predicate to the design modification. In other words, the design modification was applied to the unmodified design, for which the specification predicate was not true. Note that because of the assertion about the unmodified design, the following teleological descriptions are not equivalent:

$$(\delta \textbf{ Guarantees } \phi_1) \ \wedge \ (\delta \textbf{ Guarantees } \phi_2), \text{ and}$$

$$\delta \textbf{ Guarantees } (\phi_1 \ \wedge \ \phi_2).$$

3.7.2 Expression in Modal Logic

As we discussed in Section 3.6.2, modal logic provides an existing formalism for expressing the statement that a specification predicate holds in none, some, or all possible behaviors (worlds) of a design. Modal logic (see Chellas [Che80]) adds the operators *necessarily* (written $\Box$) and *possibly* (written $\Diamond$) to first order logic. Given a first order predicate ϕ, $\Box\phi$ (necessarily

[12] When defining teleological operators we use a special-purpose notation in which the left brace indicates a structured conjunction related to the transformation from the unmodified design to the modified design. Accordingly, the expression involving envisionment E (of the unmodified design) is written above the expression involving envisionment E′ (of the modified design), indicating the transformation from the unmodified to the modified design.

ϕ) is true if ϕ is true in all possible worlds and $\Diamond\phi$ (possibly ϕ) is true if ϕ is true in at least one possible world. To express teleological descriptions in modal logic, we recognize that the envisioning process characterizes the possible worlds for the design instance, where a possible world is one behavior or physical configuration of the design. The modal operators *possibly* and *necessarily* can express that a specification predicate is false in all possible worlds (*necessarily not*), true in every possible world (*necessarily*), or true in at least one possible world (*possibly*).

We use the model-theoretic approach to define teleological descriptions in modal logic. This approach involves a *model* – a particular instance of a set of possible worlds and truth assignments of logical sentences in these possible worlds. We derive the following benefits from the model-theoretic approach:

1. A set of logical sentences can be shown to be consistent by demonstrating a model for which the sentences are satisfiable, and

2. Model-checking techniques for verifying the truth of a set of logical sentences exist for various languages[13].

A model is expressed as an instance of a schema called a *standard model*. We use the definition of a standard model from [Che80, p. 68]. Given $\mathcal{M} = \langle W, R, P \rangle$ with

1. W a set,

[13]A model-checking algorithm has been implemented in this work, based on the behavior abstraction relations defined in Chapter 4.

2. R a binary relation on W, and

3. P a mapping from natural numbers to subsets of W,

$\mathcal{M}$ is a *standard model*. In the possible worlds interpretation, W is the set of possible worlds, R is a relation on W called the *accessibility* relation, and P represents the subsets of W for which predicates are true.

Given the envisionment of a design instance, the design specifications, and procedures[14] for determining the truth value of the design specifications in each behavior (possible world) in the envisionment, we further refine the standard model for expressing teleological descriptions in modal logic. Envisionments E and E′ define the set of possible behaviors (possible worlds) of the unmodified and modified design. The mappings P and P' define, for each specification predicate ϕ_i, the set of behaviors (subsets of E and E′, respectively) in which ϕ_i is true. For the binary relation R, we define $\alpha R\beta$ as[15]

$$\forall\alpha \in W,\ \forall\beta \in W,\ \alpha R\beta.$$

We define the model schemas $\mathcal{M} = \langle W, R, P\rangle$ and $\mathcal{M}' = \langle W', R, P'\rangle$ with

- W the set characterized by envisionment E,

- W' the set characterized by envisionment E′,

[14]This work is not directly concerned with techniques for verifying that design specifications have been met, i.e., the problem of design verification. While we have implemented a model checking algorithm based on behavior abstraction, we focus on using the results of design verification to capture design rationale, namely the purpose of design modifications and design decisions.

[15]The relation R is serial (for every α there is a β such that $\alpha R\beta$), reflexive, symmetric, transitive, and euclidean (for every α, β, and γ if $\alpha R\beta$ and $\alpha R\gamma$ then $\beta R\gamma$). These properties imply the validity of schemata D ($\Box A \rightarrow \Diamond A$), T ($\Box A \rightarrow A$), B ($A \rightarrow \Box\Diamond A$), 4 ($\Box A \rightarrow \Box\Box A$) and 5 ($\Diamond A \rightarrow \Box\Diamond A$), respectively. See [Che80, p. 80].

- $P(i) = \{\alpha \mid \alpha \in W \ \wedge \ \neg\phi_i \text{ in } \alpha\}$, and
- $P'(i) = \{\alpha \mid \alpha \in W' \ \wedge \ \phi_i \text{ in } \alpha\}$.

Now we can express teleological operator **Guarantees** in terms of the modal operators $\Box$ (*necessarily*) and $\Diamond$ (*possibly*). For specification predicate ϕ_1 and models $\mathcal{M}$ and $\mathcal{M}'$, we define **Guarantees** as:

$$\delta \textbf{ Guarantees } \phi_1 \ \Leftrightarrow \begin{cases} \mathcal{M} \text{ is a model for } \ \Diamond\neg\phi_1 \\ \text{ and} \\ \mathcal{M}' \text{ is a model for } \ \Box\phi_1. \end{cases}$$

Note that $\mathcal{M}'$ is a model for $\Box\phi_1$ if and only if $P'(1) = W$, and $\mathcal{M}$ is a model for $\Diamond\neg\phi_1$ if and only if $P(1) \neq \{\}$.

3.7.3 Example

One behavior exhibited by the initial steam boiler design (see Figure 3.3 for the structure description, Figure 3.7 for the behavior) does not conform to the design specification `(prohibited` σ`)` and associated specification predicate $\neg$`occursIn(`σ`,`b`)`, where σ is the behavior abstraction

$$\langle\{\texttt{(pressure ((Pmax* inf) ign))}\}\rangle.$$

The model checking algorithm implemented in this work determines the fact that one behavior of the steam boiler does not meet the specification. Output of model checking is given in Figure 3.8.

A modified design containing a pressure sensor component which translates pressure into an electrical voltage is proposed (see schematic in Figure 3.9, CC definition in Figure 3.10). The pressure is sensed via the connection with component `vessel`, and the electrical voltage is transmitted via the connection with component `flame`. The heat source modifies the temperature

```
Checking behaviors against

Design specification: DHF-NO-EXPLODE
 Prohibited Scenarios:
   State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
   Boolean Expression: TRUE

Design spec instantiation is #<Spec: PROHIBITED SC-0>:
  PROHIBITED:
    Scenario:
      State Sequence: ((SB_VESSEL.PRESSURE ((PMAX* INF) IGN)))
      Boolean Expression: TRUE

Behavior S-6 inconsistent with spec #<Spec: PROHIBITED SC-0>
```

Figure 3.8: Steam Boiler - Model Checking Output

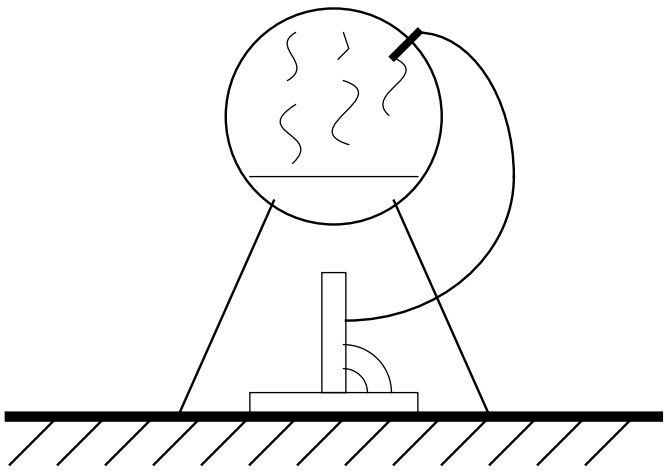


Figure 3.9: Modified Steam Boiler

```
(define-component-implementation
  2 SB
  "Steam boiler with pressure sensor"
  (components
    (Vessel boiler-vessel-modified
            (display netFlow heat pressure T
                     dTin dTout inFlow outFlow))
    (Flame  controlled-heat-source)
    (Air    heat-sink)
    (Sensor pressure-sensor (display v)))
  (connections (p1 (Flame out)  (Vessel in))
               (p2 (Vessel out) (Air in))
               (p3 (Vessel t)   (Sensor in))
               (p4 (Sensor out) (Flame ctl))))
```

Figure 3.10: Modified Steam Boiler Model in CC

based on the voltage at the control terminal. This modified design was created with the modifications shown in Figure 3.11 (possibly captured during interactive editing by a designer). The complete modified steam boiler component hierarchy is given in Appendix A.

The envisionment of the modified design (with the initial variable values given in Figure 3.5) contains three behaviors shown graphically in the behavior tree of Figure 3.12. Behaviors 2 and 3 are the same as behaviors 2 and 3 of the original design. Behavior 1 has changed (see Figure 3.13), and the variable `(SB vessel pressure)` no longer exceeds landmark `Pmax*`. The model checking algorithm determines the fact that all behaviors of the modified steam boiler meet the specification. Output of model checking is given in Figure 3.14.

We can now describe the purpose of the design modification of Figure 3.11 with respect to the design specification `(prohibited` σ`)` and associated specification predicate $\neg$`occursIn(`σ,b`)`. Letting δ denote the design

```
(create-new-component-implementation SB 1 2)

(for-component (SB 2)
  (replace-subcomponent vessel boiler-vessel-modified
    ((display netFlow heat pressure T dTin dTout inFlow outFlow)))

  (replace-subcomponent flame controlled-heat-source nil)

  (add-subcomponent sensor pressure-sensor ((display v))
                    (in (vessel t)) (out (flame ctl))))
```

Figure 3.11: Steam Boiler Modifications

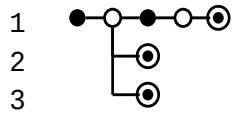
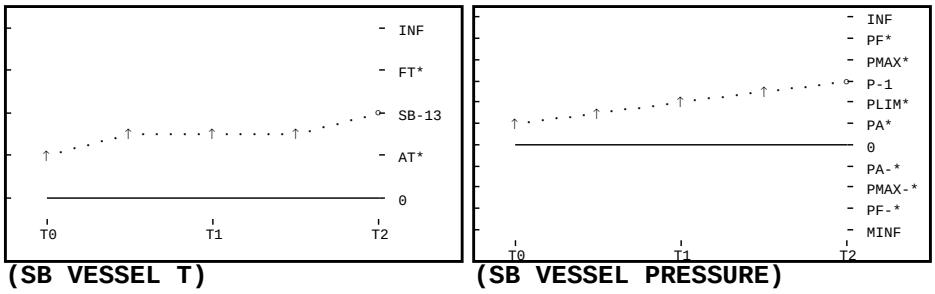


Figure 3.12: Behavior Tree - Modified Steam Boiler



Behavior 1

Figure 3.13: Qualitative Plots for Modified Steam Boiler

```
Checking behaviors against

Design specification: DHF-NO-EXPLODE
 Prohibited Scenarios:
   State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
   Boolean Expression: TRUE

Design spec instantiation is #<Spec: PROHIBITED SC-0>:
  PROHIBITED:
    Scenario:
      State Sequence: ((SB_VESSEL.PRESSURE ((PMAX* INF) IGN)))
      Boolean Expression: TRUE

Verified specifications:
  #<Spec: PROHIBITED SC-0>
```

Figure 3.14: Steam Boiler - Model Checking Output

modification that adds the pressure sensor, the teleological description is

$$\delta \textbf{ Guarantees } \neg \texttt{occursIn}(\sigma, b). \tag{3.1}$$

Design modification δ may have resulted in other behavior changes between the unmodified and the modified designs. The reason this particular purpose was identified is because one of the behavior changes accomplished by δ was defined in a specification. The steam boiler design history, evaluation steps, and acquired teleological description are shown in the context of the design process flow in Figure 3.15.

[16]Design SB1 is described in Figures 3.1 and 3.3. Envisionment E_1 is described in Figures 3.6 and 3.7. Design modification δ is described in Figure 3.11. Design SB2 is described in Figure 3.9 and 3.10. Envisionment E_2 is described in Figures 3.13 and 3.12. The acquired teleological description is equation 3.1. An sample trace of this design flow is given in Appendix A.

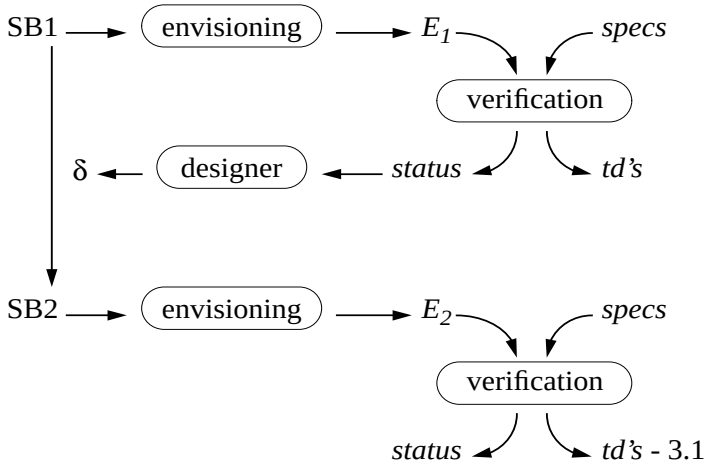


Figure 3.15: Design Flow for the Steam Boiler[16]

3.8 Additional Teleological Operators

We can define additional teleological operators so that the teleological description given for the steam boiler example in the previous section more closely matches the prose description "The pressure sensor was added to the steam boiler design to prevent the internal pressure from exceeding the safety limit `Pmax*`". These additional operators are used in defining composed teleological operators in Chapter 4.[17]

3.8.1 unGuarantees

We define the operator **unGuarantees** to facilitate the definition of other operators. This operator expresses the fact that a specification predicate true in all behaviors of the unmodified design is now false in at least one

[17]All these additional operators can be considered syntactic sugar. They are defined to demonstrate the construction of semantically more complex teleological expressions, and to demonstrate parallels to human generated descriptions of purpose. These parallels help demonstrate that teleological descriptions expressed in TeD capture the semantics of human generated descriptions, and are suitable for human consumption.

behavior of the modified design. The operator **unGuarantees** is defined as follows:

$$\delta \textbf{ unGuarantees } \phi \Leftrightarrow \begin{cases} \forall\, b \in \mathrm{E}, \quad \phi, \\ \text{ and} \\ \exists\, b' \in \mathrm{E}', \quad \neg\phi. \end{cases}$$

Expressed in the modal operators, we have

$$\delta \textbf{ unGuarantees } \phi_1 \Leftrightarrow \begin{cases} \mathcal{M} \text{ is a model for } \Box\phi_1, \\ \text{ and} \\ \mathcal{M}' \text{ is a model for } \Diamond\neg\phi_1. \end{cases}$$

3.8.2 Preventing a Behavior

Consider a design modification made to prevent or exclude an undesirable behavior, as was the case in the steam boiler example. A convenient operator is **Prevents**, defined as follows:

$$\delta \textbf{ Prevents } \phi \Leftrightarrow \begin{cases} \exists\, b \in \mathrm{E}, \quad \phi, \\ \text{ and} \\ \forall\, b' \in \mathrm{E}', \quad \neg\phi. \end{cases}$$

Operator **Prevents** can be expressed in terms of **Guarantees** as

$$\delta \textbf{ Prevents } \phi \Leftrightarrow \delta \textbf{ Guarantees } \neg\phi.$$

Applied to the design modification for the steam boiler, we have

$$\delta \textbf{ Prevents } \texttt{occursIn}(\sigma, b).$$

3.8.3 Introducing a Behavior

Consider a design modification made to introduce or enable a particular behavior abstraction, although the abstraction may not be guaranteed for all possible behaviors of the design. A convenient operator is **Introduces**, defined as follows:

$$\delta \textbf{ Introduces } \phi \;\Leftrightarrow \begin{cases} \forall\, b \in \mathrm{E}, \quad \neg\phi, \\ \text{ and} \\ \exists\, b' \in \mathrm{E}', \quad \phi. \end{cases}$$

Introduces can be expressed in terms of **unGuarantees** as

$$\delta \textbf{ Introduces } \phi \;\Leftrightarrow\; \delta \textbf{ unGuarantees } \neg\phi.$$

This is readily demonstrated by adding a negated predicate to the definition of **unGuarantees**, giving

$$\delta \textbf{ unGuarantees } \neg\phi \;\Leftrightarrow \begin{cases} \forall\, b \in \mathrm{E}, \quad \neg\phi, \\ \text{ and} \\ \exists\, b' \in \mathrm{E}', \quad \phi. \end{cases}$$

which is the definition of **Introduces**.

3.8.4 Conditional Behavior

Finally, consider a specification predicate that expresses a desired characteristic that is conditional, or if ϕ_1 is true (i.e., is observed or measured), then ϕ_2 must be true (observed or measured). To make such specifications and associated teleological descriptions more intuitive, we introduce the operator **Conditionally Guarantees** and define it directly as

$$\delta \textbf{ Conditionally } \text{when } \{\phi_1\} \textbf{ Guarantees } \phi_2 \;\Leftrightarrow \begin{cases} \exists\, b \in \mathrm{E}, \quad \neg(\phi_1 \Rightarrow \phi_2), \\ \text{ and} \\ \forall\, b' \in \mathrm{E}', \quad \phi_1 \Rightarrow \phi_2. \end{cases}$$

Note that **Conditionally Guarantees** is a single operator of three arguments. We write this operator with the conditional argument in the middle for readability. We can rewrite this operator in primitives as:

$$\delta \textbf{ Guarantees } \phi_1 \Rightarrow \phi_2.$$

We can define the operator **Conditionally Prevents**, or conditionally preventing a scenario as follows:

$$\delta \textbf{ Conditionally when } \{\phi_1\} \textbf{ Prevents } \phi_2 \Leftrightarrow \begin{cases} \exists\ b \in \mathrm{E}, & \neg(\phi_1 \Rightarrow \neg\phi_2), \\ \text{ and} & \\ \forall\ b' \in \mathrm{E}', & \phi_1 \Rightarrow \neg\phi_2. \end{cases}$$

We can rewrite this operator in primitives as:

$$\delta \textbf{ Guarantees } \phi_1 \Rightarrow \neg\phi_2.$$

We can define the operator **Conditionally Introduces**, or conditionally introducing a scenario as follows:

$$\delta \textbf{ Conditionally when } \{\phi_1\} \textbf{ Introduces } \phi_2 \Leftrightarrow \begin{cases} \forall\ b \in \mathrm{E}, & \phi_1 \Rightarrow \neg\phi_2), \\ \text{ and} & \\ \exists\ b' \in \mathrm{E}', & \neg(\phi_1 \Rightarrow \neg\phi_2). \end{cases}$$

We can rewrite this operator in primitives as:

$$\delta \textbf{ unGuarantees } \phi_1 \Rightarrow \neg\phi_2.$$

For conditional expressions involving **Prevents** and **Introduces**, one might be tempted to rewrite the expression directly in terms of **Prevents** or **Introduces**. However, directly transforming such an expression to an implication as was done for **Guarantees** violates the semantics of the **Conditionally** operator, namely that 1) the specification predicate ϕ_2 holds when the possible worlds considered are restricted to those in which specification predicate ϕ_1 holds and 2) that no statement is made about possible worlds in which the specification predicate ϕ_1 does not hold. For **Prevents**, an (incorrect) rewrite sequence is

$$\delta \textbf{ Conditionally when } \phi_1 \textbf{ Prevents } \phi_2$$

rewritten as

$$\delta \textbf{ Prevents } \phi_1 \Rightarrow \phi_2$$

which is rewritten as

$$\delta \textbf{ Guarantees } \neg(\phi_1 \Rightarrow \phi_2).$$

This result cannot be true if there are any possible worlds in which the specification predicate ϕ_1 does not hold, since an implication with a false antecedent is always true. For **Introduces**, an (incorrect) rewrite sequence is

$$\delta \textbf{ Conditionally } \text{when } \phi_1 \textbf{ Introduces } \phi_2$$

is rewritten as

$$\delta \textbf{ Introduces } \phi_1 \Rightarrow \phi_2$$

which is rewritten as

$$\delta \textbf{ unGuarantees } \neg(\phi_1 \Rightarrow \phi_2).$$

This result can be satisfied if there is one possible world in which the specification predicate ϕ_1 does not hold, since an implication with a false antecedent is always true.

Chapter 4

Behavior Abstraction

4.1 Rationale

In this chapter we formally develop the notion of behavior abstraction. In the task domain of design,[1] behavior abstraction is important because design specifications most often address only a single aspect or small number of aspects of the artifact to be designed, such as the physical dimensions (length, width, height) or the behavior of some portion of the artifact. Consequently, one needs to represent and reason about behavior descriptions that reference only part of the artifact. Further, the design specification may be given in terms more general than the details of the designed artifact, such as stating that a particular variable value should always be positive, although no specific positive value is specified.

To demonstrate behavior abstraction, recall the steam boiler example discussed in Chapter 3. The behavior in which the internal pressure of the boiler vessel instance `vessel` starts at `0` and goes to the positive value `Pmax*` is written as a sequence of three states,

```
⟨{((vessel pressure) (0 nil))},
 {((vessel pressure) ((0 Pmax*) inc))},
 {((vessel pressure) (Pmax* std))}⟩
```

[1] Hamscher [Ham91] discusses the relevance of behavior abstraction in the problem solving domain of diagnosis.

This behavior abstracts the behavior of the entire system, since variables other than `pressure` are ignored. One possible generalization of this abstract behavior is obtained by replacing the component name `vessel` with the component type `boiler-vessel`. This abstraction describes the behavior in which the internal pressure of *any* instance of `boiler-vessel` starts at `0` and goes to the positive value `Pmax*`. A further generalization eliminates the intermediate state, describing the behavior in terms of its initial and final states, or

```
⟨{((boiler-vessel pressure) (0 nil))},
 {((boiler-vessel pressure) (Pmax* std))}⟩
```

We define behavior abstraction in terms of state abstraction, and state abstraction in terms of variable value, name, and type abstraction. We prove that each abstraction relation partially orders[2] its respective space. Ordering behaviors allows us to decide whether one behavior is more or less general than another, or that no order exists between the behaviors. Table 4.1 summarizes the abstraction relations, giving their respective spaces and the relations used in each definition. For each relation, the strict inequality $a \sqsubset_* b$ is defined as

$$a \sqsubseteq_* b \ \wedge \ a \neq b.$$

We use the behavior abstraction relation to define the specification predicate `occursIn`.

Computing the abstraction relation is the basis for indexing and classifying teleological descriptions (see Chapter 8) and the model checking algorithm used in acquiring teleological descriptions (see Chapter 9). The particular

[2]We use the definition of a partial ordering relation in [Sto77, p. 82].

Relation	Space	Defined in Terms of
$\sqsubseteq_c$	component types	assumed
$\sqsubseteq_n$	variable names	component types
$\sqsubseteq_t$	variable types	generic variable types
$\sqsubseteq_v$	variable references	$\sqsubseteq_n$, $\sqsubseteq_t$
$\sqsubseteq_x$	values	qualitative/quantitative values
$\sqsubseteq_s$	states	$\sqsubseteq_v$, $\sqsubseteq_x$
$\sqsubseteq_b$	behaviors	$\sqsubseteq_s$
$\sqsubseteq_\sigma$	scenarios	$\sqsubseteq_b$

Table 4.1: Abstraction Relation Summary

details of the abstraction relations defined here are not critical for accomplishing indexing, classification, and acquisition. *The key requirement for these capabilities is that behaviors can be partially ordered.*

4.2 Variable Abstraction

Behavior abstraction described in this chapter involves variables and their values, and we describe abstraction relations for variables as well as their values. A variable (reference) is composed of a name and a type. The name is a hierarchical name, as described in Section 3.2. A fully qualified hierarchical name uniquely identifies a single variable in a design instance. The name can be abstracted by removing individual elements from the list, such as removing all elements except the component name and the variable name. Additionally, the name can be abstracted by replacing an element (e.g., component name) with a generalization of that component.[3] For example, in the domain of electronic circuits, a two input AND gate and a three input AND gate can be

[3]For example, see the model hierarchy based on behavior described by Nayak, Joskowicz, and Addanki [NJA91].

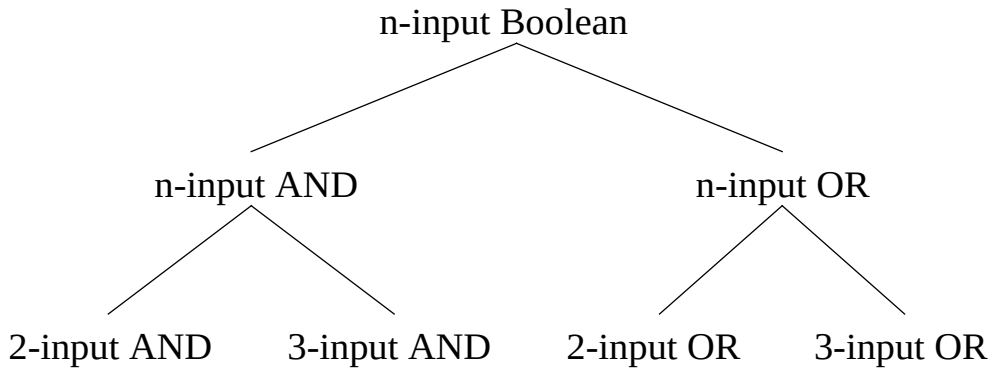


Figure 4.1: Component Type Hierarchy

generalized to n-input AND gate. Similarly, a two input OR and a three input OR gate are generalized to n-input OR, and n-input AND and n-input OR can be generalized to n-input Boolean gate, as shown in Figure 4.1.

Variable type can be generalized via the generic variable types in bond graph modeling, described by Rosenberg and Karnopp [RK83]. Domain specific types such as voltage and current (electrical domain) can be generalized to effort and flow, respectively. The most general type is designated $\top$, (read "top"). To demonstrate a variable reference and its abstraction, consider a circuit design containing a 2-input AND gate (instance name `g1`), with variable `Vin` representing the input voltage of an AND gate. The variable reference for the input voltage of `g1` is `(g1 Vin)`. The reference can be abstracted to `(2-input-AND Vin)`, referencing the input voltage of any 2-input AND gate in the circuit. Another possible abstraction is `(g1 voltage)`, referencing any voltage variable of gate `g1`.

We now define the abstraction relations for variable names and for variable types. We assume the abstraction relation $\sqsubseteq_c$ partially orders the

space of component types.[4]

The relation $\sqsubseteq_n$ (read "is a variable name less general or equal to") captures the notion of variable name abstraction. For variable name $n = (n_1, \ldots, n_k)$, $n' = (n'_1, \ldots, n'_l)$

$$n \sqsubseteq_n n' \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : n' \rightarrow n \text{ such that} \\ \\ \forall\ n'_i \in\ n', \text{ if } \mathcal{F}(n'_i) = n_{j_i}, \text{ then} \\ \qquad j_i < j_{i+1}, \quad (\textit{Order Preservation and Uniqueness}) \\ \qquad n'_i \text{ is a generalization of } n_{j_i}, \quad (\textit{Name Abstraction}) \end{cases}$$

Lemma 4.1 $\sqsubseteq_n$ *is reflexive.*
Proof: For $n = (n_1, \ldots, n_k)$, define $\mathcal{F}(n_i) = n_i$. Now, the order preservation, uniqueness, and name abstraction (we assumed that the component abstraction is a partial order, and hence reflexive) conditions are satisfied. □

Lemma 4.2 $\sqsubseteq_n$ *is anti-symmetric.*
Proof: For $n = (n_1, \ldots, n_k)$, $n' = (n'_1, \ldots, n'_l)$, first observe that if $n \sqsubseteq_n n'$ and $n' \sqsubseteq_n n$, then we must have $k = l$ in order to satisfy the uniqueness condition. With $k = l$, we can see that the mappings $\mathcal{F}'$ and $\mathcal{F}$ by which $n \sqsubseteq_n n'$ and $n' \sqsubseteq_n n$ hold, respectively, must be exactly $\mathcal{F}'(n'_i) = n_i$ and $\mathcal{F}(n_i) = n'_i$ for the order preservation condition to hold. Now, n'_i must be a component abstraction of n_i (for all i) and n_i must be a component abstraction of n'_i (for all i), which with the fact that component abstraction relation is a partial order (an assumption) gives us $n_i = n'_i$ for all i, and $n = n'$. □

Lemma 4.3 $\sqsubseteq_n$ *is transitive.*
Proof: For variable names $a = (a_1, \ldots, a_l)$, $b = (b_1, \ldots, b_m)$, $c = (c_1, \ldots, c_n)$, let $a \sqsubseteq_n b$ via mapping $\mathcal{F}_b$ and $b \sqsubseteq_n c$ via mapping $\mathcal{F}_c$. Define mapping $\mathcal{F} : c \rightarrow a$ as $\mathcal{F}(c_i) = \mathcal{F}_b(\mathcal{F}_c(c_i))$. Since both $\mathcal{F}_b$ and $\mathcal{F}_c$ satisfy the order preservation and uniqueness conditions, then $\mathcal{F}$ also satisfies the order preservation and uniqueness conditions. From the name abstraction conditions of $\mathcal{F}_b$ and $\mathcal{F}_c$

[4]This assumption is reasonable, requiring only that the component abstraction relation be reflexive (a component type is "less general than or equal to" itself), anti-symmetric (if component type A is "less general than or equal to" component type B, and B is "less general than or equal to" A, then A and B are the same component type), and transitive (if component type A is "less general than or equal to" component type B, and B is "less general than or equal to" component type C, then A is "less general than or equal to" C).

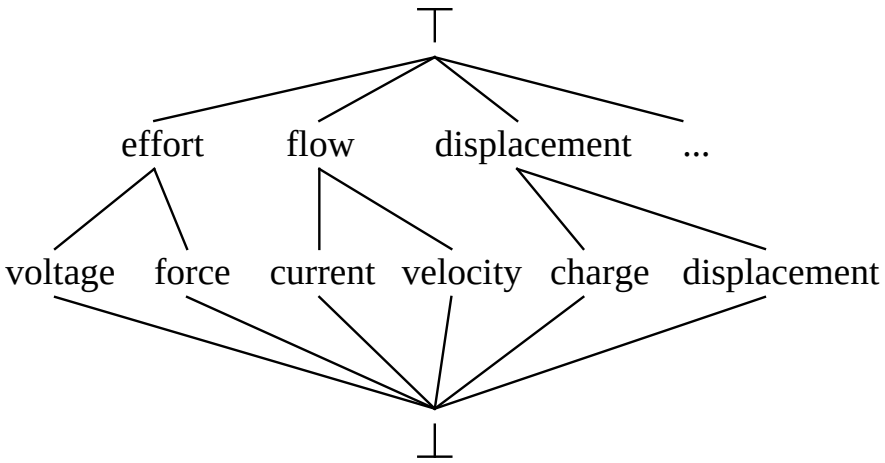


Figure 4.2: Variable Type Hierarchy

(i.e., b_i is more general than $\mathcal{F}_b(b_i)$ and c_j is more general than $\mathcal{F}_c(c_j)$) and from the transitivity of the component abstraction relation, we have c_k is more general than $\mathcal{F}(c_k)$. Now, $a \sqsubseteq_n c$. □

Theorem 4.4 $\sqsubseteq_n$ *is a partial order.*
Proof: From Lemmas 4.1, 4.2, and 4.3. □

Theorem 4.5 *The variable type abstraction relation (* $\sqsubseteq_t$ *) is a partial order.*
Proof: Domain specific variable types such as voltage, current, and charge for the electrical domain and force, velocity, and displacement for the mechanical domain are generalized as effort, flow, and displacement. These generic variable types are then abstracted to $\top$. This hierarchy is shown in part in Figure 4.2, and the properties reflexive, anti-symmetric, and transitive can be demonstrated. □

We define the relation $\sqsubseteq_v$ (read "is a variable less general or equal to") based on the relations $\sqsubseteq_n$ and $\sqsubseteq_t$ (both partial orders) as follows: For variables $v = (n, t)$ and $v' = (n', t')$,

$$v \sqsubseteq_v v' \Leftrightarrow \begin{cases} n \sqsubseteq_n n', \\ \text{and} \\ t \sqsubseteq_t t'. \end{cases}$$

Variable abstraction requires both variable name and variable type abstraction since the type of a variable name that is not fully qualified can be ambiguous. For example, a variable name can be abstracted to `(X)` in a design containing two component instances (of different component type) that each define a variable named `X`, but with different variable type. A variable reference intended to abstract only variables named `(X)` of a specific type such as `voltage` requires the variable type as well. We now prove the assertion that $\sqsubseteq_v$ is a partial order.

Lemma 4.6 $\sqsubseteq_v$ *is reflexive.*
Proof: For variable $v = (n, t)$, $n \sqsubseteq_n n$ (Lemma 4.1) and $t \sqsubseteq_t t$ (Theorem 4.5). Now, $v \sqsubseteq_v v$. □

Lemma 4.7 $\sqsubseteq_v$ *is anti-symmetric.*
Proof: For variables $v = (n, t)$ and $v' = (n', t')$. Suppose $v \sqsubseteq_v v'$ and $v' \sqsubseteq_v v$. From the definition of $\sqsubseteq_v$, $n \sqsubseteq_n n'$, $n' \sqsubseteq_n n$, $t \sqsubseteq_t t'$, and $t' \sqsubseteq_t t$. From Lemma 4.2, we have $n = n'$, and from Theorem 4.5 we have $t = t'$. Now, $v = v'$. □

Lemma 4.8 $\sqsubseteq_v$ *is transitive.*
Proof: For variables $u = (n_u, t_u)$, $v = (n_v, t_v)$, and $w = (n_w, t_w)$, suppose $u \sqsubseteq_v v$ and $v \sqsubseteq_v w$. From the definition of $\sqsubseteq_v$, $n_u \sqsubseteq_n n_v$, $n_v \sqsubseteq_n n_w$, $t_u \sqsubseteq_t t_v$, and $t_v \sqsubseteq_t t_w$. From Lemma 4.3, we have $n_u \sqsubseteq_n n_w$, and from Theorem 4.5 we have $t_u \sqsubseteq_t t_w$. Now, $u \sqsubseteq_v w$. □

Theorem 4.9 $\sqsubseteq_v$ *is a partial order.*
Proof: From Lemmas 4.6, 4.7, and 4.8. □

4.3 Partial States

A *partial state* is an abstraction of a state, possibly equal to the state. To the variable abstraction relation described in the previous section, we add two abstraction relations to achieve state abstraction. First, a partial state can abstract a state by abstracting the variable value. This is accomplished by

specifying a range for a variable value, such as $x > 0$. The relation $\sqsubseteq_x$ (read "is a value less general than or equal to") captures this notion of variable value abstraction. For example, for variable x with $\texttt{domain}(x)$ the union of real numbers ($\mathcal{R}$) and open intervals on real numbers, the values $5, (4,6), (0, \texttt{inf})$, and $\mathcal{R}$ are related by $\sqsubseteq_x$ as follows:

$$5 \sqsubseteq_x (4,6) \sqsubseteq_x (0, \texttt{inf}) \sqsubseteq_x \mathcal{R}.$$

We define the relation $\sqsubseteq_x$ for the space of qualitative values (qm, qd), where qm is a magnitude represented as a point or open interval taken from the union of symbolic quantity spaces and $\mathcal{R}$, and qd is one of $\{\texttt{dec}, \texttt{std}, \texttt{inc}, \texttt{nil}\}$. The relation $\sqsubseteq_x$ is defined for infinite domains (reals, rationals, integers) via the $<$ and $\leq$ relations defined for these domains, and for finite domains (quantity spaces) via the partial order imposed by the quantity space (which we will denote with the relational operators $<, \leq$). Considering only the magnitudes of the qualitative values, for point values x and y,

$$x \sqsubseteq_x y \;\Leftrightarrow\; x = y;$$

For point value x and open interval value (y_1, y_2),

$$x \sqsubseteq_x (y_1, y_2) \;\Leftrightarrow\; y_1 < x \;\wedge\; x < y_2;$$

For open interval values (x_1, x_2) and (y_1, y_2),

$$(x_1, x_2) \sqsubseteq_x (y_1, y_2) \;\Leftrightarrow\; y_1 \leq x_1 \;\wedge\; x_2 \leq y_2.$$

The direction of change values `dec`, `std`, and `inc` are all pairwise unordered, and `nil` is more general than the other three values. To complete the definition of $\sqsubseteq_x$, $x \sqsubseteq_x y$ if the magnitude relationships described above hold, and either the direction of change of x and y are the same or the direction of change of y is `nil`. We now prove the assertion that $\sqsubseteq_x$ is a partial order.

Lemma 4.10 $\sqsubseteq_x$ *is reflexive.*
Proof: For point value x, $x = x \Rightarrow x \sqsubseteq_x x$. For open interval value $y = (y_1, y_2)$, $y_1 \leq y_1 \ \wedge \ y_2 \leq y_2 \Rightarrow y \sqsubseteq_x y$. $\square$

Lemma 4.11 $\sqsubseteq_x$ *is anti-symmetric.*
Proof: For point values x and y, suppose $x \sqsubseteq_x y$ and $y \sqsubseteq_x x$. From the definition of $\sqsubseteq_x$, $x = y$. For point value x and interval value ;$y = (y_1, y_2)$, $y \sqsubseteq_x x$ is false. For interval values $x = (x_1, x_2)$ and $y = (y_1, y_2)$, suppose $x \sqsubseteq_x y$ and $y \sqsubseteq_x x$. From definition of $\sqsubseteq_x$, $y_1 \leq x_1$, $x_2 \leq y_2$, $x_1 \leq y_1$, and $y_2 \leq x_2$, which gives $x_1 = y_1$ and $x_2 = y_2$, and $x = y$. $\square$

Lemma 4.12 $\sqsubset_x$ *is transitive.*
Proof: Suppose $x \sqsubseteq_x y$ and $y \sqsubseteq_x z$. If z is a point value, then $z = y$, y is also a point value, and $x = y$. Now, $x \sqsubseteq_x z$. If y is a point value, then $x = y$, and $x \sqsubseteq_x z$. If x is a point value and y and z are interval values, then $z_1 \leq y_1 < x < y_2 \leq z_2$, and $x \sqsubseteq_x z$. If x, y, and z are all interval values, then $z_1 \leq y_1 \leq x_1$ and $x_2 \leq y_2 \leq z_2$, and $x \sqsubseteq_x z$. $\square$

Theorem 4.13 $\sqsubseteq_x$ *is a partial order.*
Proof: From Lemmas 4.10, 4.11, and 4.12. $\square$

The second abstraction relation for states eliminates variables from consideration (hence the name *partial state*). A variable is eliminated from consideration by assigning the variable a value that represents any possible value for the variable. The relation $\sqsubseteq_s$ (read "is a state less general than or equal to") defines an abstraction relation for states and partial states. For (partial) state s with variable set $\mathcal{V}_s$, and partial state p with variable set $\mathcal{V}_p$,

$$s \sqsubseteq_s p \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : \mathcal{V}_p \rightarrow \mathcal{V}_s \text{ such that} \\ \\ \forall\ v \in \mathcal{V}_p, \\ \quad \mathcal{F}(v) \sqsubseteq_v v, \quad (\textit{Variable Abstraction}) \\ \quad s(\mathcal{F}(v)) \sqsubseteq_x p(v), \quad (\textit{Value Abstraction}) \\ \text{and} \\ \forall\ v_1, v_2 \in \mathcal{V}_p, v_1 \neq v_2 \Rightarrow \mathcal{F}(v_1) \neq \mathcal{F}(v_2). \quad (\textit{Uniqueness}) \end{cases}$$

Lemma 4.14 $\sqsubseteq_s$ *is reflexive.*
Proof: For state s with variable set $\mathcal{V}$, define mapping $\mathcal{F}$ such that for $v \in \mathcal{V}$, $\mathcal{F}(v) = v$. Clearly, the mapping satisfies the uniqueness condition. Further, the variable and value abstraction conditions are satisfied, since both $\sqsubseteq_v$ and $\sqsubseteq_x$ are reflexive (Lemmas 4.6 and 4.10, respectively), giving $\mathcal{F}(v) \sqsubseteq_v v$ and $s(\mathcal{F}(v)) \sqsubseteq_x s(v)$. □

Lemma 4.15 $\sqsubseteq_s$ *is transitive.*
Proof: Let p_i be (partial) states. Suppose $p_1 \sqsubseteq_s p_2$ via mapping $\mathcal{F}_2$ and $p_2 \sqsubseteq_s p_3$ via mapping $\mathcal{F}_3$. From the definition of $\sqsubseteq_s$ we have

$$\forall\ v \in \mathcal{V}_2,\ \mathcal{F}_2(v) \sqsubseteq_v v,\ p_1(\mathcal{F}_2(v)) \sqsubseteq_x p_2(v), \text{ and}$$
$$\forall\ v \in \mathcal{V}_3,\ \mathcal{F}_3(v) \sqsubseteq_v v,\ p_2(\mathcal{F}_3(v)) \sqsubseteq_x p_3(v).$$

Define $\mathcal{F} : p_3 \rightarrow p_1$ as $\mathcal{F}_2 \circ \mathcal{F}_3$. From transitivity of $\sqsubseteq_v$ (Lemma 4.8) we have $\forall\ v \in \mathcal{V}_3, \mathcal{F}(v) \sqsubseteq_v v$, and from transitivity of $\sqsubseteq_x$ (Lemma 4.12) we have $\forall\ v \in \mathcal{V}_3, p_1(\mathcal{F}(v)) \sqsubseteq_x p_3(v)$. Further, for $v_1, v_2 \in \mathcal{V}_3$, $v_1 \neq v_2 \Rightarrow \mathcal{F}_3(v_1) \neq \mathcal{F}_3(v_2) \Rightarrow \mathcal{F}_2(\mathcal{F}_3(v_1)) \neq \mathcal{F}_2(\mathcal{F}_3(v_2))$, and $\mathcal{F}(v_1) \neq \mathcal{F}(v_2)$. Now, $p_1 \sqsubseteq_s p_3$. □

Lemma 4.16 $\sqsubseteq_s$ *is anti-symmetric.*
Proof: For states r and s with variable sets $\mathcal{V}_r$ and $\mathcal{V}_s$, respectively, suppose $r \sqsubseteq_s s$ via mapping $\mathcal{F}_s$ and $s \sqsubseteq_s r$ via mapping $\mathcal{F}_r$. The uniqueness property of $\sqsubseteq_s$ implies that $|\mathcal{V}_r| = |\mathcal{V}_s|$. Now consider the mapping $\mathcal{H} = \mathcal{F}_r \circ \mathcal{F}_s$, noting that $\mathcal{H} : \mathcal{V}_s \rightarrow \mathcal{V}_s$ and is one-to-one (from the uniqueness property of $\sqsubseteq_s$).
Claim: $\mathcal{H}$ is the identity mapping on $\mathcal{V}_s$.
Claim proof: Suppose that $\mathcal{H}$ is not the identity mapping on $\mathcal{V}_s$. Then there exists $v \in \mathcal{V}_s$ such that $\mathcal{H}(v) \neq v$. Now consider the sequence $\mathcal{H}(v), \mathcal{H}^2(v), \ldots$. For each step in the sequence, we know $\mathcal{H}^{i+1}(v) \sqsubseteq_x \mathcal{H}^i(v)$, since $\mathcal{H}(v) = \mathcal{F}_r(\mathcal{F}_s(v))$ and $\sqsubseteq_s$ is transitive (from Lemma 4.15), implying that $\mathcal{H}(v) \sqsubseteq_s v$. If at any point the two are equal, we have violated the one-to-one property of $\mathcal{H}$, a contradiction. However, we cannot have $\sqsubset_x$ at every point, since there are finitely many elements of $\mathcal{V}_s$. Therefore, $\mathcal{H}$ is the identity mapping on $\mathcal{V}_s$. □.
Now, $\mathcal{H} = \mathcal{F}_s \circ \mathcal{F}_r =$ the identity map, and $\mathcal{F}_r = \mathcal{F}_s^{-1}$. Similarly, we can show that $\mathcal{F}_s = \mathcal{F}_r^{-1}$. Now, $\forall v \in \mathcal{V}_s$,

$$s(v) = s(\mathcal{F}_r(\mathcal{F}_s(v))) \sqsubseteq_x r(\mathcal{F}_s(v)) \sqsubseteq_x s(v), \text{ and}$$
$$v = \mathcal{F}_r(\mathcal{F}_s(v)) \sqsubseteq_v \mathcal{F}_s(v) \sqsubseteq_v v$$

and we have $r(\mathcal{F}_s(v)) = s(v)$ and $\mathcal{F}_s(v) = v$. Similarly, $\forall v \in \mathcal{V}_r$,

$$r(v) = r(\mathcal{F}_s(\mathcal{F}_r(v))) \sqsubseteq_x s(\mathcal{F}_r(v)) \sqsubseteq_x r(v), \text{ and}$$
$$v = \mathcal{F}_s(\mathcal{F}_r(v)) \sqsubseteq_v \mathcal{F}_r(v) \sqsubseteq_v v$$

and we have $s(\mathcal{F}_r(v)) = r(v)$ and $\mathcal{F}_r(v) = v$. Now, $r = s$. □

Theorem 4.17 $\sqsubseteq_s$ *is a partial order.*
Proof: From Lemmas 4.14, 4.16, and 4.15. □

4.4 Abstract Behaviors

In addition to variable abstraction, value abstraction (point value to an interval, or interval to "wider" interval), and state abstraction (eliminating variables), behavior abstraction generalizes by eliminating certain states in the sequence. Hence, partial states of an abstract behavior need not be adjacent states as defined in Section 3.5.1. To demonstrate this point, consider (in the behavior language of QSIM) the behavior abstraction

$$\langle\{(x, (0, \texttt{dec}))\}, \{(x, (0, \texttt{inc}))\}\rangle.$$

The states abstracted in this scenario cannot be adjacent in a behavior of the model, but are allowed as a behavior abstraction The abstraction describes behavior in which x at some time had the qualitative value `(0, dec)`, and at some later time (with an unspecified number of intervening values) had the value `(0, inc)`. In the semantics of QSIM, the variable x must take on a qualitative value whose direction of change is `std` between `dec` and `inc`.

The basic idea of behavior abstraction is that there exists a correspondence between the abstraction and the behavior such that

1. a partial state of the abstraction is assigned to a state of the behavior which it abstracts, and

2. the order of states implied by the assignment (i.e., the order of partial states) is preserved in the abstracted behavior

The relation $\sqsubseteq_b$ (read "is a behavior less general than or equal to") defines the abstraction relation on behaviors. Formally, for behavior $b = \langle s_1, s_2, \ldots\rangle$

and behavior $b' = \langle s'_1, s'_2, \ldots \rangle$,

$$b \sqsubseteq_b b' \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : b' \rightarrow b \text{ such that} \\ \\ \forall\ s'_i \in\ b',\ \mathcal{F}(s'_i) = s_{j_i}, \\ \qquad j_i < j_{i+1}, \quad (\textit{Order Preservation}) \\ \qquad s_{j_i} \sqsubseteq_s s'_i, \quad (\textit{State Abstraction}) \\ \text{and} \\ \forall\ s'_i, s'_j \in\ b', i \neq j \Rightarrow \mathcal{F}(s'_i) \neq \mathcal{F}(s'_j). \quad (\textit{Uniqueness}) \end{cases}$$

The symbol $\sqsubseteq_b$ is read "is a behavior less general than" and requires for some $s'_i \in b'$ that $s_{j_i} \sqsubset_s s'_i$.

Lemma 4.18 $\sqsubseteq_b$ *is reflexive.*
Proof: For behavior $b = \langle s_1, s_2, \ldots \rangle$, define $\mathcal{F} : b \rightarrow b$ as $\mathcal{F}(s_i) = s_i$. $\mathcal{F}$ clearly satisfies the order preservation and uniqueness properties. Further, $s_i \sqsubseteq_s s_i$ from Lemma 4.14, and now $b \sqsubseteq_b b$. □

Lemma 4.19 $\sqsubseteq_b$ *is anti-symmetric.*
Proof: For behaviors $b = \langle s_1, \ldots, s_i \rangle$ and $b' = \langle s'_1, \ldots, s_j \rangle$, suppose $b \sqsubseteq_b b'$ via mapping $\mathcal{F}'$ and $b' \sqsubseteq_b b$ via mapping $\mathcal{F}$. The uniqueness property requires that $i = j$, and the order preservation property requires that $\mathcal{F}(s_k) = s'_k$ and $\mathcal{F}'(s'_k) = s_k$. From the definition of $\sqsubseteq_b$ we have $s_k \sqsubseteq_s s'_k$ and $s'_k \sqsubseteq_s s_k$, which with Lemma 4.16 gives $s_k = s'_k$ for all k. Now, $b = b'$. □

Lemma 4.20 *For behaviors $b \sqsubseteq_b b'$, (partial) states s'_i and s'_j of b', and (partial) states $s_{k_i} = \mathcal{F}(s'_i)$ and $s_{k_j} = \mathcal{F}(s'_j)$ of b, $i < j \ \Rightarrow\ k_i < k_j$.*
Proof: Straightforward, from definition of $\sqsubseteq_b$ and induction. □

Lemma 4.21 $\sqsubseteq_b$ *is transitive.*
Proof: For behaviors a, b, and c, suppose $a \sqsubseteq_b b \ \wedge\ b \sqsubseteq_b c$. From the definition of $\sqsubseteq_b$, there exist functions $\mathcal{F}_b : b \rightarrow a$ and $\mathcal{F}_c : c \rightarrow b$ such that

$$\forall\ b_i \in\ b,\ \mathcal{F}_b(b_i) = a_{j_i},\ j_i < j_{i+1},\ a_{j_i} \sqsubseteq_s b_i, \text{ and}$$

$$\forall\ c_i \in\ c,\ \mathcal{F}_c(c_i) = b_{j_i},\ j_i < j_{i+1},\ b_{j_i} \sqsubseteq_s c_i.$$

Define $\mathcal{F} = \mathcal{F}_b(\mathcal{F}_c)$, so that $\mathcal{F} : c \rightarrow a$. Now for $\mathcal{F}(c_i) = a_{j_i}$, we need to show $j_i < j_{i+1}$ (order preservation property), $a_{j_i} \sqsubseteq_s c_i$ (state abstraction property), and the uniqueness property holds. Let $b_{k_i} = \mathcal{F}_c(c_i)$ and $a_{j_i} = \mathcal{F}_b(b_{k_i})$. From the definition of $\sqsubseteq_b$, we have $k_i < k_{i+1}$, and from Lemma 4.20 we have

$j_i < j_{i+1}$. From definition of $\sqsubseteq_b$, we have $b_{k_i} \sqsubseteq_s c_i$ and $a_{j_i} \sqsubseteq_s b_{k_i}$. From Lemma 4.15, $a_{j_i} \sqsubseteq_s c_i$. Finally, for $c_i, c_j \in c$, $i \neq j \Rightarrow \mathcal{F}_c(c_i) \neq \mathcal{F}_c(c_j) \Rightarrow \mathcal{F}_b(\mathcal{F}_c(c_i)) \neq \mathcal{F}_b(\mathcal{F}_c(c_j))$, and $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$. Now we have $a \sqsubseteq_b c$. □

Theorem 4.22 $\sqsubseteq_b$ *is a partial order.*
Proof: From Lemmas 4.18, 4.19, and 4.21. □

4.5 Scenarios

A *scenario* is a pair (p, β) where $p = \langle p_1, p_2, \ldots \rangle$ is an abstract behavior and β is a Boolean expression. We introduce the Boolean expression β in order to add constraints on the behavior expressed in terms of logical connectives $(\wedge, \vee, \neg)$, relational operators, constants, and variable values (e.g., $s_1(v)$, where v is a variable in b). β can constrain the set of behaviors that the scenario abstracts. For example, if β is $(p_2(time) - p_1(time)) < 10$, then scenario (p, β) abstracts behavior $b = \langle b_1, \ldots \rangle$ if $b \sqsubseteq_b p$ via mapping $\mathcal{F}$, and for $b_i = \mathcal{F}(p_1)$, $b_j = \mathcal{F}(p_2)$, the inequality $(b_j(time) - b_i(time)) < 10$ holds.[5]

For convenience in defining composed teleological operators later in this chapter, notation for scenario concatenation and scenario merging is introduced. Let $\mathcal{V}$ be the set of variables of a design. For scenarios

$$\sigma = (p = \langle p_1, \ldots, p_n \rangle, \beta), \sigma' = (p' = \langle p_1', \ldots, p_m' \rangle, \beta'),$$

where p references[6] variables in $\mathcal{V}_p \subseteq \mathcal{V}$ and p' references variables in $\mathcal{V}_{p'} \subseteq \mathcal{V}$, we define $[\sigma; \sigma']$ as follows:

$$[\sigma; \sigma'] = (\langle p_1, \ldots, p_n, p_1', \ldots, p_m' \rangle, (\beta \ \wedge \ \beta')).$$

[5]For variables with interval values, the expression is quantified over all possible values in the interval.

[6]By "reference a variable" we mean the partial state defines a value for the variable other than "any possible value".

For $n = m$, we define $[\sigma \parallel \sigma^{'}]$ as follows:

$$[\sigma \parallel \sigma^{'}] = \begin{cases} (\langle q_1, \ldots, q_n \rangle, (\beta \ \wedge \ \beta')) \\ \quad \text{where } q_i = p_i \cup p_i^{'} \text{ and} \\ \quad \text{for } v \in (\mathcal{V}_p \cap \mathcal{V}_{p'}), q_i(v) = g.l.b.\{p_i(v), p_i^{'}(v)\}. \end{cases}$$

As was done for values, states, and behaviors, we define an abstraction relation (partial order) on scenarios. The relation $\sqsubseteq_\sigma$ (read "is a scenario less general than or equal to") is defined as follows: For scenarios $\sigma = (p, \beta)$ and $\sigma' = (p', \beta')$, with $p = \langle p_1, \ldots \rangle$ and $p' = \langle p'_1, \ldots \rangle$,

$$\sigma \sqsubseteq_\sigma \sigma' \Leftrightarrow \begin{cases} p \sqsubseteq_b p' \text{ via mapping } \ \mathcal{F}' : p' \rightarrow p \ (\mathit{BehaviorAbstraction}) \\ \ \text{and} \\ \beta \Rightarrow \mathcal{F}'(\beta') \ \ (\mathit{ConditionAbstraction}). \end{cases}$$

where $\mathcal{F}'(\beta')$ denotes the rewriting of β' with respect to the mapping $\mathcal{F}' : p' \rightarrow p$ (i.e., variable reference $p'_i(v)$ in β' is replaced by $p_j(v)$, where $p_j = \mathcal{F}'(p'_i)$). Scenarios σ and σ' are equivalent if $p \sqsubseteq_b p'$ and $\beta \Leftrightarrow \mathcal{F}'(\beta')$.

Lemma 4.23 $\sqsubseteq_\sigma$ *is reflexive.*
Proof: For scenario $\sigma = (p, \beta)$, define mapping $\mathcal{F}' : p \rightarrow p$ as the identity mapping. From Lemma 4.18 we have $p \sqsubseteq_b p$. Since $\mathcal{F}'$ is the identity mapping on p, $\mathcal{F}'(\beta) = \beta$, in which case $\beta \Rightarrow \mathcal{F}'(\beta)$. Now, $\sigma \sqsubseteq_\sigma \sigma$. □

Lemma 4.24 $\sqsubseteq_\sigma$ *is anti-symmetric.*
Proof: For scenarios $\sigma = (p, \beta)$ and $\sigma' = (p', \beta')$, suppose $\sigma \sqsubseteq_\sigma \sigma'$ via mapping $\mathcal{F}'$ and $\sigma' \sqsubseteq_\sigma \sigma$ via mapping $\mathcal{F}$. From $p \sqsubseteq_b p'$, $p' \sqsubseteq_b p$, and Lemma 4.19, we have $p = p'$. We can now choose the identity mapping for both $\mathcal{F}$ and $\mathcal{F}'$ since $p = p'$ and $\sqsubseteq_\sigma$ is reflexive (Lemma 4.23). Now, $\beta \Rightarrow \mathcal{F}'(\beta')$, $\beta' \Rightarrow \mathcal{F}(\beta)$, and $\sigma \equiv \sigma'$. □

Lemma 4.25 $\sqsubseteq_\sigma$ *is transitive.*
Proof: For $\sigma_i = (p_i, \beta_i)$, suppose $\sigma_1 \sqsubseteq_\sigma \sigma_2$ and $\sigma_2 \sqsubseteq_\sigma \sigma_3$. From $p_1 \sqsubseteq_b p_2$, $p_2 \sqsubseteq_b p_3$, and Lemma 4.21 we have $p_1 \sqsubseteq_b p_3$. Given

$$\mathcal{F}_2 : p_2 \rightarrow p_1 \text{ such that } \beta_1 \Rightarrow \mathcal{F}_2(\beta_2),$$
$$\mathcal{F}_3 : p_3 \rightarrow p_2 \text{ such that } \beta_2 \Rightarrow \mathcal{F}_3(\beta_3),$$

define $\mathcal{F} = \mathcal{F}_2 \circ \mathcal{F}_3$. We need to show that $\beta_1 \Rightarrow \mathcal{F}(\beta_3)$. Recalling that the interpretation for variables with interval values is that the expression is true

for all possible point values of the interval, note that the mapping $\mathcal{F}_2$ restricts interval values to tighter intervals or to point values. Hence, $(\beta_2 \Rightarrow \mathcal{F}_3(\beta_3)) \Rightarrow (\mathcal{F}_2(\beta_2) \Rightarrow \mathcal{F}_2(\mathcal{F}_3(\beta_3)))$. Since $\beta_1 \Rightarrow \mathcal{F}_2(\beta_2)$, we have $\beta_1 \Rightarrow \mathcal{F}(\beta_3)$. Now, $\sigma_1 \sqsubseteq_\sigma \sigma_3$. □

Theorem 4.26 $\sqsubseteq_\sigma$ *is a partial order.*
Proof: From Lemmas 4.23, 4.24, and 4.25. □

4.6 Design Specifications for Behavior

A design specification states whether a scenario is required or prohibited. The syntax of a design specification is one of

`(required` σ`)` or `(prohibited` σ`)`

where σ is a scenario. A design specification participates in a teleological description in the form of an associated specification predicate. We define the specification predicate `occursIn(`σ`,`b`)` for scenario $\sigma = (p, \beta)$ as follows:

$$\texttt{occursIn}(\sigma, b) \Leftrightarrow \begin{cases} b \sqsubseteq_b p \text{ via mapping } \mathcal{F} : p \rightarrow b \\ \text{ and} \\ \mathcal{F}(\beta) \text{ is } \texttt{true}. \end{cases}$$

A scenario is said to *occur in* an envisionment if it occurs in at least one behavior of the envisionment. A set of scenarios $\{\sigma_1, \ldots, \sigma_n\}$ occurs in a behavior b if each σ_i occurs in b. Note that each scenario can occur in the behavior via a different instance of the function $\mathcal{F}$. The specification predicates associated with the basic design specifications are, respectively,

`occursIn(`σ`,`b`)`, `¬occursIn(`σ`,`b`)`

A design description can contain a precondition for the desired behavior as follows:

(conditionally ⟨*list of scenarios*⟩
(required ⟨*scenario*⟩)
(prohibited ⟨*scenario*⟩))

To simplify the expression of design specifications, we allow a symbol to be bound to a component type and then used in variable names in scenarios. The syntax for such a design specification is:

(for-component ⟨*symbol*⟩ ⟨*type descriptor*⟩
(conditionally ⟨*list of scenarios*⟩
(required ⟨*scenarios*⟩)
(prohibited ⟨*scenarios*⟩)))

For each instance of component type ⟨*type descriptor*⟩, the design specification is instantiated. Considering the steam boiler example of Chapter 3, the design specification concerning the variable `pressure` in component type `boiler-vessel` is

```
(for-component X (boiler-vessel)
  (prohibited (((((X pressure) ((Pmax* inf) ign)))) true)))
```

4.7 Composed Teleological Operators

The operators described here demonstrate the ability to define semantically richer operators in terms of the three operators **Guarantees**, **Prevents**, and **Conditionally Guarantees**.[7] When examining descriptions of purpose generated by designers, verbs such as introduce, control, regulate, maximize, reduce, allow, order, and synchronize occur. The following definitions decompose such verbs into the teleological primitives directly or via previously defined verbs. This permits decomposition of such operators into a predetermined, small set of domain independent primitives.

[7]Recall the fact that these operators can be written in terms of the single operator **Guarantees**.

Descriptions of purpose should be able to express constraints on scenarios with respect to the time domain. For purposes involving temporal relationships, one representation approach for specification predicates is temporal logic, such as those described by Chandra and Misra [CM88], Emerson (CTL*) [ES85], Moszkowski [Mosz85], and Turner [Tur84] (temporal logics of McDermott, Allen, and Halpern, Manna, and Moszkowski). We give thirteen definitions limited to expressions involving scenarios (σ_i) and the specification predicate `occursIn(`σ_i`,`b`)`.

The first two operators describe the manner in which two scenarios may be related in time. These two operators are not concerned with the magnitude of time intervals other than the distinction between 0, finite, and infinite time.[8] Recall that σ_i are scenarios, d and d' are design instances, δ a design modification such that d' is the design obtained by applying δ to d, and E and E$'$ are the envisionments of d and d', respectively. In these and subsequent teleological descriptions, the specification predicate `occursIn(`σ_i`,`b`)` is abbreviated as σ_i.

1. δ **Orders** $\sigma_1, \sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E} & (\texttt{occursIn}(\sigma_1, b)\ \wedge\ \texttt{occursIn}(\sigma_2, b) \wedge \\ & \quad \neg\texttt{occursIn}([\sigma_1; \sigma_2], b) \\ \text{and} & \\ \forall\ b' \in \mathrm{E}' & (\texttt{occursIn}(\sigma_1, b') \wedge \texttt{occursIn}(\sigma_2, b'))\ \Rightarrow \\ & \quad \texttt{occursIn}([\sigma_1; \sigma_2], b'). \end{cases}$$

This can be written in terms of primitives as

δ **Conditionally** when $\{\sigma_1, \sigma_2\}$ **Guarantees** $[\sigma_1; \sigma_2]$.

[8] The particular operators defined here are not derived from any particular temporal logic, and are merely hypothesized as useful in constructing teleological descriptions. For example, Hamscher [Ham91] identifies *synchronize* as an interesting abstraction in describing the behavior of electronic circuits.

2. δ **Synchronizes** $\sigma_1, \sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E} & (\texttt{occursIn}(\sigma_1, b)\ \wedge\ \texttt{occursIn}(\sigma_2, b)) \wedge \\ & \neg\texttt{occursIn}([\sigma_1 \parallel \sigma_2], b) \\ \text{and} & \\ \forall\ b' \in \mathrm{E}' & (\texttt{occursIn}(\sigma_1, b') \wedge \texttt{occursIn}(\sigma_2, b'))\ \Rightarrow \\ & \texttt{occursIn}([\sigma_1 \parallel \sigma_2], b'). \end{cases}$$

This can be written in terms of primitives as

δ **Conditionally** when $\{\sigma_1, \sigma_2\}$ **Guarantees** $[\sigma_1 \parallel \sigma_2]$.

Expressed with the modal operators,

1. δ **Orders** σ_1,$\sigma_2 \Leftrightarrow$

$$\begin{cases} \mathcal{M} \text{ is a model for} \\ \quad \Diamond(\texttt{occursIn}(\sigma_1, b) \wedge \texttt{occursIn}(\sigma_2, b) \wedge \\ \quad\quad \neg\texttt{occursIn}([\sigma_1; \sigma_2], b)) \\ \text{and} \\ \mathcal{M}' \text{ is a model for} \\ \quad \Box((\texttt{occursIn}(\sigma_1, b') \wedge \texttt{occursIn}(\sigma_2, b'))\ \Rightarrow \\ \quad\quad \texttt{occursIn}([\sigma_1; \sigma_2], b')). \end{cases}$$

2. δ **Synchronizes** σ_1,$\sigma_2 \Leftrightarrow$

$$\begin{cases} \mathcal{M} \text{ is a model for} \\ \quad \Diamond(\texttt{occursIn}(\sigma_1, b) \wedge \texttt{occursIn}(\sigma_2, b) \wedge \\ \quad\quad \neg\texttt{occursIn}([\sigma_1 \parallel \sigma_2], b)) \\ \text{and} \\ \mathcal{M}' \text{ is a model for} \\ \quad \Box((\texttt{occursIn}(\sigma_1, b') \wedge \texttt{occursIn}(\sigma_2, b'))\ \Rightarrow \\ \quad\quad \texttt{occursIn}([\sigma_1 \parallel \sigma_2], b')). \end{cases}$$

The next four composed operators involve time, specifically the reduction or increase of the time between the occurrence of states in behaviors, and guarantees of minimum or maximum values for time intervals between

the occurrence of states in behaviors. The Boolean expression of a scenario expresses the temporal constraint. In the next four examples,

$$\sigma_1 \;=\; (\langle p_1, \ldots, p_n \rangle, \beta_1),$$

$$\sigma_2 \;=\; (\langle q_1, \ldots \rangle, \beta_2)$$

$$b = \langle s_1, s_2, \ldots \rangle$$

The variable t has value equal to the time at which the state occurred.

3. δ **Guarantees Minimum Latency (n) Between** $\sigma_1, \sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E} & (\texttt{occursIn}([\sigma_1;\sigma_2], b) \;\wedge\; \mathcal{F}(p_n) = s_i \;\wedge\; \mathcal{F}(q_1) = s_j) \;\wedge \\ & \quad (s_j(t) - s_i(t)) < \mathbf{n} \\ \text{and} & \\ \forall\, b' \in \mathrm{E}' & (\texttt{occursIn}([\sigma_1;\sigma_2], b') \;\wedge\; \mathcal{F}'(p_n) = s_i \;\wedge\; \mathcal{F}'(q_1) = s_j) \;\Rightarrow \\ & \quad (s_j(t) - s_i(t)) \geq \mathbf{n}. \end{cases}$$

This can be written in primitives as

δ **Conditionally** when $[\sigma_1;\sigma_2]$ **Guarantees** $(\langle p_n, q_1 \rangle, (q_1(t) - p_n(t)) \geq \mathbf{n})$.

4. δ **Guarantees Maximum Latency (n) Between** $\sigma_1, \sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E} & (\texttt{occursIn}([\sigma_1;\sigma_2], b) \;\wedge\; \mathcal{F}(p_n) = s_i \;\wedge\; \mathcal{F}(q_1) = s_j) \;\wedge \\ & \quad (s_j(t) - s_i(t)) > \mathbf{n} \\ \text{and} & \\ \forall\, b' \in \mathrm{E}' & (\texttt{occursIn}([\sigma_1;\sigma_2], b') \;\wedge\; \mathcal{F}'(p_n) = s_i \;\wedge\; \mathcal{F}'(q_1) = s_j) \;\Rightarrow \\ & \quad (s_j(t) - s_i(t)) \leq \mathbf{n}. \end{cases}$$

This can be written in primitives as

δ **Conditionally** when $[\sigma_1;\sigma_2]$ **Guarantees** $(\langle p_n, q_1 \rangle, (q_1(t) - p_n(t)) < \mathbf{n})$.

5. δ **Guarantees Minimum Duration (n) For** $\sigma_1 \;\Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E} & (\texttt{occursIn}(\sigma_1, b) \;\wedge\; \mathcal{F}(p_1) = s_i \;\wedge\; \mathcal{F}(p_n) = s_j) \;\wedge \\ & \quad (s_j(t) - s_i(t)) < \mathbf{n} \\ \text{and} & \\ \forall\, b' \in \mathrm{E}' & (\texttt{occursIn}(\sigma_1, b') \;\wedge\; \mathcal{F}'(p_1) = s_i \;\wedge\; \mathcal{F}'(p_n) = s_j) \;\Rightarrow \\ & \quad (s_j(t) - s_i(t)) \geq \mathbf{n}. \end{cases}$$

This can be written in primitives as

δ **Conditionally** when σ_1 **Guarantees** $(\langle p_1, p_n \rangle, (p_n(t) - p_1(t)) \geq \mathbf{n})$.

6. δ **Guarantees Maximum Duration (n) For** $\sigma_1 \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E} & (\texttt{occursIn}(\sigma_1, b)\ \wedge\ \mathcal{F}(p_1) = s_i\ \wedge\ \mathcal{F}(p_n) = s_j) \wedge \\ & \quad (s_j(t) - s_i(t)) > \mathbf{n} \\ \text{and} & \\ \forall\ b' \in \mathrm{E}' & (\texttt{occursIn}(\sigma_1, b')\ \wedge\ \mathcal{F}'(p_1) = s_i\ \wedge\ \mathcal{F}'(p_n) = s_j)\ \Rightarrow \\ & \quad (s_j(t) - s_i(t)) \leq \mathbf{n}. \end{cases}$$

This can be written in primitives as

δ **Conditionally** when σ_1 **Guarantees** $(\langle p_1, p_n \rangle, (p_n(t) - p_1(t)) < \mathbf{n})$.

The remaining operator definitions make statements about scenarios over time, such as maintaining a scenario, or guaranteeing that a scenario will occur exactly once or infinitely many times. In the definition of operator **Maintains**, the specification predicate `occursIn` is generalized to apply to single states as well as sequences of states (behaviors).

7. δ **Maintains** $\sigma \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E},\ \exists\ s \in b\ \ \neg\texttt{occursIn}(\sigma, s), \\ \text{and} \\ \forall\ b' \in \mathrm{E}',\ \forall\ s' \in b'\ \texttt{occursIn}(\sigma, s'). \end{cases}$$

This can be written in primitives as

$$\delta\ \textbf{Guarantees}\ \forall\ b_i \in b, \texttt{occursIn}(\sigma, b_i)$$

The **Maintains** operator can be used to express purposes such as regulate or control, namely *maintaining* some condition such as a temperature

between some specified minimum and maximum. Such a scenario would look like

$$(\langle\{(temp, (\texttt{minTemp}, \texttt{maxTemp}))\}\rangle, \texttt{true}).$$

The next two operators express the purpose that once a particular event or event sequence occurs, then some other event or event sequence is guaranteed to occur or prevented from occurring. For example, if some mechanism parameter goes beyond a specified range, then it is brought back into range, possibly within some time constraint.

8. δ **Subsequently** (in σ_1) **Guarantees** $\sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E}\ \ \texttt{occursIn}(\sigma_1, b)\ \wedge\ \neg\texttt{occursIn}([\sigma_1; \sigma_2], b) \\ \text{and} \\ \forall\ b' \in \mathrm{E}'\ \ \texttt{occursIn}(\sigma_1, b')\ \Rightarrow \texttt{occursIn}([\sigma_1; \sigma_2], b'). \end{cases}$$

This can be written in primitives as

$$\delta\ \textbf{Conditionally}\ \text{when}\ \sigma_1\ \textbf{Guarantees}\ [\sigma_1; \sigma_2].$$

9. δ **Subsequently** (in σ_1) **Prevents** $\sigma_2 \Leftrightarrow$

$$\begin{cases} \exists\ b \in \mathrm{E}\ \ \texttt{occursIn}(\sigma_1, b)\ \wedge\ \texttt{occursIn}([\sigma_1; \sigma_2], b) \\ \text{and} \\ \forall\ b' \in \mathrm{E}'\ \ \texttt{occursIn}(\sigma_1, b')\ \Rightarrow \neg\texttt{occursIn}([\sigma_1; \sigma_2], b'). \end{cases}$$

This can be written in primitives as

$$\delta\ \textbf{Prevents}\ [\sigma_1; \sigma_2].$$

The next three operators express the purpose that a particular event or event sequence occurs exactly once (**Guarantees Single Occurrence**), occurs infinitely many times if it occurs at all (**Guarantees Recurrence**), or always occurs infinitely many times (**Guarantees Recurring**). For example, a mechanism may require a periodic signal for synchronizing events or resetting itself.

10. δ **Guarantees Single Occurrence** $\sigma \Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E}\ \neg\texttt{occursIn}(\sigma, b)\ \vee\ \texttt{occursIn}([\sigma;\sigma], b) \\ \text{and} \\ \forall\, b' \in \mathrm{E}'\ \texttt{occursIn}(\sigma, b')\ \wedge\ \neg\texttt{occursIn}([\sigma;\sigma], b'). \end{cases}$$

This can be written in primitives as

$$\delta \textbf{ Guarantees } \sigma\ \wedge\ \textbf{Subsequently} \text{ when } \sigma \textbf{ Prevents } \sigma$$

or as

$$\delta \textbf{ Guarantees } \sigma\ \wedge\ \textbf{Prevents } [\sigma;\sigma].$$

In the following two definitions, behaviors b and b' are written as $\langle s_0, s_1, \ldots \rangle$ and $\langle s'_0, s'_1, \ldots \rangle$ respectively.

11. δ **Guarantees Recurrence** $\sigma \Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E}\ \texttt{occursIn}(\sigma, b)\ \wedge\ \exists\, i \geq 0\ \neg\texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle) \\ \text{and} \\ \forall\, b' \in \mathrm{E}'\ \texttt{occursIn}(\sigma, b')\ \Rightarrow\ \forall\, i \geq 0\ \texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle). \end{cases}$$

This can be written in primitives as

$$\delta \textbf{ Subsequently} \text{ when } \sigma \textbf{ Guarantees } \forall\, i \geq 0\ \texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle).$$

12. δ **Guarantees Recurring** $\sigma \Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E}\ \exists\, i \geq 0\ \neg\texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle) \\ \text{and} \\ \forall\, b' \in \mathrm{E}'\ \forall\, i \geq 0\ \texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle). \end{cases}$$

This can be rewritten as

$$\delta \textbf{ Guarantees Recurrence } \sigma\ \wedge\ \delta \textbf{ Guarantees } \sigma,$$

or as

$$\delta \textbf{ Guarantees } \forall\, i \geq 0\ \texttt{occursIn}(\sigma, \langle s_i, \ldots \rangle).$$

The last operator expresses the purpose that a particular event or event sequence (σ_2) should always be preceded by some other event or event sequence (σ_1). This predecessor event or event sequence becomes a necessary condition.

13. δ **Guarantees** σ_2 **Requires** σ_1 $\Leftrightarrow$

$$\begin{cases} \exists\, b \in \mathrm{E} \;\; \texttt{occursIn}(\sigma_2, b) \;\wedge\; \neg\texttt{occursIn}([\sigma_1;\sigma_2], b) \\ \text{and} \\ \forall\, b' \in \mathrm{E}' \;\; \texttt{occursIn}(\sigma_2, b') \;\Rightarrow \texttt{occursIn}([\sigma_1;\sigma_2], b'). \end{cases}$$

This can be written in primitives as

δ **Conditionally** when σ_2 **Guarantees** $[\sigma_1;\sigma_2]$.

Chapter 5

Language Properties

5.1 Generalization and Specialization

In this chapter we investigate some properties of teleological descriptions with respect to design specification generalization and scenario generalization. These properties will be of interest when constructing an index of designs and design modifications for reuse, when navigating this index, and when selecting designs and design modifications for reuse.

For example, it will often be the case that the exact specification a designer is addresing does not appear in the database of teleological descriptions. A design reuse implementation can identify teleological descriptions referencing specifications "similar" to the desired specification and propose these "similar" teleological descriptions to the designer. In particular, we exploit the logical relationship $\phi_1 \Rightarrow \phi_2$ between specification predicates ϕ_1 and ϕ_2. To modify a design to prevent specification predicate ϕ_1 from holding, a designer might directly apply a modification that prevents the weaker specification predicate ϕ_2 or augment the modification to prevent ϕ_1. The propositions described in this chapter demonstrate how the TeD behavior and teleology languages and the behavior abstraction relation $\sqsubseteq_\sigma$ support reuse via "similar" specifications.

5.1.1 Generalizing a Guarantee

A design modification that guarantees a specification predicate also guarantees any generalization (i.e. weaker predicate) if the generalized predi-

cate is false in at least one possible world of the unmodified design.

Proposition 5.1 *For specification predicates ϕ_1, ϕ_2, design modification δ, if*

- $\phi_1 \Rightarrow \phi_2$,
- δ **Guarantees** ϕ_1, and
- $\exists\ b \in$ E, $\neg\phi_2$,

then δ **Guarantees** *ϕ_2.*
Proof: δ **Guarantees** $\phi_1 \Rightarrow \forall\, b' \in \mathrm{E}', \phi_1$. With $\phi_1 \Rightarrow \phi_2$, we have $\forall\, b' \in \mathrm{E}', \phi_2$. Given that $\exists\ b \in \mathrm{E}$, $\neg\phi_2$, we have δ **Guarantees** ϕ_2. □

5.1.2 Specializing a Prevention

A design modification that prevents a specification predicate also prevents any specialization (i.e. stronger predicate) if the specialized predicate is true in at least one possible world of the unmodified design.

Proposition 5.2 *For specification predicates ϕ_1, ϕ_2, design modification δ, if*

- $\phi_1 \Rightarrow \phi_2$,
- δ **Prevents** ϕ_2, and
- $\exists\ b \in \mathrm{E}, \phi_1$,

then δ **Prevents** *ϕ_1.*
Proof: δ **Prevents** $\phi_2 \Rightarrow \forall\, b' \in \mathrm{E}', \neg\phi_2$. With $\phi_1 \Rightarrow \phi_2$, we have $\forall\, b' \in \mathrm{E}', \neg\phi_1$. Given that $\exists\ b \in \mathrm{E}$, ϕ_1, we have δ **Prevents** ϕ_1. □

5.1.3 Generalizing an Introduction

A design modification that introduces a specification predicate also introduces any generalization (i.e. weaker predicate) if the generalized predicate is false for all possible worlds of the unmodified design.

Proposition 5.3 *For specification predicates ϕ_1, ϕ_2, design modification δ, if*

- $\phi_1 \Rightarrow \phi_2$,
- δ **Introduces** ϕ_1, and
- $\forall\ b \in \mathrm{E}, \neg\phi_2$,

then δ **Introduces** ϕ_2.
Proof: δ **Introduces** $\phi_1 \Rightarrow \exists b' \in \mathrm{E}', \phi_1$. With $\phi_1 \Rightarrow \phi_2$, we have $\exists\, b' \in \mathrm{E}', \phi_2$. Given that $\forall\, b \in \mathrm{E}, \neg\phi_2$, we have δ **Introduces** ϕ_2. □

5.1.4 Specializing a Conditional

A design modification that conditionally guarantees (prevents) a specification predicate also guarantees (prevents) the specification predicate conditional upon any specialization (i.e. stronger predicate) of the condition, given that the guaranteed (prevented) predicate is false in at least one possible world of the unmodified design in which the specialized condition is true.

Proposition 5.4 *For specification predicates* ϕ, ϕ_1, ϕ_2, *design modification* δ, *if*

- $\phi_1 \Rightarrow \phi_2$,
- δ **Conditionally** when ϕ_2 **Guarantees** ϕ, and
- $\exists\, b \in \mathrm{E}, \phi_1 \ \wedge\ \neg\phi$,

then δ **Conditionally** (in ϕ_1) **Guarantees** ϕ.
Proof: δ **Guarantees** $(\phi_2 \Rightarrow \phi) \Rightarrow \forall\, b' \in \mathrm{E}', \phi_2 \Rightarrow \phi$. With $\phi_1 \Rightarrow \phi_2$, we have $\forall\, b' \in \mathrm{E}', \phi_1 \Rightarrow \phi$. Given that $\exists\, b \in \mathrm{E}, \phi_1 \wedge \neg\phi$, we have δ **Guarantees** $\phi_1 \Rightarrow \phi$, and δ **Conditionally** (in ϕ_1) **Guarantees** ϕ. □

5.2 Generalizing Behavior Specifications

We prove an implication from scenario generalization (abstraction) to the `occursIn` specification predicate, and then use this implication with the previous propositions of the chapter.

Proposition 5.5 *If* $\sigma_1 \sqsubseteq_\sigma \sigma_2$, *then* $\texttt{occursIn}(\sigma_1, b) \Rightarrow \texttt{occursIn}(\sigma_2, b)$.
Proof: For behavior (possible world) b, consider scenario $(b, \texttt{true})$. From $\texttt{occursIn}(\sigma_1,b)$, we have $(b, \texttt{true}) \sqsubseteq_\sigma \sigma_1$. With $\sigma_1 \sqsubseteq_\sigma \sigma_2$ and Lemma 4.25, we have $(b, \texttt{true}) \sqsubseteq_\sigma \sigma_2$. Now, $\texttt{occursIn}(\sigma_2,b)$. □

Proposition 5.6 *For scenarios σ_1 and σ_2 and design modification δ, if*

- $\sigma_1 \sqsubseteq_\sigma \sigma_2$,
- δ **Guarantees** $\texttt{occursIn}(\sigma_1, b)$, and
- $\exists\ b \in \mathrm{E}, \neg\texttt{occursIn}(\sigma_2, b)$,

then δ **Guarantees** $\texttt{occursIn}(\sigma_2, b)$.
Proof: From Proposition 5.1 and Proposition 5.5. □

Proposition 5.7 *For scenarios σ_1 and σ_2 and design modification δ, if*

- $\sigma_1 \sqsubseteq_\sigma \sigma_2$,
- δ **Prevents** $\texttt{occursIn}(\sigma_2, b)$, and
- $\exists\ b \in \mathrm{E}, \texttt{occursIn}(\sigma_1, b)$,

then δ **Prevents** $\texttt{occursIn}(\sigma_1, b)$.
Proof: From Proposition 5.2 and Proposition 5.5. □

Proposition 5.8 *For scenarios σ_1 and σ_2 and design modification δ, if*

- $\sigma_1 \sqsubseteq_\sigma \sigma_2$,
- δ **Introduces** $\texttt{occursIn}(\sigma_1, b)$, and
- $\forall\ b \in \mathrm{E}, \neg\texttt{occursIn}(\sigma_2, b)$,

then δ **Introduces** $\texttt{occursIn}(\sigma_2, b)$.
Proof: From Proposition 5.3 and Proposition 5.5. □

Proposition 5.9 *For scenarios σ_1, σ_2, σ and design modification δ, if*

- $\sigma_1 \sqsubseteq_\sigma \sigma_2$,
- δ **Conditionally** (in σ_2) **Guarantees** $\texttt{occursIn}(\sigma, b)$, and
- $\exists\ b \in \mathrm{E},\ \texttt{occursIn}(\sigma_1, b)\ \wedge\ \neg\texttt{occursIn}(\sigma, b)$,

then δ **Conditionally** when σ_1 **Guarantees** $\texttt{occursIn}(\sigma, b)$.
Proof: From Proposition 5.4 and Proposition 5.5. □

Chapter 6

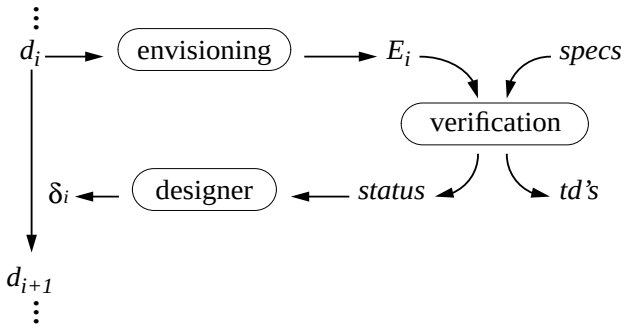
Examples

6.1 Design Examples

In this chapter we examine two detailed design examples to demonstrate the teleology language and behavior abstraction discussed in Chapters 3 and 4. These examples demonstrate acquisition of teleological descriptions in the design process. We use these examples and the acquired teleological descriptions in subsequent discussions of explanation, design reuse, and diagnosis. Acquisition issues are discussed in Chapter 9, and indexing teleological descriptions for explanation, design reuse, and diagnosis is discussed in Chapter 8. The two examples explored here are

- Input selection logic from a CMOS arithmetic logic unit (ALU) design [Ray86].
- An electric motor design [KTY91].

Both examples start with initial designs and specifications characterizing aspects of the desired behavior. The initial designs are found to be inconsistent with the design specifications, and a series of design modifications are made to make the designs meet their respective specifications. The teleological descriptions associated with these modifications are developed for each set of modifications. A diagram describing this design flow is given in Figure 6.1. For each design d_i we give a schematic, a CC model, and the initial values used



d_i - i^{th} version of the design
E_i - envisionment for design d_i
$specs$ - design specifications
$td's$ - teleological descriptions captured in verification
$status$ - results of verification
δ_i - modifications generated by the designer

Figure 6.1: Design Process Flow (Single Step)

in envisioning. For each envisionment E_i we give a prose description of the behaviors, a graphical representation of the envisionment (the QSIM behavior tree), and some representative qualitative plots generated by QSIM. For both examples we give the design specifications and describe the verification results for each design d_i. Design modifications, δ_i, generated by the designer are given in the structure modification language described in Section 3.3. In these examples, envisioning is performed with QSIM, and verification (determining whether specifications are satisfied, and acquiring and classifying teleological descriptions) is performed by code implemented by the author. Design modifications are hand generated.

6.2 Circuit Example

Consider the input selection circuit in Figure 6.2, extracted from a CMOS arithmetic logic unit (ALU) design. The circuit contains a pass tran-

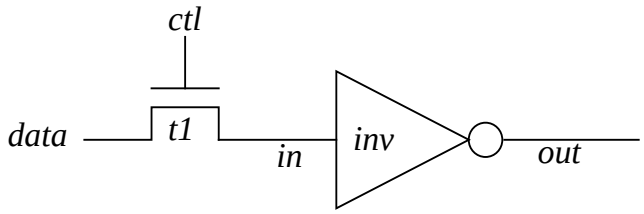


Figure 6.2: CMOS Input Selection Circuit (`ISC1`)- Schematic

sistor $t1$ and an inverter inv. In the ALU design, the signal ctl determines (controls) whether the signal $data$ is passed into the logic portion of the ALU. The desired behavior of this circuit in terms of signals $data$, ctl, in, and out is:

> When the value of signal ctl is HIGH, the value of signal $data$ is transmitted to signal in (i.e., they are electrically connected). This value is then inverted by inv (HIGH $\rightarrow$ LOW or LOW $\rightarrow$ HIGH), and the inverted value then becomes the value of signal out.

The (logic) values of HIGH and LOW are landmarks of the quantity space in which the parameters $data$, ctl, in, and out range. These landmarks represent the desired values for signals in the circuit when no signal transitions are occurring. The CC quantity space for voltages in this circuit is shown in Figure 6.3. HIGH is represented by landmark `Vhi` and LOW is represented by landmark `0`.

Structure

The top level of the hierarchical structural description (in structure language CC) of the input selection circuit is shown in Figure 6.4. The complete structural description is expressed in several levels of hierarchy, and is given in Appendix B.

```
(define-quantity-space MOS-voltage-qspace
  (Vhi- Vhi-Vtn Vtp 0 Vtn VhiVtp Vhi)
  (conservation-correspondences
    (Vhi- Vhi) (Vhi-Vtn VhiVtp) (Vtp Vtn)))
```

Figure 6.3: Circuit Model Voltage Quantity Space (in CC)

```
(define-component-interface
  ISC "Input select circuit" electrical
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current base-quantity-space))
   (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  1 ISC
  "N-trans for input select, capacitor for output load."
  (components
    (RV1 reference-voltage)
    (RV2 reference-voltage)
    (RV3 reference-voltage (ignore-qdir i))
    (t1  (MOS-transistor (impl N-channel-bidirectional))
         (display Ids))
    (inv Inverter)
    (C   (capacitor (impl current-qspace)) (ignore-qdir i i2)))
  (connections
    (data (RV1 t)   (t1 d))
    (ctl  (RV2 t)   (t1 g))
    (in   (t1 s)    (inv in))
    (out  (inv out) (C t1))
    (w1   (RV3 t)   (c t2))))
```

Figure 6.4: Input Selection Circuit (`ISC1`) - CC Model (Top Level of Hierarchy)

```
(for-component X (inverter MOS-transistor)
    (prohibited (((((X Vg) ((0 Vhi) std)))) true)))
```

Figure 6.5: Input Selection Circuit - Design Specification

Design Specifications

A general domain specification for CMOS design is: "signals should take on an intermediate value between `0` and `Vhi` only during a transition from `0` to `Vhi` or `Vhi` to `0`". In particular, a logic component such as the inverter should not have an input signal with an intermediate value between `0` and `Vhi` and unchanging. The rationale for this design rule comes from the operating characteristics of the CMOS inverter implementation, in which current flows when the input has value between `0` and `Vhi` but does not flow when the input has value either `0` or `Vhi`.[1] Hence, CMOS circuits consume power only when switching, as opposed to other implementation technologies such as nMOS that consume power during signal transition and at other times as well. We express this general domain specification in Figure 6.5, which says that for transistor components inside inverters, the scenario in which the gate voltage (`Vg`) of the transistor is in the interval `(0 Vhi)` and steady is prohibited.

Behavior

We first examine the behavior of the circuit when the signal *in* has value `0`, signal *out* has the value `Vhi`, signal *ctl* has the value `Vhi`, and the signal *data* has just assumed the value `Vhi`. The desired behavior is: "value `Vhi` is transmitted to *in*, and the inverter *inv* changes *out* to `0`".

[1]The actual value at which the inverter stops drawing current is determined by a threshold value set by the manufacturing process used to produce the circuit.

data	(ISC RV1 V)	= (Vhi std)
ctl	(ISC RV2 V)	= (Vhi std)
in	(ISC t1 Vs)	= (0 nil)
out	(ISC C V1)	= (Vhi nil)
	(ISC RV3 V)	= (0 std)
	(ISC C Q)	= (Q* nil)
	(ISC C C)	= (C* std)
	(ISC inv Vdd V)	= (Vhi std)
	(ISC inv Vss V)	= (0 std)
	(ISC inv Nt Cg)	= (Cg* std)
	(ISC inv Nt Qg)	= (0 nil)
	(ISC inv PT Cg)	= (Cg* std)
	(ISC inv PT Qg)	= (0 nil)
	(ISC t1 Cg)	= (Cg* std)
	(ISC t1 Qg)	= (Qg* std)

Figure 6.6: Initial Variable Values

The operating characteristics of $t1$ (an n-channel MOS transistor) are such that when the signal *data* has value `Vhi` and the signal *ctl* has the value `Vhi`, the value transmitted to signal *in* is `Vhi` minus the threshold value (> 0) of $t1$ [MC80]. Landmark `VhiVtp` represents the value `Vhi` minus the threshold value of $t1$. The landmark `VhiVtp` is between the landmark `0` and `Vhi`, and hence not a desired value for signal *in*.

This operating characteristic is captured in the CC description of the n-channel MOS transistor, and the associated behavior is demonstrated in the envisionment of the `ISC` model generated by QSIM from initial conditions shown in Figure 6.6. The envisionment predicts six qualitatively unique behaviors, shown in the behavior tree in Figure 6.7. The qualitative plot of variable `(ISC t1 Vs)` in behavior 6 is shown in Figure 6.8. This variable represents the voltage at terminal `s` of transistor `t1` and at terminal `in` of inverter `inv`. The qualitative plot in Figure 6.8 shows a feature of all of the behaviors when signal

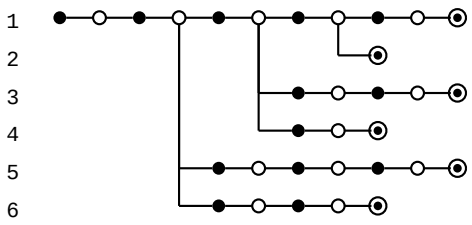


Figure 6.7: Behavior Tree of Initial Circuit (ISC1)

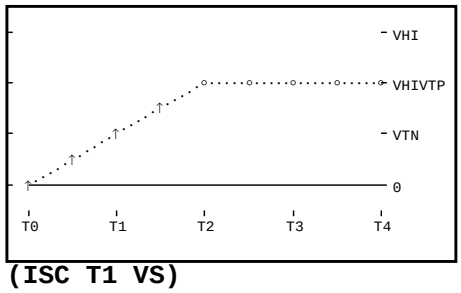


Figure 6.8: Qualitative Plot for Initial Circuit (ISC1)

values stabilize in the circuit, namely that signal (ISC t1 Vs) has the value (VhiVtp_std).

6.2.1 Evaluation 1

The model checking algorithm[2] implemented in this work determines that no behaviors satisfy the specification of Figure 6.5. The designer's task is to modify the design either structurally or via changes in parameter values (landmarks) to bring the behaviors in line with the specification.

[2]Based on the behavior abstraction relation $\sqsubseteq_\sigma$ given in Chapter 4.

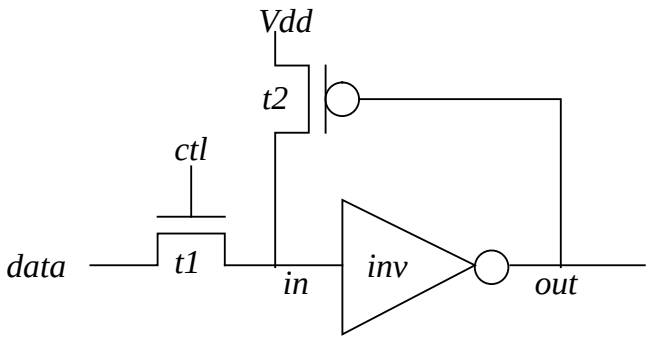


Figure 6.9: Circuit with Feedback Transistor (`ISC2`) - Schematic

Structure Modification

The addition of the feedback transistor $t2$, a p-channel MOS transistor (see the schematic in Figure 6.9) modifies the circuit behavior in the following way (in terms of signals *in* and *out*):

> As signal *in* transitions from `0` to `Vhi`, signal *out* transitions from `Vhi` to `0`. As signal *out* moves away from `Vhi` and towards `0`, transistor $t2$ electrically connects *in* with Vdd, enabling a current flow from Vdd to *in* which in turn increases the value of *in* to that of Vdd (`Vhi`).

The operating characteristics of a p-channel transistor are such that the value `Vhi` can be transmitted without degradation (i.e., not subject to any threshold value). Consequently, the designer's modification, the addition of $t2$, prevents the scenario in which *in* reaches a value less than `Vhi` and remains steady. This design modification is automatically captured in the design modification language (see Figure 6.10) and added to the circuit's design history. The modification produces the new structure description shown in Figure 6.11.

```
(for-component ISC
  (create-new-implementation 1 2))

(for-component (ISC (impl 2))
  (remove-connection (C t1))
  (add-component t2 p-channel-feedback () (in (t2 s)))
  (add-component S (split (impl equipotential-current-space))
                   ((ignore-qdir I I1 I2) (display V)
                    (no-new-landmarks I I1 I2))
                   (out (S m))
                   (w2  (S s1) (t2 g))
                   (w3  (S s2) (C t1))))
```

Figure 6.10: Design Modification (δ_1) Adding Feedback Transistor

Modified Behavior

The envisionment of the modified design characterizes 22 qualitatively distinct behaviors, shown in Figure 6.12, with all behaviors having a final, quiescent state in which the signal *in* is `(Vhi std)`, the desired result. This would seem to meet the design specification of prohibiting a steady but intermediate value for signal *in*. However, the model checking algorithm identifies a behavior (see Figure 6.13) in which signal *in* (variable `(ISC t1 Vs)`) takes on an intermediate steady value during the transition from `0` to `Vhi`, and then moves on to value `(Vhi std)`. This behavior is legal under the constraints imposed by the qualitative model of the circuit, and represents incomplete knowledge of the specific capacitance values for the capacitors in the circuit.[3] We modify the specification as shown in Figure 6.14 to state the condition "for transistor components inside inverters, the scenario in which the gate voltage (`Vg`) of

[3]Capacitance for "wires" in the circuit are represented on the transistor gates, and do not appear explicitly as capacitor components.

```
(define-component-implementation
  2 ISC
  "N-trans for input select, P-trans for feedback."
  (components
    (RV1  reference-voltage)
    (RV2  reference-voltage)
    (RV3  reference-voltage (ignore-qdir I))
    (t1   (MOS-transistor (impl N-channel-bidirectional))
          (initable Qg Vs) (display Ids Vs))
    (t2   P-channel-feedback)
    (inv    Inverter)
    (C    (Capacitor (impl current-qspace)) (ignore-qdir i i2))
    (S    (Split (impl equipotential-current-qspace))
          (ignore-qdir I I1 I2) (display V)
          (no-new-landmarks I I1 I2)))
  (connections
    (data (RV1 t)   (t1 d))
    (ctl  (RV2 t)   (t1 g))
    (in   (t1 s)    (inv in) (t2 s))
    (out  (inv out) (S m))
    (w1   (RV3 t)   (C t2))
    (w2   (S s1)    (t2 g))
    (w3   (S s2)    (C t1))))
```

Figure 6.11: Circuit with Feedback Transistor (`ISC2`) - CC Model

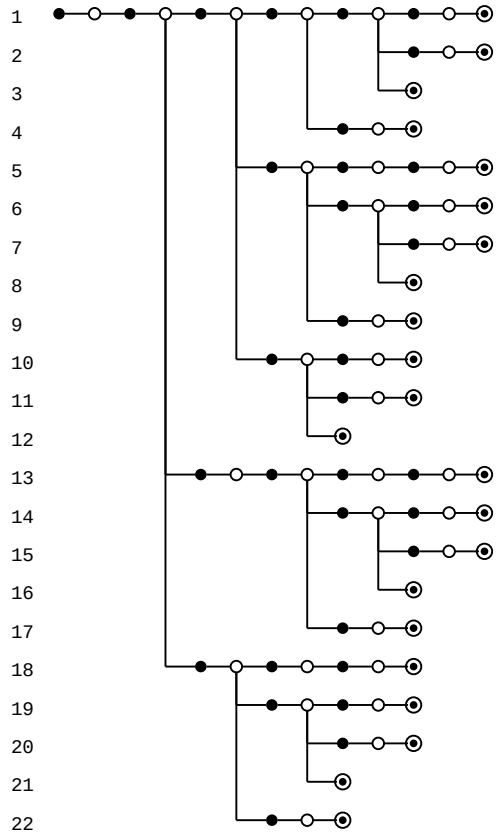


Figure 6.12: Behavior Tree of Circuit with Feedback (ISC2)

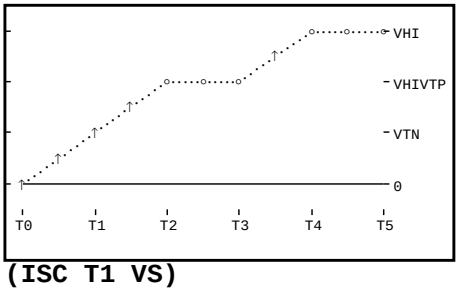


Figure 6.13: Qualitative Plot for Circuit with Feedback (ISC2)

```
(for-component X (inverter MOS-transistor)
  (conditionally ((((((X Vg) (0 std)))
                     (((X Vg) ((0 Vhi) std)))) true))
    (required (((((X Vg) (Vhi std))))
              true))))
```

Figure 6.14: Input Selection Circuit - Design Specification 2

the transistor is in the interval `(0 Vhi)` and steady must be followed by the scenario in which the gate voltage has value `(Vhi std)`". The model checking algorithm confirms that all behaviors meet this specification.

Teleology

The purpose of the design modification that adds $t2$ to the input selection circuit can be expressed in TeD as follows. Let δ_1 represent the design modification of adding $t2$ to the design (Figure 6.10), let σ_1 be the scenario

$$(\langle\{\texttt{((inv MOS-transistor Vg) ((0 Vhi) std))}\}\rangle,\texttt{true})$$

and let σ_2 be the scenario

$$(\langle\{\texttt{((inv MOS-transistor Vg) (Vhi,std))}\}\rangle,\texttt{true}).$$

Then

$$\delta_1 \textbf{ Conditionally } \text{when } \{\sigma_1\} \textbf{ Guarantees } [\sigma_1;\sigma_2].^4 \qquad (6.1)$$

The behavior that electrically connects in to Vdd also addresses another problem that occurs when in has value `Vhi` and ctl transitions from `Vhi`

[4]Recall that the specification predicate `occursIn(`σ`,`b`)` is abbreviated as σ.

```
(for-component X (ISC)
  (for-component Y (X inverter MOS-transistor)
    (conditionally ((((((Y Vg) (Vhi ign))
                       ((X Nt Vg) (0 std)))) true))
      (prohibited (((((Y Vg) ((0 Vhi) std))
                     ((X Nt Vg) (0 std))))
                true)))))
```

Figure 6.15: Input Selection Circuit - Design Specification 3

to `0`. In this situation, *in* is no longer electrically connected to *data*, and becomes a memory element which should preserve its value, `Vhi`. However, in the absence of $t2$, the charge at *in* will dissipate and move the signal value away from the landmark value `Vhi`, resulting in the value of signal *out* changing also (i.e., moving away from `0`). By introducing $t2$, the charge at *in* is maintained at `Vhi`, and hence the behavior in which *in* decreases in value is prevented. The design specification describing the desired behavior is given in Figure 6.15.

The purpose of the design modification that adds $t2$ can be expressed in TeD as follows. Let σ_3 and σ_4 be, respectively, the scenarios

(⟨{((inv MOS-transistor Vg) (Vhi std)),((t1 Vg) (0 std))}⟩,true),

(⟨{((inv MOS-transistor Vg) ((0 Vhi) ign)),((t1 Vg) (0 std))}⟩,true).

Then

$$\delta_1 \textbf{ Conditionally when } \{\sigma_3\} \textbf{ Prevents } \sigma_4. \tag{6.2}$$

The design modification of adding $t2$ to the circuit has been assigned the purpose of *guaranteeing* the behavior that a steady value between `0` and `Vhi` for signal *in* will be followed by the value `Vhi`, The steady, intermediate value will be transitory. Further, from starting conditions where *in* has value

`Vhi` and *ctl* has value `0`, the modification *prevents* signal *in* from changing its value.

6.2.2 Evaluation 2

While the first design modification has addressed problems associated with signal *in* achieving and maintaining value `Vhi`, a new problem has been introduced by the first design modification. When signal *in* has value `Vhi`, signal *data* has value `0`, and signal *ctl* transitions from `0` to `Vhi`, the charge stored at *in* (representing the value `Vhi`) should be drawn off via the connection through $t1$. However, recall that current can flow from Vdd to *in* via the connection provided by $t2$. If current flows from Vdd to *in* at a sufficient rate, an intermediate value will be reached for *in* such that the complementary value at *out* is not high enough to "turn-off" $t2$ (a p-channel transistor is off when the gate voltage is `Vhi`). This behavior, shown in Figure 6.17, occurs in the attainable envisionment, shown in Figure 6.16, generated from the initial conditions shown in Figure 6.18.

The second design modification changes the channel resistance of $t2$ (to a high resistance value) to impede the current flow and hence prevent the scenario in which *in* reaches an equilibrium point between `Vhi` and `0` during the `Vhi` to `0` transition of *in*. A qualitative plot of the desired behavior is shown in Figure 6.19. The design specification describing this desired behavior is given in Figure 6.20.

This purpose of the particular channel resistance value for $t2$ can be expressed in TeD as follows. Let δ_2 represent the design modification of increasing the channel resistance of $t2$, and let σ_5 be the scenario

$$(\langle\{\texttt{((t1 Vg) (Vhi std))},\texttt{((t1 Vd) (0 std))}\}\rangle,\texttt{true}).$$

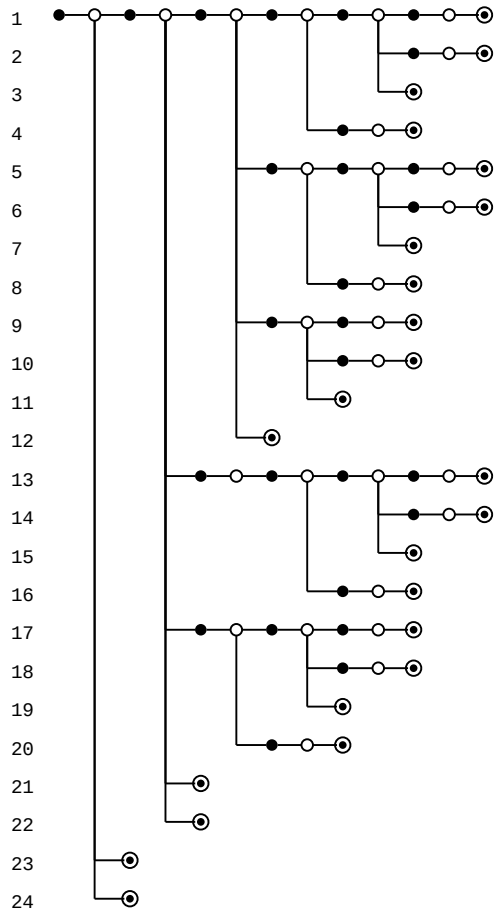


Figure 6.16: Behavior Tree of Circuit with Feedback (ISC2) - Discharging

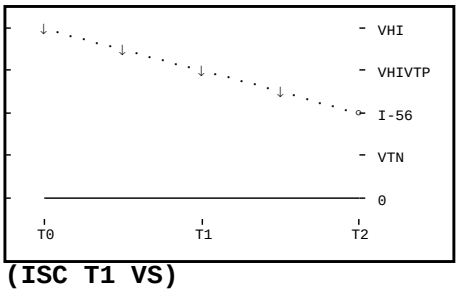


Figure 6.17: Qualitative Plot for Circuit with Feedback (ISC2) - Discharging

```
data      (ISC RV1 V)  = (0 std)
ctl       (ISC RV2 V)  = (Vhi std)
in        (ISC t1 Vs)  = (Vhi nil)
Vdd   (ISC t2 Vdd V)  = (Vhi std)
       (ISC t2 Pt Vg)  = (0 nil)
      (ISC inv Vdd V)  = (Vhi std)
      (ISC inv Vss V)  = (0 std)
      (ISC inv Nt Cg)  = (Cg* std)
      (ISC inv Nt Qg)  = (Qg* nil)
      (ISC inv Pt Cg)  = (Cg* std)
      (ISC inv Pt Qg)  = (Qg* nil)
          (ISC t1 Cg)  = (Cg* std)
          (ISC t1 Qg)  = (Qg* std)
       (ISC t2 Pt Qg)  = (0 nil)
       (ISC t2 Pt Cg)  = (Cg* std)
out        (ISC C V1)  = (0 nil)
          (ISC RV3 V)  = (0 std)
            (ISC C Q)  = (0 nil)
            (ISC C C)  = (C* std)
```

Figure 6.18: Initial Variable Values - Vhi to 0 Transition

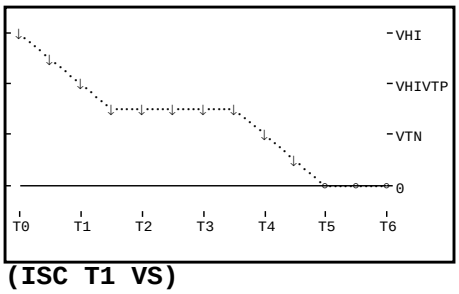


Figure 6.19: Qualitative Plot for Circuit with High Resistance Feedback

```
(for-component X (ISC)
  (for-component Y (X inverter MOS-transistor)
    (conditionally ((((((X Nt Vg) (Vhi std))
                       ((X Nt Vd) (0 std)))) true))
      (required (((((Y Vg) (0 std))))
                 true)))))
```

Figure 6.20: Input Selection Circuit - Design Specification 3

and let σ_6 be the scenario

$$(\langle\{\texttt{((inv MOS-transistor Vg) (0 std))}\}\rangle,\texttt{true}).$$

Then

$$\delta_2 \textbf{ Conditionally when } \{\sigma_5\} \textbf{ Guarantees } \sigma_6. \tag{6.3}$$

6.2.3 Modification Teleology Summary

To summarize this example, $t2$ was added 1) to prevent the scenario in which *in* reaches a steady value between `0` and `Vhi` when transitioning from `0` to `Vhi`, and 2) to prevent the scenario in which the value of *in* decreases from `Vhi` when *in* is acting as a memory element storing the value `Vhi`. The channel-resistance of $t2$ was set high to prevent the scenario in which *in* reaches an equilibrium value between `0` and `Vhi` during the transition from `Vhi` to `0`. The input selection circuit design history, evaluation steps, and the acquired teleological description are shown in the context of the design process flow in Figure 6.21.

[5]Design ISC1 is described in Figures 6.2 and 6.4. Envisionment E_1 is described in Figures 6.7 and 6.8. Design modification δ_1 is described in Figure 6.10. Design ISC2 is described in Figure 6.9 and 6.11. Envisionment E_2 is described in Figures 6.13 and 6.12. Design ISC3 is described in Section 6.2.2. Envisionment E_3 is described in Figures 6.19. The acquired teleological descriptions are equations 6.1, 6.2, and 6.3

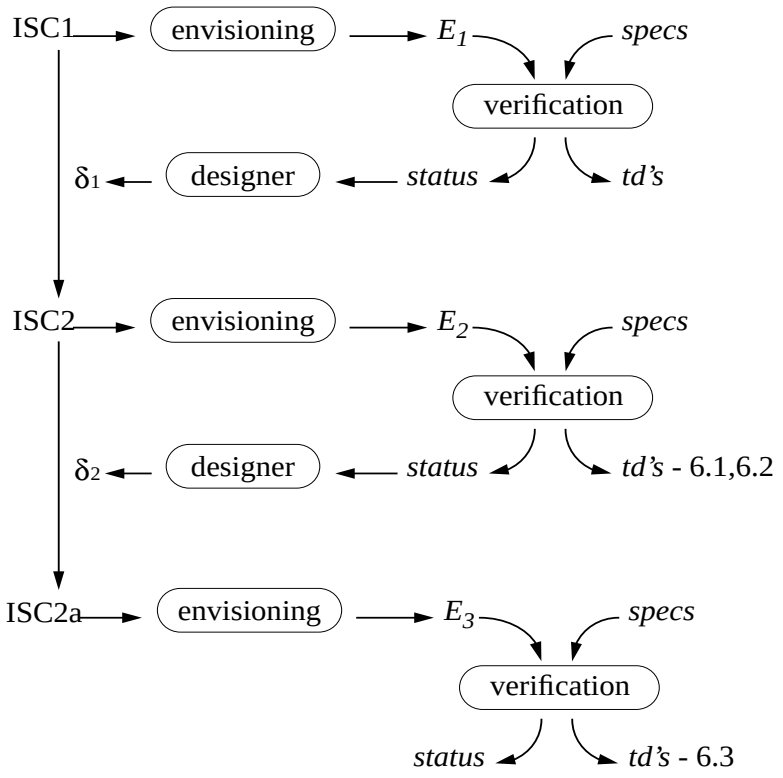


Figure 6.21: Design Flow for the Input Selection Circuit[5]

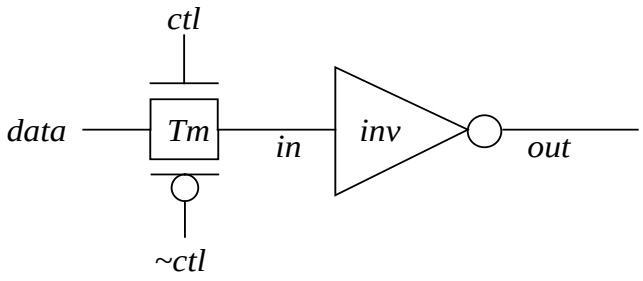


Figure 6.22: Circuit with Transmission Gate - Schematic

```
(for-component ISC
  (create-new-implementation 1 3))

(for-component (ISC (impl 3))
  (remove-component t1)
  (add-component RV4 reference-voltage ((ignore-qdir I)))
  (add-component Tm transmission-gate ((display Isd Vsd))
                   (data (Tm in))
                   (ctl  (Tm ctl))
                   (in   (Tm out)))
  (add-connection (w2 (RV4 t) (Tm ctl-bar))))
```

Figure 6.23: Design Modifications to Replace Pass Transistor

6.2.4 Alternate Design History

Given the original input selection circuit design, an alternative design modification can be made to address the initial problem observed by the designer. Consider the circuit schematic in Figure 6.22, in which the pass transistor $t1$ has been replaced by a CMOS transmission gate. The transmission gate has operating characteristics such that it can transmit both `Vhi` and `0` values without degradation with respect to voltage. The design modifications that add the transmission gate are shown in Figure 6.23, with the resulting structure description shown in Figure 6.24.

```
(define-component-implementation
  3 ISC
  "Transmission-gate for input selection."
  (components (RV1 reference-voltage)
              (RV2 reference-voltage)
              (RV3 reference-voltage (ignore-qdir I))
              (RV4 reference-voltage (ignore-qdir I))
              (Tm  transmission-gate (display Isd Vsd))
              (I   Inverter)
              (C   (Capacitor (impl base-qspace))
                   (ignore-qdir i i2)))
  (connections (data (RV1 t)  (Tm in))
               (ctl  (RV2 t)  (Tm ctl))
               (in   (Tm out) (I in))
               (out  (I out)  (C t1))
               (w1   (C t2)   (RV3 t))
               (w2   (RV4 t)  (Tm ctl-bar))))
```

Figure 6.24: Circuit with Transmission Gate - CC Model

The envisionment of this new design characterizes 24 qualitatively distinct behaviors (see Figure 6.25), all of which assign value `(Vhi std)` to signal *in* in the final (quiescent) state, as shown in Figure 6.26. Using the first design specification, given in Figure 6.5, the purpose of the design modification which replaces pass transistor ($t1$) with transmission gate (Tm) can be expressed in TeD as follows. Let δ_3 represent the design modification of replacing $t1$ with transmission gate Tm (Figure 6.23), Then

$$\delta_3 \textbf{ Prevents } \sigma_1. \tag{6.4}$$

6.3 Electric Motor Example

The electric motor example discussed here is taken from Kiriyama, Tomiyama, and Yoshikawa [KTY91], who give an initial design and a series

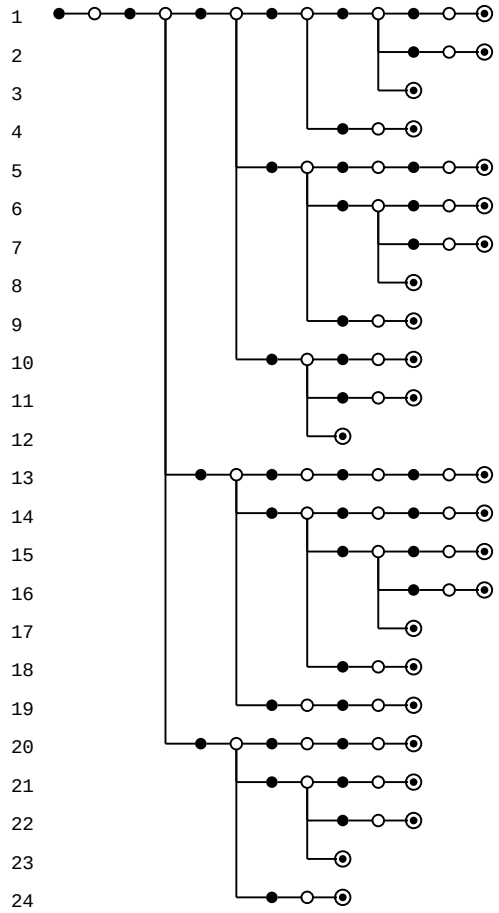


Figure 6.25: Behavior Tree of Circuit with Transmission Gate

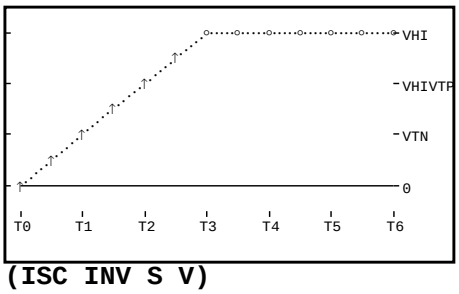


Figure 6.26: Qualitative Plot for Circuit with Transmission Gate

of design modifications in an investigation of model building techniques for analysis in various domains, specifically electrical, mechanical (rotation), and thermal.[6] The desired behavior of the electric motor is to translate electric current into mechanical rotation for use in some larger system in which the electric motor is embedded. The particular design specifications regarding the motor behavior are:

- Mechanical rotation in the positive direction at a specified velocity.
- No dead points (starting positions at which the motor will not rotate).
- No unbalanced lateral forces on the motor shaft.

6.3.1 Structure

The initial motor design (see schematic in Figure 6.27) contains three components, a shaft, rotor, and magnet. An electric current through the coil of the rotor establishes a magnet field for the rotor, and the rotor-shaft connection gives the rotor-shaft assembly one degree of freedom, namely rotation. The magnetic attraction between the south pole of the rotor and the north pole of the magnet provides the force that initiates rotation of the rotor-shaft assembly. The top level of the hierarchical structure description (in CC) is given in Figure 6.28, for positive positions of the rotor-shaft assembly. Appendix C gives the complete motor example.

[6]The larger goal discussed in [KTY91] is an intelligent CAD system *expected to serve as an integrated modeling environment in which aspect models are automatically generated and their consistency are maintained.* Such an environment provides the essential capabilities for capturing teleological descriptions, namely model building and model analysis.

[7]Mode variable declarations are ommitted in this figure for space considerations. The complete description is given in Appendix C.

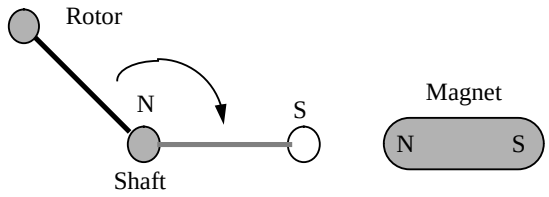


Figure 6.27: Electric Motor (`motor1`) - Initial Design

6.3.2 Design Specifications

The design specifications for the motor design given previously are captured in the following, precise specifications:

```
(for-component S (shaft)
  (conditionally ((((((S V) (0 ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (conditionally ((((((S V) ((0 V*) ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (conditionally ((((((S V) (V* ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (prohibited (((((S V) ((0 inf) ign))
                 ((S Cum-F-lat) ((0 inf) std))))
           true))
```

The first three specifications describe the behavior that for any component of type shaft, if the rotational velocity of the shaft starts between `0` and the desired velocity `V*` (inclusive), then the velocity should become constant at the desired velocity, `V*`. The last specification states that for any positive shaft velocity, the cumulative lateral force exerted on the shaft should be `0` and constant.

```
(define-component-interface
  motor "Electro-mechanical motor" mechanical
  (quantity-spaces
    (defaults ((magnetic force)            polarity-qspace)
              ((electrical current)        motor-current-qspace)
              ((mechanical-rotation force) angular-force-qspace))))

(define-component-implementation
  1 motor "Single magnet, single rotor"
  (quantity-spaces
    (default ((mechanical-rotation velocity) motor-velocity-qspace)))
  (component-variables (PE energy (quantity-space (0 PE+ PE*)))
                       (TE energy))
  (components (magnet magnet)
              (rotor (rotor (impl 1)) (no-new-landmarks F-lat F-ang)
                     (ignore-qdir F-ang)
                     (quantity-spaces (X position-X-qspace)))
              (shaft (one-terminal-shaft (impl 1))
                     (no-new-landmarks F-lat F-ang Cum-F-ang)
                     (ignore-qdir F-ang Cum-F-ang)
                     (quantity-spaces (X position-X-qspace))))
  (constraints
    ((ADD PE (shaft KE) TE))
    ((constant TE))
    ((constant (shaft I) Imax+))
    ((position positive)
      ->
      ((U- (shaft X) (rotor Orientation) (X+ Omax+)) (0 0) (X180+ 0))
      ((S- (shaft X) PE (0 PE*) (X180+ 0)) (X+ PE+)))
    ((position negative)
      ->
      ((U+ (shaft X) (rotor Orientation) (X- Omax-)) (X180- 0) (0 0))
      ((S+ (shaft X) PE (X180- 0) (0 PE*)) (X- PE+)) )
  (connections (c1 (rotor magnet) (magnet north))
               (c2 (rotor shaft)  (shaft t))))
```

Figure 6.28: Motor - Initial Design (`motor1`) - CC Model[7]

6.3.3 Behavior

The first envisionment to examine is the attainable envisionment from the assertions that the initial velocity of the shaft is `0` and the initial shaft position is between landmarks `0` and `X180+`, specifically

```
(shaft V)  =  (0 nil)
(shaft X)  =  ((0 X180+) nil)
```

This envisionment (see Figure 6.29) has three initial states corresponding to the three initial values `(0 X+)`, `X+`, and `(X+ X180+)` for the shaft position. The behaviors with an initial shaft position at `0` or `X180+` are quiescent in the initial state, demonstrating dead points at which no rotation is provided. All three initial states shown in Figure 6.29 produce cyclic behaviors in which the desired velocity `V*` may or may not be achieved, and possibly even exceeded. Further, the cumulative lateral force on the shaft is non-zero. These behavior characteristics are shown in the qualitative plots in Figure 6.30.

The attainable envisionment generated for an initial velocity of `0` and the shaft starting in a negative position is essentially the same as the attainable envisionment for a positive starting position. The attainable envisionments for initial rotational velocities of `(0 V*)` and `V*` also show cyclic behaviors, some oscillating between positive and negative velocities (like a pendulum or undamped spring), and the others completing rotations. As was the case for initial velocity `0`, some behaviors have velocity exceeding the desired velocity, `V*`.

6.3.4 Evaluation 1

The first design modification makes two changes to the initial motor design. First, a second magnet is added to the design, placed opposite the

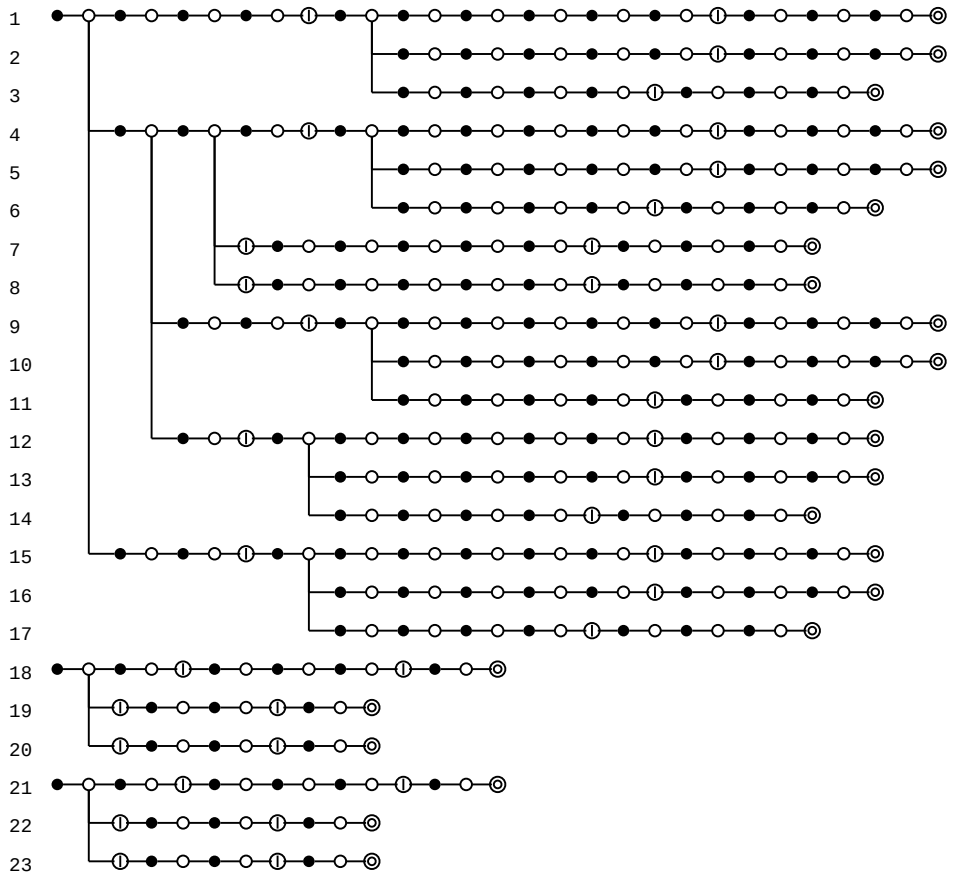


Figure 6.29: Behavior Tree (`motor1`) - Positive Starting Positions

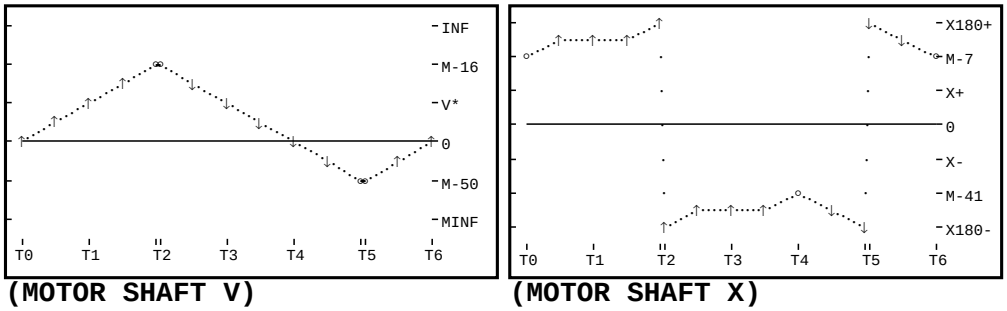


Figure 6.30: Qualitative Plots (`motor1`) - Positive Starting Positions

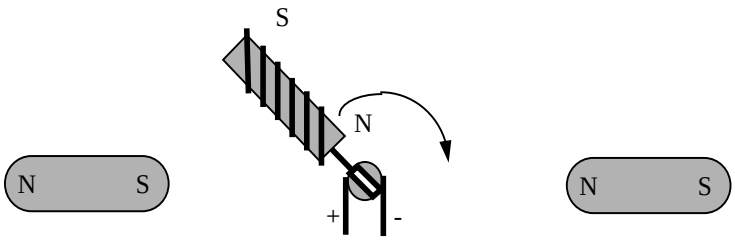


Figure 6.31: Electric Motor (`motor2`) - Second Design

shaft from the first magnet and given the opposite magnetic orientation (with respect to the shaft), as shown in the schematic in Figure 6.31. Further, the shaft is modified so that the polarity of the rotor can be reversed as the shaft rotates through positions `0` and `X180` (i.e., a commutator).

Structure Modification

The first design modification includes the addition of a second magnet and the replacement of the shaft and rotor components. The new shaft component models the commutator for reversing the current flowing through the rotor, and the new rotor component allows the rotor to be connected to both magnets.[8] The new shaft component definition is shown in Figure 6.32. The CC definition for the new motor design (`motor2`) is given in Figure 6.33. Note that the `S+` (`S-`) constraint relates the current to the shaft velocity, thereby constraining the velocity to be less than or equal to landmark `V*`.

[8]These connections provide for the interaction of the magnetic fields of the rotor and the magnets, as the component-connection modeling approach dictates that such interactions must be explicitly declared.

[9]Mode variable declarations are ommitted in this figure for space considerations. The complete description is given in Appendix C.

```
(define-component-implementation
  2 one-terminal-shaft ""
  (terminal-variables
    (t (F-ang force)
       (V     velocity)
       (F-lat (mechanical-translation force)
              (quantity-space motor-lateral-force-qspace))
       (I     (electrical current))))
  (component-variables
    (X         displacement)
    (Cum-F-ang force (quantity-space angular-force-qspace)))
  (constraints ((d/dt X V))
               ((d/dt V Cum-F-ang))
               ((minus Cum-F-ang F-ang) (F- F+) (0 0) (F+ F-))))
```

Figure 6.32: Single Rotor Commutator Shaft Component - CC Model

Modified Behavior

To generate the envisionment we assert the initial velocity of the shaft as `0` and the initial shaft position as `X90+` (between `0` and `X180+`), specifically

```
(shaft V)  =  (0 nil)
(shaft X)  =  (X90+) nil)
```

From this initial state, the envisionment is infinite as indicated in the behavior tree in Figure 6.34. This corresponds the the number of rotations of the motor before the desired velocity, `V*`, is reached. The desired velocity is reached, after some number of rotations, as shown in the qualitative plots in Figure 6.35. Starting positions `(0 X90+)` and `(X90+ X180+)` generate attainable envisionments similar to that for starting position `X90+`. Starting positions of `0` and `X180+` are still dead points. Similarly, for the negative starting shaft positions, two initial states demonstrate dead points (positions `0` and `X180-`) and the other three generate cyclic behaviors in which the velocity has the desired value `V*`. For all starting positions, an initial velocity of either `(0 V*)` or `V*`

```
(define-component-implementation
  2 motor
  "Double magnet, single rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (components
    (magnet1 magnet)
    (magnet2 magnet)
    (rotor   2-field-rotor (no-new-landmarks F-lat F-ang)
             (ignore-qdir F-ang))
    (shaft   (one-terminal-shaft (impl 2))
             (no-new-landmarks F-lat F-ang Cum-F-ang)
             (ignore-qdir F-ang Cum-F-ang)
             (quantity-spaces (X position-90-qspace))))
  (constraints
    ((position positive)
      ->
      ((U- (shaft X) (rotor Orientation) (X90+ Omax+))
                                         (0 0) (X180+ 0))
      ((S- (shaft V) (shaft I) (0 Imax+) (V* 0))))
    ((position negative)
      ->
      ((U+ (shaft X) (rotor Orientation) (X90- Omax-))
                                         (X180- 0) (0 0))
      ((S+ (shaft V) (shaft I) (0 Imax-) (V* 0)))))
  (connections (c1 (rotor magnet+) (magnet1 north))
               (c2 (rotor magnet-) (magnet2 south))
               (c3 (rotor shaft)   (shaft t))))
```

Figure 6.33: Second Motor Design (`motor2`) - CC Model[9]

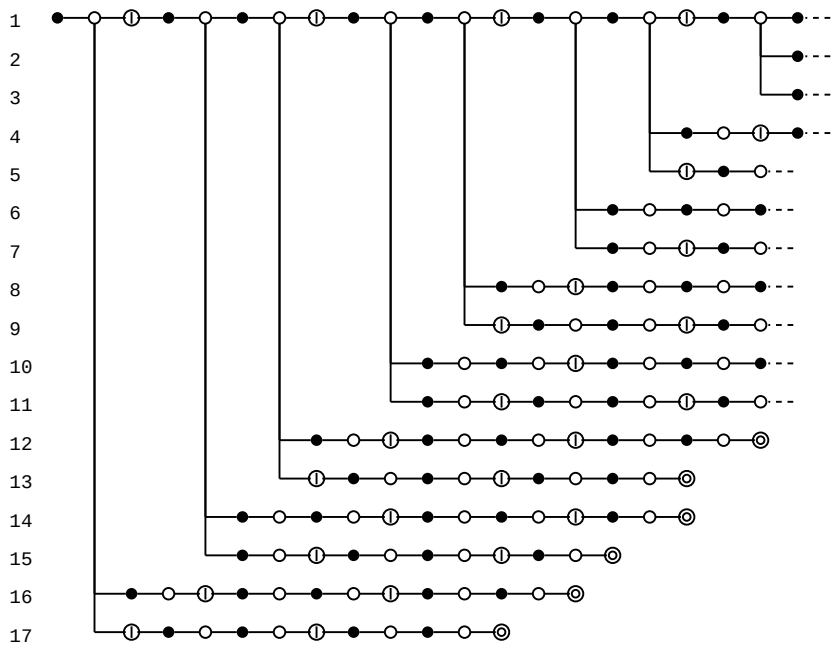


Figure 6.34: Behavior Tree (`motor2`) - Starting Position `X90+`

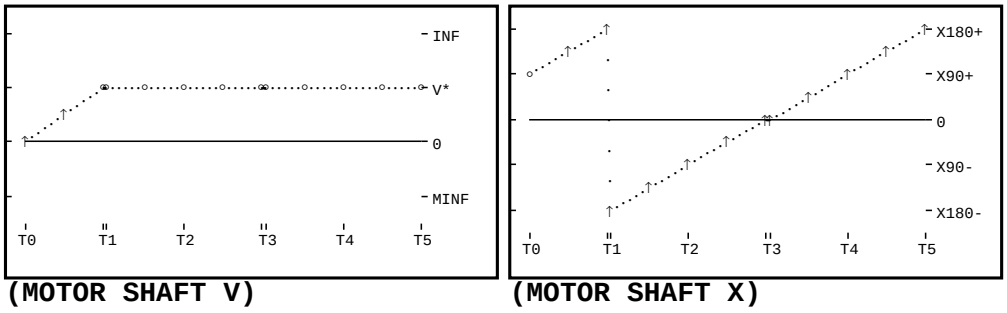


Figure 6.35: Qualitative Plots (`motor2`) - Starting Position `X90+`

results in cyclic behavior with the desired velocity `V*`. In all behaviors with a non-zero velocity, the lateral force exerted on the shaft is non-zero.

Teleology

While none of the design specifications have been met for all behaviors of the design, we can identify some initial conditions for which a design specification has been met. Specifically, for starting positions other than `0` and `X180`, the design specification regarding the desired rotational velocity for the shaft has been met. This purpose can be expressed in TeD as follows: Let δ_4 represent the design modification described previously, namely addition of a second magnet and replacement of the shaft with a commutator shaft. Let σ_7, σ_8, and σ_9 be the conditional scenarios of the first three design specifications, namely

$$\begin{aligned}
\sigma_7 &= (\langle\{\texttt{((shaft V) (0 ign))}\}\rangle,\texttt{true}),\\
\sigma_8 &= (\langle\{\texttt{((shaft V) ((0 V*) ign))}\}\rangle,\texttt{true}), \text{ and}\\
\sigma_9 &= (\langle\{\texttt{((shaft V) (V* ign))}\}\rangle,\texttt{true}).
\end{aligned}$$

Let σ_{10} be the required scenario of the first three design specifications, namely

$$\sigma_{10} = (\langle\{\texttt{((shaft V) (V* std))}\}\rangle,\texttt{true}).$$

Let σ_{11} and σ_{12} be the scenarios describing the shaft positions between `0` and `180`, namely

$$\begin{aligned}
\sigma_{11} &= (\langle\{\texttt{((shaft X) ((0 X180+) ign))}\}\rangle,\texttt{true}), \text{ and}\\
\sigma_{12} &= (\langle\{\texttt{((shaft X) ((X180- 0) ign))}\}\rangle,\texttt{true}).
\end{aligned}$$

Then we can claim the following teleological descriptions involving δ_4:

$$\delta_4 \textbf{ Conditionally } \text{when } \{\sigma_7, \sigma_{11}\} \textbf{ Guarantees } \sigma_{10} \tag{6.5}$$

$$\delta_4 \textbf{ Conditionally } \text{when } \{\sigma_7, \sigma_{12}\} \textbf{ Guarantees } \sigma_{10} \tag{6.6}$$

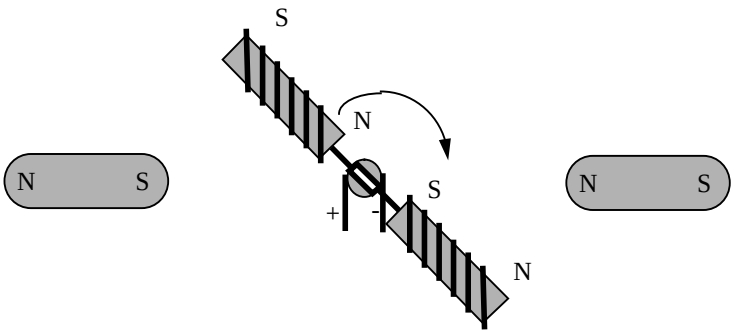


Figure 6.36: Electric Motor (`motor3`) - Third Design

$$\delta_4 \textbf{ Conditionally } \text{when } \{\sigma_8\} \textbf{ Guarantees } \sigma_{10} \tag{6.7}$$

$$\delta_4 \textbf{ Conditionally } \text{when } \{\sigma_9\} \textbf{ Guarantees } \sigma_{10} \tag{6.8}$$

6.3.5 Evaluation 2

The second design modification makes essentially one change to the previous motor design, the addition of a second rotor to the shaft placed opposite the first rotor (see schematic in Figure 6.36). The shaft component is modified to accommodate the second rotor.

Structure Modification

The second design modification adds a second rotor and modifies the shaft. The new shaft component models the commutator for reversing the current flowing through both rotors, as the shaft rotates through positions `0` and `X180`, and maintains opposite polarity in the two rotors. The definition for the new motor design (`motor3`) is given in Figure 6.37.[10]

[10]Mode variable declarations are ommitted in this figure for space considerations. The complete description is given in Appendix C.

[11]Mode variable declarations are ommitted in this figure for space considerations. The complete description is given in Appendix C.

```
(define-component-implementation
  3 motor
  "Double magnet, double rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation force)    angular-force-qspace)
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (components
    (magnet1 magnet)
    (magnet2 magnet)
    (rotor1  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang) (display I Orientation Polarity)
             (quantity-spaces (Orientation orientation-qspace)))
    (rotor2  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang) (display I Orientation Polarity)
             (quantity-spaces (Orientation orientation-qspace)))
    (shaft   (2-terminal-shaft (impl 1))
             (no-new-landmarks F-lat1 F-lat2 F-ang1 F-ang2
                               F-ang-sum Cum-F-ang)
             (display V1 X Cum-F-lat Cum-F-ang Position)
             (ignore-qdir F-ang1 F-ang2 F-ang-sum Cum-F-ang)
             (quantity-spaces (X position-90-qspace))))
  (constraints
    ((position positive)
      -> ((U- (shaft X) (rotor1 Orientation) (X90+ Omax+))
                                             (0 0) (X180+ 0))
         ((U+ (shaft X) (rotor2 Orientation) (X90+ Omax-))
                                             (0 0) (X180+ 0)))
    ((position negative)
      -> ((U+ (shaft X) (rotor1 Orientation) (X90- Omax-))
                                             (X180- 0) (0 0))
         ((U- (shaft X) (rotor2 Orientation) (X90- Omax+))
                                             (X180- 0) (0 0))))
  (connections (c1 (rotor1 magnet+) (rotor2 magnet+) (magnet1 north))
               (c2 (rotor1 magnet-) (rotor2 magnet-) (magnet2 south))
               (c3 (rotor1 shaft) (shaft t1))
               (c4 (rotor2 shaft) (shaft t2))))
```

Figure 6.37: Third Motor Design (`motor3`) - CC Model[11]

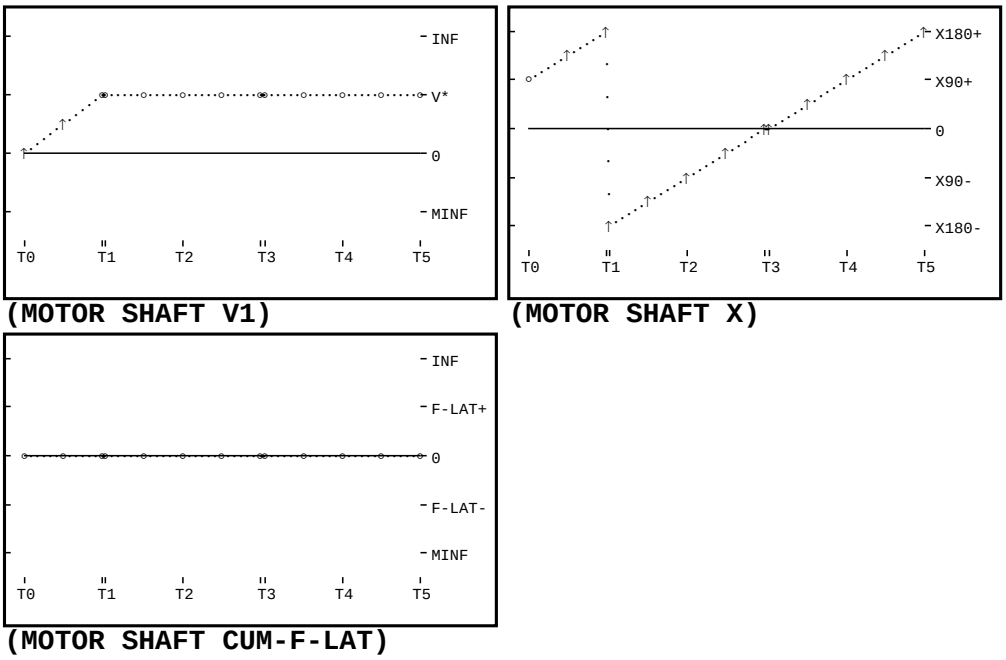


Figure 6.38: Qualitative Plots (`motor3`)

Modified Behavior

To generate the envisionment we assert the initial velocity of the shaft as `0` and the initial shaft position as `X90+` (between `0` and `X180+`), specifically

```
(shaft V)  =  (0 nil)
(shaft X)  =  (X90+ nil)
```

The behavior of the `motor3` model matches the behavior of `motor2` except that the cumulative lateral force of the shaft is now `0`, since the lateral force imparted by each rotor is balanced by the other rotor.[12] This can be seen in the value of variable `(S Cum-F-lat)`, which is `(0 std)` in all behaviors (see qualitative plot in Figure 6.38. The behavior tree for `motor3` matches the behavior tree for `motor2` (Figure 6.34).

[12]Balanced lateral force is asserted in the model via a CONSTANT constraint. The model checking algorithm does not use this information, since it looks only at the behaviors and the specifications.

Teleology

In the `motor3` design, we have satisfied the design specification regarding lateral force on the shaft. This purpose can be expressed in TeD as follows: Let δ_5 represent the design modification described previously, namely addition of a second rotor. Let σ_{13} be the required scenario of the design specification regarding lateral force on the shaft, namely

$$\sigma_{13} = (\langle\{\texttt{((shaft V) ((0 inf) ign)),} \\ \texttt{((shaft Cum-F-lat) (0 std))}\}\rangle,\texttt{true}).$$

Then we can claim the following teleological description involving δ_5:

$$\delta_5 \textbf{ Guarantees } \sigma_{13}. \tag{6.9}$$

6.3.6 Evaluation 3

The third design modification addresses the remaining design problem, the dead points, by adding a third rotor (see Figure 6.39). The shaft component is modified to accommodate the third rotor.

Structure Modification

The third design modification adds a third rotor and modifies the shaft. The new shaft component models the commutator for reversing the current flowing through all rotors as the shaft rotates and passes each rotor through positions `0` and `X180`. The rotors are distributed around the shaft so as to balance the lateral force exerted by each rotor. The definition for the new motor design (`motor4`) is given in Figure 6.40.[13]

[13]Mode variable declarations are ommitted in this figure for space considerations. The complete description is given in Appendix C.

[14]Mode variable declarations and some constraints are ommitted in this figure for space considerations. The complete description is given in Appendix C.

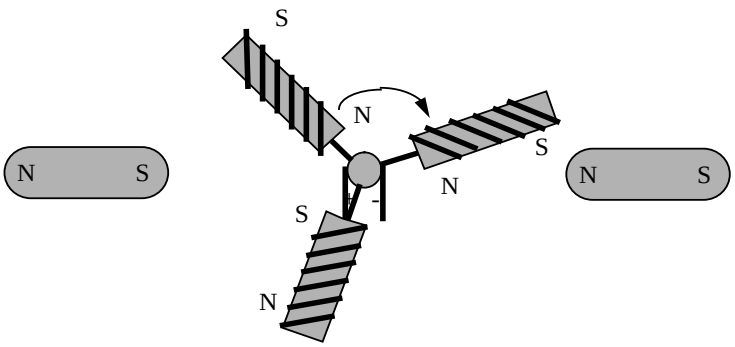


Figure 6.39: Electric Motor (`motor4`) - Fourth Design

Modified Behavior

To generate the envisionment we assert the initial velocity of the shaft as `0` and the initial shaft position as `X90+` (between `0` and `X180+`), specifically

```
(shaft V)  =  (0 nil)
(shaft X)  =  (X90+ nil)
```

The behavior of the `motor4` model matches the behavior of `motor3` except that all starting positions result in the desired shaft velocity (`V*`). This is shown in the qualitative plot in Figure 6.41, where the starting position is `0`. The behavior tree for `motor4` is similar in structure to the behavior tree for `motor3` and `motor2` (Figure 6.34), and is shown in Figure 6.42.

Teleology

In the `motor4` design, we have satisfied the first design specification in the case where the shaft starts in positions `0` and `180`, and the specification is still satisfied for the other starting positions. The purpose can be expressed in TeD as follows: Let δ_6 denote the design modification that adds the third rotor, and let σ_{14} and σ_{15} be the scenarios describing the starting shaft positions of `0` and `180`, namely

```
(define-component-implementation
  4 motor
  "Double magnet, triple rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation force)    angular-force-qspace)
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (components
    (magnet1 magnet)
    (magnet2 magnet)
    (rotor1  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang) (display I Orientation Polarity)
             (quantity-spaces (Orientation orientation-60-qspace)))
    (rotor2  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang) (display I Orientation)
             (quantity-spaces (Orientation orientation-60-qspace)))
    (rotor3  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang) (display I Orientation)
             (quantity-spaces (Orientation orientation-60-qspace)))
    (shaft   (3-terminal-shaft (impl 1))
             (no-new-landmarks F-lat1 F-lat2 F-lat3
                               F-ang1 F-ang2 F-ang3 Cum-F-ang)
             (display V1 X Cum-F-lat Cum-F-ang Position)
             (ignore-qdir F-ang1 F-ang2 F-ang3 Cum-F-ang)
             (quantity-spaces (X position-30-qspace))))
  (constraints
    ((position X0toX60+)
      -> ((M+ (shaft X) (rotor1 Orientation)) (0 0) (X60+ 060+))
         ((M- (shaft X) (rotor2 Orientation)) (0 060+) (X60+ 0))
         ((U+ (shaft X) (rotor3 Orientation) (X30+ 0max-))
                                              (0 060-) (X60+ 060-)))
    ((position X60+toX120+)
      -> ((U- (shaft X) (rotor1 Orientation) (X90+ 0max+))
                                            (X60+ 060+) (X120+ 060+))
         ((M- (shaft X) (rotor2 Orientation)) (X60+ 0) (X120+ 060-))
         ((M+ (shaft X) (rotor3 Orientation)) (X60+ 060-) (X120+ 0)))
    ...)
  (connections (c1 (rotor1 magnet+) (rotor2 magnet+)
                   (rotor3 magnet+) (magnet1 north))
               (c2 (rotor1 magnet-) (rotor2 magnet-)
                   (rotor3 magnet-) (magnet2 south))
               (c3 (rotor1 shaft) (shaft t1))
               (c4 (rotor2 shaft) (shaft t2))
               (c5 (rotor3 shaft) (shaft t3))))
```

Figure 6.40: Fourth Motor Design (`motor4`) - CC Model[14]

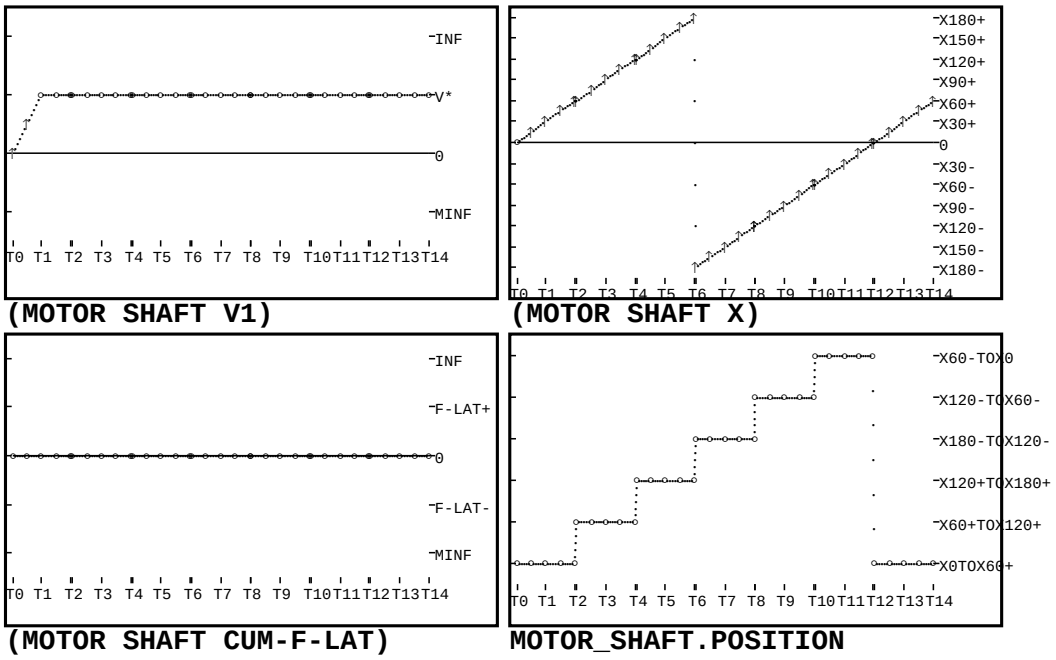


Figure 6.41: Qualitative Plots (`motor4`)

$$\begin{aligned}\sigma_{14} &= (\langle\{\texttt{((shaft X) (0 ign))}\}\rangle,\texttt{true}),\\ \sigma_{15} &= (\langle\{\texttt{((shaft X) (X180+ ign))}\}\rangle,\texttt{true}).\end{aligned}$$

Then we can claim the following teleological description involving δ_6:

$$\delta_6 \textbf{ Conditionally } \text{when } \{\sigma_7, \sigma_{14}\} \textbf{ Guarantees } \sigma_{10}, \tag{6.10}$$

$$\delta_6 \textbf{ Conditionally } \text{when } \{\sigma_7, \sigma_{15}\} \textbf{ Guarantees } \sigma_{10}. \tag{6.11}$$

The motor design history, evaluation steps, and the acquired teleological description are shown in the context of the design process flow in Figure 6.43.

[15]Design Motor1 is described in Figures 6.27 and 6.28. Envisionment E_1 is described in Figures 6.29 and 6.30. Design Motor2 is described in Figure 6.31 and 6.33. Envisionment E_2 is described in Figures 6.35 and 6.34. Design Motor3 is described in Figure 6.36 and 6.37. Envisionment E_3 is described in Figures 6.38 and 6.34. Design Motor4 is described in Figure 6.39 and 6.40. Envisionment E_4 is described in Figures 6.41 and 6.42. The acquired teleological descriptions are equations 6.5 through 6.11

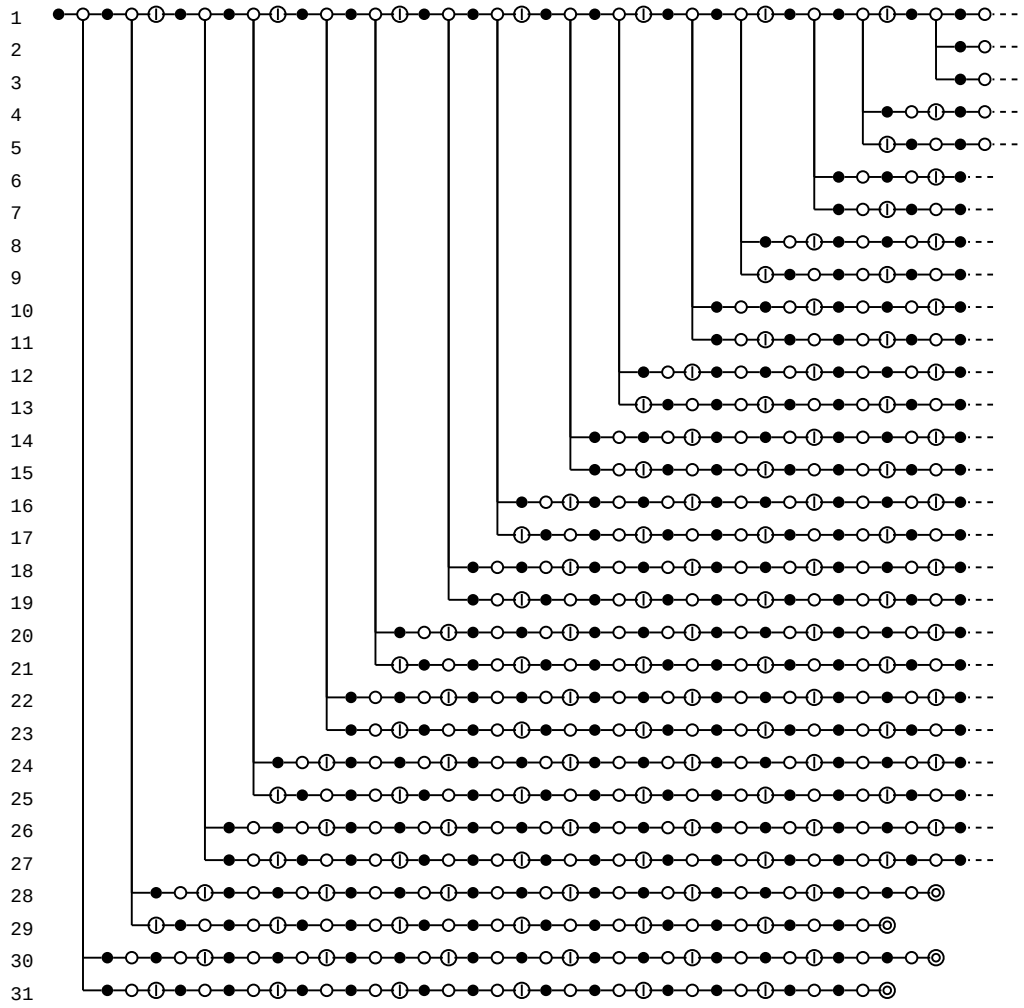


Figure 6.42: Behavior Tree (`motor4`) - Starting Position `0`

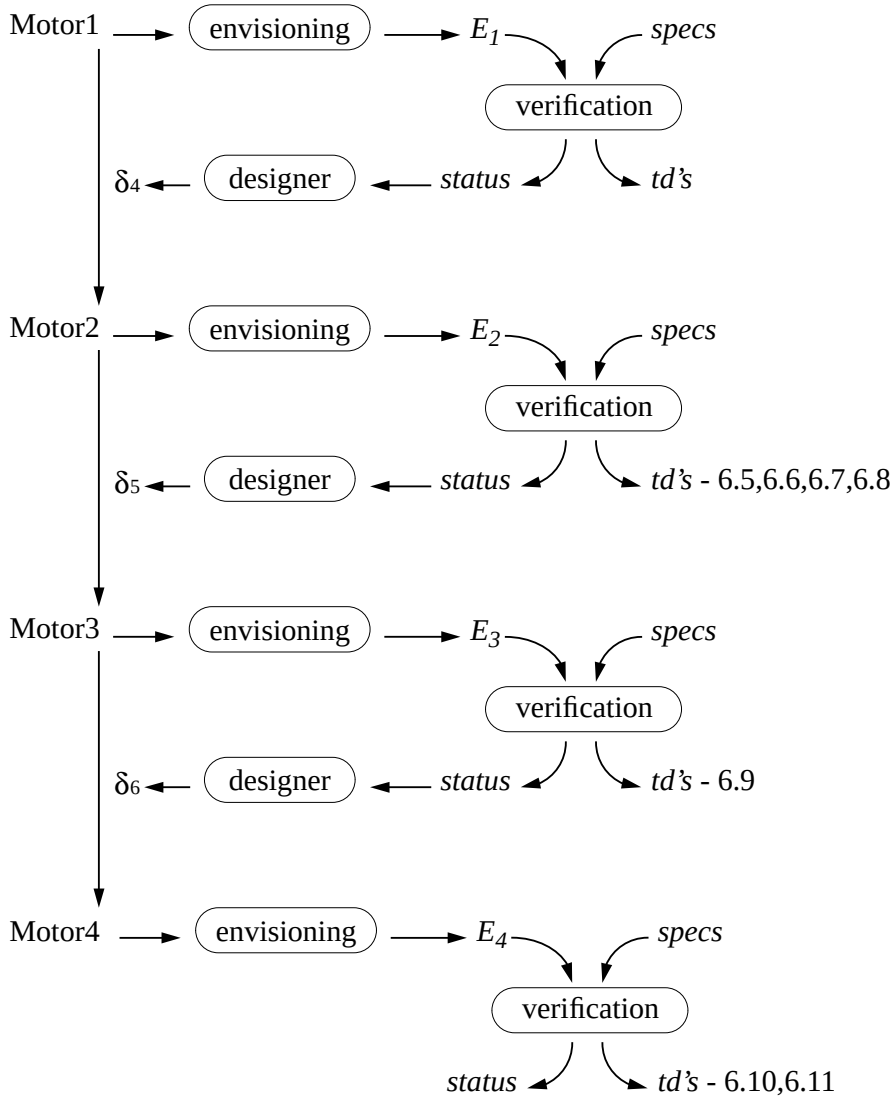


Figure 6.43: Design Flow for the Motor[15]

Chapter 7

Applications

7.1 Reusing Designs

Consider the initial input selection circuit design (Figure 6.2) in the context of an accompanying database of design modifications and components which have associated teleological descriptions.[1] If the design modifications of 1) adding the feedback transistor or 2) replacing the pass transistor with a transmission gate are recorded in a database with the associated teleological description of preventing the intermediate signal value, then these design modifications are available to the designer via a query of the form "Show me design modifications that prevent the behavior in which a signal maintains an intermediate value between `0` and `Vhi`". This query could be generated by the designer, or alternatively generated by a design critic that examines the behaviors of the design and compares those behaviors with the specifications. This design critic can then present discrepancies between the design's behavior and its specifications, and suggest possible modifications to correct these discrepancies.

More generally, the goals of the designer at each individual design step can be used to index a database of existing design modifications (including complete components) to retrieve modifications or components for reuse. In

[1] A design modification or component may have several teleological descriptions associated with it, since the modification may have resolved more than one design specification.

particular, at each step the designer is attempting to modify some aspect of the design structure and/or behavior (via structure modifications) to bring the design in line with the specifications. The significant capability introduced by this work is that the designer can access existing design modifications and components in terms relevant to the task at hand, namely modifying the design so that it meets a particular design specification. Indexing existing design information in terms of structural or behavioral aspects alone cannot provide this relevance. This index is discussed in Chapter 8, and addresses a significant problem in analogy and case-based reasoning systems.

7.1.1 Analogy

In analogy-based design reuse, behaviors referenced in teleological descriptions can be abstracted beyond particular domains by abstracting variable types, and hence provide a mechanism for retrieving design solutions from other domains (e.g. electrical versus mechanical). When such design modifications or components are retrieved, they provide the designer with design solutions from other design domains that can be applied in the current domain when suitable structural analogies are made.

For example, consider the design modification made to the steam boiler (Figure 3.11). The pressure sensor translates pressure (effort in the hydraulic domain) to voltage. An analogous component to the pressure sensor, a temperature sensor, translates temperature (effort in the thermal domain) to voltage. In undertaking a design in which temperature as opposed to pressure is to be regulated, the teleological description giving the purpose of (the addition of) the pressure sensor as regulating the pressure can be recognized and retrieved, providing a candidate analogous design, namely the steam boiler and

pressure sensor. Further, the abstraction of the teleological description that matched the new design specification (i.e., regulated temperature) gives part of the mapping from the base (analogous) design to the target design, namely that pressure (effort) maps to temperature (effort).

7.1.2 Redesign

The design flow of Figure 1.2 is also relevant when an existing design is being modified to meet a new set of specifications. Teleological descriptions can assist in the redesign task in two ways. First, as a specification is changed, any design modifications and components with teleological descriptions that reference the specification (e.g., required or prohibited behavior) are primary candidates for modification to meet the new specifications. Similar approaches to redesign are *design plans* (Steinberg and Mitchell [SM84]) and *functional representations* (Goel and Chandrasekaran [Goe89]). Second, as the designer explores the space of possible design modifications, teleological descriptions associated with the current design structure provide the designer with information concerning what other behaviors of the design might be affected if a particular component is modified.

7.1.3 Cased-Based Reasoning

Case-based reasoning systems address reuse in a manner similar to analogical reasoning systems by retrieving previous cases and *adapting* these cases to the current situation. As Riesbeck and Schank state in [RS89], "A case-based reasoner:

- finds those cases in memory that solved problems similar to the current problem, and

- adapts the previous solution or solutions to fit the current problem, taking into account any difference between the current and previous situations."

Riesbeck and Schank [RS89] also point out that "Finding the relevant cases involves:

- characterizing the input problem by assigning the appropriate features, and

- retrieving the cases from memory with those features."

The behavior and teleological description languages provide a means for characterizing the (input) problem, namely what (behavior) specification is of interest, and the indexing capability built on these languages provides the means for retrieving cases. We discuss indexing in Chapter 8.

7.2 Diagnosis

The role of teleological descriptions in diagnosis is essentially that described for redesign, namely providing focus for selecting structural components that are likely to account for observed or desired behaviors. This selection task is called *candidate* or *hypothesis generation* in model-based diagnosis. When performing model-based diagnosis (see Davis and Hamscher [DH88]), a set of candidate structure components is generated. This set contains those structural components that can possibly account for the missing or undesirable behaviors. Candidate evaluation is performed to determine whether each candidate can account for the aberrant behavior. Finally, candidate selection chooses a single candidate or set of candidates that best accounts for the aberrant behavior.

Techniques for generating the candidate set include dependency tracing and causal analysis. For devices with highly interconnected structure, this set can be a large percentage of the structural components of the device, possibly all structural elements. Since all of these candidates may require evaluation, it is important to focus the candidate generation process where possible. Domain specific heuristics can be applied to select among potential causes, but are not applicable outside their particular domain.

Teleological descriptions provide an initial focus for candidate generation, allowing an initial candidate set to be generated based on those structural components known to have been placed in the design for the purpose of affecting the aberrant behavior. If an observed symptom of a mechanism is considered as an unwanted behavior (missing behavior), then a teleological description which relates a component of the mechanism with the prevention (introduction or guarantee) of that behavior provides a heuristic for selection among potential causes. Hence, teleological descriptions can provide a more productive initial focus of attention for candidate generation in diagnosis. Such a candidate generation process is not claimed to be complete, since it is possible that the candidate set generated in this way will not always contain the structural component(s) causing the aberrant behavior.

In the modified input selection circuit (Figure 6.9), if the behavior in which *in* takes on the value `((0 Vhi) std)` is observed (i.e., an aberrant behavior occurs), then component $t2$ is a likely candidate to examine, since the purpose of adding $t2$ to the design was to eliminate the observed behavior.

Chapter 8

Indexing

8.1 Goal

In this chapter we describe an organization, for teleological descriptions, that facilitates retrieval for explanation, reuse, and diagnosis, as well as classification of newly acquired descriptions. This organization, or index structure, provides two perspectives on the database of teleological descriptions,

- specification predicates and their abstractions, and
- design history (modification sequence for a design).

We have implemented the computation of the abstraction relations described in Chapter 4 to achieve classification of new teleological descriptions and querying for teleological descriptions. We conclude this chapter by describing query or use scenarios of the teleological description database for explanation, design reuse, and diagnosis. These queries answer to following questions:

- Explanation: "What is the purpose of component X?"
- Design Reuse: "How have previous designs addressed specification ϕ?"
- Diagnosis: "What components have purposes referencing behavior B?"

8.2 Specification Predicate Lattice

The primary organizing mechanism for the teleological description index is the specification predicate lattice, which is based on the partial order $\sqsubseteq_\sigma$ (Theorem 4.26). The partial orders $\sqsubseteq_\sigma$, $\sqsubseteq_b$, $\sqsubseteq_s$, and $\sqsubseteq_v$ define the generalization and specialization relationships between scenarios and their abstractions. The abstractions used to generate the index are:

1. Generalize the qualitative direction of change of variable values (`dec`, `std`, and `inc` become `ign`).

2. Generalize the magnitudes of variable values to the quantity space `(minf - 0 + inf)`[1]

3. Generalize variable types:

   (a) Generalize the hierarchical variable name (remove a prefix from the hierarchical name), followed by

   (b) Generalize the variable type per the type hierarchy in Figure 4.2.

8.2.1 Variable Value Abstraction

Variable values are abstracted with respect to the qualitative direction of change, and with respect to the qualitative magnitude. The quantity space `(minf - 0 + inf)` is selected for magnitude abstraction because it can express the following abstract values:

[1]The landmarks `-` and `+` are a notational convenience for expressing values such as "bounded below by `0` and above by some finite value", or `(0 +)`. A finite, positive value can be expressed as `(0 inf)`, but this form makes `(0 (0 inf))` somewhat awkward.

- Negative, zero, or positive. This quantity space is used in the qualitative reasoning approaches of de Kleer and Brown [dKB85] and Williams [Wil85][2] and can be expressed in this QSIM quantity space as `(minf 0)`, `0`, and `(0 inf)`.

- A positive value bounded above, or `(0 +)`.

- A negative value bounded below, or `(- 0)`.

- A finite, positive value with a positive lower bound, or `(+ inf)`.

- A finite, negative value with an negative upper bound, or `(minf -)`.

- A finite value with a negative lower bound, or `(- inf)`.

- A finite value with a positive upper bound, or `(minf +)`.

- A finite value with a negative lower bound and a positive upper bound, or `(- +)`.

- A value with a positive lower bound and positive upper bound. Such a value has abstractions `(0 +)` and `(+ inf)`, representing the upper and lower bounds, respectively.

- A value with a negative lower bound and negative upper bound. Such a value has abstractions `(minf -)` and `(- 0)`, representing the upper and lower bounds, respectively.

The abstraction hierarchy for the values of this quantity space is shown in Figure 8.1.

[2]These qualitative values are often referred to as +,0,-.

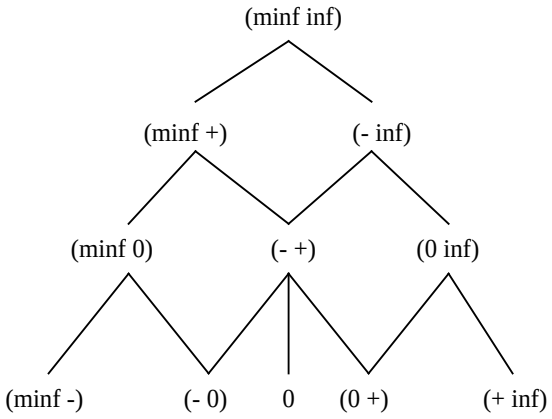


Figure 8.1: Generic Magnitude Abstraction Hierarchy

8.2.2 Variable Abstraction

The relation $\sqsubseteq_v$ is defined in terms of variable name abstraction and variable type abstraction (see Section 4.2), and gives two ways of abstracting the variable. First, a variable instance can be abstracted to represent any occurrence of the variable, such as the input voltage of an inverter instance or the input voltage of a Boolean logic gate instance.[3] This abstraction is accomplished by abstracting the (hierarchical) variable name. Second, the variable type can be abstracted from its specific domain to the corresponding generic type (see Table 8.1). For example, voltage in the electrical domain is abstracted to the generic type effort. Variable type abstraction is applied only after the variable value magnitude has been abstracted, since specific landmarks are not meaningful across domains (e.g., electrical and thermal).

[3]This abstraction is very useful because it allows a general behavior description to be written that can be intantiated for particular occurrences of a variable. Recall the general design specification for CMOS circuit design given in Chapter 6, which stated that the input voltage of a (any) logic gate in a design should not have the value `((0 Vhi) std)`.

8.2.3 Design History Index

A design history provides a (sequential) index for accessing the teleological descriptions associated with the design modifications referenced by the history. Explanation queries search the design history for design modifications involving the addition, deletion, or modification of specific design components or parameters. When found, the design modification will give the teleological descriptions associated with the modification.

8.2.4 Initial Index Structure

The initial index structure contains entries for single variable, single state behaviors representing the domains and variable types shown in Table 8.1 and the qualitative values shown in Figure 8.1. Some statistics on the initial specification predicate lattice are shown in Table 8.2.

8.3 Classification

The classification algorithm implements the behavior abstraction relations described in Chapter 4 and determines the position for teleological descriptions in the specification lattice. The algorithm computes the sets representing the most specific generalizations (*msg*) and the most general specializations (*mgs*) of the specification predicates referenced by the teleological description. Each specification predicate is added to the specification predicate lattice "between" the *msg* and the *mgs* (i.e., as a specialization of each element of *msg* and as a generalization of each element of *mgs*).

A scenario σ appears in the specification predicate `occursIn(`σ,b`)`, either as the object of the teleological description or as the condition predicate of a conditional teleological description. When classifying a scenario, we gen-

	effort	flow	momentum	displacement
Acoustic	pressure	flow	momentum	amount
Electrical	voltage	current	flux-linkage	charge
Hydraulic	pressure	flow	momentum	amount
Mechanical	force	velocity	momentum	displacement
Mech-Rotation	force	velocity	momentum	displacement
Mech-Translation	force	velocity	momentum	displacement
Thermal	temperature	flow		entropy

	power	capacitance	resistance
Acoustic	power	volume	resistance
Electrical	power	capacitance	resistance
Hydraulic	power	volume	resistance
Mechanical	power	capacitance	resistance
Mech-Rotation	power	capacitance	resistance
Mech-Translation	power	capacitance	resistance
Thermal	power	capacitance	resistance

Table 8.1: Domain-specific variable type names

Nodes	3304
Links	9502
Leaf Nodes	1050
Maximum Depth	7
Average # Generalizations	2.9
Average # Specializations	4.2
Maximum # Generalizations	4
Maximum # Specializations	15

Table 8.2: Initial Index - Metrics

erate two generalizations of the scenario and add them to the index as well. These generalizations are:

1. The scenario with values abstracted to the quantity space `(minf - 0 + inf)`,[4] abstracting away the details of particular quantity spaces.

2. Scenario 1 with variable types abstracted to the generic types shown in Table 8.1, abstracting away from a specific domain and allowing identification of analogous scenarios across different domains.

We select these generalizations out of the many that can be generated for two reasons. First, we abstract scenarios away from specific quantity spaces since quantity spaces are likely to differ in landmark names and landmark order. This abstraction provides a connection or grouping in the lattice for scenarios containing variables of the same domain specific types with similar time varying behavior. Second, we abstract away from domain specific variable types to provide a connection or grouping among scenarios containing variables of the same generic types with similar time varying behavior. This second abstraction provides the basis for design reuse across domains, namely using a teleological description originally captured in one domain (e.g., hydraulic) when designing in another domain (e.g., thermal).

8.4 Queries

In this section we describe query or use scenarios of the teleological description database for explanation, design reuse, and diagnosis. These queries

[4]All possible abstractions based on wider intervals from a particular quantity space are not recorded in the index, but are considered during search.

answer the following questions:

- Explanation: "What is the purpose of component X?"
- Design Reuse: "How have previous designs addressed specification ϕ?"
- Diagnosis: "What components have purposes referencing behavior B?"

For presentation purposes, we express these queries via the following Prolog [CM84] predicates:

- `td(`*design-history*`,`*modification*`,`*operator*`,` *spec*`)`
  Succeeds when teleological description *modification operator spec* occurs in *design-history*.
- `references(`*modification*`,`*component*`)`
  Succeeds when *modification* references structural entity *component*.
- `msg(`*spec*`,`*generalization*`)`
  Succeeds when *generalization* is a most specific generalization of specification *spec*.
- `mgs(`*spec*`,`*specialization*`)`
  Succeeds when *specializations* is a most general specialization of specification *spec*.

8.4.1 Explanation Queries

For explanation, queries of the teleological description database are restricted to those descriptions involving modifications from the design history of the mechanism being examined. In this case, queries are of the form

$$\texttt{td}(d, \texttt{Mod}, \texttt{Op}, \texttt{SpecPredicate}), \texttt{references}(\texttt{Mod}, c) \qquad (8.1)$$

where d is the design history of the mechanism being examined, and c is the component or parameter for which an explanation is desired.

For the steam boiler example from Chapter 3, an explanation query asks "What is the purpose of the pressure sensor instance `Sensor` in the steam boiler design?". Using Query Form 8.1, let d be the steam boiler design history and c the pressure sensor instance `sensor`. Having added the teleological description derived for the pressure sensor (Section 3.7.3) to the database, the query variables are bound by the query as follows:

- `Mod` - the modification that adds the pressure sensor (Figure 3.11),
- `Op` - **Prevents**, and
- `SpecPredicate` - `occursIn(`σ`,`b`)`, where σ is the scenario in which the internal pressure exceeds the landmark `Pmax*`.

This query and the retrieved teleological description are shown near the end of Section A.5.

For the input selection circuit from Chapter 6, an explanation query asks "What is the purpose of the p-channel transistor instance `PtFb` in the input selection circuit?". In Query Form 8.1, let d be the input selection circuit design history and c the p-channel transistor instance `t2`. Having added the teleological descriptions derived for the input selection circuit (Section 6.2.1) to the database, the query variables are bound by the query as follows:

- `Mod` - the modification that adds the transistor (δ_1),
- `Op` - **Conditionally** when $\{\sigma_1\}$ **Guarantees**, and
- `SpecPredicate` - $[\sigma_1;\sigma_2]$.

To further expand the set of explanations, the restriction confining the search to the current design history can be lifted, allowing search for explanations of the purpose of the component in other design histories that employ that component. For the steam boiler example, the unrestricted query for teleological descriptions referencing instances of `pressure-sensor` is

$$\texttt{td}(\texttt{Dh}, \texttt{Mod}, \texttt{Op}, \texttt{SpecPredicate}), \texttt{references}(\texttt{Mod}, \texttt{pressure-sensor}) \quad (8.2)$$

In this query, the design history is unrestricted, and the component type `pressure-sensor` is explicit. Such queries can help in understanding the uses of the component, but are not guaranteed to explain its purpose in the design of interest.

8.4.2 Reuse Queries

In a design reuse context, the designer is faced with the problem of modifying a design so that it meets specifications. Hence, the initial query made by the designer will be one based on a specification predicate of the design. While one can specify the appropriate teleological operator (**Guarantees**, if the specification predicate is to hold everywhere, or **Prevents** if the specification predicate is to be prohibited), retrieving modifications based solely on the specification predicate can be of interest to the designer regardless of the teleological operator. For example, a design modification that introduced a specification predicate in a previous design may be of interest when the reusing designer is attempting to prevent the specification predicate from holding, because the reusing designer may be able to reverse the design modification that introduced the predicate and hence prevent it. The base query is

$$\texttt{td}(\texttt{Dh}, \texttt{Mod}, \texttt{Op}, \phi) \quad (8.3)$$

where ϕ is the specification predicate of interest.

Having added the teleological descriptions captured for the steam boiler and input selection circuit designs to the database, a reuse query retrieves the appropriate design modifications for each design. For the steam boiler design, let ϕ be the design specification in which the internal pressure exceeds `Pmax*`. The variables of Query Form 8.3 are bound by the query as follows:

- `Dh` - the design history containing modification `Mod`
- `Mod` - the modification that adds the pressure sensor (Figure 3.11), and
- `Op` - **Prevents**.

If the database contains other descriptions referencing the specification predicate ϕ, these are also retrieved. A more specific query can be constructed by replacing the query variable `Op` with the explicit operator `prevents`.

For the input selection circuit, let ϕ be $[\sigma_1; \sigma_2]$. The variables of Query Form 8.3 are bound by the query as follows:

- `Dh` - the design history containing modification `Mod`
- `Mod` - δ_1, the modification that adds the feedback transistor, and
- `Op` - **Conditionally** when σ_1 **Guarantees**.

It is likely, however, that the exact specification predicate does not appear in the database, in which case the designer would like to retrieve design modifications (i.e., teleological descriptions) concerned with specification predicates "close to" the one of interest. This "closeness" property is realized by generalization and specialization links among specification predicates in the database of teleological descriptions. Consequently, the initial set of teleological descriptions retrieved for potential reuse should contain the most specific

generalizations (msg) and the most general specializations (mgs) of the specification predicate of interest, ϕ. Note that if ϕ appears exactly in the database, this set will be $\{\phi\}$, the specification predicate itself. This set is the union of solutions to the queries:

$$\texttt{msg}(\phi, \texttt{Spec}), \texttt{td}(\texttt{Dh}, \texttt{Mod}, \texttt{Op}, \texttt{Spec}) \tag{8.4}$$

and

$$\texttt{mgs}(\phi, \texttt{Spec}), \texttt{td}(\texttt{Dh}, \texttt{Mod}, \texttt{Op}, \texttt{Spec}). \tag{8.5}$$

To constrain the query with respect to a specific teleological operator such as **Guarantees**, one can use the queries

$$\texttt{td}(\texttt{Dh}, \texttt{Mod}, \texttt{guarantees}, \texttt{Spec}) \tag{8.6}$$

or

$$\texttt{td}(\texttt{Dh}, \texttt{Mod}, [\texttt{conditionally}, \phi_1, \texttt{guarantees}], \texttt{Spec}). \tag{8.7}$$

For queries involving conditional scenarios, the conditional scenario can be replaced in the query (e.g., Query Form 8.7) by a query variable. This permits teleological descriptions with different conditions, possibly empty or not a generalization or specialization of the desired condition, to be retrieved from the database. The modifications referenced in these teleological descriptions can *potentially* be of use to the designer. For example, consider a design in which temperature must be maintained between prescribed limits (denote this specification ϕ_1), and in which a modification (denoted δ) has been made that brings the temperature back within the prescribed range when perturbations push the temperature above the upper limit. Letting ϕ_2 denote the condition where the upper limit for temperature has been exceeded, the teleological description involving $\delta$is

δ **Conditionally** when ϕ_2 **Guarantees** $[\phi_2; \phi_1]$.

Design modification δcan be useful to the design when attempting to make the design respond to perturbations that push the temperature below the lower limit.

8.4.3 Diagnosis Queries

In the context of diagnosis, we make the assumption that the specification predicates were satisfied by the design, and that some portion of the mechanism is broken. Consequently, queries of the teleological description database should be restricted to those descriptions involving modifications from the design history of the mechanism under diagnosis. In this case, queries are of the form

$$\texttt{td}(d, \texttt{Mod}, \texttt{Op}, \phi) \tag{8.8}$$

where d is the design history of the mechanism under diagnosis, and ϕ is the specification predicate which no longer holds (i.e., the object of the diagnosis). If the specification predicate describes a condition that should not occur, the query can further restrict candidate descriptions by restricting the operator in the query as either

$$\texttt{td}(d, \texttt{Mod}, \texttt{prevents}, \phi) \tag{8.9}$$

or

$$\texttt{td}(d, \texttt{Mod}, [\texttt{conditionally}, \texttt{Spec}, \texttt{prevents}], \phi). \tag{8.10}$$

If the condition under diagnosis is not expressed precisely in terms of the specification predicates of the design, then `msg` and `mgs` may be required in the query.

For the steam boiler example from Chapter 3, if the internal pressure of the boiler vessel is exceeding the desired maximum pressure `Pmax*`, an diagnosis query asks "What component or subsystem of the steam boiler has the purpose of enforcing the specification in which the internal pressure exceeds

landmark `Pmax*`?". In Query Form 8.8, let d be the steam boiler design history and ϕ the design specification describing the scenario in which the internal pressure exceeds `Pmax*`. Having added the teleological description derived for the pressure sensor (Section 3.7.3) to the database, the query variables are bound by the query as follows:

- `Mod` - the modification that adds the pressure sensor (Figure 3.11), and
- `Op` - **Prevents**.

The design modification identifies the addition of the pressure sensor (and associated connections) as preventing the undesirable behavior, thereby giving an initial focus for diagnosis, namely the pressure sensor and its connections.

For the case in which no teleological description referencing specification predicate ϕ was captured during the mechanism design, and ϕ is the object of diagnosis (i.e. ϕ is not satisfied by the malfunctioning mechanism), retrieving teleological descriptions relative to ϕ for other designs can *potentially* provide some insight into the current diagnosis task. If the specification predicate was established in a similar manner in both designs, then information captured for one design can be applied in diagnosis of instances of the related design. In this case, "similar manner" means the design modifications were similar, such as adding an instance of a particular component or modifying a specific parameter of the design.

Chapter 9

Acquisition

9.1 The Problem

For knowledge-based systems, acquiring the knowledge in a form usable by that system is a principal concern. This is particularly true for systems that rely on a database (knowledge base) of examples, such as analogical reasoning systems [Hel88], design support systems, and case-based reasoning systems [RS89]. Once captured, an effective indexing technique for classifying and retrieving this knowledge is required. The semantics and form of teleological descriptions developed in this research provide a means for addressing the acquisition problem in the context of the design process model of Figure 1.2. In particular, the essential elements of teleological descriptions are available in this design process, namely design specifications and design modifications. Further, the process includes evaluation steps, points at which it is determined whether the design meets the specifications and at which the corresponding teleological descriptions can be captured.

Several acquisition approaches are possible, and one has been implemented in this work. These approaches can be applied either interactively during design or to a replay of the design history. The acquisition approaches are:

- *Explicit description* - the designer identifies the design specification and the modification that comprise the teleological description. The acqui-

sition system can verify the fact that the modification did result in the specification being met.

- *Explicit cue* or *Learn now* - the designer explicitly invokes generation of teleological descriptions at those points in the design process where a specification has been satisfied.

- *Implicit cue* - the designer states to the design system that the design specification being addressed is X. The system proposed by Abelson et al. [AEH*89] provides such an approach, with designer interactions such as "Add an active stabilizer to damp the family B motions."

- *Automatic* - the acquisition program observes design activity, noting design modifications and evaluations, and automatically generates teleological descriptions.

The approach implemented in this work can be described as the *explicit cue* approach, and provides the implementation core for the other approaches.

9.2 Comparative Analysis

Comparative and differential analyses are used to recognize the situation that a design specification has been met as a result of a design modification. We compare design evaluations performed before and after the modification to determine if a previously unsatisfied specification is now satisfied. Satisfaction of a specification predicate is determined by a model checking algorithm that computes the abstraction relations given in Chapter 4. In the designs examined in this work, we use QSIM to model and simulate designs. The choice of QSIM

allows strong statements about guarantees of the presence or absence of particular behaviors since QSIM guarantees that all possible behaviors of the model appear in the QSIM generated behavior tree [Kui89a]. To evaluate a design, we compare each behavior with the scenario σ to determine the truth value of the specification predicate `occursIn(`σ`,`b`)`. After evaluating the unmodified and modified designs, design specifications not met by the unmodified design and now met by the modified design are attributed to the design modification.

It is possible that a modification does not ensure a specification for all possible behaviors of a design, but does so for some behaviors. In this case, a conditional teleological description can be generated, with the condition describing an initial state or state sequence common to the behaviors now meeting the specification and not occurring in (i.e. abstracting) the behaviors that do not meet the specification. For example, the first modification to the motor design (see Figure 6.31) solved the dead point specification for starting positions between `0` and `180` and velocity `0`, but did not eliminate the dead points for starting positions `0` or `180` with velocity `0`.

Acquisition in this manner can be applied to modeling and evaluation techniques that do not guarantee the condition that all possible behaviors are represented *if* the evaluation technique can state those initial conditions under which it can guarantee the condition that the specification predicate is true. For example, a quantitative modeling and simulation approach may be restricted to making assertions about the truth value of a specification predicate given a set of initial, quantitative values for the model (design).

9.3 The Issue of Scope

In considering acquisition of teleological descriptions, we must consider the appropriate level of behavior or specification at which to attribute a purpose of a design component or modification. The example of a spark plug's purpose in an automobile, suggested by Mooney [Moo89], best demonstrates this issue. What is the purpose of a spark plug in an automobile? To make the car go? To make the engine produce force? To make a piston go up and down? In this example, the number of possible specifications or desired behaviors would seem to be endless, given all the potential behaviors of the automobile. We resolve this issue in the following paragraphs via a discussion concerning the nature of large system specifications, how they are developed, and how they evolve.

9.3.1 Design Specification Hierarchy

Although not always explicitly represented, the design specifications for a large, complex system describe the desired behavior and physical characteristics of the system at the level at which a user interfaces with that system. In the case of the automobile, these specifications (implicit or explicit) state such things as the expected behavior when the steering wheel is turned or when the accelerator pedal is pushed down, or the miles per gallon achieved by the vehicle. The designer or design team elaborates the design specifications based on past knowledge of such designs and on the initial functional and structural decompositions of the design (see discussions by Alford [Alf82] and Rich and Shrobe [RS84]). For the automobile example, more detailed specifications for the steering column and linkage are generated (e.g., X degrees of rotation of the steering wheel translates to Y degrees of deflection in the front tires), the en-

gine (e.g., power curve characterization), and other functional and structural components of the design. Each of these elaborations can be related to the higher level specification to which it contributes.

Teleological descriptions can be generated for any level of the specification hierarchy. For a specific component or modification, the associated specification (associated via the teleological description) will usually make a statement about the desired behavior of the functional or structural level at which the component is included. For example, a specification for the automobile might be that it translates chemical energy (gasoline) into mechanical energy (motion). The purpose of the engine is then to guarantee this behavior. As the engine design is created (via functional and/or structural decomposition), the specification is decomposed, eventually resulting in a specification for each individual cylinder of the engine. This level of specification will be referenced by a teleological description for the spark plug, namely to guarantee the behavior that the compressed fuel and air mixture is ignited and burns. Consequently, a teleological description will associate a modification with a specification of the "nearest" structural parent (hierarchically) within which the modification is made.

9.4 Planning

Planning systems (*cf.* Cohen and Feigenbaum [CF82], Nilsson [Nil80]) including linear planning, nonlinear planning (*cf.* Fikes, Nilsson [FN71], and Hart [FHN72]), and hierarchical planning (*cf.* Sacerdoti [Sac74, Sac77]) provide additional examples of initial and evolved specifications in design. Planning systems attempt to achieve some goal state, given an initial state and a set of operators on states. An example planning domain is robot manipulation, a

simple characterization of which is the blocks world.

In the blocks world, a goal state (specification) and pre- and post-conditions of operators are given in terms of predicates such as `on`, `onTable`, `holding`, and `emptyArm`. For example, the goal state in which three blocks (`A`, `B`, and `C`) are stacked on one another on the table is written as

$$\texttt{on(A,B)} \ \wedge \ \texttt{on(B,C)} \ \wedge \ \texttt{onTable(C)}.$$

To achieve this goal state, a planner can manipulate the state of the blocks world via operators, such as `pickUp`, `putDown`, `stack`, and `unStack`, that transform one state into another. Each operator has a set of preconditions that determine when the operator may be applied to a state. For example, the operator `pickUp(`x`)` requires the predicates `clear(`x`)` and `emptyArm` be true. In addition to these two sources of specifications, namely

1. An initial specification in the form of the goal state, and
2. Preconditions for the application of operators

the planning system may decompose individual specifications during the course of problem solving. In our example, the goal specification might be decomposed into `on(A,B)`, `on(B,C)`, and `onTable(C)` for purposes of the planning task.

In planning tasks where operators and their preconditions are known, acquiring teleological descriptions is straightforward. If the application of an operator achieves a particular specification (operator precondition or subgoal), then the teleological description references that operator and that specification. If a sequence of operator applications is required, then the entire sequence is referenced by the teleological description as the modification. In this way, the operator applications (each with their respective teleological descriptions) used

to achieve the various subgoals of a goal g can be referenced as a group, or *macro* modification (see Huhns and Acosta [HA88]), by a teleological description that also references the goal g.

In our blocks world example, assume that the initial state is described by `onTable(C)` $\wedge$ `onTable(B)` $\wedge$ `on(A,B)`. The planning system (hopefully!) will generate a plan like:

1. `pickUp(A)`
2. `putDown(A)`
3. `pickUp(B)`
4. `stack(B,C)`
5. `pickUp(A)`
6. `stack(A,B)`

Assuming the planning system generated the subgoals `on(B,C)` and `on(A,B)` and recognized the need to achieve subgoal `on(B,C)` first, we can generate the following teleological descriptions (in the context of the plan), where I represents the initial state:

1. `pickUp(A)`

- (1) **Guarantees** `clear(B)`

2. `putDown(A)`

- (2) **Guarantees** `emptyArm`

- (1,2)[1] **Conditionally** when I[2] **Guarantees** `clear(B)` $\wedge$ `emptyArm`, preconditions for moving `B`.

3. `pickUp(B)`

- (3) **Guarantees** `holding(B)`, precondition for operation 4.

4. `stack(B,C)`

- (4) **Guarantees** `on(B,C)`, a subgoal generated by the planner.
- (1,2,3,4) **Conditionally** when I **Guarantees** `on(B,C)`.

5. `pickUp(A)`

- (5) **Guarantees** `holding(A)`, precondition for operation 6.

6. `stack(A,B)`

- (6) **Guarantees** `on(A,B)`, a subgoal generated by the planner.
- (1,2,3,4,5,6) **Conditionally** when I **Guarantees** `on(A,B)` $\wedge$ `on(B,C)`.

There are two interesting observations to make from this planning example. First, the "regressive" operation of removing `A` from `B` (operation 1) does not cause a problem in expressing the purpose of operations, since

[1]This notation indicates that all operators are applied, in the specified order.

[2]An explicit condition is given here to state that the operators are applied from the initial state. A single operator (modification) is assumed to be applied to the previous state of the design history

the operation alone can be related to a specification (`clear(B)`). Second, the context of solving specific goals provided by the planner allows teleological descriptions involving a composite modification such as (1,2,3,4) to be recognized and generated.

Finally, with respect to the planning task, the greatest potential for teleological descriptions and their acquisition is to provide a database of example design modifications and their purpose, from which design modification operators can be learned.

Chapter 10

Previous and Related Work

10.1 Introduction

Although a large number of potential uses of teleological descriptions have been cited in work on design explanation, design reuse, design by analogy, case-based reasoning, and diagnosis, few researchers have directly addressed the formal representation and acquisition problems for descriptions of purpose. The two most significant contributions discussed in the literature are de Kleer's EQUAL system [deK85] and the *Functional Representation* [SC85] work at Ohio State University. More recent efforts in representing purpose have been undertaken by the Conservation of Design Knowledge (CDK) Project [BSZ89] at NASA Ames Research Center and in Gruber's ASK system [Gru91], and in medical reasoning research in Downing's BIOTIC system [Dow90]. Representing purpose in design systems is addressed in the REDESIGN system [SM84] and an approach to diagnosis called the *theory of responsibilities* has been developed by Milne [Mil85]. We compare the work described in this dissertation to these systems and approaches, and we point out where the TeD language has extended previous capabilities and how TeD represents the descriptions used in previous work.

With respect to related research, the work described in this dissertation achieves a set of capabilities that no related research has collected together. Subsets of these capabilities can be found in related research. However, the classification and retrieval capabilities supported by TeD are not provided

elsewhere, and the ability to described purposes regarding behaviors not exhibited by the mechanism (i.e., prevented behaviors such as explosion of the steam boiler) and components removed from the mechanism is unique to TeD. We list here the key capabilities supported by the TeD language.

- Formal language for representing purpose, with clearly defined semantics, as opposed to an ad hoc representation
- Domain independent teleology language
- Teleologies are not prescribed
- Ability to express teleology regarding missing behaviors
- Ability to express teleology regarding component removed from a design
- Teleological descriptions reference behavior of any level of the structure hierarchy
- Support for indexing and classification
- Teleological descriptions can be acquired by automated techniques
- Teleological descriptions are task independent, and can be applied to explanation, design reuse, redesign, analogical design, case-based reasoning, and diagnosis.

10.2 Function versus Teleology

As pointed by Kuipers [Kui85], the existing literature frequently obscures the distinction between *purpose* and *behavior* by using the term *function* to refer to *behavior*. For example, in their introduction to [CM85] Chandrasekaran and Milne state:

> *In simple cases, the behavior [...] can be the function, but in general, functional specifications involve teleology, i.e., an account of the intentions for which the device is used.*

In this dissertation, the term *function* has not been used to avoid this confusion. In comparing this research to other work, a clear definition of the term *function* used in other work is required. In particular, the term function is used to describe behavior, teleology, or various combinations of behavior and teleology. We will attempt to clarify the meaning of the term *function* for work referenced in this chapter.

10.3 EQUAL (de Kleer)

de Kleer's EQUAL system [deK85] expresses teleological descriptions in terms of behaviors of a component. Each such description is based on a causal assumption(s) on the parameters of the component. For example, if a resistor in an electrical circuit causally relates changes in voltage to changes in current, then the resistor is characterized as a *voltage-sensor*. A functional characterization (teleological description) is identified by matching derived behavior with prescribed behavior prototypes which have been enumerated, named, and added as domain specific knowledge. Two limitations of this approach are

- Teleological descriptions are prescribed and domain specific, and
- Teleological descriptions are limited to describing relationships among variables of a single component.

To clarify this second point, a teleological description (in EQUAL) of a component cannot reference behaviors and parameters of other components of the

system in which the component under analysis is embedded. For example, the purpose of a valve in a system of pipes and tanks may be to prevent overflow of a specific tank. The parameters of the valve are a control input, a measure of the valve aperture, and a flow rate. The level of a tank is sensed (via pressure, a level indicator, or some other technique) and relayed to the control input of the valve. EQUAL can only express teleology in terms of the parameters of the component, and in this example the level of the associated tank is not a parameter of the valve. Hence, the teleological description of the valve generated by EQUAL would be "control flow through the valve".

TeD addresses both of these problems. First, a teleological description (in the TeD language) for a component (i.e., the design modification that added the component) is generated from the specifications of the design in which the component in included, and hence is not prescribed *for the component.*[1] Second, TeD provides for abstraction of teleological descriptions, thereby allowing these descriptions to be applied across domains (electrical, thermal, etc.) in support of design by analogy.

For purposes of design explanation (a goal of the EQUAL system), EQUAL teleological descriptions can provide answers to queries of the form "What is the behavior of component X in the circuit?". However, queries of the form "Why is component X in this circuit?" cannot be answered beyond "to provide (component) behavior Y". EQUAL teleological descriptions will not give any insight into the component's contribution to a design specification beyond one that defines the desired behavior observed at the component terminals.

[1]One could say that a design specification is *prescribed*, but only in the context of the design in which the component is included, and not for the component itself.

The EQUAL approach does attempt to capture causal information (implicitly, via the derivation process) in a teleological description, while the TeD language does not. The relevance of this approach to causal reasoning is debated in [IS86a, IS86b] and [dKB86].

10.3.1 Function vs. Teleology

In the EQUAL work, de Kleer uses the terms *function* and *teleology* interchangeably (*"... the function of a circuit (i.e., its purpose) ..."*, [deK85, p. 205]). In EQUAL, function is a combination of causal information (changes in voltage "cause" changes in current) and behavior (voltage and current change). Because the teleological descriptions of a component are defined in terms of the behaviors observed at the component terminals, teleological descriptions in EQUAL will necessarily map one-to-one to these behaviors.

10.4 Functional Representation, Functional Modeling

The *functional representation* (FR) [SC85, Goe89, SCB89] and *functional modeling* (FM) [ST90, SKB90] address 1) representing *"how a device functions"* and 2) applying this information to explanation, diagnosis, and design. The functional representation expresses functional knowledge at multiple levels of abstraction in the following ontology [SC85]:

- *Structure - relationships among components and abstractions of components*

- *Function - the response* (what we call behavior) *of the component to external or internal stimuli*

```
FUNCTIONS:
   buzz: TOMAKE buzzing(buzzer)
   IF pressed (manual-switch)*
PROVIDED assumption1
BY behavior1
   ...
END FUNCTIONS
```

Figure 10.1: Function in FR (Functional Representation)

- *Behavior - how a device achieves its function.* We call this a causal explanation or representation

- *Generic knowledge - causal knowledge compiled from various domains, such as Kirchoff's laws in electrical circuits*

- *Assumptions.* We call these constraints or preconditions.

With respect to teleological descriptions, the interesting ontological element of FR is Function. Each function definition in FR contains a **ToMake** clause (see Figure 10.1, taken from [SC85]) which references a particular behavior (in the QSIM or TeD sense) that the FR function is supposed to achieve. The **ToMake** clause states the purpose of the FR function.

In TeD, given a state description of the behavior (buzzing(buzzer)), the IF condition behavior (pressed(switch)), and the assumptions in the PROVIDED clause, we can write a teleological description for the FR function. Let σ_{Make}, σ_{If}, and σ_i be the respective state descriptions listed above. The teleological description for the FR function is

$$\delta \textbf{ Conditionally } \{\text{in } \sigma_{If}, \sigma_i\} \textbf{ Guarantees } \sigma_{Make}$$

where δ is a design modification that incorporates the FR function into the larger design. Consequently, TeD provides a formal language for expressing the **ToMake** clause and condition clauses in FR. Keuneke [Keu91] extends the **ToMake** clause of FR to include the "function types" **ToMaintain**, **ToPrevent**, and **ToControl**. The TeD language can formally capture the semantics of these "purposes", or function types as shown in Chapters 3 and 4.

By expressing an FR purpose in TeD, we combine several clauses of the FR description with a formal language for which we have indexing and classification capabilities. This formal language also provides a means for clearly defining the semantics of Keuneke's function types.

10.4.1 Function vs. Teleology

As is obvious from the definition of FR ontological elements, *function* in FR describes a desired behavior via the **ToMake** clause and condition clauses, and also references a causal description (the FR behavior) of how the FR function (behavior) is achieved.

10.5 Responsibilities (Milne)

[Mil85] describes an approach to automated troubleshooting called the *theory of responsibilities*. Responsibilities relate a particular component of a design (e.g., analog circuit) to a desired output (behavior) in the form

$$\langle output \rangle \ \langle time\text{-}interval \rangle \ \langle value \rangle \ \text{by} \ \langle components \rangle$$

Responsibilities are assigned automatically based on *second principles* which have been provided to the system. These second principles represent "the type of description that an electronics engineer uses to describe various building

blocks of circuits", and are used in causal simulation to develop the responsibility assignments. These second principles represent domain specific knowledge that must be elicited from designers and represented. The thoroughness of the responsibility assignments depends on the depth of understanding provided in the second principles. If only limited understanding is available in the form of second principles, then responsibilities can only be assigned in a limited way.

The approach taken in the theory of responsibilities resembles the FR work and the work described in this dissertation in that responsibilities associate components with behaviors of the system incorporating the component. The TeD language provides a formal basis for these representations, and the TeD implementation provides an acquisition technique that does not rely on the availability of second principles.

10.5.1 Function vs. Teleology

The theory of responsibilities work uses the term function to mean the (expected) behavior at the component terminals. A responsibility is analogous to a teleological description in this work.

10.6 CDK Project (NASA Ames)

The Conservation of Design Knowledge (CDK) Project addresses the problems of representing and acquiring design rationale. The CDK acquisition approach is very similar to the approach taken in this work, and is based on a similar philosophy of design rationale. [BSZ89] states:

> *This work assumes that the goal of the designer is to satisfy a set of (changing) requirements and that the rationale for design decisions can be inferred by comparing how different design alternatives*

> *meet the design requirements. ... Our first strategy is to simulate the behavior of two designs, collect the set of requirements that were affected, and compare the ways in which they are met. To accomplish this, our system must represent design requirements and model the structure and behavior of alternative designs. Then the impact of these designs on their respective requirements could be evaluated.*

TeD provides a formal language for representing design rationale descriptions captured in the CDK acquisition work. The CDK work complements the work on TeD by providing acquisition techniques. Insufficient information regarding the details of the CDK project representation is available to judge its strengths and weaknesses with respect to TeD.

10.7 BIOTIC (Downing)

BIOTIC [Dow90] critiques natural (e.g., human, reptilian) circulatory models with respect to *teleologies*, desired global behaviors of a system. For circulatory systems, example teleologies are oxygen transport or carbon dioxide dissipation. BIOTIC critiques circulatory systems from two perspectives, a static or "zero-order" perspective and a dynamic or "first-order" perspective, called the Bipartite Teleological Model (BTM). The formalization of BTM identifies four teleologies, *transport*, *conservation*, *accumulation*, and *dissipation*. The topology (structure) of the circulatory systems are described in terms of *producers*, *consumers*, *flow mixers*, and connections among these elements. Quantities modeled in these topologies are *concentrations*, *gradients*, *exchange rates*, and *flows*. For the static perspective, "recommended" behaviors of producer and consumer flows and gradients are enumerated. For the dynamic perspective, *tendencies* for producer and consumer quantities are enumerated.

Given these recommended behaviors for each teleology, BIOTIC can then critique various circulatory systems with respect to each teleology. These critiques evaluate each topology's ability to meet the teleology by assigning ratings in the range -1 to 1 to the systems behaviors (compared to the behaviors recommended for the specific teleology), and produces an explanation that relates the salient topological relationships to teleological satisfaction.

The teleologies of BIOTIC correspond to design specifications of the work described in this dissertation[2]. The explanations generated by BIOTIC correspond to the teleological descriptions of this work. For example, in critiquing a model of the reptilian circulatory system, the following explanation is generated:

> *The steady ventricular output along with the parallelism of GM and GL permits the desired a) increase of flow to the consumption region, GL, and b) decrease of flow to the production region.*

In the TeD language, the structural and component behavioral features of *steady ventricular output* and *parallelism of GM and GL* comprise the design modification, and the desired behaviors expressed in a) and b) comprise the specification predicate of a teleological description. Note that the desired behaviors (specifications) a) and b) are derived from the top level specification (BIOTIC teleology) concerning (carbon dioxide) dissipation.

With respect to the work described herein, the most interesting capabilities of BIOTIC are 1) application to natural systems and 2) the acquisition

[2]It should be noted that BIOTIC addresses teleology in natural, evolved systems while this work addresses teleology in engineered systems. We avoid the concomitant philosophical and theological debate.

of teleological descriptions (BIOTIC explanations).

10.8 ASK (Gruber)

The ASK [Gru91] system elicits justifications from experts via an interactive dialogue with the expert. The characteristics of these justifications are [Gru91, p. 73]:

1. Justifications are represented in terms of SITUATIONS (structure, assumed operating conditions), CHOICES (design alternatives), and FEATURES (models, initial conditions, predicted behaviors).

2. The representation is implemented in a TASK-SPECIFIC ARCHITECTURE that can apply justifications to perform some task

3. Examples are elicited in a COMPUTATIONAL CONTEXT OF USE (design evaluation), where situations and choices are reflected in the state of the system

4. Justifications are elicited by asking for RELEVANT FEATURES, selected from a finite set of possible features provided by the system

5. EXPLANATIONS ARE GENERATED by mapping from relevant features to intended behaviors.

These characteristics can be realized as a combination of TeD teleological descriptions, the design environment in which they are captured, and some task environment in which they will be used (e.g., design explanation). ASK *situations* and *choices* correspond to TeD design histories (designs and modifications). ASK *features* (as predicted behaviors) correspond to TeD design specifications. The ASK *computational context of use* is the design evaluation step in

acquiring TeD descriptions. The step of asking for *relevant features* identifies the structural elements (modifications to structure, component behavior, or parameters), and *explanation generation* associates the *relevant features* (TeD design modifications) with intended behaviors (TeD design specifications).

The contribution of TeD is a formal language for representing teleological descriptions (ASK explanations) and the associated indexing capabilities provided by the language. The TeD representation also supports more automated acquisition of teleological descriptions during design.

10.9 REDESIGN (Steinberg, Mitchell)

The REDESIGN system [SM84] utilizes representations of purpose to focus the selection of candidate components for the redesign task. These teleological descriptions occur in a *design plan*, which "is characterized in terms of implementation rules that embody [...] general knowledge about circuit design tactics". These rules specify decomposition steps for realizing a design in available components. For example, a rule specifies how the OR of two functions is accomplished (introduce an OR-gate) or specifies how parallel input can be converted into serial input (introduce a shift register). All such rules applied to create the design can be organized into the *design plan*, which then records the role of individual components in the larger design.

Steinberg and Mitchell give the following example to demonstrate the kind information they attempt to capture:

> "Because the address inputs to the ROM6475 must be stable for at least 500 nsec, while the input Characters are stable for only 300 nsec, a latch (LATCH74175) is used to capture the input Character,

and hold these data values for an acceptable duration."

To represent this description in the TeD language, let δ denote the addition of "LATCH74175" to the circuit, and let σ denote the behavior "capture the input Characters and hold the address inputs to ROM6475 stable for at least 500 nsec". The teleological description is then

δ **Guarantees** σ.

Descriptions of purpose in the REDESIGN system are represented in the form of implementation rules. These rules are captured independently of the design process and added to the design system as domain specific knowledge. Automatic acqusition of implementation rules is proposed as future work. The TeD language supports acquisition of these descriptions as design occurs, providing a means for capturing the information from which such implementation rules can be derived (as described in Section 9.4). Given state representations of the desired behaviors of a circuit (e.g., OR'ed functions or parallel to serial transmission), REDESIGN descriptions of purpose can be expressed in the TeD language.

10.9.1 Function and Teleology

The REDESIGN work makes a distinction between *function* and *purpose*, where function defines the (expected) behavior at component terminals, and purpose identifies the component's role in the larger context of the behavior of the design that includes the component.

10.10 Purpose-Directed Analogy (Kedar-Cabelli)

The purpose-directed analogy system of [Ked85] requires a representation of purpose or function of artifacts to direct the construction of analogies. In particular, these descriptions of purpose are used to identify the relevant attributes of artifacts to be used in the analogy. The example given is for the concept HOT-CUP, representing objects whose purpose is to enable the drinking of hot liquids. If a cup were to be used for some other purpose, say ornamental, then different attributes would be meaningful in the analogy. The approach to representing function uses predicates such as *enables* with arguments such as operations (e.g. drinking) and substances (e.g., hot liquid). A domain theory is developed in which attributes imply structural features which in turn imply preconditions to actions which can be combined to achieve some goal (i.e., the purpose of the artifact). In this system, the relationships between attributes, structure, and function are prescribed and domain specific.

The TeD language provides a formal, domain independent representation of purpose that could be used in purpose directed analogy. Further, the TeD representation provides a means for retrieving potential analogy examples. In the case of the concept HOT-CUP, one aspect or specification of the goal to drink a hot liquid is the insulating property of the material from which the cup is constructed. If σ represents the scenario in which heat is rapidly transmitted from the liquid in the cup to the hand holding the cup, then we can write the teleological description

$$\delta \textbf{ Prevents } \sigma$$

where δ represents the selection of styrofoam as the material from which the cup is constructed. The behavior involved in this teleological description, namely the transmission of heat, can assist in the selection of analogy candidates by

identifying those designs which incorporate an insulator, i.e., something that prevents the transmission of heat.

Chapter 11

Conclusions

11.1 Accomplishments

The contribution of this work is the ability to *represent* descriptions of purpose so that these descriptions can be *reasoned about* (acquired, classified, and retrieved) and *reasoned with* in design (explanation, reuse, analogy), case-based reasoning, and diagnosis. We believe that this is an important endeavor for artificial intelligence, as Schank points out more generally:

> *...the AI [is] also in collecting the actual experiences of the experts and indexing them so that reminding and, hence, learning [can] take place.* [Sch91, p. 45]

> *The crux of AI is in the representation of [this] knowledge, the content-based indexing of [this] knowledge, and the adaptation and modification of this knowledge through the exercise of this knowledge.* [Sch91, p. 47]

This work has addressed the first two points here, namely the representation of teleological descriptions (knowledge) and the content-based indexing of this knowledge. The claims laid out in Chapter 1 for this work were:

1. Descriptions of purpose can be represented formally in a language that is independent of a particular domain of mechanisms or behavior description

language (specifically the Teleological Description (TeD) language), and these descriptions of purpose can be expressed in terms of the primitive operators **Guarantees** and **unGuarantees**,

2. Descriptions of purpose can be effectively acquired in the design process given information available in current design methodologies, and

3. The representation language facilitates the classification and retrieval of descriptions of purpose for use in design explanation, design reuse, design by analogy, case-based reasoning, and diagnosis.

These claims are supported as follows:

1. The teleological description and behavior abstraction languages described in Chapters 3 and 4, respectively, provide a formal language that is independent of any specific domain of mechanisms. This independence is demonstrated in the examples described in Chapters 2 and 6.

2. An acquisition technique is described in Chapter 9 and has been implemented. Other research (*cf.* [Gru91], [BSZ89]) addresses the acquisition problem directly, providing additional acquisition approaches.

3. Teleological descriptions can be classified and retrieved as described in Chapter 8. Further, retrieval via this index is well suited for the tasks of design explanation, design reuse, design by analogy, case-based reasoning, and diagnosis because queries are posed in terms of the problem to be solved as opposed to the technique for solving the problem.

11.2 Implementation

The ideas put forth in this dissertation have been implemented for the structure language CC [FD90] and the behavior language of QSIM [Kui85, Kui86]. The example designs for the steam boiler (Chapter 3), input selection circuit (Chapter 6), and electromechanical motor (Chapter 6) are represented in CC and simulated in QSIM. The teleological descriptions at each modification step described for the designs are captured, and classified in the index structure described in Chapter 8. Diagnosis and reuse queries via the index are also implemented.

11.3 Scaling Up

A critical question regarding the representation, acquisition, and indexing approaches is "Do they scale up?". We believe they will, and support this claim with observations about design specifications for real world, engineered systems and the design methodologies employed to produce the designs.

First, consider the issues of representation and indexing. Although the complete specification of a real world system may be many pages long in its text description, it is structured (usually hierarchically) with each individual specification making a concise statement about some static property or dynamic property (behavior) the designed system should exhibit. Hence, the scenario representation used to express behaviors will be adequate for the task. Research and progress in

- requirement (specification) representation and capture, and
- methodological (design process) support

will also contribute to the scale up of this work, since each of these endeavor to formally represent, reason about, and reason with design specifications.

Now, consider the issue of acquisition. As with representation and indexing, work in requirements and methodology support consider specifications as first class objects, hence making them accessible to acquisition techniques. To capture the fact that system requirements have in fact been met in a system design (currently available in products such ad RDD-100 and Teamwork, and called *requirements traceability*), requirements and methodology support systems will (automatically or via human intervention) trace verification steps with respect to requirements and hence identify those points at which teleological descriptions can be acquired.

Finally, with respect to acquisition, our experience has been that individual designers make design modifications whose purpose is to satisfy one or a small number of individual specifications, simply because the high complexity of making and verifying changes that address many specifications makes the design process unmanageable. For example, software development organizations employ control systems for software maintenance that require changes to the source code to be identified with the specification (usually in the form of a user problem report, i.e., a specification the software does not meet) the change is intended to address. The designers (maintainers) are then assigned the task of addressing individual problem reports, or a small number of related problem reports.

11.4 Future Work

This work provides a basis for the following related research activities:

- *Extension to other domains*, particularly software design. Issues of design explanation, design by analogy, design reuse, and diagnosis are also of concern in the domain of software engineering. Investigation should focus on the differences (if any) in specifications, behavior description, and verification techniques from the design of physical systems.

- *Integration with task specific problem solvers*: design explanation, case-based reasoners, and diagnosis systems.

- *Integration into a design environment*: address integration issues including: explicit representation of and reasoning with design process or methodology models; explicit representation of teleological description acquisition and use (explanation, design reuse) in the design process model; access to various representations of design data.

- *Scaling up*: building a knowledge-base of descriptions for real, working systems that can subsequently be used in real world design, case-based reasoning, and diagnosis systems.

- *Ex post facto acquisition*: acquisition of teleological descriptions from designs for which only the final design and no design history is available.

- *Probabilistic Guarantees*: representation and acquisition of teleological descriptions that describe the purpose of increasing or decreasing the probability of a particular behavior (suggested by Michael Huhns).

11.5 Epilogue

In reviewing this work to compose a conclusion, many of the details seem straightforward now, partly from focusing on the problem for several years

and partly because the goals of this research have been refined along with the results. A more fundamental reason for this observation is that much of what has been formalized here is understood and practiced intuitively by designers. This type of contribution is noted by Polya [Pol73, p. 57] when he quotes the nineteenth century mathematician Bernard Bolanzo:

> *I do not think at all that I am able to present here any procedure of investigation that was not perceived long ago by all men of talent; and I do not promise at all that you will find here anything quite new of this kind. But I shall take pains to state in clear words the rules and ways of investigation which are followed by all able men, who in most cases are not even conscious of following them. Although I am free from the illusion that I shall fully succeed even in doing this, I still hope that the little that is presented here may please some people and have some applications afterwards.*

In summary, this work attempts to formalize an aspect of intelligent behavior, namely reasoning about and reasoning with descriptions of purpose.

Appendix A

Steam Boiler Example

A.1 Quantity Space Definitions

```
(define-quantity-space temperature-qspace (0 AT* FT* inf))

(define-quantity-space heat-qspace (0 Ha* Hf* inf))

(define-quantity-space heat-flow-qspace
  (minf F-* 0 F* inf)
  (conservation-correspondences (F-* F*)))

(define-quantity-space liquid-flow-qspace
  (minf Fmax-* 0 Fmax* inf)
  (conservation-correspondences (Fmax-* Fmax*)))

(define-quantity-space simple-pressure-qspace
  (minf Pf-* 0 Pf* inf)
  (conservation-correspondences (Pf-* Pf*)))

(define-quantity-space pressure-qspace
  (minf Pf-* Pmax-* Pa-* 0 Pa* Pmax* Pf* inf)
  (parent simple-pressure-qspace)
  (conservation-correspondences
    (Pf-* Pf*) (Pa-* Pa*) (Pmax* Pmax-*)))

(define-quantity-space modified-pressure-qspace
  (minf Pf-* Pmax-* Pa-* 0 Pa* Plim* Pmax* Pf* inf)
  (parent pressure-qspace)
  (conservation-correspondences
    (Pf-* Pf*) (Pa-* Pa*) (Pmax* Pmax-*)))

(define-quantity-space voltage-qspace (0 Vmax*))
```

A.2 Component Definitions

```
(define-component-interface
  Heat-Source
  "Heat source in thermal domain" thermal
  (terminals out))

(define-component-implementation
  primitive Heat-Source
  "Heat source in thermal domain, in QSIM primitives"
  (terminal-variables (out (f heat-flow)
                           (t temperature independent))))

(define-component-interface
  Heat-Sink
  "Heat sink in thermal domain" thermal
  (terminals in))

(define-component-implementation
  primitive Heat-Sink
  "Heat sink in thermal domain, in QSIM primitives"
  (terminal-variables (in (f heat-flow)
                          (t temperature independent))))

(define-component-interface
  Boiler-Vessel
  "Boiler Vessel in thermal domain" thermal
  (terminals in out)
  (quantity-spaces
   (defaults (temperature temperature-qspace)
             (entropy heat-qspace))))

(define-component-implementation
  primitive Boiler-Vessel
  "Boiler Vessel for heat flow, in QSIM primitives"
  (terminal-variables
     (in  (inFlow  heat-flow (lm-symbol IF))
          (Tin     temperature))
     (out (outFlow heat-flow (lm-symbol OF))
          (Tout    temperature)))
  (component-variables
    (netFlow  heat-flow   display (lm-symbol NF))
    (heat     entropy     display (lm-symbol H))
    (pressure (hydraulic pressure)
```

```
                          display (lm-symbol P)
                          (quantity-space pressure-qspace) )
    (T        temperature display)
    (dTin     temperature display
                          (quantity-space base-quantity-space))
    (dTout    temperature display
                          (quantity-space base-quantity-space)))
  (constraints
    ((ADD T dTin Tin)   (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTin inFlow)   (0 0))
    ((ADD T dTout Tout) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTout outFlow) (0 0))
    ((ADD inFlow outFlow netFlow) (0 0 0))
    ((d/dt heat netFlow))
    ;; Assume constant fluid/gas mass, so heat follows temperature
    ((M+ heat T)     (0 0) (Ha* AT*) (Hf* FT*))
    ((M+ pressure T) (0 0) (Pa* AT*) (Pf* FT*))
    ))

(define-component-interface
  Controlled-Heat-Source
  "Controlled heat source in thermal domain" thermal
  (terminals out ctl))

(define-component-implementation
  1 Controlled-Heat-Source
  "Controlled heat source, in QSIM primitives"
  (terminal-variables (out (f heat-flow)
                           (t temperature))
                      (ctl (v (electrical voltage)
                              (quantity-space voltage-qspace))))
  (constraints ((S- v t (0 FT*) (Vmax* AT*)))))

(define-component-interface
  Boiler-Vessel-Modified
  "Boiler Vessel with Instrumentation Terminal" thermal
  (terminals in out t)
  (quantity-spaces
   (defaults (temperature temperature-qspace)
             (entropy heat-qspace))))

(define-component-implementation
  primitive Boiler-Vessel-Modified
```

```
  "Boiler Vessel with instrumentation terminal, in QSIM primitives"
  (terminal-variables
    (in  (inFlow  heat-flow (lm-symbol IF))
         (Tin     temperature))
    (out (outFlow heat-flow (lm-symbol OF))
         (Tout    temperature))
    (t   (p       (hydraulic pressure)
                  (quantity-space modified-pressure-qspace))))
  (component-variables
    (netFlow  heat-flow   display (lm-symbol NF))
    (heat     entropy     display (lm-symbol H))
    (pressure (hydraulic pressure)
                          display (lm-symbol P)
                          (quantity-space modified-pressure-qspace))
    (T        temperature display)
    (dTin     temperature display
                          (quantity-space base-quantity-space))
    (dTout    temperature display
                          (quantity-space base-quantity-space)))
  (constraints
    ((ADD T dTin Tin)   (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTin inFlow)   (0 0))
    ((ADD T dTout Tout) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ dTout outFlow) (0 0))
    ((ADD inFlow outFlow netFlow) (0 0 0))
    ((d/dt heat netFlow))
    ;; Assume constant fluid/gas mass, so heat follows temperature
    ((M+ heat T)     (0 0) (Ha* AT*) (Hf* FT*))
    ((M+ pressure T) (0 0) (Pa* AT*) (Pf* FT*))
    ((M+ pressure p) (Pf-* Pf-*) (Pmax-* Pmax-*) (Pa-* Pa-*) (0 0)
                     (Pa* Pa*) (Plim* Plim*) (Pmax* Pmax*) (Pf* Pf*))
    ))


(define-component-interface
  Pressure-Sensor
  "Pressure sensor, voltage output" hydraulic
  (terminals in out))

(define-component-implementation
  1 Pressure-Sensor
  "Pressure sensor, voltage output, in QSIM primitives"
  (terminal-variables
```

```
    (in  (p pressure (quantity-space modified-pressure-qspace)))
    (out (v (electrical voltage) (quantity-space voltage-qspace))))
  (constraints ((S+ p v (Plim* 0) (Pmax* Vmax*)))))
```

A.3 Model Definition

```
(define-component-interface
  SB "Steam Boiler" thermal
  (quantity-spaces
    (defaults (temperature temperature-qspace)
              (entropy heat-qspace)
              (heat-flow base-quantity-space))))

(define-component-implementation
  1 SB
  "Simple steam boiler"
  (components
    (Vessel boiler-vessel (display netflow heat pressure T
                                   dTin dTout inFlow outFlow))
    (Flame  heat-source)
    (Air    heat-sink))
  (connections (p1 (Flame out) (Vessel in))
               (p2 (Vessel out) (Air in))))

(define-component-implementation
  2 SB
  "Steam boiler with pressure sensor"
  (components
    (Vessel boiler-vessel-modified
           (display netFlow heat pressure T
                    dTin dTout inFlow outFlow))
    (Flame  controlled-heat-source)
    (Air    heat-sink)
    (Sensor pressure-sensor (display v)))
  (connections (p1 (Flame out) (Vessel in))
               (p2 (Vessel out) (Air in))
       (p3 (Vessel t) (Sensor in))
       (p4 (Sensor out) (Flame ctl))))

(defun Steam-Boiler-Sim (model text)
```

```
  (let* ((initial-values (translate-cc-name-alist
                          model
                          '(((Vessel t) (AT* nil))
                            ((Flame f)  ((minf 0) nil))
                            ((Flame t)  (FT* std))
                            ((Air t)    (AT* std)))))
         (sim (make-sim))
         (initial-state (make-new-state
                         :from-qde model
                         :assert-values initial-values
                         :text text
                         :sim sim)))
    (qsim initial-state)
    (qsim-display initial-state)
    sim))
```

A.4 Design Specifications

```
(define-design-specification
  DHF-No-Explode
  (prohibited (((((Pressure) ((PMax* inf) ign)))) true))
  )
```

A.5 Sample Trace

CC and TeD demo - Steam Boiler example. Initial Steam Boiler definition (in CC):

```
(define-component-interface SB
  "Steam Boiler"
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE)
              ((THERMAL HEAT-FLOW) BASE-QUANTITY-SPACE))))

(define-component-implementation SB 1
  "Simple steam boiler"
  (components
```

```
    (VESSEL BOILER-VESSEL (DISPLAY NETFLOW HEAT PRESSURE T DTIN
                                   DTOUT INFLOW OUTFLOW))
    (FLAME HEAT-SOURCE)
    (AIR HEAT-SINK))
  (connections
    (P1 (FLAME OUT) (VESSEL IN))
    (P2 (VESSEL OUT) (AIR IN))))
```

Boiler Vessel component definition (in CC):

```
(define-component-interface BOILER-VESSEL
  "Boiler Vessel in thermal domain"
  (terminals IN OUT)
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE))))

(define-component-implementation BOILER-VESSEL PRIMITIVE
  "Boiler Vessel for heat flow, in QSIM primitives"
  (terminal-variables
    (IN (INFLOW HEAT-FLOW (LM-SYMBOL IF)) (TIN TEMPERATURE))
    (OUT (OUTFLOW HEAT-FLOW (LM-SYMBOL OF)) (TOUT TEMPERATURE)))
  (component-variables
    (NETFLOW HEAT-FLOW DISPLAY (LM-SYMBOL NF))
    (HEAT ENTROPY DISPLAY (LM-SYMBOL H))
    (PRESSURE (HYDRAULIC PRESSURE)
              DISPLAY (LM-SYMBOL P)
              (QUANTITY-SPACE PRESSURE-QSPACE))
    (T TEMPERATURE DISPLAY)
    (DTIN TEMPERATURE DISPLAY (QUANTITY-SPACE BASE-QUANTITY-SPACE))
    (DTOUT TEMPERATURE DISPLAY
           (QUANTITY-SPACE BASE-QUANTITY-SPACE)))
  (constraints
    ((ADD T DTIN TIN) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ DTIN INFLOW) (0 0))
    ((ADD T DTOUT TOUT) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
    ((M+ DTOUT OUTFLOW) (0 0))
    ((ADD INFLOW OUTFLOW NETFLOW) (0 0 0))
    ((D/DT HEAT NETFLOW))
    ((M+ HEAT T) (0 0) (HA* AT*) (HF* FT*))
    ((M+ PRESSURE T) (0 0) (PA* AT*) (PF* FT*))))
```

Heat source and heat sink definitions (in CC):

```
(define-component-interface HEAT-SOURCE
  "Heat source in thermal domain"
  (terminals OUT))

(define-component-implementation HEAT-SOURCE PRIMITIVE
  "Heat source in thermal domain, in QSIM primitives"
  (terminal-variables
     (OUT (F HEAT-FLOW) (T TEMPERATURE INDEPENDENT))))

(define-component-interface HEAT-SINK
  "Heat sink in thermal domain"
  (terminals IN))

(define-component-implementation HEAT-SINK PRIMITIVE
  "Heat sink in thermal domain, in QSIM primitives"
  (terminal-variables
     (IN (F HEAT-FLOW) (T TEMPERATURE INDEPENDENT))))
```

Design specification for the Steam Boiler. This specification states that the pressure should not exceed a maximum expressed as the landmark Pmax*.

```
Design specification: DHF-NO-EXPLODE
 Prohibited Scenarios:
   State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
   Boolean Expression: TRUE
```

Constructing the QDE for the Steam Boiler from the CC definition. Information regarding the model variables is displayed.

```
Model stats: 12 variables, 0 mode variables, 10 constraints.
Model variable quantity spaces:
  Hierarchical name        Quantity Space  Internal CC name
  (SB AIR F)               (MINF 0 INF)    SB_AIR.F@P2
  (SB VESSEL OUTFLOW) [d]  (MINF 0 INF)    SB_VESSEL.OUTFLOW@P2
```

```
  (SB VESSEL INFLOW) [d]   (MINF 0 INF)    SB_VESSEL.INFLOW@P1
  (SB FLAME F)             (MINF 0 INF)    SB_FLAME.F@P1
  (SB VESSEL DTOUT) [d]    (MINF 0 INF)    SB_VESSEL.DTOUT
  (SB VESSEL DTIN) [d]     (MINF 0 INF)    SB_VESSEL.DTIN
  (SB VESSEL T) [d]        (0 AT* FT* INF) SB_VESSEL.T
  (SB VESSEL PRESSURE) [d] (MINF PF-* PMAX-* PA-* 0
                            PA* PMAX* PF* INF)
                                           SB_VESSEL.PRESSURE
  (SB VESSEL HEAT) [d]     (0 HA* HF* INF) SB_VESSEL.HEAT
  (SB VESSEL NETFLOW) [d]  (MINF 0 INF)    SB_VESSEL.NETFLOW
 Effort variable equivalence classes:
  Quantity space: (0 AT* FT* INF)          SB.EFFORT_THERMAL@P2
    (SB VESSEL TOUT)
    (SB AIR T)
  Quantity space: (0 AT* FT* INF)          SB.EFFORT_THERMAL@P1
    (SB FLAME T)
    (SB VESSEL TIN)
```

The QDE constructed from the CC model.

```
(define-QDE SB_1
  (quantity-spaces
    (SB_VESSEL.NETFLOW (minf 0 inf) "(SB VESSEL NETFLOW)")
    (SB_VESSEL.HEAT (0 ha* hf* inf) "(SB VESSEL HEAT)")
    (SB_VESSEL.PRESSURE (minf pf-* pmax-* pa-* 0 pa* pmax* pf* inf)
                        "(SB VESSEL PRESSURE)")
    (SB_VESSEL.T (0 at* ft* inf) "(SB VESSEL T)")
    (SB_VESSEL.DTIN (minf 0 inf) "(SB VESSEL DTIN)")
    (SB_VESSEL.DTOUT (minf 0 inf) "(SB VESSEL DTOUT)")
    (SB.EFFORT_THERMAL@P1 (0 at* ft* inf) "(SB FLAME T)")
    (SB_FLAME.F@P1 (minf 0 inf) "(SB FLAME F)")
    (SB_VESSEL.INFLOW@P1 (minf 0 inf) "(SB VESSEL INFLOW)")
    (SB.EFFORT_THERMAL@P2 (0 at* ft* inf) "(SB VESSEL TOUT)")
    (SB_VESSEL.OUTFLOW@P2 (minf 0 inf) "(SB VESSEL OUTFLOW)")
    (SB_AIR.F@P2 (minf 0 inf) "(SB AIR F)"))
  (constraints
   ((ADD SB_VESSEL.T SB_VESSEL.DTIN SB.EFFORT_THERMAL@P1)
    (0 0 0) (at* 0 at*) (ft* 0 ft*))
   ((M+ SB_VESSEL.DTIN SB_VESSEL.INFLOW@P1) (0 0))
   ((ADD SB_VESSEL.T SB_VESSEL.DTOUT SB.EFFORT_THERMAL@P2)
    (0 0 0) (at* 0 at*) (ft* 0 ft*))
```

```
   ((M+ SB_VESSEL.DTOUT SB_VESSEL.OUTFLOW@P2) (0 0))
   ((ADD SB_VESSEL.INFLOW@P1 SB_VESSEL.OUTFLOW@P2 SB_VESSEL.NETFLOW))
   ((D/DT SB_VESSEL.HEAT SB_VESSEL.NETFLOW))
   ((M+ SB_VESSEL.HEAT SB_VESSEL.T) (0 0) (ha* at*) (hf* ft*))
   ((M+ SB_VESSEL.PRESSURE SB_VESSEL.T) (0 0) (pa* at*) (pf* ft*))
   ((MINUS SB_FLAME.F@P1 SB_VESSEL.INFLOW@P1)
    (minf inf) (inf minf) (0 0))
   ((MINUS SB_VESSEL.OUTFLOW@P2 SB_AIR.F@P2)
    (minf inf) (inf minf) (0 0)))
  (independent SB.EFFORT_THERMAL@P2 SB.EFFORT_THERMAL@P1)
  (text (("Simple steam boiler")))
  (layout
    (SB_VESSEL.OUTFLOW@P2 SB_VESSEL.INFLOW@P1 SB_VESSEL.DTOUT)
    (SB_VESSEL.DTIN SB_VESSEL.T SB_VESSEL.PRESSURE)
    (SB_VESSEL.HEAT SB_VESSEL.NETFLOW))
  (other
    (IGNORE-QDIRS)
    (NO-NEW-LANDMARKS)
    (CC-INFO . (SB (impl 1)))
    (CC-MODE-ASSUMPTIONS))
  )
```

Simulating the model in QSIM. Behavior tree and qualitative plots are shown in Figures 3.6 and 3.7.

```
Run time: 0.200 seconds to initialize a state.
Run time: 0.940 seconds to simulate 7 states.
Send Images to [s screen / f file / b both / n nowhere] -> s

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: t

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: 1

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: q
```

Checking the behavior tree against the design specification. Any discrepencies will be noted.

```
Checking behaviors against

Design specification: DHF-NO-EXPLODE
 Prohibited Scenarios:
   State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
   Boolean Expression: TRUE

Design spec instantiation is #<Spec: PROHIBITED SC-0>:
  PROHIBITED:
    Scenario:
      State Sequence: ((SB_VESSEL.PRESSURE ((PMAX* INF) IGN)))
      Boolean Expression: TRUE

Behavior S-6 inconsistent with spec #<Spec: PROHIBITED SC-0>
```

Modifying the design. This modification involves the following editing operations:

- First operation: replace the vessel component with another that has a sensor terminal (component type boiler-vessel-modified).
- Second operation: replace the heat source (flame) component with another that has a control input (component-type controlled-heat-source).
- Third operation: add a pressure sensor that translates pressure sensed in the boiler vessel to voltage at the control input of the heat source.

```
Edit command 1:
(REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED
         ((DISPLAY NETFLOW HEAT PRESSURE T
              DTIN DTOUT INFLOW OUTFLOW)))

Edit command 2:
(REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)

Edit command 3:
(ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR
```

```
                ((DISPLAY V)) (IN (VESSEL T)) (OUT (FLAME CTL)))
```

Modified boiler vessel component definition.

```
(define-component-interface BOILER-VESSEL-MODIFIED
  "Boiler Vessel with Instrumentation Terminal"
  (terminals IN OUT T)
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE))))

(define-component-implementation BOILER-VESSEL-MODIFIED PRIMITIVE
  "Boiler Vessel with instrumentation terminal, in QSIM primitives"
  (terminal-variables
     (IN (INFLOW HEAT-FLOW (LM-SYMBOL IF)) (TIN TEMPERATURE))
     (OUT (OUTFLOW HEAT-FLOW (LM-SYMBOL OF)) (TOUT TEMPERATURE))
     (T (P (HYDRAULIC PRESSURE)
        (QUANTITY-SPACE MODIFIED-PRESSURE-QSPACE))))
  (component-variables
    (NETFLOW HEAT-FLOW DISPLAY (LM-SYMBOL NF))
    (HEAT ENTROPY DISPLAY (LM-SYMBOL H))
    (PRESSURE (HYDRAULIC PRESSURE)
              DISPLAY (LM-SYMBOL P)
              (QUANTITY-SPACE MODIFIED-PRESSURE-QSPACE))
    (T TEMPERATURE DISPLAY)
    (DTIN TEMPERATURE DISPLAY (QUANTITY-SPACE BASE-QUANTITY-SPACE))
    (DTOUT TEMPERATURE DISPLAY (QUANTITY-SPACE BASE-QUANTITY-SPACE)))
  (constraints
     ((ADD T DTIN TIN) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
     ((M+ DTIN INFLOW) (0 0))
     ((ADD T DTOUT TOUT) (0 0 0) (AT* 0 AT*) (FT* 0 FT*))
     ((M+ DTOUT OUTFLOW) (0 0))
     ((ADD INFLOW OUTFLOW NETFLOW) (0 0 0))
     ((D/DT HEAT NETFLOW))
     ((M+ HEAT T) (0 0) (HA* AT*) (HF* FT*))
     ((M+ PRESSURE T) (0 0) (PA* AT*) (PF* FT*))
     ((M+ PRESSURE P) (PF-* PF-*) (PMAX-* PMAX-*) (PA-* PA-*) (0 0)
                 (PA* PA*) (PLIM* PLIM*) (PMAX* PMAX*) (PF* PF*))))
```

Steam boiler definition after applying edit command:

```
(REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED
         ((DISPLAY NETFLOW HEAT PRESSURE T
              DTIN DTOUT INFLOW OUTFLOW)))
```

```
(define-component-interface SB
  "Steam Boiler"
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE)
              ((THERMAL HEAT-FLOW) BASE-QUANTITY-SPACE))))

(define-component-implementation SB MODIFIED
  "Simple steam boiler"
  (components
     (VESSEL BOILER-VESSEL-MODIFIED (DISPLAY NETFLOW HEAT PRESSURE T
                                          DTIN DTOUT INFLOW OUTFLOW))
     (FLAME HEAT-SOURCE)
     (AIR HEAT-SINK))
  (connections
     (P1 (FLAME OUT) (VESSEL IN))
     (P2 (VESSEL OUT) (AIR IN))))
```

Modified heat source component definition.

```
(define-component-interface CONTROLLED-HEAT-SOURCE
  "Controlled heat source in thermal domain"
  (terminals OUT CTL))

(define-component-implementation CONTROLLED-HEAT-SOURCE 1
  "Controlled heat source, in QSIM primitives"
  (terminal-variables
     (OUT (F HEAT-FLOW) (T TEMPERATURE))
     (CTL (V (ELECTRICAL VOLTAGE) (QUANTITY-SPACE VOLTAGE-QSPACE))))
  (constraints
     ((S- V T (0 FT*) (VMAX* AT*)))))
```

Steam boiler definition after applying edit command:

```
(REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)
```

```
(define-component-interface SB
  "Steam Boiler"
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE)
              ((THERMAL HEAT-FLOW) BASE-QUANTITY-SPACE))))

(define-component-implementation SB MODIFIED
  "Simple steam boiler"
  (components
     (FLAME CONTROLLED-HEAT-SOURCE)
     (VESSEL BOILER-VESSEL-MODIFIED (DISPLAY NETFLOW HEAT PRESSURE T
                                        DTIN DTOUT INFLOW OUTFLOW))
     (AIR HEAT-SINK))
  (connections
     (P1 (FLAME OUT) (VESSEL IN))
     (P2 (VESSEL OUT) (AIR IN))))
```

Pressure sensor component definition.

```
(define-component-interface PRESSURE-SENSOR
  "Pressure sensor, voltage output"
  (terminals IN OUT))

(define-component-implementation PRESSURE-SENSOR 1
  "Pressure sensor, voltage output, in QSIM primitives"
  (terminal-variables
     (IN (P PRESSURE (QUANTITY-SPACE MODIFIED-PRESSURE-QSPACE)))
     (OUT (V (ELECTRICAL VOLTAGE) (QUANTITY-SPACE VOLTAGE-QSPACE))))
  (constraints
     ((S+ P V (PLIM* 0) (PMAX* VMAX*)))))
```

Steam boiler definition after applying edit command 3:

```
(ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR
                  ((DISPLAY V)) (IN (VESSEL T)) (OUT (FLAME CTL)))
```

```
(define-component-interface SB
  "Steam Boiler"
  (quantity-spaces
    (defaults ((THERMAL TEMPERATURE) TEMPERATURE-QSPACE)
              ((THERMAL ENTROPY) HEAT-QSPACE)
              ((THERMAL HEAT-FLOW) BASE-QUANTITY-SPACE))))

(define-component-implementation SB MODIFIED
  "Simple steam boiler"
  (components
     (SENSOR PRESSURE-SENSOR (DISPLAY V))
     (FLAME CONTROLLED-HEAT-SOURCE)
     (VESSEL BOILER-VESSEL-MODIFIED (DISPLAY NETFLOW HEAT PRESSURE T
                                        DTIN DTOUT INFLOW OUTFLOW))
     (AIR HEAT-SINK))
  (connections
     ((SENSOR OUT) (FLAME CTL))
     ((SENSOR IN) (VESSEL T))
     (P1 (FLAME OUT) (VESSEL IN))
     (P2 (VESSEL OUT) (AIR IN))))
```

Constructing the QDE for the modified Steam Boiler from the CC definition. Information regarding the model variables is displayed.

```
Model stats: 14 variables, 0 mode variables, 13 constraints.
Model variable quantity spaces:
  Hierarchical name        Quantity Space  Internal CC name
  (SB AIR F)               (MINF 0 INF)    SB_AIR.F@P2
  (SB VESSEL OUTFLOW) [d]  (MINF 0 INF)    SB_VESSEL.OUTFLOW@P2
  (SB VESSEL INFLOW) [d]   (MINF 0 INF)    SB_VESSEL.INFLOW@P1
  (SB FLAME F)             (MINF 0 INF)    SB_FLAME.F@P1
  (SB VESSEL DTOUT) [d]    (MINF 0 INF)    SB_VESSEL.DTOUT
  (SB VESSEL DTIN) [d]     (MINF 0 INF)    SB_VESSEL.DTIN
  (SB VESSEL T) [d]        (0 AT* FT* INF) SB_VESSEL.T
  (SB VESSEL PRESSURE) [d] (MINF PF-* PMAX-* PA-* 0
```

```
                              PA* PLIM* PMAX* PF* INF)
                                                SB_VESSEL.PRESSURE
   (SB VESSEL HEAT) [d]      (0 HA* HF* INF) SB_VESSEL.HEAT
   (SB VESSEL NETFLOW) [d]   (MINF 0 INF)    SB_VESSEL.NETFLOW
  Effort variable equivalence classes:
   Quantity space: (0 AT* FT* INF)           SB.EFFORT_THERMAL@P2
     (SB VESSEL TOUT)
     (SB AIR T)
   Quantity space: (0 AT* FT* INF)           SB.EFFORT_THERMAL@P1
     (SB FLAME T)
     (SB VESSEL TIN)
   Quantity space: (MINF PF-* PMAX-* PA-* 0 PA* PLIM* PMAX* PF* INF)
                                             SB.EFFORT_HYDRAULIC@C-7669
     (SB SENSOR P)
     (SB VESSEL P)
   Quantity space: (0 VMAX*)                 SB.EFFORT_ELECTRICAL@C-7668
     (SB SENSOR V) [d]
     (SB FLAME V)
```

The QDE constructed from the CC model.

```
(define-QDE SB_MODIFIED
  (quantity-spaces
    (SB_VESSEL.NETFLOW (minf 0 inf) "(SB VESSEL NETFLOW)")
    (SB_VESSEL.HEAT (0 ha* hf* inf) "(SB VESSEL HEAT)")
    (SB_VESSEL.PRESSURE
        (minf pf-* pmax-* pa-* 0 pa* plim* pmax* pf* inf)
        "(SB VESSEL PRESSURE)")
    (SB_VESSEL.T (0 at* ft* inf) "(SB VESSEL T)")
    (SB_VESSEL.DTIN (minf 0 inf) "(SB VESSEL DTIN)")
    (SB_VESSEL.DTOUT (minf 0 inf) "(SB VESSEL DTOUT)")
    (SB.EFFORT_ELECTRICAL@C-7668 (0 vmax*) "(SB SENSOR V)")
    (SB.EFFORT_HYDRAULIC@C-7669
        (minf pf-* pmax-* pa-* 0 pa* plim* pmax* pf* inf)
        "(SB SENSOR P)")
    (SB.EFFORT_THERMAL@P1 (0 at* ft* inf) "(SB FLAME T)")
    (SB_FLAME.F@P1 (minf 0 inf) "(SB FLAME F)")
    (SB_VESSEL.INFLOW@P1 (minf 0 inf) "(SB VESSEL INFLOW)")
    (SB.EFFORT_THERMAL@P2 (0 at* ft* inf) "(SB VESSEL TOUT)")
    (SB_VESSEL.OUTFLOW@P2 (minf 0 inf) "(SB VESSEL OUTFLOW)")
    (SB_AIR.F@P2 (minf 0 inf) "(SB AIR F)"))
```

```
(constraints
 ((S+ SB.EFFORT_HYDRAULIC@C-7669 SB.EFFORT_ELECTRICAL@C-7668
  (PLIM* 0) (PMAX* VMAX*))  (plim* 0) (pmax* vmax*))
 ((S- SB.EFFORT_ELECTRICAL@C-7668 SB.EFFORT_THERMAL@P1
  (0 FT*) (VMAX* AT*))  (0 ft*) (vmax* at*))
 ((ADD SB_VESSEL.T SB_VESSEL.DTIN SB.EFFORT_THERMAL@P1)
  (0 0 0) (at* 0 at*) (ft* 0 ft*))
 ((M+ SB_VESSEL.DTIN SB_VESSEL.INFLOW@P1) (0 0))
 ((ADD SB_VESSEL.T SB_VESSEL.DTOUT SB.EFFORT_THERMAL@P2)
  (0 0 0) (at* 0 at*) (ft* 0 ft*))
 ((M+ SB_VESSEL.DTOUT SB_VESSEL.OUTFLOW@P2) (0 0))
 ((ADD SB_VESSEL.INFLOW@P1 SB_VESSEL.OUTFLOW@P2 SB_VESSEL.NETFLOW))
 ((D/DT SB_VESSEL.HEAT SB_VESSEL.NETFLOW))
 ((M+ SB_VESSEL.HEAT SB_VESSEL.T) (0 0) (ha* at*) (hf* ft*))
 ((M+ SB_VESSEL.PRESSURE SB_VESSEL.T) (0 0) (pa* at*) (pf* ft*))
 ((M+ SB_VESSEL.PRESSURE SB.EFFORT_HYDRAULIC@C-7669)
  (pf-* pf-*) (pmax-* pmax-*) (pa-* pa-*) (0 0) (pa* pa*)
  (plim* plim*) (pmax* pmax*) (pf* pf*))
 ((MINUS SB_FLAME.F@P1 SB_VESSEL.INFLOW@P1)
  (minf inf) (inf minf) (0 0))
 ((MINUS SB_VESSEL.OUTFLOW@P2 SB_AIR.F@P2)
  (minf inf) (inf minf) (0 0)))
(independent SB.EFFORT_THERMAL@P2)
(text (("Simple steam boiler")))
(layout
  (SB_VESSEL.OUTFLOW@P2 SB_VESSEL.INFLOW@P1
                        SB.EFFORT_ELECTRICAL@C-7668)
  (SB_VESSEL.DTOUT SB_VESSEL.DTIN SB_VESSEL.T)
  (SB_VESSEL.PRESSURE SB_VESSEL.HEAT SB_VESSEL.NETFLOW))
(other
  (IGNORE-QDIRS)
  (NO-NEW-LANDMARKS)
  (CC-INFO . (SB (impl MODIFIED)))
  (CC-MODE-ASSUMPTIONS))
)
```

Simulating the model in QSIM. Behavior tree and qualitative plots are shown in Figures 3.12 and 3.13.

```
Run time: 0.240 seconds to initialize a state.
Run time: 1.200 seconds to simulate 7 states.
```

```
Send Images to [s screen / f file / b both / n nowhere] -> s

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: t

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: 1

Qualitative time plots.  Enter T=behavior Tree,
         Space or N=Next behavior (1 of 3), behavior number,
         O=Other commands, Q=Quit: q
```

Checking the behavior tree against the design specification. Any discrepencies will be noted.

```
 Checking behaviors against

 Design specification: DHF-NO-EXPLODE
  Prohibited Scenarios:
    State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
    Boolean Expression: TRUE

 Design spec instantiation is #<Spec: PROHIBITED SC-0>:
   PROHIBITED:
     Scenario:
       State Sequence: ((SB_VESSEL.PRESSURE ((PMAX* INF) IGN)))
       Boolean Expression: TRUE

 Verified specifications:
   #<Spec: PROHIBITED SC-0>
```

Classifying the teleological description for the verified specification.

```
 Classifying <TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>))
 Value (PMAX* INF) abstracted to (+ INF)
 Variable type (HYDRAULIC EFFORT) abstracted to EFFORT
```

```
 Classifying <IN "Abstract (generic type, qmag) scenario.">
 Classifying <IN "Abstract (domain type, generic qmag) scenario.">
 Classifying <IN "Full scenario.">
```

Query design history for purpose of a modification. What were the purpose(s) of design modification Delta1?

```
   <TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>))
```

Structure inspection:

```
#<Structure DESIGN-MODIFICATION 1670C2B>

[0: NAME] DELTA1
[1: EDIT-COMMANDS]
    ((REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED (#))
     (REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)
     (ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR (#) (IN #) (OUT #)))
[2: HISTORY] <DH for (SB (IMPL 1))>
[3: TDS] (<TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>)))
>> 1
#<List 1670C49>

[0] (REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED
      ((DISPLAY NETFLOW HEAT PRESSURE T DTIN DTOUT INFLOW OUTFLOW)))
[1] (REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)
[2] (ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR
      ((DISPLAY V)) (IN (VESSEL T)) (OUT (FLAME CTL)))
>> :U
#<Structure DESIGN-MODIFICATION 1670C2B>

[0: NAME] DELTA1
[1: EDIT-COMMANDS]
    ((REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED (#))
     (REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)
     (ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR (#) (IN #) (OUT #)))
[2: HISTORY] <DH for (SB (IMPL 1))>
[3: TDS] (<TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>)))
```

```
>> 2
#<Structure DESIGN-HISTORY 1670C0B>

[0: INITIAL-DESIGN] (SB (IMPL 1))
[1: MODIFICATIONS] (DELTA1)
>> :U
#<Structure DESIGN-MODIFICATION 1670C2B>

[0: NAME] DELTA1
[1: EDIT-COMMANDS]
    ((REPLACE-SUBCOMPONENT VESSEL BOILER-VESSEL-MODIFIED (#))
     (REPLACE-SUBCOMPONENT FLAME CONTROLLED-HEAT-SOURCE NIL)
     (ADD-SUBCOMPONENT SENSOR PRESSURE-SENSOR (#) (IN #) (OUT #)))
[2: HISTORY] <DH for (SB (IMPL 1))>
[3: TDS] (<TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>)))
>> 3
#<List C7C801>

[0] <TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>))
>> 0
#<Structure TD C7C78B>

[0: MODIFICATION] DELTA1
[1: CONDITION] NIL
[2: RESULT] ((<IV SB_VESSEL.PRESSURE>))
[3: OPERATOR-NEGATED?] NIL
[4: RESULT-NEGATED?] T
>> 2
#<List C7C721>

[0] (<IV SB_VESSEL.PRESSURE>)
>> 0
#<List C7C719>

[0] <IV SB_VESSEL.PRESSURE>
>> 0
#<Structure INDEX-VARIABLE C7C6FB>

[0: NAME] SB_VESSEL.PRESSURE
[1: TYPE] (HYDRAULIC EFFORT)
[2: QMAG] (PMAX* INF)
[3: QDIR] IGN
[4: QSPACE] (MINF PF-* PMAX-* PA-* 0 PA* PLIM* PMAX* PF* INF)
```

```
>> :Q
```

Query index for teleological descriptions matching a spec.

```
Design specification: DHF-NO-EXPLODE
 Prohibited Scenarios:
   State Sequence: (((PRESSURE) ((PMAX* INF) IGN)))
   Boolean Expression: TRUE



Td's addressing the specification:
  <TD DELTA1 PREVENTS ((<IV SB_VESSEL.PRESSURE>))
```

Appendix B

Circuit Example

B.1 Quantity Space Definitions

```
(define-quantity-space MOS-voltage-qspace
  (Vhi- Vhi-Vtn Vtp 0 Vtn VhiVtp Vhi)
  (conservation-correspondences
    (Vhi- Vhi) (Vhi-Vtn VhiVtp) (Vtp Vtn)))


(define-quantity-space MOS-positive-voltage-qspace
  (0 Vtn VhiVtp Vhi)
  (parent MOS-voltage-qspace))


(define-quantity-space MOS-current-qspace
  (Imax- 0 Imax)
  (conservation-correspondences (Imax- Imax)))
```

B.2 Component Definitions

```
(define-component-interface
  Reference-Voltage
  "Reference voltage" electrical
  (terminals t)
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace)))))

(define-component-implementation
  primitive Reference-Voltage
  "Reference Voltage in QSIM primitives"
  (terminal-variables (t (v voltage independent)
                         (i current)))
  (constraints ((CONSTANT v))))
```

```
(define-component-interface
  Split
  "Split one flow into two" electrical
  (terminals m s1 s2)
  (quantity-spaces
   (defaults (voltage MOS-positive-voltage-qspace)
             (current base-quantity-space))))

(define-component-implementation
  Equipotential-base-qspace Split
  "Flows are synchronized in direction and 0 value"
  (terminal-variables (m (v voltage)
                         (i current))
                      (s1 (v1 voltage)
                          (i1 current))
                      (s2 (v2 voltage)
                          (i2 current)))
  (constraints
    ((SUM-ZERO i i1 i2) (0 0 0))
    ((M- i i1) (minf inf) (0 0) (inf minf))
    ((M- i i2) (minf inf) (0 0) (inf minf))
    ((M+ v v1) (0 0) (Vtn Vtn) (VhiVtp VhiVtp) (Vhi Vhi))
    ((M+ v v2) (0 0) (Vtn Vtn) (VhiVtp VhiVtp) (Vhi Vhi))))

(define-component-implementation
  Equipotential-current-qspace Split
  "Flows are synchronized in direction and 0 value"
  (terminal-variables
    (m (v voltage)
       (i current (quantity-space MOS-current-qspace)))
    (s1 (v1 voltage)
        (i1 current (quantity-space MOS-current-qspace)))
    (s2 (v2 voltage)
        (i2 current (quantity-space MOS-current-qspace))))
  (constraints
    ((SUM-ZERO i i1 i2) (0 0 0) (Imax Imax- 0) (Imax 0 Imax-)
                                (Imax- Imax 0) (Imax- 0 Imax))
    ((M- i i1) (0 0))
    ((M- i i2) (0 0))
    ((M+ v v1) (0 0) (Vtn Vtn) (VhiVtp VhiVtp) (Vhi Vhi))
    ((M+ v v2) (0 0) (Vtn Vtn) (VhiVtp VhiVtp) (Vhi Vhi))))
```

```
(define-component-interface
  capacitor
  "Electrical Capacitor" electrical
  (terminals t1 t2)
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current MOS-current-qspace))
   (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  current-qspace capacitor
  "Electrical capacitor in QSIM primitives"
  (terminal-variables
    (t1 (v1 voltage (quantity-space MOS-positive-voltage-qspace))
        (i  current display))
    (t2 (v2 voltage (quantity-space MOS-positive-voltage-qspace))
        (i2 current)))
  (component-variables
    (v voltage display (quantity-space MOS-voltage-qspace))
    (c capacitance independent (quantity-space (0 C*)))
    (q charge (quantity-space (0 Q*))))
  (constraints
    ((ADD v v2 v1) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                   (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                   (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                   (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                   (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
    ((MULT v c q) (Vhi C* Q*))
    ((d/dt q i))
    ((MINUS i i2) (imax imax-) (0 0) (Imax- Imax))
    ((CONSTANT c))))

(define-component-implementation
  base-qspace capacitor
  "Electrical capacitor in QSIM primitives"
  (terminal-variables
    (t1 (v1 voltage (quantity-space MOS-positive-voltage-qspace))
        (i  current display (quantity-space base-quantity-space)))
    (t2 (v2 voltage (quantity-space MOS-positive-voltage-qspace))
        (i2 current (quantity-space base-quantity-space))))
  (component-variables
    (v voltage display (quantity-space MOS-voltage-qspace))
```

```
    (c capacitance independent (quantity-space (0 C*)))
    (q charge (quantity-space (0 Q*))))
  (constraints
    ((ADD v v2 v1) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                   (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                   (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                   (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                   (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
    ((MULT v c q) (Vhi C* Q*))
    ((d/dt q i))
    ((MINUS i i2) (inf minf) (0 0) (minf inf))
    ((CONSTANT c))))


(define-component-interface
  MOS-transistor
  "MOS transistor" electrical
  (terminals g s d)
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current MOS-current-qspace))
   (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  N-channel-bidirectional MOS-transistor
  "N channel, bidirectional transistor"
  (terminal-variables
    (g (Vg  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ig  current (quantity-space base-quantity-space)))
    (s (Vs  voltage (quantity-space MOS-positive-voltage-qspace))
       (Isd current))
    (d (Vd  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ids current)))
  (component-variables
    (Vsd       voltage     (quantity-space MOS-voltage-qspace))
    (Vds       voltage     (quantity-space MOS-voltage-qspace))
    (Vgs       voltage     (quantity-space MOS-voltage-qspace))
    (Vgd       voltage     (quantity-space MOS-voltage-qspace))
    (channel1  resistance  (quantity-space (0 C1*))
                           (landmark-symbol ch))
    (channel2  resistance  (quantity-space (0 C2*))
                           (landmark-symbol ch))
    (channel12 resistance  (quantity-space (0 C12* C3*))
```

```
                          (landmark-symbol ch))
    (channel   resistance  (quantity-space (0 Ch*))
                          (landmark-symbol ch))
    (Qg        charge      (quantity-space (0 Qg*)))
    (Cg        capacitance independent (quantity-space (0 Cg*))))
  (constraints
   ((ADD Vsd Vd Vs) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                    (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                    (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                    (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                    (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
   ((MINUS Vsd Vds) (Vhi- Vhi) (Vhi-Vtn VhiVtp) (Vtp Vtn) (0 0)
                    (Vhi Vhi-) (VhiVtp Vhi-Vtn) (Vtn Vtp))
   ((ADD Vgd Vd Vg) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                    (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                    (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                    (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                    (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
   ((ADD Vgs Vs Vg) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                    (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                    (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                    (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                    (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
   ((S+ Vgd channel1 (Vtn 0) (Vhi C1*)))
   ((S+ Vgs channel2 (Vtn 0) (Vhi C2*)))
   ((ADD channel1 channel2 channel12) (0 0 0) (C1* 0 C12*)
                                      (0 C2* C12*) (C1* C2* C3*))
   ((S+ channel12 channel (0 0) (C12* Ch*)))
   ((MULT Vsd channel Isd) (Vhi Ch* Imax) (Vhi- Ch* Imax-))
   ((MINUS Ids Isd) (Imax- Imax) (0 0) (Imax Imax-))
   ;; Gate capacitance constraints
   ((MULT Vg Cg Qg) (Vhi Cg* Qg*))
   ((D/DT Qg Ig))
   ))

(define-component-implementation
  P-channel-bidirectional MOS-transistor
  "P channel transistor in QSIM primitives"
  (terminal-variables
    (g (Vg  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ig  current (quantity-space base-quantity-space)))
    (s (Vs  voltage (quantity-space MOS-positive-voltage-qspace))
       (Isd current))
```

```
  (d (Vd  voltage (quantity-space MOS-positive-voltage-qspace))
     (Ids current)))
(component-variables
  (Vsd       voltage     (quantity-space MOS-voltage-qspace))
  (Vds       voltage     (quantity-space MOS-voltage-qspace))
  (Vgs       voltage     (quantity-space MOS-voltage-qspace))
  (Vgd       voltage     (quantity-space MOS-voltage-qspace))
  (channel1  resistance  (quantity-space (0 C1*))
                         (landmark-symbol ch))
  (channel2  resistance  (quantity-space (0 C2*))
                         (landmark-symbol ch))
  (channel12 resistance  (quantity-space (0 C12* C3*))
                         (landmark-symbol ch))
  (channel   resistance  (quantity-space (0 Ch*))
                         (landmark-symbol ch))
  (Qg        charge      (quantity-space (0 Qg*)))
  (Cg        capacitance independent (quantity-space (0 Cg*))))
(constraints
 ((ADD Vsd Vd Vs) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                  (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                  (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                  (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                  (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
 ((MINUS Vsd Vds) (Vhi- Vhi) (Vhi-Vtn VhiVtp) (Vtp Vtn) (0 0)
                  (Vhi Vhi-) (VhiVtp Vhi-Vtn) (Vtn Vtp))
 ((ADD Vgd Vd Vg) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                  (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                  (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                  (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                  (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
 ((ADD Vgs Vs Vg) (Vhi- Vhi 0) (Vhi-Vtn VhiVtp 0) (Vhi-Vtn Vhi Vtn)
                  (Vtp Vtn 0) (Vtp Vhi VhiVtp)
                  (0 0 0) (0 Vtn Vtn) (0 VhiVtp VhiVtp) (0 Vhi Vhi)
                  (Vtn 0 Vtn) (Vtn VhiVtp Vhi)
                  (VhiVtp 0 VhiVtp) (VhiVtp Vtn Vhi) (Vhi 0 Vhi))
 ((S- Vgd channel1 (Vhi- C1*) (Vtp 0)))
 ((S- Vgs channel2 (Vhi- C2*) (Vtp 0)))
 ((ADD channel1 channel2 channel12) (0 0 0) (C1* 0 C12*)
                                    (0 C2* C12*) (C1* C2* C3*))
 ((S+ channel12 channel (0 0) (C12* Ch*)))
 ((MULT Vsd channel Isd) (Vhi Ch* Imax) (Vhi- Ch* Imax-))
 ((MINUS Ids Isd) (Imax- Imax) (0 0) (Imax Imax-))
 ;; Gate capacitance constraints
```

```
   ((MULT Vg Cg Qg) (Vhi Cg* Qg*))
   ((D/DT Qg Ig))
   ))

(define-component-implementation
  N-channel-source-at-Vss MOS-transistor
  "N channel transistor - source at Vss (0)"
  (terminal-variables
    (g (Vg  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ig  current (quantity-space base-quantity-space)))
    (s (Vs  voltage (quantity-space (0)))
       (Isd current))
    (d (Vd  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ids current)))
  (component-variables
    (Vsd     voltage     (quantity-space MOS-voltage-qspace))
    (channel resistance  (quantity-space (0 Ch*))
                         (landmark-symbol ch))
    (Qg      charge      (quantity-space (0 Qg*)))
    (Cg      capacitance independent (quantity-space (0 Cg*))))
  (constraints
   ((M- Vsd Vd) (Vhi- Vhi) (Vhi-Vtn VhiVtp) (Vtp Vtn) (0 0))
   ((CONSTANT Vs 0))
   ((S+ Vg channel (Vtn 0) (Vhi Ch*)))
   ((MULT Vsd channel Isd) (Vhi- Ch* Imax-))
   ((MINUS Ids Isd) (0 0) (Imax Imax-))
   ;; Gate capacitance constraints
   ((MULT Vg Cg Qg) (Vhi Cg* Qg*))
   ((D/DT Qg Ig))
   ))

(define-component-implementation
  P-channel-drain-at-Vdd MOS-transistor
  "P channel transistor - drain at Vdd (Vhi)"
  (terminal-variables
    (g (Vg  voltage (quantity-space MOS-positive-voltage-qspace))
       (Ig  current (quantity-space base-quantity-space)))
    (s (Vs  voltage (quantity-space MOS-positive-voltage-qspace))
       (Isd current))
    (d (Vd  voltage (quantity-space (0 Vhi)))
       (Ids current)))
  (component-variables
    (Vsd     voltage     (quantity-space MOS-voltage-qspace))
```

```
    (Vgd     voltage     (quantity-space MOS-voltage-qspace))
    (channel resistance  (quantity-space (0 Ch*))
                         (landmark-symbol ch))
    (Qg      charge      (quantity-space (0 Qg*)))
    (Cg      capacitance independent (quantity-space (0 Cg*))))
  (constraints
   ((M+ Vsd Vs) (Vhi- 0) (Vhi-Vtn Vtn) (Vtp VhiVtp) (0 Vhi))
   ((CONSTANT Vd Vhi))
   ((M+ Vgd Vg) (Vhi- 0) (Vhi-Vtn Vtn) (Vtp VhiVtp) (0 Vhi))
   ((S- Vgd channel (Vhi- Ch*) (Vtp 0)))
   ((MULT Vsd channel Isd) (Vhi- Ch* Imax-))
   ((MINUS Ids Isd) (0 0) (Imax Imax-))
   ;; Gate capacitance constraints
   ((MULT Vg Cg Qg) (Vhi Cg* Qg*))
   ((D/DT Qg Ig))
   ))


(define-component-interface
  P-Channel-Feedback
  "P-channel transistor with Vdd at drain" electrical
  (terminals g s)
  (quantity-spaces
    (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
              (current MOS-current-qspace))
    (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  1 P-Channel-Feedback
  "P-Channel-Feedback from P-Channel with drain at Vdd, and Vdd"
  (components
    (Vdd reference-voltage (ignore-qdir I))
    (Pt  (MOS-transistor (impl P-channel-drain-at-Vdd))
         (ignore-qdir Ids Isd Ig) (display Ids Vg)))
  (connections (w (Vdd t) (Pt d))
               (g (Pt g))
               (s (Pt s)))))


(define-component-interface
  Transmission-Gate
  "CMOS transmission gate (N, P in parallel)" electrical
  (terminals in out ctl ctl-bar)
```

```
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current MOS-current-qspace))
   (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  transistors Transmission-Gate
  "P-channel and N-channel transistors in parallel."
  (components
    (Pt  (MOS-transistor (impl P-channel-bidirectional))
         (display Ids Qg))
    (Nt  (MOS-transistor (impl N-channel-bidirectional))
         (display Ids Qg)))
  (connections (in  (Pt s) (Nt s))
               (out (Pt d) (Nt d))
               (ctl (Pt g))
               (ctl-bar (Nt g))))


(define-component-interface
  Inverter "Inverter composed from transistors" electrical
  (terminals in out)
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current MOS-current-qspace))
   (hierarchical-parents (voltage MOS-voltage-qspace))))

(define-component-implementation
  transistors Inverter
  "P-channel and N-channel transistors."
  (components
    (Vdd reference-voltage (ignore-qdir i))
    (Vss reference-voltage (ignore-qdir i))
    (Pt  (MOS-transistor (impl P-channel-drain-at-Vdd))
 (initable Qg Vg Vs) (ignore-qdir Ids Isd Ig)
 (display Ids Qg) (no-new-landmarks Ig))
    (Nt  (MOS-transistor (impl N-channel-source-at-Vss))
 (initable Qg Vg Vd) (ignore-qdir Ids Isd Ig)
 (display Ids Qg) (no-new-landmarks Ig))
    (S   (Split (impl equipotential-base-qspace))
 (ignore-qdir I I1 I2) (display V I)
 (no-new-landmarks I I1 I2)))
  (connections (w1  (Vdd t) (Pt d))
```

```
              (w2  (Vss t) (Nt s))
              (w3  (S s1)  (Pt g))
              (w4  (S s2)  (Nt g))
              (in  (S m))
              (out (Pt s)  (Nt d))))
```

B.3 Model Definitions

```
(define-component-interface
  ISC "Input selection circuit" electrical
  (quantity-spaces
   (defaults (voltage (0 Vhi) (parent MOS-voltage-qspace))
             (current base-quantity-space))
   (hierarchical-parents (voltage MOS-voltage-qspace))))


(define-component-implementation
  1 ISC
  "N-trans for input select, capacitor for output load."
  (components
    (RV1  reference-voltage)
    (RV2  reference-voltage)
    (RV3  reference-voltage (ignore-qdir i))
    (t1   (MOS-transistor (impl N-channel-bidirectional))
          (display Ids Vs))
    (inv  Inverter)
    (C    (capacitor (impl current-qspace))
          (ignore-qdir i i2)))
  (connections (w1 (RV1 t)   (t1 d))
       (w2 (RV2 t)   (t1 g))
       (w3 (t1 s)    (inv in))
       (w4 (inv out) (C t1))
       (w5 (RV3 t)   (c t2))))

(define-component-implementation
  2 ISC
  "N-trans for input select, P-trans for feedback."
  (components
    (RV1  reference-voltage)
    (RV2  reference-voltage)
    (RV3  reference-voltage (ignore-qdir I))
```

```
    (t1   (MOS-transistor (impl N-channel-bidirectional))
          (display Ids Vs))
    (t2   P-channel-feedback)
    (inv  Inverter)
    (C    (Capacitor (impl current-qspace)) (ignore-qdir i i2))
    (S    (Split (impl equipotential-current-qspace))
          (ignore-qdir I I1 I2) (display V)
          (no-new-landmarks I I1 I2)))
  (connections (w1 (RV1 t)   (t1 d))
               (w2 (RV2 t)   (t1 g))
               (w3 (t1 s)    (inv in) (t2 s))
               (w4 (inv out) (S m))
               (w5 (C t2)    (RV3 t))
               (w6 (S s1)    (t2 g))
               (w7 (S s2)    (C t1))))

(define-component-implementation
  3 ISC
  "Transmission-gate for ramp input."
  (components
    (RV1 reference-voltage)
    (RV2 reference-voltage)
    (RV3 reference-voltage (ignore-qdir I))
    (RV4 reference-voltage (ignore-qdir I))
    (Tm  transmission-gate (display Isd Vsd))
    (inv Inverter)
    (C   (Capacitor (impl base-qspace))
         (ignore-qdir i i2)))
  (connections (w1 (RV1 t)   (Tm in))
               (w2 (RV2 t)   (Tm ctl))
               (w3 (RV4 t)   (Tm ctl-bar))
               (w4 (Tm out)  (inv in))
               (w5 (inv out) (C t1))
               (w6 (C t2)    (RV3 t))))
```

B.4 Design Specifications

```
(for-component X (inverter MOS-transistor)
    (prohibited (((((X Vg) ((0 Vhi) std)))) true)))

(for-component X (inverter MOS-transistor)
  (conditionally ((((((X Vg) (0 std)))
                     (((X Vg) ((0 Vhi) std)))) true))
    (required (((((X Vg) (Vhi std))))
              true))))

(for-component X (ISC)
  (for-component Y (X inverter MOS-transistor)
    (conditionally ((((((Y Vg) (Vhi ign))
                       ((X Nt Vg) (0 std)))) true))
      (prohibited (((((Y Vg) ((0 Vhi) std))
                     ((X Nt Vg) (0 std))))
               true)))))

(for-component X (ISC)
  (for-component Y (X inverter MOS-transistor)
    (conditionally ((((((X Nt Vg) (Vhi std))
                       ((X Nt Vd) (0 std)))) true))
      (required (((((Y Vg) (0 std))))
                true)))))
```

Appendix C

Electric Motor Example

C.1 Quantity Space Definitions

```
(define-quantity-space position-X-qspace (X180- X- 0 X+ X180+)
  (conservation-correspondences (X180- X180+) (X- X+)))

(define-quantity-space positive-qspace (0 inf))

(define-quantity-space polarity-qspace (South 0 North)
  (conservation-correspondences (South North)))

(define-quantity-space angular-force-qspace (F- 0 F+)
  (conservation-correspondences (F- F+)))

(define-quantity-space orientation-qspace (Omax- 0 Omax+)
  (conservation-correspondences (Omax- Omax+)))

(define-quantity-space orientation-60-qspace
  (Omax- O60- 0 O60+ Omax+)
  (conservation-correspondences (Omax- Omax+) (O60- O60+)))

(define-quantity-space motor-current-qspace (Imax- 0 Imax+)
  (conservation-correspondences (Imax- Imax+)))

(define-quantity-space motor-velocity-qspace (minf 0 V* inf))

(define-quantity-space position-90-qspace (X180- X90- 0 X90+ X180+)
  (conservation-correspondences (X180- X180+) (X90- X90+)))

(define-quantity-space position-30-qspace
  (X180- X150- X120- X90- X60- X30- 0
   X30+ X60+ X90+ X120+ X150+ X180+)
  (conservation-correspondences (X180- X180+) (X150- X150+)
    (X120- X120+) (X90- X90+) (X60- X60+) (X30- X30+)))

(define-quantity-space motor-velocity-qspace (minf 0 V*))
```

```
(define-quantity-space motor-lateral-force-qspace
  (minf F-lat- 0 F-lat+ inf)
  (conservation-correspondences (F-lat- F-lat+)))
```

C.2 Component Definitions

```
(define-component-interface
  rotor "Rotor for electromechanical motor" mechanical-rotation
  (terminals shaft magnet))

(define-component-implementation
  1 rotor ""
  (terminal-variables
    (shaft  (F-ang force)
            (V     velocity)
            (F-lat (mechanical-translation force))
            (I     (electrical current)))
    (magnet (F-mag (magnetic force))))
  (component-variables
    (Polarity    (magnetic force) (quantity-space polarity-qspace))
    (Repulsion   force (quantity-space angular-force-qspace))
    (PotentialF  force (quantity-space angular-force-qspace))
    (Orientation displacement (quantity-space orientation-qspace)))
  (constraints
    ((M+ I Polarity) (Imax- South) (0 0) (Imax+ North))
    ((mult F-mag Polarity Repulsion)     (0 0 0) (South North F-)
               (North South F-) (South South F+) (North North F+))
    ((minus Repulsion PotentialF)        (F- F+) (0 0) (F+ F-))
    ((mult PotentialF Orientation F-ang) (F- Omax- F+) (F- Omax+ F-)
                                (0 0 0) (F+ Omax- F-) (F+ Omax+ F+))
    ((U- V F-lat (0 0))                  (minf minf) (inf minf))))


(define-component-interface
  magnet "Magnet for electromechanical motor" magnetic
  (terminals north south))

(define-component-implementation
  1 magnet ""
  (terminal-variables
```

```
    (north (F-north force (quantity-space polarity-qspace)))
    (south (F-south force (quantity-space polarity-qspace))))
  (constraints ((constant F-north north))
               ((constant F-south south))))


(define-component-interface
  one-terminal-shaft
  "Single terminal shaft for electromechanical motor"
  mechanical-rotation
  (terminals t))

(define-component-implementation
  1 one-terminal-shaft ""
  (terminal-variables
    (t (F-ang force)
       (V     velocity (quantity-space motor-velocity-qspace))
       (F-lat (mechanical-translation force))
       (I     (electrical current))))
  (component-variables
    (X         displacement)
    (Cum-F-ang force (quantity-space angular-force-qspace))
    (KE        energy))
  (constraints ((d/dt X V))
               ((d/dt V Cum-F-ang))
               ((minus Cum-F-ang F-ang) (F- F+) (0 0) (F+ F-))
               ((U+ V KE (0 0))         (minf inf) (inf inf))))

(define-component-interface
  2-field-rotor
  "Rotor for electromechanical motor, two magnetic fields"
  mechanical-rotation
  (terminals shaft magnet- magnet+))

(define-component-implementation
  1 2-field-rotor
  "Rotor for electromechanical motor, two magnetic fields"
  (terminal-variables
    (shaft   (F-ang force)
             (V     velocity (quantity-space motor-velocity-qspace))
             (F-lat (mechanical-translation force)
                    (quantity-space motor-lateral-force-qspace))
             (I     (electrical current)))
```

```
    (magnet+ (F-mag+ (magnetic force)))
    (magnet- (F-mag- (magnetic force))))
  (component-variables
    (Polarity    (magnetic force) (quantity-space polarity-qspace))
    (Repulsion-  force (quantity-space angular-force-qspace))
    (Repulsion+  force (quantity-space angular-force-qspace))
    (PotentialF  force (quantity-space angular-force-qspace))
    (Orientation displacement (quantity-space orientation-qspace)))
  (constraints
    ((M+ I Polarity) (Imax- South) (0 0) (Imax+ North))
    ((mult F-mag- Polarity Repulsion-)   (0 0 0) (South North F-)
              (North South F-) (South South F+) (North North F+))
    ((mult F-mag+ Polarity Repulsion+)   (0 0 0) (South North F-)
              (North South F-) (South South F+) (North North F+))
    ((add Repulsion+ PotentialF Repulsion-)
                                    (0 0 0) (F- F+ F+) (F+ F- F-))
    ((mult PotentialF Orientation F-ang) (F- Omax- F+) (F- Omax+ F-)
                                (0 0 0) (F+ Omax- F-) (F+ Omax+ F+))
    ((U- V F-lat (0 0))                 (minf minf) (V* F-lat-))))


(define-component-implementation
  2 one-terminal-shaft ""
  (terminal-variables
    (t (F-ang force)
       (V     velocity)
       (F-lat (mechanical-translation force)
              (quantity-space motor-lateral-force-qspace))
       (I     (electrical current))))
  (component-variables
    (X         displacement)
    (Cum-F-ang force (quantity-space angular-force-qspace)))
  (constraints ((d/dt X V))
               ((d/dt V Cum-F-ang))
               ((minus Cum-F-ang F-ang) (F- F+) (0 0) (F+ F-))))

(define-component-interface
  2-terminal-shaft "Two terminal motor shaft" mechanical-rotation
  (terminals t1 t2)
  (quantity-spaces
    (defaults
      (velocity                     motor-velocity-qspace)
      (force                        angular-force-qspace)
```

```
      ((mechanical-translation force) motor-lateral-force-qspace)
      ((electrical current)           motor-current-qspace))))

(define-component-implementation
  1 2-terminal-shaft "Two terminal motor shaft"
  (terminal-variables (t1 (F-ang1 force)
                          (V1     velocity)
                          (F-lat1 (mechanical-translation force))
                          (I1     (electrical current)))
                      (t2 (F-ang2 force)
                          (V2     velocity)
                          (F-lat2 (mechanical-translation force))
                          (I2     (electrical current))))
  (component-variables (X         displacement)
                       (Cum-F-ang force)
                       (Cum-F-lat (mechanical-translation force)))
  (mode-variables
    (position
      (positive <- (or (X ((0 X180+) nil))
                       (X (X180+ dec))
                       (X (X180+ std))))
      (negative <- (or (X ((X180- 0) nil))
                       (X (X180- inc))
                       (X (X180- std))))
      (:discontinuous-transition <- (X (X180+ inc))
                                 negative (X (X180- inc)))
      (:discontinuous-transition <- (X (X180- dec))
                                 positive (X (X180+ dec)))))
  (constraints
   ((d/dt X V1))
   ((d/dt V1 Cum-F-ang))
   ((sum-zero F-ang1 F-ang2 Cum-F-ang) (F- F- F+) (0 0 0) (F+ F+ F-))
   ((minus I1 I2)                 (Imax- Imax+) (0 0) (Imax+ Imax-))
   ((equal V1 V2)                 (0 0) (V* V*))
   ((constant Cum-F-lat 0))
   ((position positive) -> ((S- V1 I1 (0 Imax+) (V* 0))))
   ((position negative) -> ((S- V1 I2 (0 Imax+) (V* 0))))))

(define-component-interface
  3-terminal-shaft "Three terminal motor shaft" mechanical-rotation
  (terminals t1 t2 t3)
  (quantity-spaces
    (defaults
```

```
      (velocity                       motor-velocity-qspace)
      (force                          angular-force-qspace)
      ((mechanical-translation force) motor-lateral-force-qspace)
      ((electrical current)           motor-current-qspace))))

(define-component-implementation
  1 3-terminal-shaft "Three terminal motor shaft"
  (terminal-variables (t1 (F-ang1 force)
                          (V1     velocity)
                          (F-lat1 (mechanical-translation force))
                          (I1     (electrical current)))
                      (t2 (F-ang2 force)
                          (V2     velocity)
                          (F-lat2 (mechanical-translation force))
                          (I2     (electrical current)))
                      (t3 (F-ang3 force)
                          (V3     velocity)
                          (F-lat3 (mechanical-translation force))
                          (I3     (electrical current))))
  (component-variables (X         displacement)
                       (Cum-F-ang force)
                       (Cum-F-lat (mechanical-translation force)))
  (mode-variables
    (position
      (X0toX60+     <- (or (X ((0 X60+) nil))
                           (X (X60+ dec))
                           (X (0 inc))
                           (X (X60+ std))
                           (X (0 std))))
      (X60+toX120+  <- (or (X ((X60+ X120+) nil))
                           (X (X120+ dec))
                           (X (X60+ inc))
                           (X (X120+ std))
                           (X (X60+ std))))
      (X120+toX180+ <- (or (X ((X120+ X180+) nil))
                           (X (X180+ dec))
                           (X (X120+ inc))
                           (X (X180+ std))
                           (X (X120+ std))))
      (X180-toX120- <- (or (X ((X180- X120-) nil))
                           (X (X120- dec))
                           (X (X180- inc))
                           (X (X120- std))
```

```
                          (X (X180- std))))
      (X120-toX60-  <- (or (X ((X120- X60-) nil))
                          (X (X60- dec))
                          (X (X120- inc))
                          (X (X60- std))
                          (X (X120- std))))
      (X60-toX0     <- (or (X ((X60- 0) nil))
                          (X (0 dec))
                          (X (X60- inc))
                          (X (X60- std))))
      (:discontinuous-transition <- (X (X180+ inc))
                                    X180-toX120- (X (X180- inc)))
      (:discontinuous-transition <- (X (X180- dec))
                                    X180+toX120+ (X (X180+ dec)))))
  (constraints
    ((d/dt X V1))
    ((d/dt V1 Cum-F-ang))
    ((sum-zero F-ang1 F-ang2 F-ang3 Cum-F-ang)
     (F- F- F- F+) (0 0 0 0) (F+ F+ F+ F-))
    ((equal V1 V2)                 (0 0) (V* V*))
    ((equal V1 V3)                 (0 0) (V* V*))
    ((constant Cum-F-lat 0))
    ((position X0toX60+) ->
     ((S- V1 I1 (0 Imax+) (V* 0)))
     ((equal I1 I2) (Imax- Imax-) (0 0) (Imax+ Imax+))
     ((minus I1 I3) (Imax- Imax+) (0 0) (Imax+ Imax-)))
    ((position X60+toX120+) ->
     ((S- V1 I1 (0 Imax+) (V* 0)))
     ((equal I2 I3) (Imax- Imax-) (0 0) (Imax+ Imax+))
     ((minus I1 I3) (Imax- Imax+) (0 0) (Imax+ Imax-)))
    ((position X120+toX180+) ->
     ((S- V3 I3 (0 Imax+) (V* 0)))
     ((equal I1 I3) (Imax- Imax-) (0 0) (Imax+ Imax+))
     ((minus I1 I2) (Imax- Imax+) (0 0) (Imax+ Imax-)))
    ((position X180-toX120-) ->
     ((S- V3 I3 (0 Imax+) (V* 0)))
     ((equal I1 I2) (Imax- Imax-) (0 0) (Imax+ Imax+))
     ((minus I1 I3) (Imax- Imax+) (0 0) (Imax+ Imax-)))
    ((position X120-toX60-) ->
     ((S- V2 I2 (0 Imax+) (V* 0)))
     ((equal I2 I3) (Imax- Imax-) (0 0) (Imax+ Imax+))
     ((minus I1 I3) (Imax- Imax+) (0 0) (Imax+ Imax-)))
    ((position X120-toX60-) ->
```

```
    ((S- V2 I2 (0 Imax+) (V* 0)))
    ((equal I1 I3) (Imax- Imax-) (0 0) (Imax+ Imax+))
    ((minus I1 I2) (Imax- Imax+) (0 0) (Imax+ Imax-)))))
```

C.3 Model Definitions

```
(define-component-interface
  motor "Electromechanical motor" mechanical
  (quantity-spaces
    (defaults ((magnetic force)            polarity-qspace)
              ((electrical current)        motor-current-qspace)
              ((mechanical-rotation force) angular-force-qspace))))

(define-component-implementation
  1 motor
  "Single magnet, single rotor"
  (quantity-spaces
    (default
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (component-variables (PE energy (quantity-space (0 PE+ PE*)))
                       (TE energy))
  (mode-variables
    (position
      (positive <- (or ((shaft X) ((0 X180+) nil))
                       ((shaft X) (X180+ dec))
                       ((shaft X) (X180+ std))))
      (negative <- (or ((shaft X) ((X180- 0) nil))
                       ((shaft X) (X180- inc))
                       ((shaft X) (X180- std))))
      (:discontinuous-transition <- ((shaft X) (X180+ inc))
                                    negative ((shaft X) (X180- inc)))
      (:discontinuous-transition <- ((shaft X) (X180- dec))
                                    positive ((shaft X) (X180+ dec)))))
  (components (magnet magnet)
              (rotor (rotor (impl 1)) (no-new-landmarks F-lat F-ang)
                     (ignore-qdir F-ang)
                     (quantity-spaces (X position-X-qspace)))
              (shaft (one-terminal-shaft (impl 1))
                     (no-new-landmarks F-lat F-ang Cum-F-ang)
                     (ignore-qdir F-ang Cum-F-ang)
                     (quantity-spaces (X position-X-qspace)))))
```

```
  (constraints
    ((ADD PE (shaft KE) TE)              (0 0 0))
    ((constant TE))
    ((constant (shaft I) Imax+))
    ((position positive)
      ->
      ((U- (shaft X) (rotor Orientation) (X+ Omax+))
                                         (0 0) (X180+ 0))
      ((S- (shaft X) PE (0 PE*) (X180+ 0)) (X+ PE+)))
    ((position negative)
      ->
      ((U+ (shaft X) (rotor Orientation) (X- Omax-))
                                         (X180- 0) (0 0))
      ((S+ (shaft X) PE (X180- 0) (0 PE*)) (X- PE+))))
  (connections (c1 (rotor magnet) (magnet north))
               (c2 (rotor shaft)  (shaft t))))

(define-component-implementation
  2 motor
  "Double magnet, single rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (mode-variables
    (position
      (positive <- (or ((shaft X) ((0 X180+) nil))
                       ((shaft X) (X180+ dec))
                       ((shaft X) (X180+ std))))
      (negative <- (or ((shaft X) ((X180- 0) nil))
                       ((shaft X) (X180- inc))
                       ((shaft X) (X180- std))))
      (:discontinuous-transition <- ((shaft X) (X180+ inc))
                                 negative ((shaft X) (X180- inc)))
      (:discontinuous-transition <- ((shaft X) (X180- dec))
                                 positive ((shaft X) (X180+ dec)))))
  (components
    (magnet1 magnet)
    (magnet2 magnet)
    (rotor   2-field-rotor (no-new-landmarks F-lat F-ang)
             (ignore-qdir F-ang))
    (shaft   (one-terminal-shaft (impl 2))
             (no-new-landmarks F-lat F-ang Cum-F-ang)
             (ignore-qdir F-ang Cum-F-ang)
```

```
            (quantity-spaces (X position-90-qspace))))
  (constraints
    ((position positive)
      ->
      ((U- (shaft X) (rotor Orientation) (X90+ Omax+))
                                         (0 0) (X180+ 0))
      ((S- (shaft V) (shaft I) (0 Imax+) (V* 0))))
    ((position negative)
      ->
      ((U+ (shaft X) (rotor Orientation) (X90- Omax-))
                                         (X180- 0) (0 0))
      ((S+ (shaft V) (shaft I) (0 Imax-) (V* 0)))))
  (connections (c1 (rotor magnet+) (magnet1 north))
               (c2 (rotor magnet-) (magnet2 south))
               (c3 (rotor shaft)   (shaft t))))

(define-component-implementation
  3 motor
  "Double magnet, double rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation force)    angular-force-qspace)
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (mode-variables
    (position
      (positive <- (or ((shaft X) ((0 X180+) nil))
                       ((shaft X) (X180+ dec))
                       ((shaft X) (X180+ std))
                       ((shaft X) (0 inc))
                       ((shaft X) (0 std))
                       ))
      (negative <- (or ((shaft X) ((X180- 0) nil))
                       ((shaft X) (X180- inc))
                       ((shaft X) (X180- std))
                       ((shaft X) (0 dec))
                       ((shaft X) (0 std))
                       ))
      (:discontinuous-transition <- ((shaft X) (X180+ inc))
                                    negative ((shaft X) (X180- inc)))
      (:discontinuous-transition <- ((shaft X) (X180- dec))
                                    positive ((shaft X) (X180+ dec)))))
  (components
    (magnet1 magnet)
```

```
    (magnet2 magnet)
    (rotor1  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang)
             (quantity-spaces (Orientation orientation-qspace)))
    (rotor2  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
             (ignore-qdir F-ang)
             (quantity-spaces (Orientation orientation-qspace)))
    (shaft   (2-terminal-shaft (impl 1))
             (no-new-landmarks F-lat1 F-lat2 F-ang1 F-ang2
                               F-ang-sum Cum-F-ang)
             (ignore-qdir F-ang1 F-ang2 F-ang-sum Cum-F-ang)
             (quantity-spaces (X position-90-qspace))))
  (constraints
    ((position positive)
      -> ((U- (shaft X) (rotor1 Orientation) (X90+ Omax+))
                                             (0 0) (X180+ 0))
         ((U+ (shaft X) (rotor2 Orientation) (X90+ Omax-))
                                             (0 0) (X180+ 0)))
    ((position negative)
      -> ((U+ (shaft X) (rotor1 Orientation) (X90- Omax-))
                                             (X180- 0) (0 0))
         ((U- (shaft X) (rotor2 Orientation) (X90- Omax+))
                                             (X180- 0) (0 0))))
  (connections (c1 (rotor1 magnet+) (rotor2 magnet+) (magnet1 north))
               (c2 (rotor1 magnet-) (rotor2 magnet-) (magnet2 south))
               (c3 (rotor1 shaft) (shaft t1))
               (c4 (rotor2 shaft) (shaft t2))))

(define-component-implementation
  4 motor
  "Double magnet, triple rotor"
  (quantity-spaces
    (defaults
      ((mechanical-rotation force)    angular-force-qspace)
      ((mechanical-rotation velocity) motor-velocity-qspace)))
  (mode-variables
    (position
      (X0toX60+     <- (or ((shaft X) ((0 X60+) nil))
                           ((shaft X) (X60+ dec))
                           ((shaft X) (0 inc))
                           ((shaft X) (X60+ std))
                           ((shaft X) (0 std))))
      (X60+toX120+  <- (or ((shaft X) ((X60+ X120+) nil))
```

```
                        ((shaft X) (X120+ dec))
                        ((shaft X) (X60+ inc))
                        ((shaft X) (X120+ std))
                        ((shaft X) (X60+ std))))
    (X120+toX180+ <- (or ((shaft X) ((X120+ X180+) nil))
                        ((shaft X) (X180+ dec))
                        ((shaft X) (X120+ inc))
                        ((shaft X) (X180+ std))
                        ((shaft X) (X120+ std))))
    (X180-toX120- <- (or ((shaft X) ((X180- X120-) nil))
                        ((shaft X) (X120- dec))
                        ((shaft X) (X180- inc))
                        ((shaft X) (X120- std))
                        ((shaft X) (X180- std))))
    (X120-toX60-  <- (or ((shaft X) ((X120- X60-) nil))
                        ((shaft X) (X60- dec))
                        ((shaft X) (X120- inc))
                        ((shaft X) (X60- std))
                        ((shaft X) (X120- std))))
    (X60-toX0     <- (or ((shaft X) ((X60- 0) nil))
                        ((shaft X) (0 dec))
                        ((shaft X) (X60- inc))
                        ((shaft X) (X60- std))))
    (:discontinuous-transition <- ((shaft X) (X180+ inc))
                          X180-toX120- ((shaft X) (X180- inc)))
    (:discontinuous-transition <- ((shaft X) (X180- dec))
                          X180+toX120+ ((shaft X) (X180+ dec)))))
(components
  (magnet1 magnet)
  (magnet2 magnet)
  (rotor1  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
           (ignore-qdir F-ang) (display I Orientation Polarity)
           (quantity-spaces (Orientation orientation-60-qspace)))
  (rotor2  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
           (ignore-qdir F-ang) (display I Orientation)
           (quantity-spaces (Orientation orientation-60-qspace)))
  (rotor3  2-field-rotor (no-new-landmarks F-lat F-ang Orientation)
           (ignore-qdir F-ang) (display I Orientation)
           (quantity-spaces (Orientation orientation-60-qspace)))
  (shaft   (3-terminal-shaft (impl 1))
           (no-new-landmarks F-lat1 F-lat2 F-lat3
                             F-ang1 F-ang2 F-ang3 Cum-F-ang)
           (display V1 X Cum-F-lat Cum-F-ang Position)
```

```
            (ignore-qdir F-ang1 F-ang2 F-ang3 Cum-F-ang)
            (quantity-spaces (X position-30-qspace))))
  (constraints
    ((position X0toX60+)
     -> ((M+ (shaft X) (rotor1 Orientation)) (0 0) (X60+ O60+))
        ((M- (shaft X) (rotor2 Orientation)) (0 O60+) (X60+ 0))
        ((U+ (shaft X) (rotor3 Orientation) (X30+ Omax-))
                                            (0 O60-) (X60+ O60-)))
    ((position X60+toX120+)
     -> ((U- (shaft X) (rotor1 Orientation) (X90+ Omax+))
                                            (X60+ O60+) (X120+ O60+))
        ((M- (shaft X) (rotor2 Orientation)) (X60+ 0) (X120+ O60-))
        ((M+ (shaft X) (rotor3 Orientation)) (X60+ O60-) (X120+ 0)))
    ((position X120+toX180+)
     -> ((M- (shaft X) (rotor1 Orientation)) (X120+ O60+) (X180+ 0))
        ((U+ (shaft X) (rotor2 Orientation) (X150+ Omax-))
                                           (X120+ O60-) (X180+ O60-))
        ((M+ (shaft X) (rotor3 Orientation)) (X120+ 0) (X180+ O60+)))
    ((position X180-toX120-)
     -> ((M- (shaft X) (rotor1 Orientation)) (X180- 0) (X120- O60-))
        ((M+ (shaft X) (rotor2 Orientation)) (X180- O60-) (X120- 0))
        ((U- (shaft X) (rotor3 Orientation) (X150- Omax+))
                                           (X180- O60+) (X120- O60+)))
    ((position X120-toX60-)
     -> ((U+ (shaft X) (rotor1 Orientation) (X90- Omax-))
                                            (X120- O60-) (X60- O60-))
        ((M+ (shaft X) (rotor2 Orientation)) (X120- 0) (X60- O60+))
        ((M- (shaft X) (rotor3 Orientation)) (X120- O60+) (X60- 0)))
    ((position X60-toX0)
     -> ((M+ (shaft X) (rotor1 Orientation)) (X60- O60-) (0 0))
        ((U- (shaft X) (rotor2 Orientation) (X30- Omax+))
                                            (X60- O60+) (0 O60+))
        ((M- (shaft X) (rotor3 Orientation)) (X60- 0) (0 O60-))))
  (connections (c1 (rotor1 magnet+) (rotor2 magnet+)
                   (rotor3 magnet+) (magnet1 north))
               (c2 (rotor1 magnet-) (rotor2 magnet-)
                   (rotor3 magnet-) (magnet2 south))
               (c3 (rotor1 shaft) (shaft t1))
               (c4 (rotor2 shaft) (shaft t2))
               (c5 (rotor3 shaft) (shaft t3))))
```

C.4 Design Specifications

```
(for-component S (shaft)
  (conditionally ((((((S V) (0 ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (conditionally ((((((S V) ((0 V*) ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (conditionally ((((((S V) (V* ign)))) true))
    (required (((((S V) (V* std)))) true))))

(for-component S (shaft)
  (prohibited (((((S V) ((0 inf) ign))
                 ((S Cum-F-lat) ((0 inf) std))))
          true))
```

Appendix D

Behavior Abstraction Relations

D.1 Abstraction Relation Table

Relation	Space	Defined in Terms of
$\sqsubseteq_c$	component types	assumed
$\sqsubseteq_n$	variable names	component types
$\sqsubseteq_t$	variable types	generic variable types
$\sqsubseteq_v$	variable references	$\sqsubseteq_n$, $\sqsubseteq_t$
$\sqsubseteq_x$	values	qualitative/quantitative values
$\sqsubseteq_s$	states	$\sqsubseteq_v$, $\sqsubseteq_x$
$\sqsubseteq_b$	behaviors	$\sqsubseteq_s$
$\sqsubseteq_\sigma$	scenarios	$\sqsubseteq_b$

Table D.1: Abstraction Relation Summary

D.2 Abstraction Relation Definitions

The relation $\sqsubseteq_n$ (read "is a variable name less general or equal to") partially orders the space of variable names. For variable name $n = (n_1, \ldots, n_k)$, $n' = (n'_1, \ldots, n'_l)$

$$n \sqsubseteq_n n' \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : n' \rightarrow n \text{ such that} \\ \\ \forall\ n'_i \in\ n', \text{ if } \mathcal{F}(n'_i) = n_{j_i}, \text{ then} \\ \qquad j_i < j_{i+1}, \quad (\textit{Order Preservation and Uniqueness}) \\ \qquad n'_i \text{ is a generalization of } n_{j_i}, \quad (\textit{Name Abstraction}) \end{cases}$$

The relation $\sqsubseteq_v$ (read "is a variable less general or equal to") partially orders the space of variable references (names and types). For variables

$v = (n, t)$ and $v' = (n', t')$,

$$v \sqsubseteq_v v' \Leftrightarrow \begin{cases} n \sqsubseteq_n n', \\ \\ t \sqsubseteq_t t'. \end{cases}$$

The relation $\sqsubseteq_x$ (read "is a value less general than or equal to") partially orders the space of variable values. Considering only the magnitudes of the qualitative values, for point values x and y,

$$x \sqsubseteq_x y \Leftrightarrow x = y;$$

For point value x and open interval value (y_1, y_2),

$$x \sqsubseteq_x (y_1, y_2) \Leftrightarrow y_1 < x \wedge x < y_2;$$

For open interval values (x_1, x_2) and (y_1, y_2),

$$(x_1, x_2) \sqsubseteq_x (y_1, y_2) \Leftrightarrow y_1 \leq x_1 \wedge x_2 \leq y_2.$$

The direction of change values `dec`, `std`, and `inc` are all pairwise unordered, and `nil` is more general than the other three values. To complete the definition of $\sqsubseteq_x$, $x \sqsubseteq_x y$ if the magnitude relationships described above hold, and either the direction of change of x and y are the same or the direction of change of y is `nil`.

The relation $\sqsubseteq_s$ (read "is a state less general than or equal to") partially orders the space of states and partial states. For (partial) state s with variable set $\mathcal{V}_s$, and partial state p with variable set $\mathcal{V}_p$,

$$s \sqsubseteq_s p \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : \mathcal{V}_p \rightarrow \mathcal{V}_s \text{ such that} \\ \\ \forall\ v \in \mathcal{V}_p, \\ \quad \mathcal{F}(v) \sqsubseteq_v v, \quad (\textit{Variable Abstraction}) \\ \quad s(\mathcal{F}(v)) \sqsubseteq_x p(v), \quad (\textit{Value Abstraction}) \\ \\ \forall\ v_1, v_2 \in \mathcal{V}_p, v_1 \neq v_2 \Rightarrow \mathcal{F}(v_1) \neq \mathcal{F}(v_2). \quad (\textit{Uniqueness}) \end{cases}$$

The relation $\sqsubseteq_b$ (read "is a behavior less general than or equal to") partially orders the space of behaviors. For behavior $b = \langle s_1, s_2, \ldots \rangle$ and behavior $b' = \langle s'_1, s'_2, \ldots \rangle$,

$$b \sqsubseteq_b b' \Leftrightarrow \begin{cases} \exists\ \mathcal{F} : b' \rightarrow b \text{ such that} \\ \\ \forall\ s'_i \in\ b',\ \mathcal{F}(s'_i) = s_{j_i}, \\ \qquad j_i < j_{i+1}, \quad (Order\ \ Preservation) \\ \qquad s_{j_i} \sqsubseteq_s s'_i, \quad (State\ \ Abstraction) \\ \\ \forall\ s'_i, s'_j \in\ b', i \neq j \Rightarrow \mathcal{F}(s'_i) \neq \mathcal{F}(s'_j). \quad (Uniqueness) \end{cases}$$

The relation $\sqsubseteq_\sigma$ (read "is a scenario less general than or equal to") partially orders the space of scenarios. For scenarios $\sigma = (p, \beta)$ and $\sigma' = (p', \beta')$, with $p = \langle p_1, \ldots \rangle$ and $p' = \langle p'_1, \ldots \rangle$,

$$\sigma \sqsubseteq_\sigma \sigma' \Leftrightarrow \begin{cases} p \sqsubseteq_b p' \text{ via mapping } \mathcal{F}' : p' \rightarrow p\ (BehaviorAbstraction) \\ \\ \beta \Rightarrow \mathcal{F}'(\beta')\ \ (ConditionAbstraction). \end{cases}$$

where $\mathcal{F}'(\beta')$ denotes the rewriting of β' with respect to the mapping $\mathcal{F}' : p' \rightarrow p$ (i.e. variable reference $p'_i(v)$ in β' is replaced by $p_j(v)$, where $p_j = \mathcal{F}'(p'_i)$). Scenarios σ and σ' are equivalent if $p \sqsubseteq_b p'$ and $\beta \Leftrightarrow \mathcal{F}'(\beta')$.

Appendix E

Teleology Operators

E.1 Notation

Teleological operators are the language primitives for teleological descriptions. In the context of a design modification, a single teleological operator relates the unmodified design to the modified design in terms of the specification predicates. In the following definitions, ϕ_i are specification predicates, d and d' are designs (structure descriptions), δ a design modification such that d' is the design obtained by applying δ to d, and E and E$'$ are the envisionments of d and d', respectively.

E.2 Primitive Operators

$$\delta \textbf{ Guarantees } \phi \Leftrightarrow \begin{cases} \exists\ b \in \mathrm{E}, & \neg\phi, \\ \text{and} & \\ \forall\ b' \in \mathrm{E}', & \phi. \end{cases}$$

$$\delta \textbf{ unGuarantees } \phi \Leftrightarrow \begin{cases} \forall\ b \in \mathrm{E}, & \phi, \\ \text{and} & \\ \exists\ b' \in \mathrm{E}', & \neg\phi. \end{cases}$$

E.3 Composed Operators

E.3.1 Prevents

$$\delta \textbf{ Prevents } \phi \Leftrightarrow \begin{cases} \exists\ b \in \mathrm{E}, & \phi, \\ \text{and} & \\ \forall\ b' \in \mathrm{E}', & \neg\phi. \end{cases}$$

Operator **Prevents** can be expressed in terms of **Guarantees** as

$$\delta \textbf{ Prevents } \phi \Leftrightarrow \delta \textbf{ Guarantees } \neg\phi.$$

E.3.2 Introduces

$$\delta \textbf{ Introduces } \phi \Leftrightarrow \begin{cases} \forall\, b \in \mathrm{E}, & \neg\phi, \\ \text{ and} & \\ \exists\, b' \in \mathrm{E}', & \phi. \end{cases}$$

Introduces can be expressed in terms of **unGuarantees** as

$$\delta \textbf{ Introduces } \phi \Leftrightarrow \delta \textbf{ unGuarantees } \neg\phi.$$

E.3.3 Conditionally Guarantees

$$\delta \textbf{ Conditionally when } \{\phi_1\} \textbf{ Guarantees } \phi_2 \Leftrightarrow \begin{cases} \exists\, b \in \mathrm{E}, & \neg(\phi_1 \Rightarrow \phi_2), \\ \text{ and} & \\ \forall\, b' \in \mathrm{E}', & \phi_1 \Rightarrow \phi_2. \end{cases}$$

We can rewrite this operator in primitives as:

$$\delta \textbf{ Guarantees } \phi_1 \Rightarrow \phi_2.$$

E.3.4 Conditionally Prevents

We can define the operator **Conditionally Prevents**, or conditionally preventing a scenario as follows:

$$\delta \textbf{ Conditionally when } \{\phi_1\} \textbf{ Prevents } \phi_2 \Leftrightarrow \begin{cases} \exists\, b \in \mathrm{E}, & \neg(\phi_1 \Rightarrow \neg\phi_2), \\ \text{ and} & \\ \forall\, b' \in \mathrm{E}', & \phi_1 \Rightarrow \neg\phi_2. \end{cases}$$

We can rewrite this operator in primitives as:

$$\delta \textbf{ Guarantees } \phi_1 \Rightarrow \neg\phi_2.$$

E.3.5 Conditionally Introduces

We can define the operator **Conditionally Introduces**, or conditionally introducing a scenario as follows:

$$\delta \textbf{ Conditionally } \text{when } \{\phi_1\} \textbf{ Introduces } \phi_2 \Leftrightarrow \begin{cases} \forall\ b \in \mathrm{E}, \quad \phi_1 \Rightarrow \neg\phi_2), \\ \text{ and} \\ \exists\ b' \in \mathrm{E}', \quad \neg(\phi_1 \Rightarrow \neg\phi_2). \end{cases}$$

We can rewrite this operator in primitives as:

$$\delta \textbf{ unGuarantees } \phi_1 \Rightarrow \neg\phi_2.$$

Appendix F

CC BNF

F.1 Macros

`(define-component-interface`
    interface-name string domain interface-clause$^+$`)`
`(define-component-implementation`
    implementation-name interface-name string impl-clause$^+$`)`
`(define-configuration`
    config-name config-type-reference string config-clause$^+$`)`
`(define-quantity-space`
    qspace-name qspace [*parent-clause*] [*conservation-clause*]`)`

F.2 Lower-Level Items

cc-constraint-spec `::=`
    constraint-spec
    | `(`*constraint-mode-condition* `->` *constraint-spec*$^+$`)`
component-type-reference `::=`
    interface-name
    | `(`*interface-name component-type-reference-details*`)`
component-type-reference-details `::=`
    [*instance-implementation-clause*] [*instance-mode-clause*] *parm-val**
config-name `::=` *symbol*
config-type-reference `::=` *interface-name*
    | `(`*interface-name instance-implementation-clause*`)`
config-clause `::=`
    `(`*interface-name component-type-reference-details*`)`
    | `(`*instance-name component-type-reference-details*`)`
connection-name `::=` *symbol*
conservation-clause `::=`
    `(conservation-correspondences` *conservation-correspondence*$^+$`)`
conservation-correspondence `::=` `(`*lmark lmark*$^+$`)`

constraint ::= (*name var*$^+$ *other-info**)
constraint-mode-condition ::=
(*mode-variable-reference mode-value*)
| (AND *constraint-mode-condition*$^+$)
| (NOT *constraint-mode-condition*)
| (OR *constraint-mode-condition*$^+$)
constraint-spec ::= (*constraint corresponding-values**)
corresponding-values ::= (*lmark lmark*$^+$)
default-parent-clause ::= (hierarchical-parents *default-parent-spec*$^+$)
default-parent-spec ::= (*variable-type-spec qspace-spec*)
default-qspace-clause ::= (defaults *default-qspace-spec*$^+$)
default-qspace-spec ::= (*variable-type-spec qspace-spec*)
domain ::=
acoustic | electrical | hydraulic | mechanical
| mechanical-rotation | mechanical-translation | thermal
implementation-name ::= *symbol*
impl-clause ::=
(quantity-spaces [*default-qspace-clause*] [*default-parent-clause*])
| (terminal-variables (*terminal-name variable-spec*$^+$)$^+$)
| (component-variables *variable-spec*$^+$)
| (mode-variables (*mode-variable-name mode-value-spec*$^+$)$^+$)
| (constraints *cc-constraint-spec*$^+$)
| (components (*instance-name component-type-reference*
instance-option*)$^+$)
| (connections ([*connection-name*] *terminal-reference*$^+$)$^+$)
instance-implementation-clause ::= (impl *implementation-name*)
instance-mode-clause ::= (mode (*mode-variable-name mode-value*)$^+$)
instance-name ::= *symbol*
instance-option ::=
(display *variable-name*$^+$)
| (ignore-qdir *variable-name*$^+$)
| (no-new-landmarks *variable-name*$^+$)
| (quantity-spaces (*variable-name qspace-spec*)$^+$)
interface-clause ::=
(terminals *terminal-name*$^+$)
| (parameters *parameter-spec*$^+$)
| (quantity-spaces [*default-qspace-clause*] [*default-parent-clause*])
interface-name ::= *symbol*
mag ::= *lmark* | (*lmark lmark*)
mode-value-spec ::= *mode-value* | (*mode-value* <- *mode-value-condition*)
mode-value ::= *symbol*

mode-value-condition ::=
    (*variable-reference test-val*)
    | (AND *mode-value-condition*$^+$)
    | (NOT *mode-value-condition*)
    | (OR *mode-value-condition*$^+$)
mode-variable-name ::= *symbol*
mode-variable-reference ::= *mode-variable-name*
    | (*instance-name*$^+$ *mode-variable-name*)
parameter-default ::= *symbol*
parameter-name ::= *symbol*
parameter-spec ::= *parameter-name* | (*parameter-name parameter-default*)
parameter-value ::= *symbol*
parm-val ::= (*parameter-name parameter-value*)
parent-clause ::= (parent *qspace-name*)
qspace ::= ([minf] *lmark** 0 *lmark** [inf])
qspace-name ::= *symbol*
qspace-spec ::=
    qspace-name
    | *qspace* [*parent-clause*] [*conservation-clause*]
terminal-name ::= *symbol*
terminal-reference ::= *terminal-name* | (*instance-name terminal-name*)
test-dir ::= inc | std | dec | nil
test-val ::= (*mag test-dir*)
variable-name ::= *symbol*
variable-name-alist ::= ((*variable-name test-val*)$^+$)
variable-name-list ::= (*variable-name*$^+$)
variable-option ::=
    display | ignore-qdir | no-new-landmarks
    | (landmark-symbol *symbol*)
    | (quantity-space *qspace-spec*)
variable-reference ::= *variable-name* | (*instance-name*$^+$ *variable-name*)
variable-spec ::= (*variable-name variable-type-spec variable-option**)
variable-type-spec ::= *variable-type-name* | (*variable-type-name domain*)
variable-type-name ::= *symbol*

BIBLIOGRAPHY

[AS85] Harold Abelson, Gerald J. Sussman, *The Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[AEH*89] Harold Abelson, Michael Eisenberg, Matthew Halfant, Jacob Katzenelson, Elisha Sacks, Gerald J. Sussman, Jack Wisdom, Kenneth Yip, "Intelligence in Scientific Computing" in *Communications of the ACM*, Vol. 32, No. 5 (May 1989), pp. 546-562.

[Alf82] Mack W. Alford, "A Graph Model Based Approach to Specifications", in *Distributed Systems: Methods and Tools for Specification*, M. Paul and H. J. Siegert (eds.), *Lecture Notes in Computer Science* No. 190, G. Goos and J. Hartmanis (eds.), Springer-Verlag, New York, 1982, pp. 131-201.

[BSZ89] Catherine Baudin, Cecilia Sivard, Monte Zweben, "Model-Based Approach to Design Rationale Conservation", in *Proceedings of the 1989 Workshop on Model-Based Reasoning*, Detroit, August 20, 1989, pp. 88-90.

[BR86] Ted Biggerstaff, Charles Richter, "Reusability Framework, Assessment, and Directions", MCC (Non Proprietary) Technical Report No. STP-345-86, October 1986.

[BC85] Tom Bylander, B. Chandrasekaran, "Qualitative Reasoning About Physical Structures", in *SIGART Newsletter*, Special Section on Rea-

soning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 22-24.

[CLMG85] Richard R. Cantone, W. Brent Lander, Michael P. Marrone, Michael W. Gaynor, "Automated Knowledge Acquisition in INATE Using Component Information and Connectivity", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 32-34.

[CM88] K. Many Chandra, Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Mass., 1988.

[CM85] B. Chandrasekaran, Rob Milne, (eds.) "Special Section on Reasoning About Structure, Behavior, and Function", in *SIGART Newsletter*, No. 93 (July 1985), pp. 4-54.

[Cha90] B. Chandrasekaran, "Design Problem Solving: A Task Analysis", in *AI Magazine*, Vol. 11 No. 4 (Winter 1990), pp. 59-71.

[Che80] Brian F. Chellas, *Modal Logic: An Introduction*, Cambridge University Press, 1980.

[CM84] W. F. Clocksin, C. S. Mellish, *Programming in Prolog*, Second Edition, Springer-Verlag, Berlin, 1984.

[CF82] Paul R. Cohen, Edward A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. III, Addison-Wesley, Reading, MA.

[Dav85] Randall Davis, "Diagnostic Reasoning Based on Structure and Behavior", in *Qualitative Reasoning About Physical Systems*, Daniel G.

Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 347-410. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[DH88] Randall Davis, Walter Hamscher, "Model-based Reasoning: Troubleshooting" in *Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence*, Howard Shrobe, ed., Morgan Kaufmann Publishers, San Mateo, CA 1988, pp. 297-346.

[DeJ85] Kenneth De Jong, "Expert Systems for Diagnosing Complex System Failures", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 29-32.

[deK77] Johan de Kleer, "Multiple Representation of Knowledge in a Mechanics Problem Solver, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA, pp. 299-304.

[deK85] Johan de Kleer, "How Circuits Work", in *Qualitative Reasoning About Physical Systems*, Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 205-280. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[dKB82] Johan de Kleer, John Seely Brown, "Foundations of Envisioning", in *Proceedings of the Second National Conference on Artificial Intelligence*, pp. 434-437.

[dKB85] Johan de Kleer, John Seely Brown, "A Qualitative Physics Based on Confluences", in *Qualitative Reasoning About Physical Systems*,

Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 7-83. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[dKB86] Johan de Kleer, John Seely Brown, "Theories of Causal Ordering", in *Artificial Intelligence* Vol. 29, No. 1 (July 1986), pp. 33-61.

[Dow90] Keith Downing "The Qualitative Criticism of Circulatory Models via Bipartite Teleological Analysis", in *Proceedings of the 1990 Workshop on Qualitative Reasoning.*

[Doy86] Richard J. Doyle, "Constructing and Refining Causal Explanations from an Inconsistent Domain Theory", in *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, pp. 538-544.

[ES85] Allen Emerson, A. Prasad Sistla, "Deciding Full Time Branching Logic", in *Information and Control*, Vol. 61, No. 3, pp. 175-201.

[FKKP90] Kenneth W. Fiduk, Sally Kleinfeldt, Marta Kosarchyn, Eileen B. Perez, "Design Methodology Management - A CAD Framework Initiative Perspective", in *Proceedings of the* 27^{th} *ACM/IEEE Design Automation Conference*, June 24 - 28, 1990, Orlando.

[FHN72] Richard E. Fikes, P. E. Hart, Nils J. Nilsson, "Learning and Executing Generalized Plans", in *Artificial Intelligence*, Vol. 3, (1972) pp. 251-288.

[FN71] Richard E. Fikes, Nils J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", in *Artificial Intelligence*, Vol. 2, (1971) pp. 189-208.

[For85] Kenneth D. Forbus, "Qualitative Process Theory", in *Qualitative Reasoning About Physical Systems*, Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 85-168. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[Fra89] Bruno Franck, "Qualitative Engineering at Various Levels of Conception for Design and Evaluation of Structures", in *Proceedings of the Conference on Industrial and Engineering Application of AI and ES*, ACM, 1989.

[FD90] David W. Franke, Daniel L. Dvorak, "CC: Component Connection Models for Qualitative Simulation, A User's Guide", TR AI90-126, Dept. of Computer Sciences, The University of Texas at Austin.

[Gen85] Michael R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis", in *Qualitative Reasoning About Physical Systems*, Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 411-436. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[Goe89] Ashok Goel, B. Chandrasekaran, "Functional Representation of Designs and Redesign Problem Solving", in *Proceedings of the Eleventh Joint International Conference on Artificial Intelligence*, August 1989, Detroit, pp. 1388-1394.

[Gre83] James G. Greeno, "Conceptual Entities", in *Mental Models*, Dedre Gentner, Albert L. Stevens (eds.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1983, pp. 227-252.

[Gru91] Thomas Gruber, "Learning Why by Being Told What", in *IEEE Expert* Vol. 6, No. 4 (August 1991), pp. 65-75.

[Hel88] David H. Helman (ed.), *Analogical Reasoning*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1988.

[HW87] Michael R. Herbert, Gareth H. Williams, "An Initial Evaluation of the Detection and Diagnosis of Power Plant Faults Using a Deep Knowledge Representation of Physical Behaviour", in *Expert Systems*, Vol. 4, No. 2, (May 1987), pp. 90-99.

[Ham91] Walter Hamscher, "Modeling Digital Circuits for Troubleshooting", in *Artificial Intelligence* Vol. 51, Nos. 1-3 (October 1991), pp. 223-271.

[HA88] Michael N. Huhns, Ramon D. Acosta, "ARGO: A System for Design by Analogy", in *IEEE Expert* Vol. 3, No. 3 (Fall 1988), pp. 53-68.

[IEEE84] *IEEE Transactions on Software Engineering* issue on Software Reusability, Vol. SE-10, No. 5 (September 1984).

[IEEE87] *IEEE Software* issue on Reuse Tools, Vol. 4, No. 4 (July 1987).

[IEEE88] *IEEE Software* issue on CASE, Vol. 5, No. 2 (March 1988).

[IS86a] Yumi Iwasaki, Herbert A. Simon, "Causality in Device Behavior", in *Artificial Intelligence* Vol. 29, No. 1 (July 1986), pp. 3-32.

[IS86b] Yumi Iwasaki, Herbert A. Simon, "Theories of Causal Ordering: Reply to de Kleer and Brown", in *Artificial Intelligence* Vol. 29, No. 1 (July 1986), pp. 63-72.

[Ked85] Smadar Kedar-Cabelli, "Purpose-Directed Analogy", in *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, 1985, Irvine CA, pp. 150-159.

[Keu91] Anne M. Kueneke, "Device Representation", in *IEEE Expert* Special Track on Functional Reasoning, Vol. 6, No. 2 (April 1991), pp. 22-25.

[KTY91] Takashi Kiriyama, Tetsuo Tomiyama, Hiroyuki Yoshikawa, "Model Generation in Design", in *Working Papers for QR-91, Fifth International Workshop on Qualitative Reasoning about Physical Systems*, May 19-22, 1991, Austin, TX, pp. 93-108.

[Kow85] Thaddeus J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, 1985, Kluwer Academic Publishers, Boston.

[Kui82] Benjamin J. Kuipers, "Getting the Envisionment Right", in *Proceedings of the Second National Conference on Artificial Intelligence*, pp. 209-212.

[Kui85] Benjamin J. Kuipers, "Commonsense Reasoning about Causality: Deriving Behavior from Structure", in *Qualitative Reasoning About Physical Systems*, Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 169-203. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[Kui86] Benjamin J. Kuipers, "Qualitative Simulation", in *Artificial Intelligence*, Vol. 29, No. 3, (September 1986), pp. 289-338.

[Kui87] Benjamin J. Kuipers, "Qualitative Simulation as Causal Explanation", in *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 3 (May/June 1987), pp. 432-444.

[Kui87] Benjamin J. Kuipers, "Abstraction by Time-Scale in Qualitative Simulation", in Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, July 1987, pp. 621-625.

[Kui89a] Benjamin J. Kuipers, "Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge", *Automatica*, Vol 25, No. 4 (1989), pp. 571-585.

[Kui89b] Benjamin Kuipers, "Generic Mechanisms".

[MC80] Carver Mead, Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing, Reading Mass., 1980.

[MBR89] Proceedings of the 1989 Workshop on Model-Based Reasoning.

[MBR90] Proceedings of the 1990 Workshop on Model-Based Reasoning.

[MBR91] Proceedings of the 1991 Workshop on Model-Based Reasoning.

[McC88] Anna Marguerite McCann, "The Roman Port of Cosa", in *Scientific American*, Vol. 258, No. 3, (March 1988), pp. 102-109.

[Mil85] Robert Milne, "A Theory of Responsibilities", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 25-29.

[Moo89] Raymond Mooney, private communication.

[Mos85] Jack Mostow, "Towards a Better Model of the Design Process", in *AI Magazine*, Vol. 6, No. 1 (Spring 1985), pp. 44-57.

[MB87] Jack Mostow, Mike Barley, "Automated Reuse of Design Plans", in *Proceedings of the International Conference on Engineering Design*, August 1987, Boston, MA, pp. 632-647.

[Mosz85] Ben Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware", in *Computer*, Vol. 18, No. 5 (February 1985), pp. 10-19.

[NJA91] P. Pandurang Nayak, Leo Joskowicz, Sanjaya Addanki, "Automated Model Selection using Context-Dependent Behaviors", in *Working Papers for QR-91, Fifth International Workshop on Qualitative Reasoning about Physical Systems*, May 19-22, 1991, Austin, TX, pp. 10-24.

[Nil80] Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[Pol73] George Polya, *How To Solve It: A New Aspect of Mathematical Method*, Second Edition, Princeton University Press, Princeton, NJ, 1973.

[PF87] Ruben Prieto-Diaz, Peter Freeman, "Classifying Software for Reusability", in *IEEE Software*, Vol. 4, No. 1 (January 1987), pp. 6-16.

[Ray86] Joe Raymond, private communication.

[RS84] Charles Rich, Howard E. Shrobe, "Initial Report on a LISP Programmer's Apprentice", in *Interactive Programming Environments*, D. Barstow, H. Shrobe, E. Sandewall (eds.), McGraw-Hiull, New York, 1984, pp. 443-463. Reprinted from *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 6 (November 1978), pp. 456-467.

[RS89] Christopher K. Riesbeck, Roger C. Schank, "Case-Based Reasoning: An Overview", in *Inside Case-Based Reasoning*, Lawrence Erlbaum, Hillsdale, NJ, 1989.

[RK83] Ronald C. Rosenberg, Dean C. Karnopp, *Introduction to Physical System Dynamics*, McGraw-Hill, New York, 1983.

[Sac74] Earl D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces", in *Artificial Intelligence*, Vol. 5 (1974), pp. 115-135.

[Sac77] Earl D. Sacerdoti, *A Structure for Plans and Behavior*, Elsevier, New York, 1977.

[SJD85] Ethan A. Scarl, John R. Jamieson, Carl I. Delaune, "Process Monitoring and Fault Location at the Kennedy Space Center", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 38-44.

[Sch91] Roger C. Schank, "Where's the AI", in *AI Magazine*. Vol. 12, No. 4 (Winter 1991), pp. 38-49.

[SC85] V. Sembugamoorthy, B. Chandrasekaran, "Functional Representation of Devices as Deep Models", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 21-22.

[SS88] Lawrence K. Shapiro, Howard I. Shapiro, "Construction Cranes", in *Scientific American*, Vol. 258, No. 3, (March 1988), pp. 72-79.

[Sim81] Herbert A. Simon, *Sciences of the Artificial*, Second Edition, MIT Press, Cambridge, Mass., 1981.

[SM84] Louis I. Steinberg, Tom M. Mitchell, "A Knowledge Based Approach to VLSI CAD: The REDESIGN System", in *Proceedings of the* 21^{st} *Design Automation Conference*, 1984, pp. 412-418.

[SCB89] Jon Sticklen, B. Chandrasekaran, W. E. Bond, "Applying a Functional Approach for Model-Based Reasoning", in *Proceedings of the 1989 Workshop on Model-Based Reasoning*, Detroit, August 20, 1989, pp. 165-176.

[ST90] Jon Sticklen, Rula Tufankji, "Utilizing a Functional Approach for Modeling Biological Systems", AI/KBS Laboratory Report 1990:#2, Department of Computer Science, Michigan State University, 1990.

[SKB90] Jon Sticklen, Ahmed Kamel, W. E. Bond, "A Model-Based Approach for Organizing Quantitative Computations", AI/KBS Laboratory Report 1990:#3, Department of Computer Science, Michigan State University, 1990.

[Sto77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[Tur84] Raymond Turner, "Logics for Artificial Intelligence", Halsted Press, New York, 1984.

[VHDL87] IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987.

[Wat84] Richard C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", in *Interactive Programming Environments*, D. Barstow, H. Shrobe, E. Sandewall (eds.), McGraw-Hiull, New York, 1984, pp. 464-486. Reprinted from *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1 (January 1982), pp. 1-12.

[WF85] Barbara Y. White, John R. Frederiksen, "QUEST: Qualitative Understanding of Electrical System Troubleshooting", in *SIGART Newsletter*, Special Section on Reasoning About Structure, Behavior, and Function, B. Chandrasekaran, Robert Milne eds., No. 93 (July 1985), pp. 34-37.

[Wil85] Brian C. Williams, "Qualitative Analysis of MOS Circuits", in *Qualitative Reasoning About Physical Systems*, Daniel G. Bobrow, ed., The MIT Press, Cambridge, MA 1985, pp. 281-346. Reprinted from *Artificial Intelligence* Vol. 24, 1984.

[WMK89] Howard G. Wilson, Paul B. MacCready, Chester R. Kyle, "Lessons of Sunraycer", in *Scientific American*, Vol. 260, No. 3, (March 1989), pp. 90-97.

VITA

David Wayne Franke was born on April 8, 1954, in Enid, Oklahoma. He received his high school diploma from Enid High School in May, 1972, a B.S. in Mathematics from the University of Oklahoma in May, 1976, and a M.S in Computer Science from the Pennsylvania State University in November, 1977. He has worked for Texas Instruments, Inc. (1978-1985) as a Senior Member of the Technical Staff in operating systems, computer architecture, and artificial intelligence. He has also worked for the Microelectronics and Computer Technology Corporation (1986-1991) as a Senior Member of the Technical Staff in artificial intelligence in design, design reuse, and hardware/software codesign. He currently works for the Trilogy Development Group.

In the fall of 1985, he joined the Department of Computer Sciences at The University of Texas at Austin as a graduate student where he has pursued research in qualitative modeling and representation of descriptions of purpose.

Permanent address: 8913 Scottish Pastures Dr.
Austin, Texas 78750-3571

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.