ABSTRACT

Title of dissertation: COMPUTATIONAL FOUNDATIONS FOR COMPUTER AIDED CONCEPTUAL DESIGN OF MULTIPLE INTERACTION-STATE MECHATRONIC DEVICES.

Changxin Xu, Ph.D., 2005

Directed By: Associate Professor Satyandra K. Gupta, Department of Mechanical Engineering

Increasing autonomy and intelligence in mechatronic devices requires them to be multiple interaction-state devices. Different modes of operations and different types of interactions with the use-environment require the device to have multiple interaction-states, each state capable of producing a different behavior to meet its intended requirements. For multiple interaction-state mechatronic devices, a satisfactory framework does not exist for representing, evaluating, and synthesizing design concepts. Hence, majority of mechatronic designers currently use informal methods for representing and evaluating design concepts during the conceptual design. This leads to the following problems. First, informal representation of design concepts hinders information exchange and reuse. Second, in absence of a validation methodology, it is not clear how to determine if a proposed design concept is consistent with the requirements. Finally, designers cannot perform computer aided evaluation during the conceptual design stage.

This dissertation focuses in the area of computational foundations for representing, validating, evaluating, and synthesizing design concepts of multiple interaction-state mechatronic devices. A modeling and simulation framework has been developed for representing design concepts behind multiple interaction-state mechatronic devices. The problem of consistency-checking of interaction-states has been studied and an algorithm has been developed for solving the interaction consistency-checking problem. The problem of determining the presence of unsafe parameter values has been studied and an algorithm has been developed to determine whether an interaction-state in the proposed design concept can attain unsafe parameter values. Algorithms have been developed for evaluating design concepts based on the maximum power consumption and sharability of components. Finally, algorithms have been developed for automatically synthesizing transition diagrams for meeting the desired behavior specifications, given a components library.

We believe that the results reported in this dissertation will provide the underlying foundations for constructing the next generation computer aided design tools for conceptual design of mechatronic devices. We expect that these tools would streamline the product development process, facilitate information reuse, and reduce product development time.

COMPUTATIONAL FOUNDATIONS FOR COMPUTER AIDED CONCEPTUAL DESIGN OF MULTIPLE INTERACTION-STATE MECHATRONIC DEVICES

By

Changxin Xu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:
Associate Professor Satyandra K. Gupta, Chairman / Advisor
Professor Davinder K. Anand
Associate Professor Mark A. Austin
Dr. Michael Gruninger
Professor Edward B. Magrab
Professor Dana S. Nau
Dr. Ram D. Sriram

© Copyright by

Changxin Xu

2005

Acknowledgements

I would like to thank my advisor, Dr. Satyandra K. Gupta, for his invaluable advice and suggestions through years of my study. His guidance led me through many hard times. His hardworking spirit, persistence and skilled ways of analyzing problems are imprinted in my heart. I am sure I will benefit from this in my life.

I would like to thank my parents and my wife Li Yin. It is their continuous encouragement and endless support that keep me inspired and motivated.

I would also like to thank my dissertation committee members: Drs. Anand, Austin, Gruninger, Magrab, Nau, and Sriram for accepting to serve in the committee and providing suggestions.

Finally, I would like to thank my colleagues in the Computer Integrated Manufacturing Lab: Antonio, Alok, Mukul and Ira for devoting their precious time proofreading my dissertation. I would also like to thank Drs. Lin and Yao for providing feedbacks on my research.

.

Table of Contents

Acknowlegements ii

Table of Contents iii

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1

1.1 Background 1
1.2 Motivation 4
1.3 Research Issues 10
1.3.1 Design Concept Representation 10
1.3.2 Algorithms for Design Concepts Validation 12
1.3.3 Algorithms for Design Concepts Evaluation 13
1.3.4 Design Concepts Synthesis 14
1.4 Dissertation Outline 15

Chapter 2: Related Research 18

2.1 Background 19
2.2 Motivation 27
2.3 Research Issues 29
2.4 Dissertation Outline 30
2.5 Summary 34

Chapter 3: Modeling and Simulation Framework 37

3.1 Background 37
3.2 Class Definitions for Modeling Primitives 40
3.2.1 Classes for Modeling Parameters and Parameter Interactions 42
3.2.2 Classes for Modeling Artifacts, Artifact Interactions, and Artifact Mappings 46
3.2.3 Classes for Modeling Interaction-States 48
3.2.4 Classes for Modeling Event and Event Spaces 52
3.2.5 Classes for Modeling Unsafe Parameter Value Sets 54
3.2.6 Classes for Modeling Interaction-State Transitions and Transition Diagrams 55
3.3 Elaboration Operators 58
3.4 Steps in Conceptual Design 66
3.5 Simulating Transition Diagrams 70

3.6 Example of Modeling Autonomous Vacuum Cleaner (*AVC*)........................... 73
3.7 Summary 95

Chapter 4: Consistency-Checking of Interaction-states 100

4.1 Problem Formulation 100
4.1.1 Problem Statement 100
4.1.2 Overview of Our Approach 103
4.1.3 Related Work On Finding Min Cut Of A Graph 104
4.2 Mapping Consistency Checking Problem To Minimum S-T Cut Problem In Interaction Network 105
4.2.1 Construction Of Interaction Network 105
4.2.2 Mapping Consistency-Checking Problem to Minimum Cut Problem. 106
4.3 Algorithms For Finding Minimum S-T Cut And Identifying Inconsistent Interactions... 117
4.3.1 Algorithm for finding minimum s-t cut in network G.......................... 117
4.3.2 Algorithm For Finding Inconsistent Interactions............................... 121
4.4 Implementation And Examples 122
4.5 Summary 134

Chapter 5: Detection Of Unsafe Parameter Value Sets Embedded In

Interaction-States 135

5.1 Problem Formulation 135
5.1.1 Problem Statement 135
5.1.2 Overview of Our Approach 137
5.2 Algorithm for Detecting the Presence of Unsafe Parameter Value Sets 138
5.3 Examples 147
5.4 Summary 154

Chapter 6: Design Concept Evaluation.. 156

6.1 Optimal Component Sharing 156
6.1.1 Problem Statement 156
6.1.2 Complexity Analysis of Optimal Component Sharing Problem.......... 159
6.1.3 Branch And Bound Algorithm For Solving The Problem 162
6.1.4 Example 165
6.2 Evaluating Design Concept Based On Maximum Power Consumption......... 167
6.3 Summary 169

Chapter 7: Transition Diagram Synthesis 170

7.1 Problem Formulation 170
7.1.1 Preliminaries... 170
7.1.2 Problem Statement 179
7.2 Structure of the Component library 180
7.3 Synthesis Algorithms 184

7.4 Characteristics of Algorithms 189
7.5 Example 191
7.6 Summary 205

Chapter 8: Transition Diagram Synthesis 206

8.1 Intellectual Contributions.. 206
8.2 Anticipated Benefits 209
8.3 Directions for Future Work 210

References.. 212

List of Tables

Table 3.1: Limitations on combining initialization types and value-changing modes 51
Table 3.2: Artifacts and Parameters used in *AVC* behavior specification 75
Table 3.3: Event space used in *AVC* behavioral specification 75
Table 3.4: Unsafe state used in *AVC* behavioral specification 76
Table 3.5: Event sequence for *AVC* behavior simulation 83
Table 3.6: *AVC* behavior simulation result 84
Table 3.7: Decomposed Artifacts and Parameters of *AVC* 87
Table 7.1: Standard parameters used in *IDS* example 172
Table 7.2: Parameters selection for artifact definition 175

List of Figures

Figure 1.1: Example of interaction-states in a hybrid car 2
Figure 1.2: An abstraction of information flow in design... 3
Figure 1.3: Limitations of existing CAD models 5
Figure 1.4: Applications enabled by formal design concept representations 8
Figure 1.5: Organization of the dissertation.. 17
Figure 3.1: Overview of primitives 41
Figure 3.2: Structure of interaction state transition diagram 42
Figure 3.3: Relationships between major primitives 43
Figure 3.4: Unrealizable transitions 56
Figure 3.5: Example of unsafe transition diagram 57
Figure 3.6: Usage of operator *DECOMPOSE-ARTIFACT* 60
Figure 3.7: Usage of operator *DECOMPOSE-STATE* 63
Figure 3.8: Usage of operator *DECOMPOSE-TRANSITION* 65
Figure 3.9: Elaboration of interaction transition diagrams 69
Figure 3.10: Requirements of *AVC* 74
Figure 3.11: *AVC* behavior specification #1 76
Figure 3.12: Definition of state s_0 77
Figure 3.13: Definition of state s_1 78
Figure 3.14: Definition of state s_2 79
Figure 3.15: Definition of state s_3 80
Figure 3.16: Definition of state s_4 81
Figure 3.17: Illustration of a use-environment for simulation.................................. 82
Figure 3.18: *AVC* behavior specification #2 85
Figure 3.19: Modified “Waiting” state 86
Figure 3.20: *AVC* design concept based on behavior specification #2...................... 88
Figure 3.21: Definition of state s_0 89
Figure 3.22: Definition of state s_1 90
Figure 3.23: Definition of state s_1 91
Figure 3.24: Definition of state s_3 92
Figure 3.25: Definition of state s_3 93
Figure 3.26: Definition of state s_5 94
Figure 3.27: Definition of state s_5 94
Figure 3.28: Organization of the content of the remaining chapters 99
Figure 4.1: Example of an interaction-state for hybrid car 101
Figure 4.2: Interaction network constructed from the above relationships.............. 107
Figure 4.3: Residual network 109
Figure 4.4: A cut of the network 110
Figure 4.5: A cut illustrating terminology used in Theorem 1................................ 111
Figure 4.6: An example of a cut for illustrating Theorem 1 113
Figure 4.7: A cut illustrating terminology used in Theorem 2................................ 114
Figure 4.8: A cut illustrating terminology used in Theorem 3................................ 116
Figure 4.9: Illustration of algorithm FINDMINIMUMSTCUTSIZE 118

Figure 4.10: Maximum flow of the graph 120
Figure 4.11: Residual network corresponding to maximum flow 120
Figure 4.12: Finding inconsistent relationships 123
Figure 4.13: Design alternative A of a planar mechanism 123
Figure 4.14: Design alternative B of a planar mechanism 126
Figure 4.15: Design alternative A of a spatial mechanism 127
Figure 4.16: Design alternative B of a spatial mechanism 131
Figure 5.1: Transition diagrams for behavior specification of microwave 148
Figure 5.2: Definition of state s_3 149
Figure 5.3: Transition diagrams for behavior specification of reservoir 150
Figure 5.4: Definition of state s_2 151
Figure 5.5: Transition diagrams for behavior specification of coffee maker 152
Figure 5.6: Definition of state s_3 153
Figure 5.7: Mixing state 155
Figure 6.1: Converting GCP to OCSP 161
Figure 6.2: An example illustrating the branch and bound algorithm 166
Figure 6.3: An example of estimating maximum power consumption 168
Figure 7.1: Transition diagram and event space used in *IDS* behavioral specification 173
Figure 7.2: Example of a component library 174
Figure 7.3: Working state for *CCD* in behavior specification 175
Figure 7.4: Working state for motor in behavior specification 176
Figure 7.5: Working state for lens in behavior specification 176
Figure 7.6: Working state for recognition algorithm in behavior specification 177
Figure 7.7: Working state for image improvement system in behavior specification 192
Figure 7.8: Final transition diagram for the Image Improvement System 193
Figure 7.9: State description of “Capture” 193
Figure 7.10: State description of “Track” 194
Figure 7.11: State description of “Switch” 195
Figure 7.12: Illustration of searching for components of *IDS* 196
Figure 7.13: Incorporate *CCD* into *IDS* 197
Figure 7.14: Incorporate IIS into *IDS* 197
Figure 7.15: Incorporate recognition algorithm and locate algorithm into *IDS* 198
Figure 7.16: Applying operator: generate replacement states 199
Figure 7.17: Applying operator: generate replacement components 200
Figure 7.18: Applying operator: generate use-environment components 201
Figure 7.19: State description of “Monitor” 201
Figure 7.20: State description of “Capture” 202
Figure 7.21: State description of “Track” 202
Figure 7.22: State description of “Switch” 203
Figure 7.23: State description of “Record” 204
Figure 7.24: Applying operator: generate internal transitions 204
Figure 7.25: Applying operator: generate external transitions 205

Chapter 1: Introduction

This chapter is organized in the following manner. Section 1.1 describes the background needed to introduce the problem being addressed in this dissertation. Section 1.2 describes the motivation behind the research described in this dissertation. Section 1.3 describes the major research issues being addressed in the dissertation. Section 1.4 describes the organization of the remainder of the dissertation.

1.1 Background

The industrial revolution has brought mechatronic devices into the forefront of technological advancements. Mechatronic devices refer to the devices that integrate elements from mechanical, electrical and electronic, and information domains, which are designed to provide better solutions than would be possible if components from only one domain are used [Walt01]. Use of mechatronic devices is pervasive, ranging from everyday utilities such as microwave ovens and washing machines, to intelligent robots and numerical controlled machine tools used in industry.

Increasing autonomy and intelligence in mechatronic devices requires them to be multiple interaction-state devices. Multiple interaction-state devices are those devices in which the interactions between elements of the use-environment and elements of the device can have different qualitative structures (i.e., different interaction topologies) depending upon the modes of device operation and the states of the use-environment. Different modes of operations and different types of interaction topologies with the use-environment require the device to be in different states, while each state is capable of producing a different behavior to meet its intended

requirements. For example, consider a hybrid vehicle as shown in Figure 1.1. When the vehicle is going down a hill, the engine is storing energy into the batteries. When the vehicle is going up a hill, both the batteries and the engine are providing power to the wheels. In this example, the interactions topology among device components (battery, engine, and wheel) is changing depending upon the states of the use-environment (e.g., uphill or downhill).

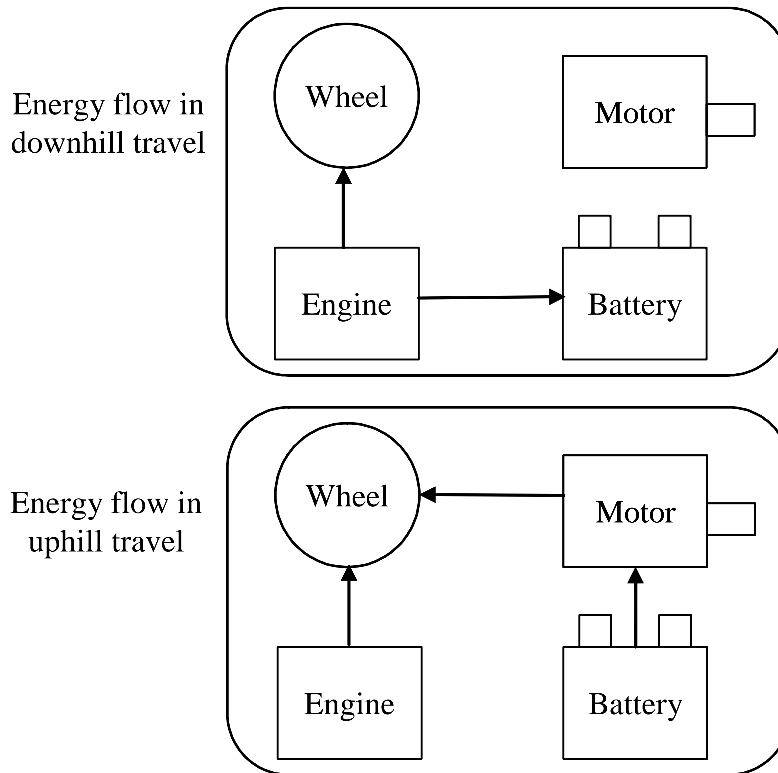


Figure 1.1: Example of interaction-states in a hybrid car

Figure 1.2 shows an abstraction of the information flow in a typical product development process [Pahl96]. This figure mainly illustrates the information flow and does not show the iterative nature of the design process. Starting from the customer, the first step is need analysis, which determines the requirements. This step establishes *why* a device should exist. The second step is to establish behavior

specifications, which creates the specifications of the desired observable behavior of the device that satisfy the requirements. This step establishes *what* a device should do. After that, the conceptual design step analyzes the desired behavior of the device and results in the specifications of the internal structure of the device. Finally the detailed design step completes the design by developing details of every component in the structure. The conceptual and the detailed design steps establish *how* the device will provide the desired behavior.

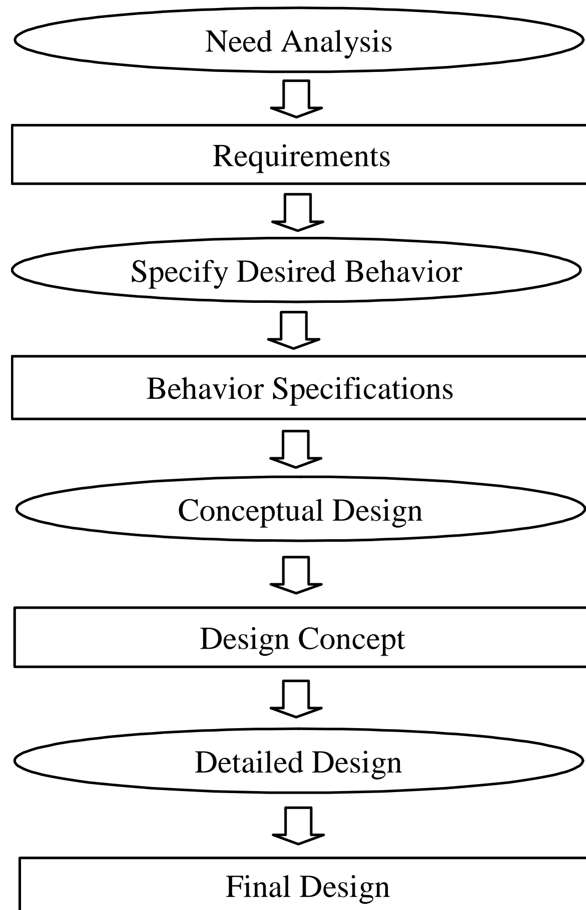


Figure 1.2: An abstraction of information flow in design (this figure only shows the information flow and does not depict loops generated by the iterative nature of the design process)

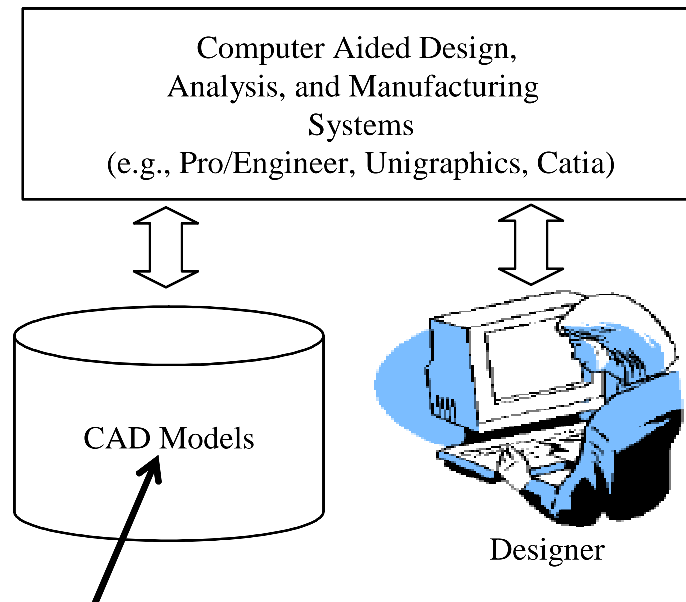
In systems engineering community, requirements engineering is often used to establish what a system will do [Hull02]. Furthermore, in this step, high level system requirements are hierarchically decomposed into lower level requirements. A particular way of decomposing the requirements may also impose constraints on how the system will be designed. Therefore, requirements engineering may overlap with the conceptual design step.

While it is well understood what constitutes a detailed design, it is not always clear what goes into describing a design concept. In this dissertation, we assume that the design concept will need to have the following three main ingredients. First, the design concept will need to identify various major components (e.g., functional units) that will be needed to meet the requirements and their roles in meeting the requirements. Second, the design concept will need to specify the basic working principles behind every main component to ensure that the component is realizable. Third, the design concept will need to specify how various components will interact with each other to achieve the requirements. We believe that these three pieces of information are necessary for evaluating complexity and cost associated with design concepts.

1.2 Motivation

Today's intensive competition in the market requires companies to deliver better quality products in shorter lead-times with limited product development budget. Computer aided design (CAD) tools are being used to satisfy such needs. However, most of the commercial CAD systems for mechanical products are aiding designers only in the detailed design step. Computer aided design tools for early stage of

mechanical design are either restricted to few specific products or only providing simple sketching functions. Figure 1.3 illustrates the current state of design tools and design data being stored. CAD models currently only store geometric information and there is no connectivity between the final product geometry and requirements.



Observations:
1. Only geometric information is stored
2. There is no connectivity between the final product geometry and requirements

Figure 1.3: Limitations of existing CAD models

Most designers use their own notations and conventions to create and represent design concepts. This informal and ad-hoc practice of creating and storing design concepts makes it very difficult for a person who was not a part of the design team to understand the design concepts underlying a product. Design of mechatronic devices is further complicated by the collaboration of engineers from different disciplines on a complex device, all of whom have their own perspective and way of working. A shared understanding between each disciplines involved is key to the success of the

integrated device. Furthermore, unless formal representations are developed for modeling design concepts, we cannot develop software tools for design concept synthesis and evaluation. On the other hand, if we consider detailed design phase of mechanical products, computer interpretable representations are widely used in forms of solid models and feature-based models. These representations have led to the development of many engineering analysis tools that are frequently used to increase designers' productivity.

Unlike mechatronic devices, formal languages for design are successfully being used in software and VLSI (Very Large Scale Integrated Circuits) industry. In the software industry, UML (Unified Modeling Language) is increasingly being used as a modeling language to model the concepts behind complex software systems. In the VLSI industry, VHDL (VLSI Hardware Description Language) is being used to model concepts behind complex computer chips.

Requirements engineering provides a systematic process for developing system requirements. However, existing representation schemes being used in requirements engineering alone are not sufficient for representing complex mechatronic device concepts. Interactions among components are viewed as important pieces of information in requirements engineering. However, detailed representations for adequately modeling all possible types of interactions that are common in mechatronic devices have not been developed. In the absence of detailed interaction models, only a limited type of computer-aided validation and evaluation can be carried out in the requirements engineering area. Usually, such validation and evaluation is sufficient for requirements engineering. However, in order to support

computer aided conceptual design, a lot more information needs to be formally represented.

For mechatronic devices, a satisfactory design concept description language does not exist. Besides, traditional functional modeling approaches that have been developed for single interaction topology based devices cannot be conveniently applied to multiple interaction-state devices. Hence, the majority of mechatronic device designers currently use informal methods for representing and evaluating design concepts. This leads to the following problems:

- Informal representation of design concepts hinders information exchange and reuse, which leads to longer development time, longer product update time and perhaps poorer product quality. Dynamic design teams and the need to constantly upgrade products increase the importance of archiving and exchanging design concepts. In the absence of formal representation, a new designer who has been given the charge of improving a device may take a very long time to understand how the existing device works and exchange ideas with his/her colleagues.
- In the absence of a formal validation methodology, it is not clear how to determine if a proposed design concept is consistent with the requirements. Such inconsistency may not be detected until the device testing stage. Hence, informal methods of validating design concepts may waste designer's energy on unpromising design concepts.
- Designers need to develop the concepts further in order to apply computer aided engineering tools. This may not only waste time and energy on unpromising

design concepts, but also limit the number of promising concepts that can be evaluated in a given development time.

It is clear that if we were to achieve a high level of automation in design of multiple interaction-state mechatronic devices we will need formal representations to describe design concepts. We believe that such a formal representation will enable computer-supported tools for aiding conceptual design. Figure 1.4 graphically shows different computational tools that can utilize the formal design concept representation.

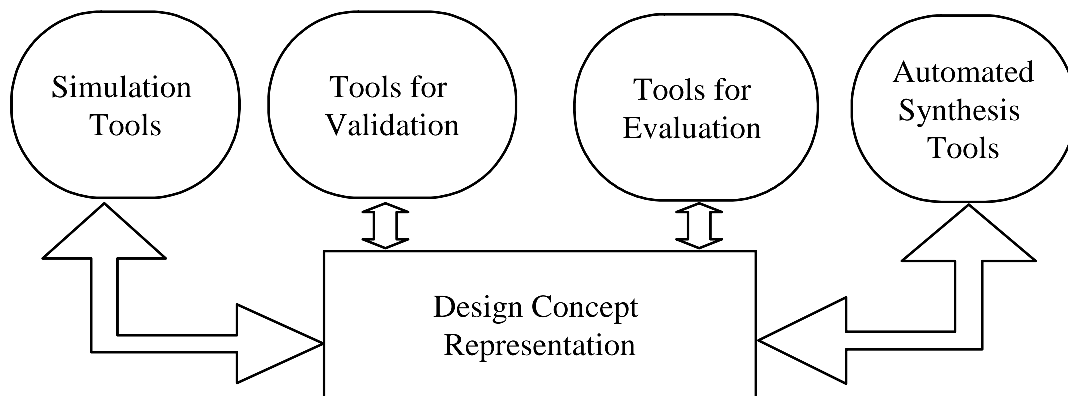


Figure 1.4: Applications enabled by formal design concept representations

In order to reduce the product development time, we also need tools that can perform automated validation [Chan90] of the proposed design concepts. These tools will ensure that only valid design concepts are transferred to the detailed design stage for further development. The importance of this step can be better understood by examining the consequence of not performing the design validation at the detailed design stage. For example, the product development gets significantly delayed if non-manufacturable shapes are passed from the detailed design step to the manufacturing step. Similarly, passing invalid design concepts to the detailed design stage leads to unnecessary delays in the product development.

Many tools have been developed that can perform validation during detailed design stages. These tools check various features in the geometric model of the proposed design to assess their validity. Such tools are significantly reducing the time to carry out the validation tasks. We are interested in developing validation tools for the conceptual design stage. Developing such tools requires the following three steps. First we need to develop a representation to model design concepts. This is analogous to the development of feature-based representations for modeling detailed designs. The next step is to develop the definition of validity. This is similar to defining what feature parameters will be considered valid during the detailed design stage. For example, very thin walls or features with zero-draft angles may not be considered valid in the context of injection molding. Finally, we need algorithms that can determine if a proposed design is invalid. This is analogous to the development of a geometric algorithm that can detect if the given design contains a feature with zero draft angles.

Design concepts generated during conceptual design stage must be evaluated before being developed further into detailed designs. Unlike detailed designs, design concepts do not carry complete design information. Therefore, evaluation methods that have been developed for evaluating detailed designs cannot be applied to the conceptual design stage. Depending upon the information available in the design concepts, different types of evaluation can be performed. Therefore, we will need to analyze design concept representations and develop the appropriate evaluation algorithms.

In order to effectively explore the design space, we need to examine a large number of design concepts. Generating design concepts manually limits the number of design concepts that can be examined. Therefore, we need to develop algorithms for automatically synthesizing design concepts to meet a given set of requirements. Formal representations, validation methods, and evaluation methods provide the necessary infrastructure for the development of automated synthesis algorithms.

1.3 Research Issues

The main research issues being considered in this dissertation are described in the following sections.

1.3.1 Design Concept Representation

Design concepts behind multiple interaction-state mechatronic devices capture designers' idea in the conceptual design stage for meeting requirements. Representation of design concepts provides the foundation of design information archiving, exchange and reuse. Design concept representation for mechatronic device cannot be simply accomplished by aggregation of existing representations. Furthermore, multiple interaction-states encountered in complex mechatronic devices need to be adequately modeled. Current representation schemes such as function based flow diagrams and bond graphs do not offer a convenient means for representing changing interaction topologies encountered in multiple interaction-state devices. State transition diagrams currently being used in modeling and analyzing concepts behind software and electronic circuits provide a starting point for capturing state transitions of multiple interaction-states mechatronic devices. However, they do

not provide adequate modeling support for describing interactions among device components.

This dissertation will focus on the following three research issues related to the representation of design concepts:

- **Modeling primitives for multiple interaction-state mechatronic devices**. Modeling design concepts of mechatronic devices with multiple interaction-states requires considering interactions that lead to continuous and discrete changes in device parameter values. Therefore, modeling primitives are needed to model interactions among device components and interaction between device components and the environment. Furthermore, the modeling framework will need to support changes in governing interactions as the device goes through different modes of operations. This dissertation provides these modeling primitives to support conceptual design.
- **Modeling operators for multiple interaction-state mechatronic devices**. We envision that during the conceptual design, the underlying modeling primitives will be manipulated to add more detail to the design concept. An example of such a manipulation is decomposing a primitive into a set of primitives. To eliminate design errors, we need to ensure that the primitives that result from manipulating existing primitives remain valid. Operators needed to manipulate the primitives will depend a great deal on the primitives themselves. Thus, new modeling operators are needed. This dissertation provides the modeling operators.

- **A framework for enabling concept simulation**. In many situations, simulation serves as a powerful tool for evaluating designs. However, simulation tools developed for detailed design simulation cannot be used during conceptual design. Instead we need a new simulation framework that only utilizes the information available during the conceptual design. The new representation developed in the dissertation enables concept simulation. This dissertation provides a framework for determining the response of the device to a given set of events in the use-environment. By creating a set of simulated events in the use-environment, users can evaluate the concept using the framework provided in this dissertation.

1.3.2 Algorithms for Design Concepts Validation

In order to reduce the product development time, we need tools that can perform automated validation of the proposed design concepts. These tools will ensure that only valid design concepts are transferred to the detailed design stage for further development. However, in conceptual design stage, often we only know the qualitative structure of the design solution instead of the knowing the exact equations. This requires design concept validation methods to work with the qualitative design information.

This dissertation will focus on the following two research issues related to the design concept validation area:

- **Algorithms for checking interaction-state consistency.** Validating interaction-states involves checking the consistency of the set of interactions in the state. This requires us to ensure that the underlying interactions in the state are not

over-constrained. Furthermore, we need to ensure that any subset of the interactions is also not over-constrained. This dissertation provides algorithms, the corresponding correctness proofs, and worst-case asymptotic complexity analysis for the interaction-state consistency-checking problem.

- **Algorithms for detecting presence of unsafe parameter values.** In many instances, unsafe parameter values are defined as a part of the requirements. In order to satisfy requirements, a valid design concept must be safe and hence should not attain unsafe parameter values. Checking presence of unsafe parameter values based on the existing discrete parameter value formulations is not expected to work in presence of interactions that involve both continuous and discrete changes in parameter values. Therefore, we need new algorithms. This dissertation presents the problem formulation for checking presence of unsafe parameter values in a design concept based on multiple interaction-states and provides algorithms for solving it.

1.3.3 Algorithms for Design Concepts Evaluation

Design concepts generated must be evaluated before being developed further in the detailed design stage. Since design evaluation consumes resources such as time and money, eliminating unpromising design concept alternatives as soon as possible is desired. Different representation schemes usually support different types of evaluations. The new representation scheme described in the dissertation enables new directions for design concept evaluations.

This dissertation will focus on the following two research issues related to the design concept evaluation area:

- **Algorithms for evaluating design concepts based on active component use**. The new representation of multiple interaction-state mechatronic design concepts makes it possible for us to determine which components are active in which states. Consider the problem of evaluating the maximum power consumed by a design concept. This cannot be simply computed by summing up the power requirements for all components. Instead, we need to figure out when components are active and when they are not active. We also need to determine the state where the maximum power is being consumed by active components. This dissertation provides an algorithm for evaluating design concepts based on maximum power consumption. The algorithm developed in this dissertation can be extended to the estimation of noise generation as well.
- **Algorithms for determining component sharability.** In many designs, two different states require components of the same type but for different usages. If two different usages are not needed at the same time, then a single physical component can fulfill both roles. Sharing of a physical component among multiple states can potentially reduce the cost of the design. Therefore, this dissertation investigates the computational complexity of the component sharability problem and presents an algorithm for solving it.

1.3.4 Design Concepts Synthesis

The representation, validation and evaluation schemes provide the infrastructure for the conceptual design of mechatronic devices with multiple interaction-states. The same schemes can be adopted for synthesis as well. This dissertation will explore the following two issues related to the synthesis area:

- **Component description**. A component library will be used for describing the known components. Components will need to be described in such a way that a synthesis algorithm will be able to identify the applicable components and connect them together to create a possible design concept. This dissertation will provide a component description scheme for describing known components. This scheme will support description of both simple as well as complex components.
- **Synthesis algorithm**. The synthesis algorithm will need to identify the appropriate components and connect them together to form valid design concepts consistent with the desired behavior. Although algorithms have been developed for synthesizing circuit and simple input/output type of electromechanical products, they will require significant extensions be useful in mechatronic devices with multiple interaction-states. This dissertation will focus on the development of sound synthesis algorithms that can utilize complex components in simple arrangements to come up with the possible design concepts. Complex components will allow the synthesis algorithm to exploit the known design concepts and make use of them in solving new design problems.

1.4 Dissertation Outline

The remainder of this dissertation is organized in the following manner.

Chapter 2 presents a literature survey on topics related to this dissertation.

Chapter 3 describes a modeling and simulation framework for representing design concepts behind multiple interaction-state mechatronic devices.

Chapter 4 describes the problem of consistency-checking of interaction-states as a step in the design concept validation. It presents an algorithm for solving the

interaction consistency-checking problem. It also presents an algorithm for analyzing inconsistent interaction-states and identifying the inconsistent interactions.

Chapter 5 describes the problem of determining whether unsafe parameter value sets are embedded in an interaction-state transition diagram. It presents an algorithm to determine whether a given interaction-state transition diagram can include unsafe parameter value sets.

Chapter 6 describes an algorithm for estimating the maximum power consumed by a design concept. It also describes an algorithm for determining the sharable components in multiple interaction-state devices.

Chapter 7 describes algorithms for automatically synthesizing detailed transition diagrams given the initial behavior specifications and a components library.

Chapter 8 describes the main research contributions of this dissertation and identifies the anticipated benefits resulting from this research. It also gives suggestions for the future extension of the work described in this dissertation. Figure 1.5 shows the organization of the dissertation.

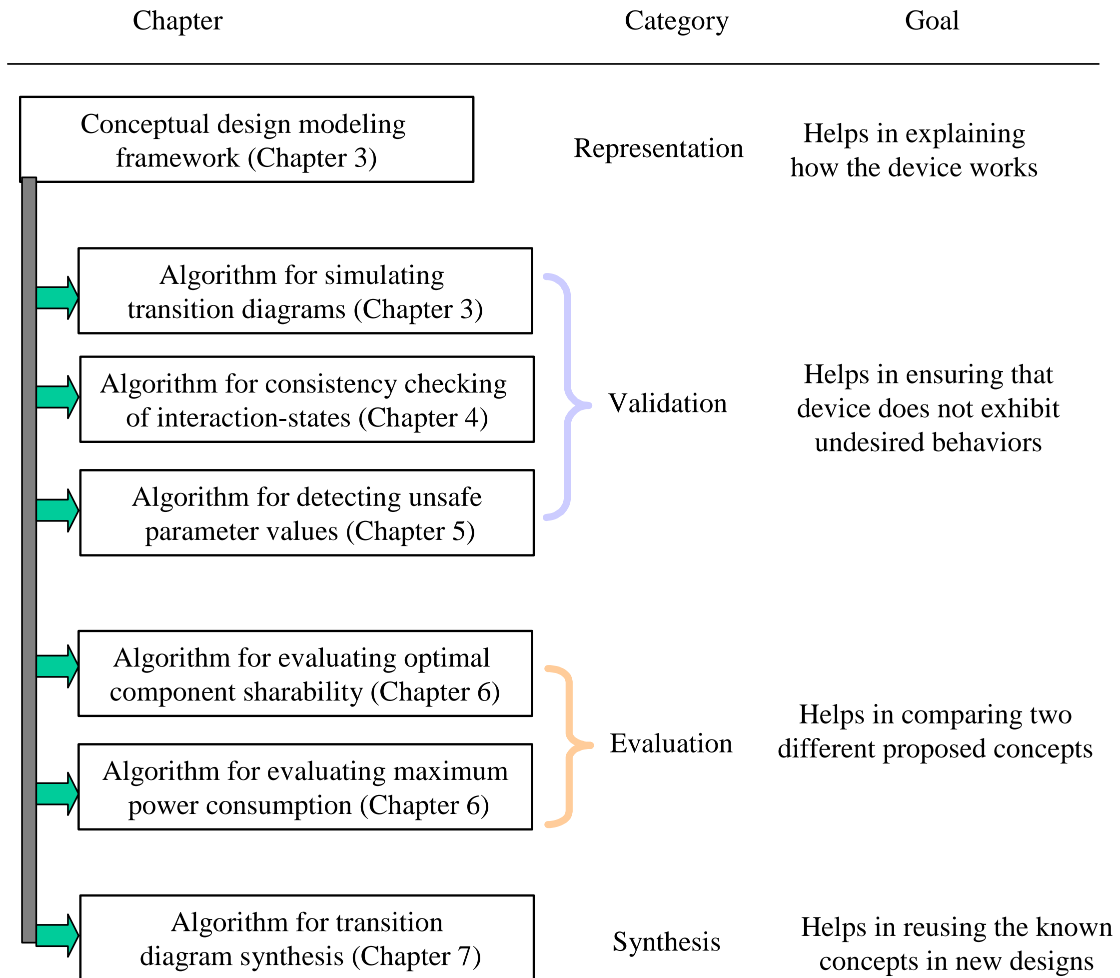


Figure 1.5: Organization of the dissertation

Chapter 2: Related Research

This chapter provides a review of the state of the art in topic areas related to this dissertation.

To facilitate reasoning about behavioral and spatial aspects of an electro-mechanical system, the process of design has been divided into the following distinct stages: (1) conceptual design, and (2) detailed design. The detailed design stage is divided into (1) parametric design, (2) configuration design, and (3) optimization and refinement of the design. Conceptual design schemes based on a variety of models have been proposed such as influence diagrams [Navi91], behavioral networks [Will89], equations, qualitative reasoning [Navi92] and bond graphs [Ulri88, Welc91]. A variety of representations such as graphs, grammars, constraint satisfaction etc. have been presented based on those models. Mechanism synthesis is addressed in [Erdm95]. The conceptual design step results in a selection of major components and their interconnections to provide its intended functionality. The choice of components is followed by parametric design wherein the components are sized for their geometry and material properties. Once a behavioral topology of components is chosen and sized, spatial orientation and location of components along with minor refinements to their form is addressed during spatial configuration design. The spatial configuration problem is highly dependent on the nature of the components, their physical forms, their material construction, constraints on spatial volume, weight and size and finally manufacturing and assembly considerations.

We review related work in the area of conceptual design representations in Section 2.1. Research in the design validation area is reviewed in Section 2.2. Previous work

in the area of evaluation is reviewed in Section 2.3. Research in design synthesis is reviewed in Section 2.4. We present the overall summary of the related work in Section 2.5.

2.1 Representations in Conceptual Design

Design representation is regarded as the description or model of design information and it is strongly coupled to the design methodology being practiced [Dym94]. Conceptual design is usually viewed as the stage that needs a lot of human intervention during the generation of creative ideas. Design information generated during the conceptual design stage is traditionally generated for human designers. Since it is in the early stage of design, most of the information is incomplete and often represented informally as sketches for human consumption. However, the broad usage of computer tools is pushing the research towards the study of formal and reusable representations during the conceptual design stage. Formal languages, ontologies and other computational representations belong to this case.

Sketches have been used as the initial representation for conceptual design in many industries because they are thought to be able to easily express innovative ideas [Mcgo98, Purc98, Yang03, Tove04]. Some progress has been made in the area of formalizing representations of planar mechanisms using sketches [Stah98]. However, they are defined using adhoc notations and may be difficult to understand for people other than their creators. Moreover, different designers may use different styles and may find it difficult to understand each other's sketches. It would be difficult to store and reuse sketches using computers.

Automated evaluation of concepts represented in sketches is also very challenging due to varying levels of details. Schematic representations are also used in which design is described schematically as a graph of its constituent elements. A common example would be the analog circuit description [Ulri02]. However, formal schematic representations are not popular in mechanical designs.

Function is an important concept in conceptual design across different domains. However, there is no universally accepted definition of term function. The function representation research was inspired by prior work from value engineering [Mile72, Akiy91]. Due to the hierarchical nature of the design problem solving, functions are often decomposed into sub-functions. Besides the hierarchical relationships between functions and sub-functions, different levels of importance for all functions are also identified. Function representation is also referred to as functional modeling. The fundamental issue in functional representation is to represent the function structure that includes the all the constituent sub-functions and the relationship between them.

Pahl and Beitz's widely accepted systematic approach to engineering design defines conceptual design as the feasible combination of working principles for sub-functions. Pahl and Beitz describe the function as the transformation from input to output in three flows: material, energy and signal [Pahl96]. They describe both functions and flows among functions. Working principles are sought for low level sub-functions and working structures are formulated by combining working principles. Compatibility of sub-functions is shown in compatibility matrix. Many variations have been proposed based on this basic framework [Ulri95, Magr97, Ullm97, Wood01].

Suh views design as the mapping between functional requirements and design parameters. Design problems are posed by defining top level functional requirements. Design parameters are defined in physical domain. Design solutions are formulated by combination of design parameters [Suh90].

Grabowski et. al. divide the traditional single function model into three layered function models with different levels of abstraction. The logical model, borrowed from electronic domain, is used to present high level topology and connectivity of sub-functions. The status model describes the working state combinations of different components. The relation model defines the mathematical or physical relations between physical variables [Grab99]. However, functions in the logical model are defined only using input/output of components. The status model only deals with discrete states of components.

An object oriented graphical representation of functions and flows is proposed in [Szyk99]. Data structures of functions and flows are defined using attribute/value pairs while the relationships between functions and flows are depicted graphically. Working principles (called artifact in [Szyk99]) are attached to functions and flows in the graph. Shooter et. al. subsequently proposed a model for design information flow in which the relationship between intended behavior and observed behavior of design artifact are described and elaborated [Shoo00]. Zeiny proposed a dynamic object-oriented model that stores form, function, behavior, taxonomy, composition and relationships. All the design information is stored in a generic container object called Design Entity. Design entities are organized hierarchically [Zein04].

The NIST Core Product Model (CPM) provides a base-level product model that is open, non-proprietary, generic, extensible, independent of any product development process and capable of capturing the full engineering context commonly shared in the product development process [Fenv01]. The CPM is intended to serve as a generic core representation for design information through the whole product development process. Specialized representations can be developed from it by adding more details to it.

Stone and Wood have proposed a functional basis language that tries to subsume the previous effort in functional modeling and provide a consistent classification scheme for functions. In this approach, functions are characterized using verb-object (function-flow) format and definitions of different classes of functions and flows are provided [Ston00, Hirt01]. Bohm and Stone recently argued that supporting functions are needed to completely represent artifacts. Supporting functions describe manufacturing, assembly and support features present in the embodied form of a product [Bohm04]. However, these 'function-transformation' methodologies have difficulty in representing a function that does not transform something. Besides, flows with a changing flow topology are difficult to model.

Some other researchers view functions as closely coupled with behaviors and present approaches for representing both. Function is defined as 'what a device is for' and behavior is defined as 'what a device does'. In this sense, function is also viewed as 'intended behavior' and sometimes function and intended behavior are used interchangeably. Chandrasekaran proposed a language called function representation for describing the function of a device, its structure and the causal processes in the

device that culminate in the achievement of the function [Chan94]. The causal process is described using simple state transitions.

Iwasaki et al. proposed the Causal Functional Representation Language (CFRL) [Iwas93, Iwas95]. They argued that this framework allows them to capture the knowledge of how the device is intended to work to achieve its function. CFRL relates intuitive functional description to behavior represented as a state transition. However, state is not defined formally. The interactions between system components are not captured either.

Sasajima et al. proposed Representation Language for Behavior and Function (FBRL) for representing function and behavior with predefined task and domain independent primitives [Sasa96]. Umeda et al. proposed Function-Behavior-State (FBS) modeling and a conceptual design support tool called FBS modeler [Umed96]. In FBS, a state is described by a set of entities and attributes and relationships between them. Behavior is described by a sequence of one or more changes of states. Qian and Gero propose a Function-Behavior-Structure path design model [Qian96]. Deng et al. proposed a representation model for desired product in terms of its function, behavior, structure and working environment [Deng99]. This Function-Environment-Behavior-Structure (FEBS) model includes initial function decomposition and conversion, causal behavioral process generation and physical phenomena library. In a subsequent paper, they argued that in some cases, material must also be considered in conceptual design. Thus conceptual design framework must also include material [Deng04a].

Approaches used in FBRL, FBS, and FEBS focus on capturing designer's intentions or behavioral processes by using simple state transitions. Basically they have two

limitations. First, states are only defined using a set of state variables. Relationships and constraints among state variables are not modeled. Second, they do not have different levels of abstraction for different levels of design information (i.e., they do not make any distinction between a higher level or lower level functions).

Bond graphs have been used in modeling behavior of dynamic system [Madh98]. The system is viewed as consisting of standard elements that have different numbers of input and output ports by which they are connected. This approach is restricted to power and signal flow based systems.

Vargas-Hernandez and Shah presents an information model called 2nd-CAD that aims at providing users with catalogs of elements to create interconnected multi layered structures of functions, behaviors, and components. The logic model of 2nd-CAD consists of entity and relationship models with corresponding transactions and constraints. Functions, behaviors and components are represented in function entity-relationship models, behavior entity-relationship models and component entity-relationship models respectively. Flow relationship models connect the output of one element and the output of another element. Composition relationship models connect parent and children elements. The mapping relationship models connect elements from different structures [Varg04]. Only energy flows are used in the function model. Behavior models are limited to bond graphs.

Current research in mechatronic system design is based on existing design methodologies. Most of the efforts focus on analyzing system performance using bond graphs [Karn00, Good02]. Diaz et al. used a hybrid representation of linear graph and block diagrams to support automatic generation of simulations from

individual components [Diaz99]. This approach is based on an augmented system graph that represents the topology of the system. Stacey et al. represented functions using concept arrays and blob diagrams [Stac96]. Chen and Jayaram extended flow diagram based functional representation schemes into mechatronic system representation by introducing two additional flows (information flow and control flow) and new relationships between functions and flows [Chen02]. They also presented a systematic approach for applying their schemes [Jaya03].

Gausemeier et. al. proposed a semi-formal specification language for modeling functions in conceptual design of mechatronic systems. Functions are viewed as transformations of discrete system states described by parameters [Gaus01]. However, functions are not represented formally. The modeling does not support hierarchy of functions either.

Verma and Wood argue that freeform text and functional basis represent extreme views for storage and reuse of functions during design [Verm03]. They suggest that a way to reconcile the two would be to use both. They proposed an augmented language to improve description of certain aspects of functions during the conceptual design.

Dori and Crawley argued that Object-Process Methodology (OPM) could serve as a domain-independent paradigm and modeling methodology that are shared among the various fields of knowledge for complex systems [Dori03]. OPM uses objects, processes and states as basic building blocks (called entities). Links are used to capture the static relations and behavioral relations between entities. They believe that in this way, structure and behavior, which are the two major aspects of system, can

co-exist in one paradigm. However, each object has its own states. Moreover, complex interactions between objects and their transformation are not captured.

Williams describes design as a process of building a network of qualitative interactions (called an interaction topology) between primitive components. Interactions are described by equations among parameters of components. He argued that new devices could be constructed by examining possible interactions producible by available components and every type of connection between components. The resulting structure is a topology of potential interactions [Will92].

Aiming at constructing the logical relationships between sub-functions at the first level of functional decomposition through information flows, Erden et. al. combine Petri nets with hybrid automata to model the logical behavior of mechatronic systems. They argue the logical relationships between sub-functions of a system can be best achieved by Petri nets. Hybrid automata are used to model both discrete and continuous state changes and evolution [Erde03]. Although energy, material and information flows are all acknowledged, only information flow is used in their first level of decomposition. Interactions among design parameters in the states are not explicitly considered.

State transition diagrams (STD), also known as state machines, are a way of describing the time-dependent behavior of a system. The basic consistency rule is: "A system's behavior in any state must be the same no matter by which path the state is arrived at" [Hare87]. STDs are good for modeling complex system behavior such as multiple entries and exits subject to different conditions. It has been successfully used in software/systems engineering (requirements engineering) [Kont98] and electrical

circuit design. The idea of simple state transitions has also been used to represent design knowledge [Chan93]. State transition matrix is used in systems engineering for analyzing complex system behavior [Haze96]. Researchers have also used hybrid automata that combine discrete transition diagram with continuous systems in modeling dynamical behavior [Broo04]. STD has been formalized in Unified Modeling Language (UML) (extended as statechart diagram) [Booc98]. However, the STD in UML cannot be directly used to represent design concepts. Each STD in UML only represents the states for one object. In order to represent design concepts, we also need to consider object hierarchy and the interactions between objects. UML provides another diagram called interaction diagram to model interactions. UML has been suggested as integration tool in design of mechatronic systems [Mroz01]. In order to conveniently model design concepts, we need a diagram that concurrently models interactions and state transition.

Researchers have also recognized the importance of a formal representation of the design process. Gorti et. al. presented a knowledge representation model for product and design process by applying and extending traditional object-oriented methodology [Gort98].

Some researchers also use shape grammar to capture the design knowledge of a certain types of artifacts. Shape grammar is a set of shape rules that could generate design step by step [Caga01].

2.2 Validation During Conceptual Design

Validation, also sometimes referred to as verification, involves checking that a design proposal satisfies functional and other specifications [Chan90]. Design validation

studies are usually closely related to design representations. For function-behavior based representations, the embedded behavior enables designers to do simulation. The behavior of the proposed system can be simulated and compared against the desired behavior [Brac96].

Deng et al. proposed a constraint-based functional design verification model based on an extension of their Function-Environment-Behavior-Structure (FEBS) design model [Deng99]. The input is the functional design model, which incorporates four aspects of functional design information: the working environment, the physical structure, the intended behavior, and the required function. A framework is developed that allows for the backward reasoning to trace the causes of system behavior. Design verification is achieved by identifying input and output design variables, developing a variable dependency graph, propagating constraints over the graph, and checking the values of the design variables against these constraints.

Efforts have also been made in the formal validation area using logic. Even though we are not aware of a validation effort that directly deals with electro-mechanical design, but efforts are being made in several other domains, particularly in the domain of process design. For example, Gruninger et al. used formal enterprise models to characterize process integration within enterprises. Customer's business requirements are transformed into logical questions and checked against the constraints within the enterprise model [Grun00].

Approaches for checking unsafe states in finite state machines are presented in [Mage99]. However, significant extensions are needed to check interaction-states that include both continuous as well as discrete variables.

2.3 Evaluation During Conceptual Design

Evaluation is needed to compare design option during the decision making. It is well known that evaluations can be either absolute or relative. Absolute evaluation evaluates the concept directly against the evaluation criteria such as feasibility judgment, technology-readiness assessment and go/no-go screening. Relative evaluation compares concepts with each other using measures defined by the criteria such as decision-matrix method [Ullm97].

Pahl and Beitz presented a general evaluation method: identifying evaluation criteria, weighting the evaluation criteria, assessing the values of alternatives and comparing alternatives. Evaluation criteria are usually derived from requirements or from general technical and economic characteristics [Pahl95]. Saaty proposed analytical hierarchy process (AHP) for evaluating multi-attributes problems [Saat90]. Hari and Weiss argued that failure modes and effect analysis should be conducted during conceptual design for evaluation. Potential failure modes and design improvements needed to eliminate these failures are estimated for concepts. Severity, frequency and detection phase are rated quantitatively. These parameters and their combinations indicate the reliability of the concepts [Hari99]. Huang and Liao proposed a method that integrates ordinal, cardinal and matrix algebra methods with normalized values based on Pahl and Beitz, and Saaty [Huan00]. Kalenchuk and Gu argued that product life cycle performance such as maintainability should also be evaluated during the conceptual design. They proposed specific maintainability metrics incorporating uncertainty for evaluating product maintenance of conceptual design alternatives [Kale02].

Pugh’s decision matrix method has been widely used for rating concepts relative to each other in their abilities to meet criteria set by customer requirements. It is an iterative evaluation method that consists of several steps. First, criteria are chosen for comparison. Then concepts are selected. Third, every designer picks the best concept that serves as a datum with which all other concepts will be compared. Finally the total score is calculated [Pugh90]. Takai and Ishii presented two modified Pugh methods [Taka02].

Due to the characteristic of incomplete information during conceptual design, evaluation uncertainties must be accounted for. Many decision support systems have been proposed for consideration of uncertainty during evaluation. See and Lewis presented a method for evaluating multiple, potentially conflicting criteria. They introduce hypothetical alternative choices to help assess decision maker’s preferences [See02, Gurn03].

2.4 Conceptual Design Synthesis

Traditionally, the design process has involved two main activities---synthesis and analysis. A general discussion of the synthesis process itself can be found in [Rooz02]. Most of the present generation CAD tools are geared towards analysis. As people want to achieve higher level of automation in design, they are beginning to investigate the possibility of automating some aspects of synthesis as well. At least in some applications, it may be desirable to have a system that takes customer needs as input and automatically designs a suitable product. There is no known general solution to synthesis problem. However, research is underway to achieve automation in specialized domains.

Particularly, in design of integrated circuits, significant level of automation has been achieved. Circuit design synthesis techniques are surveyed in [Kuma96]. Logic program synthesis techniques are surveyed in [Devi94]. Automated design synthesis is much harder for mechanical and electro-mechanical devices. We believe that the primary reason for this discrepancy is that for mechanical and electro-mechanical devices, it is more difficult to decouple the interactions among the device requirements than it is for purely digital devices.

Conceptual design synthesis investigates how design concepts are generated from the given design requirements. Usually synthesis techniques are closely associated with specific design representations. Due to the lack of availability of formal design representations during the conceptual design, many efforts are mainly focused on developing a synthesis methodology, not on the automation.

In their systematic approach, Pahl and Beitz's use the following process to generate conceptual solutions [Pahl96]. First, designers identify overall function from the design specifications. Then, the overall function is decomposed hierarchically into sub-functions, leading to a function structure. Working principles are sought for each sub-function and the solution is generated based on the feasible combination of working principles.

Suh views design as the inter-leaved mapping between function requirements and design parameters. Function requirement can not be decomposed unless its corresponding design parameter(s) is(are) determined [Suh90]. He presents two main axioms to assist designers in performing the right design decomposition. However, a procedure for automatically carrying out the decomposition is not given.

Chakrabarti et. al. view design solutions as combinations of a set of functional elements. They also argue that each element can be defined by basic element types or combinations thereof. Existing designs are investigated for distilling these element types with associated inputs and outputs. They describe design problem by functions represented by a number of inputs and outputs. Transformation chains are generated between these inputs and outputs. Each transformation can be embodied by choosing from the database of functional elements [Chak02]. They developed a program called FuncSION (Functional Synthesiser for Input Output Networks) to implement this approach. They use exhaustive search algorithm in their work. Search terminates when predetermined bounds on the number of the elements in the chain or the complexity of the chains are met.

Ward and Seering developed a synthesis algorithm for generating sets of components that combine to satisfy the design problem from a schematic input of a mechanical system, mathematical representations of the function specifications and catalog of elements. A control strategy of interval calculus and constraint propagation techniques is used to avoid exhaustive search [Ward93]. The algorithm is limited to single input single output systems.

Ulrich developed a bond graph based synthesis algorithm for single-input single-output devices. Input output chains are searched to connect the input bond graph chunk to the output chunk. The number of bond graph elements is preset to limit the search [Ulri88].

Following case-based design procedures, Madhusudan synthesizes electromechanical products based on bond graph representations. His synthesis algorithm consists of

three essential procedures: elaboration, retrieval and verification. Design specifications are described using inputs and outputs. First, internal topologies in the flow path are generated. Then cases are retrieved using retrieving keys. Finally, case verification is performed by symbolically solving the device relation and output time histories for the input time-histories [Madh98].

Campbell et. al. applied A-design methodology on synthesis of electromechanical design configurations. Starting from functional descriptions that describe the expected inputs and outputs, agents are used to generate design alternatives. Configuration-agents create design configuration by attaching various embodiment structure enhanced from that of Welch and Dixon [Welc94] to fulfill the input function parameters and output function parameters. Instantiation-agents extract the equations from the design configuration and choose actual components from a computer catalog by choosing the exact values of variables in the design [Camp00]. The algorithm can only handle problems in the form of inputs and outputs.

Qian presented an analogy-based synthesis approach based on function-behavior-structure representations. His computational model includes knowledge base. It starts with a search for an existing design based on keywords. A subsequent designer guided search is performed to find designs with similar function or behavior. New designs are generated by combining the retrieved existing designs with the retrieved analogous designs [Qian02].

Deng and Lu proposed a synthesis framework for MEMS conceptual design based on their FEBS representation schemes. Characteristics of MEMS such as chemical/biological/other reactions are also considered in the behavior representation

[Deng04b]. The approach is limited to problems defined in terms of inputs and outputs.

Graph grammars also have been used to generate design solutions. Graph grammars are a set of rules that could transform the nodes and arcs in a graph representing design specifications [Fu93, Li01]. Shridharan and Campbell applied graph grammar to create function structures. Grammar rules are extracted from studying function structures of thirty products. New solutions can be generated by applying these rules to the input given by the user [Srid04]. Schmidt and Cagan developed a synthesis algorithm that uses a graph-based abstraction grammar to create design alternatives and a recursive simulated annealing process to select a near-optimum design [Schm97]. Grammar based approaches are limited to specific types of products since different grammars have to be developed for different products.

Subramanian and Wang developed an algorithm for synthesizing single-input, single output mechanisms. They used recursive search algorithm that starts from the desired output and work backwards to the specified input. If several primitive mechanisms are identified during the process, they randomly pick one and continue [Subr95]. This method cannot be applied to complex device synthesis with multiple inputs and outputs.

2.5 Summary

The research in representation of design concepts is mainly focused on single interaction-state systems, where the relationship between different components of system is fixed during operation of the product. However, changing interaction relationships under different operation modes is the basis for multi-state mechatronic

systems to work. Representing design concepts of multi-state mechatronic systems not only requires representing the components of the system, but also new features such as changing interaction topologies among system components. Besides, mechatronic product design requires us to consider complex interactions between system elements. These interactions may not necessarily be of the form of input/output relationships. Instead, these interactions in the most general form may need to be expressed as constraints or other types of complex relationships among parameters of various components. In order to describe complex requirements, we will also need to explicitly model the use-environments. In order to simulate the design concept, we will also need to formally define the events that trigger different interactions. Finally, we will need to add explicit notions of unsafe states in the representations to ensure that the concepts being represented are safe. This requires us to develop a new representation based on the combinations of existing representations to capture the behavior exhibited by multi-state devices.

Evaluation and validation is closely coupled to representation. Current design validation methods available as a part of function-based design will not be able to perform validation due to the additional validation requirements imposed on the new representation. Thus a new validation methodology will be needed for the new representation. Currently available evaluation techniques cannot fully exploit additional information present in the new representation. Moreover, most evaluation methods do not consider design concepts explicitly and do not consider how the evaluation will be performed using the available information. Thus, we will need to develop new evaluation algorithms that can exploit the additional information

available in the design concepts and assist the users in comparing two different possible concepts.

Increasingly, a large number of new electro-mechanical devices are designed using off-the-shelf components and sub-systems. The main challenge in designing such products is to arrive at the optimal product configuration that satisfies the functional specifications and at the same time is cost effective. Typically, a large number of potential product configurations exist for a required end-user functionality. For example, linear mechanical motion can be realized through a number of different product configurations consisting of a combination of motors and mechanisms such as cams, screw systems, piston drives etc. Synthesizing the optimal configuration from a large number of alternatives is a challenging task. Currently, designers tend to make decisions based on their intuition and past experiences with the previous designs. Such design practices lead to long delays in incorporating new components and new solutions into current product designs. The development of a synthesis algorithm will significantly assist designers in effectively exploiting new off-the-shelf components.

Synthesis algorithms are also closely related to specific design representations. The new representation developed as a part of this dissertation will enable us to describe behavior of complex components. Moreover, explicit modeling of use-environment will aid in selection of the appropriate components from the component library. Hence, we plan to develop a new synthesis algorithm that can effectively utilize existing basic and complex components.

Chapter 3: Modeling and Simulation Framework

This chapter provides a framework for modeling design concepts of mechatronic devices with multiple interaction-states to facilitate computer-aided conceptual design of such devices. This chapter introduces the primitives and operators used in the modeling framework, and illustrates the modeling process by an example.

This chapter has been organized in the following manner. Section 3.1 describes the background information related to this chapter. Section 3.2 describes the class definitions for the primitives used in the framework to model behavior specifications. Behavior specifications describe how the device interacts with different components of the use-environment under different conditions. We use transition diagrams to capture the behavior specifications. Every component and device is modeled as an artifact. After defining the behavior specifications, conceptual design is carried out. This step entails decomposing the device into components and further elaborating the interactions among different components of the device and the use-environment. Operators for elaboration to support the conceptual design process are described in Section 3.3. Section 3.4 describes how the primitives can be used to model the behavior specifications and how the elaboration operators can be used to transform the behavior specifications into a design concept. Section 3.5 describes an algorithm for simulating a transition diagram. Section 3.6 presents an example. Finally, Section 3.7 summarizes this chapter.

3.1 Background

The following terminology will be used in the remainder of this chapter.

***Use-environments*:** We define the use-environment as part of the world with which the mechatronic device interacts in its lifetime. Modeling the use-environment becomes necessary to describe the desired behavior of complex devices. Consider the energy flow in a hybrid vehicle. The road is a part of the use-environment in this case. The changes in road conditions alter the operating modes of the hybrid vehicle. Therefore, road conditions are needed to describe the desired behavior of the hybrid vehicle.

***Mechatronic Devices*:** Mechatronic devices refer to the devices that integrate components from mechanical, electrical and electronic, and information domains. A mechatronic device interacts with its use-environment to meet the customer needs. Components in a mechatronic device interact with each other and components of the use-environment to produce the desired behavior. For example, a hybrid car is a mechatronic device, which is composed of a passenger cabin, an engine, a battery, a motor, a control system, a transmission system, and four wheels.

***Design Worlds*:** A design world consists of entities that need to be modeled to carry out the device design and describes how the device produces the desired behavior. In our case, it is the combination of a mechatronic device and its use-environment. In the hybrid vehicle design example, the hybrid vehicle to be designed and the road comprise the design world.

***Component Interactions*:** Components in a mechatronic device and its use-environment may interact with each other in different ways. Such interactions include energy, signal, and mass flows between components, and two or more components mutually constraining each other's motions. For example, the engine transmits power

to the transmission system in the hybrid vehicle. Therefore the engine interacts with the transmission system.

Mechatronic Device Behaviors**:** The behavior of a mechatronic device is the way in which the device interacts with its use-environment over time by responding to the changes in the use-environment, and affects the use-environment. For example, the behavior of a hybrid vehicle is as following: (1) the engine and the battery both provide power to the wheels when the vehicle travels uphill or accelerates, (2) the engine provides power to the wheels and it charges the battery when the vehicle travels downhill or decelerates, and (3) the engine provides power to the wheels when the vehicle travels along a horizontal road.

Requirements: Requirements define the customers' needs for a product. In this chapter, requirements are considered as the description of the services provided by a device to its use environment. Subject and verb pairs typically describe these requirements. Subjects are typically components of the device and the use-environments. Additional specifications are used to include constraints on the requirements. For example, requirements for a hybrid vehicle may be stated as following: (1) hybrid vehicle stores spare power being produced by engine when vehicle encounters reduced load; (2) hybrid vehicle delivers stored energy to wheels when vehicle encounters increased load.

Behavior Specifications**:** Behavior specifications formally state the specifications of the observable behavior of a device that would satisfy the stated requirements. These specifications include how the device will interact with use-environment under different conditions and how the device and components of the use-environment will

mutually affect each other. In the subsequent sections of this chapter, we use transition diagrams to represent the behavior specifications of the device being designed.

Working Principles**:** We define working principles as the basic physical principles behind device components.

Design Concepts**:** We call the result of conceptual design stage a design concept. A design concept needs to have the following three main ingredients. First, the design concept will need to identify various major components that will be needed to meet the requirements. Second, the design concept will need to specify basic working principle behind every main component to ensure that the component is realizable. Third, the design concept will need to specify how various components will interact with each other.

3.2 Class Definitions for Modeling Primitives

In this section we define classes for the modeling primitives used in our framework. Figure 3.1 shows the overall conceptual design modeling framework and main primitives used in this framework. The rationale behind the main primitives shown in this figure is as following. We need to be able to models *events* in the use-environments to which the design concept will respond. To ensure the safety of the device operation, we need to be able to model *unsafe parameter value sets*. These are the parameter value sets that the device should never enter because it can cause significant operational difficulties or create hazardous conditions. For example, when the door of a machine tool is open, the spindle should not rotate. *Engineering*

characteristics are needed to specify quantitative constraints associated with the operation of the device being designed.

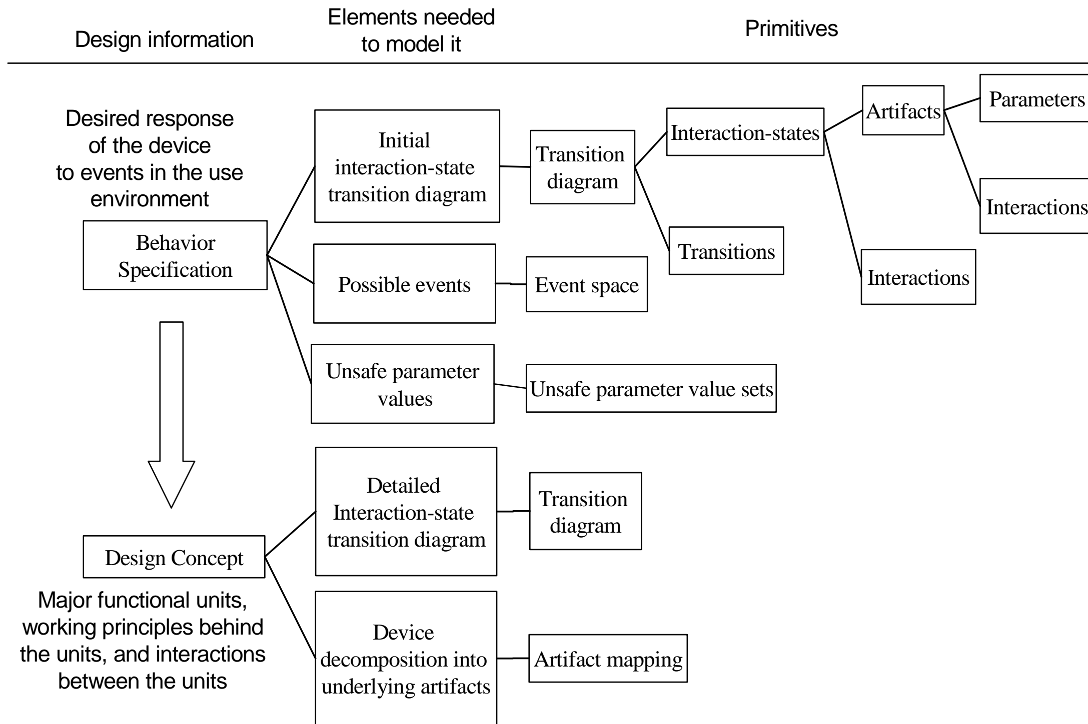


Figure 3.1: Overview of primitives

We need *transition diagrams* to model how the device (or components of the device) interacts with the use-environment in response to various events in the use-environment. Figure 3.2 shows the primitives used in defining interaction-state transition diagrams. Dependency among various primitive definitions is depicted in Figure 3.3. The primitive at the start of an arrow is needed for defining the primitive at the end of the arrow.

Every class instance will have a name that will serve as the identification for the class instance. We use notation "*name.member*" in this chapter to refer to a member of a class instance. For example, notation *a.p* refers to member *p* of class instance with

name *a*. In the following subsections we introduce the class definitions for various primitives.

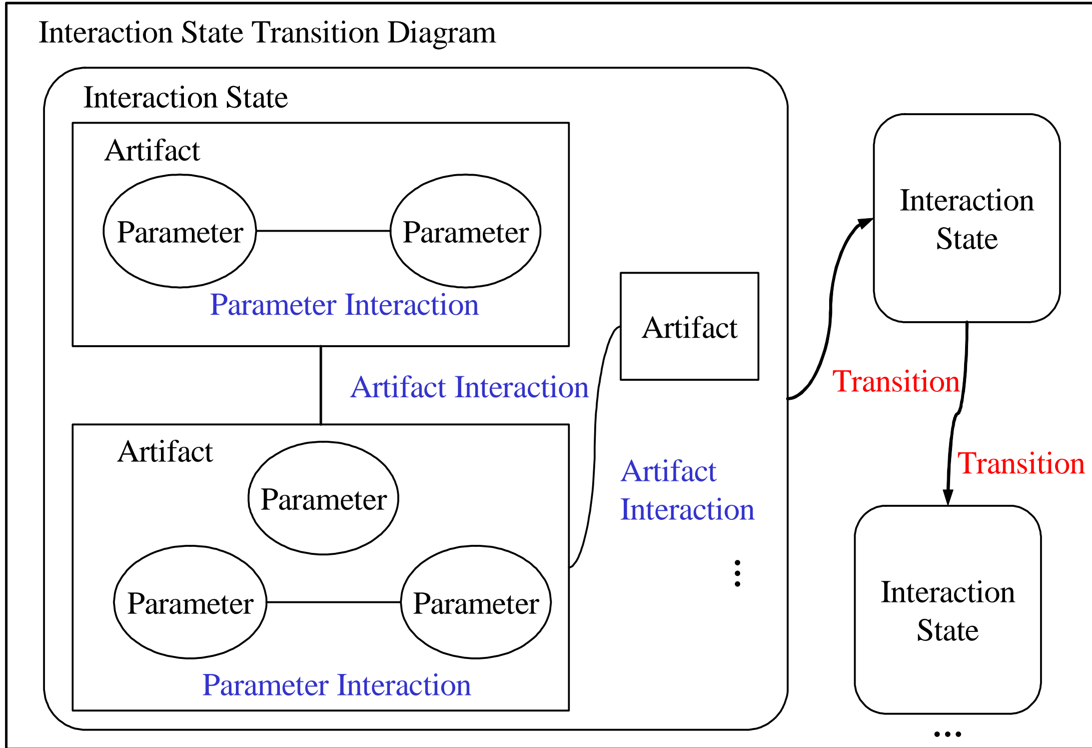


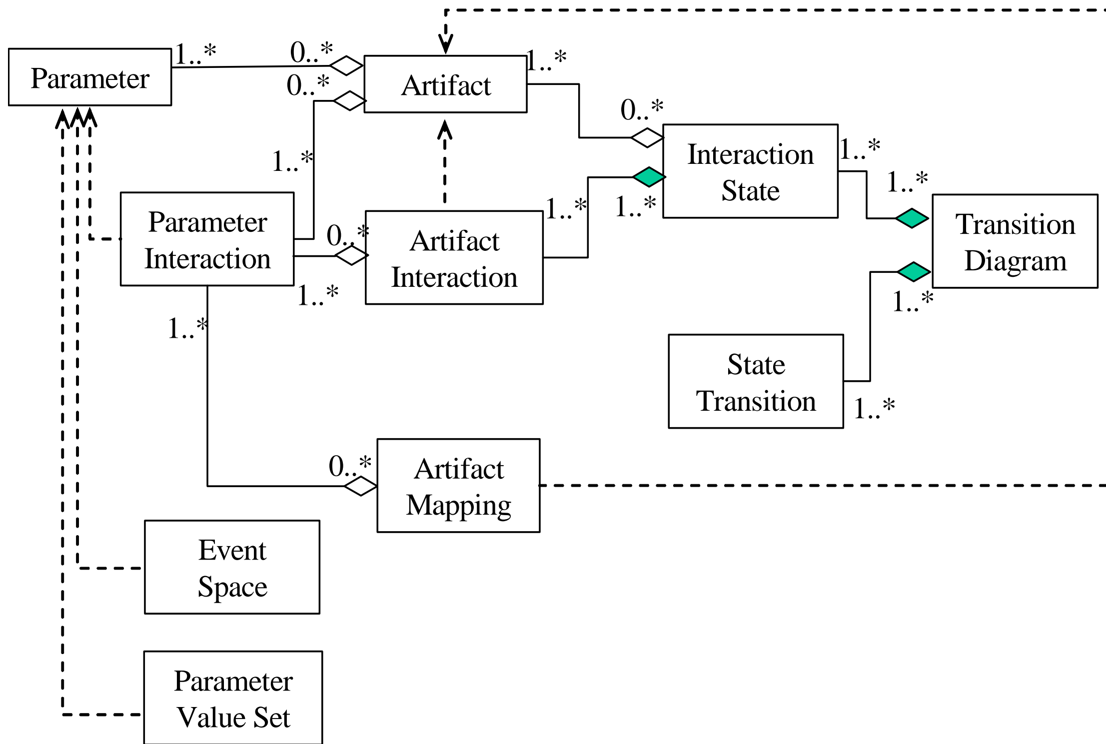
Figure 3.2: Structure of interaction state transition diagram

3.2.1 Classes for Modeling Parameters and Parameter Interactions

A parameter is a type of observation of an artifact. For example, a table is an artifact, and the height of the table is a parameter. Class **Parameter** is defined using the following members:

- *DataType* indicates the data type of this parameter. Parameter can be of several different data types. Our framework supports basic data types such as *INTEGER*, *REAL*, *BOOLEAN*, and *STRING*. We also support user-defined data types that are defined by using class **UserDefinedDataType** in terms of basic data types.

- *Unit* is a string that describes the unit of a piece of data. If the unit is not required, then it is set to *NONE*.



Legend:

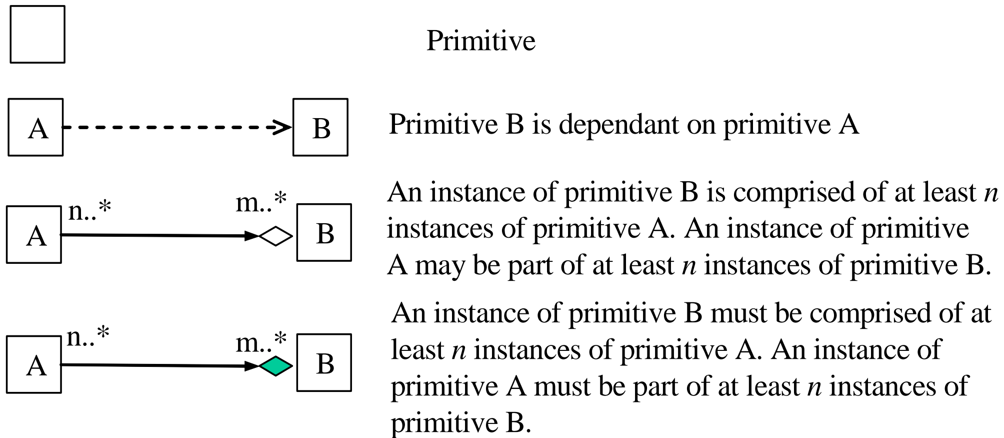


Figure 3.3: Relationships between major primitives

Class **UserDefinedDataType** is defined using member *Fields,* a set of names of *Parameter* instances.

For example, **Parameter** *position* can be defined in the following manner:

<table>
<tr><td colspan="3"><u>position : Parameter</u>
DataType = positionVector
Unit = NONE</td><td colspan="3"><u>positionVector : UserDefinedDataType</u>
Fields = {x, y, z}
Unit = NONE</td></tr>
<tr><td colspan="2"><u>x : Parameter</u>
DataType = REAL
Unit = “mm”</td><td colspan="2"><u>y : Parameter</u>
DataType = REAL
Unit = “mm”</td><td colspan="2"><u>z : Parameter</u>
DataType = REAL
Unit = “mm”</td></tr>
</table>

As will be introduced in detail in the subsequent sections, parameters will be assigned values within states at specific time instances. If the data type of a parameter is a basic data type, then the value of this parameter is represented as a number, string, or symbol (e.g., 3.002, “mm”, *TRUE*). Values for user defined data types are represented as sets of expressions. For example, {(x=2), (y=4), (z=3)} represents a possible value for a user defined data type *position vector.*

Parameter may also take *NONE* or *NA* (not available) as a value for convenience. When a parameter does not have a value, we assign its value as *NONE*. When the value of a parameter is not known at the time of modeling, we assign the value as *NA*.

Relationships among parameters are called parameter interactions. From the perspective of the governing equations behind the relationships, there are two types of interactions:

- *Declarative Interactions*: These can be modeled using algebraic or ordinary differential equations. For example, the interaction of the mass parameter and the volume parameter of an artifact with uniform density is given by $m=dv$, where m is the mass, d is the density, and v is the volume of the artifact. However, in conceptual design stage, the exact equation may not be available. A qualitative structure that describes the characteristics of the interaction is then used.

- *Procedural Interactions*: These cannot be modeled explicitly using algebraic or ordinary differential equations during conceptual design. In most of the cases, a procedure is needed to describe these interactions. If simulation of a design concept is necessary, then simplified numerical simulation can be used as surrogates for these interactions. For example, the interaction among a light source, a person, and, image at the camera lens (i.e., light from the light source reflects from person's face and forms an image at the camera lens) cannot be modeled by algebraic equations or ordinary differential equations.

We define class **ParameterInteraction** using the following members:

- *InteractionReason* is a tag taken from the following options:
  - *ENERGY FLOW* indicates energy flow.
  - *SIGNAL FLOW* indicates signal flow.
  - *MASS FLOW* indicates mass flow.
  - *SPATIAL CONSTRAINT* indicates spatial constraints among a set of components.
  - *LAW* indicates physical laws governing relationships among physical parameters of a component.
  - *OTHER* indicates all other types of interactions.
- *InteractionType* is a tag taken from the following options:
  - *NON-CAUSAL INTERACTION*: For these interactions, there is no need to specify the dependence among parameters. For example, the interaction of the mass and the volume of an artifact with uniform density is a non-causal interaction.

- *CAUSAL INTERACTION*: For these interactions, we have to specify the dependent relationships between parameters.
- *ParameterSet* is a set of names of **Parameter** instances that interact with each other.
- *DependantParameter* is the name of a **Parameter** instance whose value is dependent on the other parameters belonging to a *ParameterSet* as a result of the interaction. For non-causal interactions, *DependantParameter* is set to *NONE.*
- *Equation* is an algebraic or ordinary differential equation (in terms of parameters) if the interaction is declarative. In this case, it is defined as an instance of class **Expression**. If the interaction is procedural or the exact form of the equation is not available, then we don't capture the equation. Therefore this field is set to *NA*.

Class **Expression** is defined using a member called *Content*, a special type of string that starts and ends with a parenthesis symbol. It includes numbers, standard and user defined function names, logical symbols, and mathematical symbols.

3.2.2 Classes for Modeling Artifacts, Artifact Interactions, and Artifact Mappings

An artifact is a finite collection of parameters and the interactions among these parameters. Class **Artifact** is defined using the following members:

- *InputParameterSet* is a set of names of **Parameter** instances. These parameters serve as the input ports for flow types of interactions among artifacts.

- *OutputParameterSet* is a set of names of **Parameter** instances. These parameters serve as the output ports for energy and signal flow types of interactions among artifacts.
- *GeneralParameterSet* is a set of names of **Parameter** instances. These parameters do not play input or output role.
- *ParameterInteractionSet* is the set of names of **ParameterInteraction** instances describing interactions among parameters belonging to the artifact.
- *ArtifactType* is a tag assigned to either *USE-ENVIRONMENT* or *DEVICE* to classify two different types of artifacts.

For example, let us consider a DC motor without load. It can be represented by

motor **: Artifact**	***c*** **: ParameterInteraction**
InputParameterSet = {*v*, *k*}	*InteractionReason* = *LAW*
OutputParameterSet = {ω}	*InteractionType* = *CAUSAL INTERACTION*
GeneralParameterSet = {*weight*}	*ParameterSet* = {*v*, *k*, ω}
ParameterInteractionSet = {*c*}	*DependantParameter* = ω
ArtifactType = *DEVICE*	*Equation* = ($\omega = v/k$)

v **: Parameter**	***k*** **: Parameter**	***ω*** **: Parameter**	***weight*** **: Parameter**
DataType = *REAL*	*DataType* = *REAL*	*DataType* = *REAL*	*DataType* = *REAL*
Unit = “m/s”	*Unit* = *NONE*	*Unit* = “rad/s”	*Unit* = “kg”

Where *v* is the input voltage, *k* is the motor constant, ω is the no-load speed.

If *a* is the name of an **Artifact**, then we use notation *a*::*p* to refer to **Parameter** *p* of **Artifact** *a*.

Artifacts interact with each other to affect their mutual behaviors. Complex artifacts can also be decomposed into simple artifacts. These two relationships about artifacts are modeled using classes **ArtifactInteraction** and **ArtifactMapping**.

Artifact interactions usually result due to interactions among their parameters. We define class **ArtifactInteraction** using the following members:

- *ArtifactSet* is the set of names of the **Artifact** instances in the interaction.
- *InteractionInfo* is defined as the set of names of **ParameterInteraction** instances that describe the parameter interactions between the artifacts.

All artifact interactions can finally be modeled as parameter interactions

An artifact mapping is defined as the relationship between an artifact and its children artifacts. The relationship includes artifact hierarchy and parameter mapping between parent artifact and children artifacts. We define class **ArtifactMapping** using the following members:

- *Artifact* is the name of the **Artifact** instance being decomposed.
- *ChildrenArtifactSet* is the set of the names of children **Artifact** instances resulting from the decomposition of *Artifact*.
- *ParameterMappingSet* is a set of **Expression** instances. Each **Expression** instance defines the relationship between the parent artifact's parameters and its children artifacts' parameters. For example, suppose parameter p_1 of parent a_1 is mapped to parameter p_2 of children artifact a_2 and parameter p_3 of children artifact a_3. Then, a possible expression can be ($a_1::p_1 = a_2::p_2 + a_3::p_3$).

3.2.3 Classes for Modeling Interaction-States

An interaction-state describes the invariant interactions between a set of artifacts. For example, if a motor is driving a gearbox to transmit mechanical energy, then the interaction-state of this set of artifacts is the description of the motor, the power source, the gearbox, and their interactions. Every artifact in the artifact set of this

interaction-state must participate in at least one artifact interaction in this state. An artifact is ***active*** in the interaction-state if it belongs to the artifact set of the state. Otherwise, the artifact is considered *inactive* in the state. Usually when we refer to the artifacts in a state, we refer to the active artifacts in the state.

We use symbol t to denote the time variable associated with the internal clock of the state. We call t the ***local time variable*** because t only exists with respect to a specific state. On the other hand, when simulating a design concept, we need another variable to indicate the time in the design world, which includes all states of the device. This time variable is denoted as T and is called the ***global time variable***. At a given global time $T=T^*$, the device is in a particular state with its own corresponding local time $t=t^*$. Within a state, t starts from 0. Ending time of a state is denoted by symbol t_e. At a particular time t', the value of a parameter p of artifact a is denoted by $a::p(t=t')$. $a::p(t)$ is used to represent the value of a parameter parametrically. On the other hand, if the global time variable is used to indicate the value of a parameter, we use $a::p(T=T')$ for a specific time, and $a::p(T)$ to represent it parametrically.

Notation $s::a::p$ will be used to refer to the **Parameter** p of **Artifact** a in **State** s.

We define class **InteractionState** using the following members:

- *ArtifactSet* is a set of names of **Artifact** instances that are active in the state.
- *ArtifactInteractionSet* is a set of names of **ArtifactInteraction** instances between the active artifacts in this state.
- *InitialValueSet* is a set of names of class **ValueAssignment** instances that describes how parameter values are initialized.

- *ChangeModeSet* is a set of names of class **ChangeMode** instances that describes how parameter values will change inside of the interaction-state.

Class **ValueAssignment** is defined using the following members:

- *ParameterName* is the name of a **Parameter** instance.
- *InitializationType* is a tag taken from the following options:
    - *INHERIT* indicates that the parameter inherits its value from the previous state. Let the current state be s, and its previous state be s', then the initial value of a parameter $a::p$ belonging to this artifact can be obtained in the following manner: $s::a::p\ (t = 0) = s'::a::p\ (t = t_e)$, where t_e is the ending time of state s'.
    - *DERIVE* indicates the value of a parameter is derived from other parameter values that belong to some artifacts in the same state.
    - *ASSIGN* indicates the value of a parameter is assigned to a particular value.
- *Value* denotes the value of a parameter. If the *InitializationType* is set to *INHERIT* or *DERIVE*, *Value* is set to *NA*.

Class **ChangeMode** is defined using the following members:

- *ParameterName* is the name of the **Parameter** instance.
- *ChangeType* is a tag taken from the following options:
    - *CONSTANT* indicates the value is a constant within the state.
    - *DERIVE* indicates the value changes according to the values of parameters it interacts with.
    - *EQUATION* indicates the value is changing according to time variable t.

- *Equation* is an equation in terms of a parameter with respect to the local time in a state. In this case, it is defined using class **Expression**. If the *ChangeType* is set to *CONSTANT* or *DERIVE*, it is set to *NA*.

Some limitations may apply on combining initialization types and value-changing modes as shown in the Table 3.1

Table 3.1: Limitations on combining initialization types and value-changing modes

Initialization type	**Value changing mode**
ASSIGN	*CONSTANT*
	EQUATION
INHERIT	*CONSTANT*
	EQUATION
DERIVE	*DERIVE*

States may be inconsistent if the underlying interactions are inconsistent. An *Interaction-state s* is ***inconsistent*** if equations defined in *ArtifactInteractionSet* are inconsistent. Equations may turn out to be inconsistent if the system of equations is over-constrained.

Let X be the set of parameters belonging to all the artifacts in an interaction-state s. Let F be the set of interactions in state s defined over X. Each f in F is a subset of X and describes an interaction. During the conceptual design stage we only consider the qualitative nature of interactions.

- Set $X = \{x_1, \dots x_n\}$
- Set $F = \{f_1, \dots f_m\}$, where each $f_i \subseteq X$ and $\cup F = X$
- $n \geq m$

The problem of interaction consistency is to determine if there exists $F' \subset F$ such that cardinality(F') > cardinality($\cup F'$). If such F' exists, then the given set of interactions is considered inconsistent.

For example, consider an interaction-state that has seven parameters. Let the set of parameters for this state be defined as $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$. Let the interactions among these parameters be characterized by the following set of relationships:

$f_1(x_1, x_2) = 0,\ f_2(x_2, x_3) = 0,\ f_3(x_3, x_4) = 0, f_4\ (x_2, x_4) = 0,\ f_5(x_1, x_4) = 0, f_6(x_5, x_6, x_7) = 0.$

Although overall there are seven parameters and only six relationships, but the first five relationships (i.e., $f_1(x_1, x_2) = 0, f_2(x_2, x_3) = 0, f_3(x_3, x_4) = 0, f_4(x_2, x_4) = 0, f_5(x_1, x_4) = 0$) only involve four variables x_1, x_2, x_3, and x_4. Therefore these relationships are over-constrained. Thus, the interactions in this state are inconsistent and this state is inconsistent.

3.2.4 Classes for Modeling Event and Event Spaces

An event occurs when a use-environment artifact becomes active or inactive, or a parameter or parameters of the use-environment artifacts change their values. Event space refers to the set of all possible events that can happen in the use-environment. Class *Event Space* is defined using the following members:

- *ParameterRangeSet* is a set of names of **ParameterValueRange** instances. These parameters belong to the use-environment artifacts.

Class **ParameterValueRange** is defined using the following members:

- *Parameter* is the name of a **Parameter** instance.
- *RangeType* is a tag taken from the following options:

  - *CONTINOUS* means that values are bound between *ValueLowerLimit* and *ValueUpperLimit.*
  - *DISCRETE* means that values are assigned from a *ValueSet.*
- *ValueSet* is a set of **Expression** instances. If *RangeType* is set to *CONTINOUS*, *ValueSet* is set to *NA*.
- *ValueLowerLimit* is a value for the parameter. If *RangeType* is set to *DISCRETE*, *Value LowerLimit* is set to *NA*.
- *ValueUpperLimit* is a value for the parameter. If *RangeType* is set to *DISCRETE*, *ValueUpperLimit* is set to *NA*.

During the simulation of a design concept, global time variable *T* represents time in the design world, which includes the device and the use-environment. Every event happens in the use-environment at a certain specific value of *T*.

We define class **Event** using the following members:

- *GlobalTime* is the value of the global timer that describes the time when this event happens.
- *EventCondition* is defined as an instance of class **Expression** that describes parameter value changes during the event.

The following **Expression** instances show several examples of event conditions:

(*a* = ACTIVE). This expression means that **Artifact** *a* became active at design world time *T*=4. (*a*::*p* = 3). This expression means that **Parameter** *p* of **Artifact** *a* takes value of 3 at design world time *T*=5. (*a*::*p* = 3+). This expression means that the value of **Parameter** *p* of **Artifact** *a* takes is incremented by 3 at design world time *T*=6.

To facilitate efficient simulations, the current modeling framework has the following limitations. Events can only involve use-environment parameters. Use-environment parameters that are used to define events are called event parameters. Event parameters affect device parameters only during initialization of a state. Device parameters cannot affect values of event parameters.

3.2.5 Classes for Modeling Unsafe Parameter Value Sets

A parameter value set is a snapshot of an interaction-state. In a transition diagram, interaction-states may contain a set (possibly infinite) of parameter value sets. A unique parameter value set can be extracted from an interaction-state by selecting a specific time instant in the interaction-state. For example at $T = 5$, the values of all parameters belonging to both the device and the use-environment artifacts define the world-state at $T = 5$. An unsafe parameter value set is a parameter value set that is forbidden by requirements.

Class **UnsafeParameterValueSet** is defined using member *ParameterValueSet*, where *ParameterValueSet* is a set of **Expression** instances that indicates the forbidden parameter values or value ranges by the requirements.

A design concept should never enter an unsafe parameter value set. Therefore, a design concept should be such that in response to all possible events contained in the event space, it should never enter an interaction-state that will contain unsafe parameter value sets.

3.2.6 Classes for Modeling Interaction-State Transitions and Transition Diagrams

An interaction-state transition is the indication of changes from one interaction-state to another interaction-state. We define class **InteractionStateTransition** using the following members:

- *StartState* is the **InteractionState** instance where the transition starts.
- *EndState* is the **InteractionState** instance where the transition ends.
- *TransitionCondition* is an **Expression** instance that indicates the condition under which the transition occurs. This is a composite expression that may contain (1) sub-expressions indicating the internal time clock of a state reaching a particular value, such as (t=4) or (2) sub-expressions indicating some parameters taking particular values, such as ($a::p(t)$=5).
- *ClosureActionSet* is a set of **Expression** instances that describes how the parameters value will be set in the starting state before leaving it. For example, $\{(a_1::p_1(t=t_e) = 1), (a_1::p_2(t=t_e) = 2), (a_2::p_1(t=t_e) = 3)\}$.
- *Initialization Action Set* is a set of **Expression** instances that describe how the parameters value will be set in the ending state before entering it. $\{(a_1::p_1(t=0) = 2), (a_1::p_2(t=0) = 3), (a_2::p_1(t =0) = 3)\}$. Expressions in this set override the initialization expressions defined for a state.

InteractionStateTransition r is ***realizable*** for **InteractionState** s if there exists a sequence of events such that the device reaches s and transition condition for r is satisfied. If a transition is not realizable, then it is called ***unrealizable***. Unrealizable transitions should be eliminated from the design concept, as they do not contribute anything to the behavior.

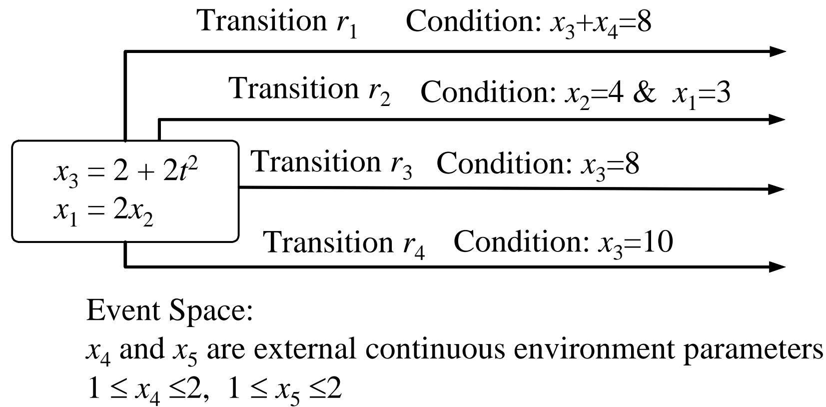


Figure 3.4: Unrealizable transitions

A transition may be unrealizable because of a variety of reasons:

- The condition for some other transition will always be satisfied before condition for this transition is satisfied. Transition r_4 in Figure 3.4 is unrealizable because condition for transition r_3 is always satisfied before condition for transition r_4 is satisfied. Therefore, transition r_4 never takes place.
- Interactions in the state rule out the possibility of the transition condition being satisfied. Transition r_2 in Figure 3.4 is unrealizable because $x_2 = 4$ and $x_1 = 3$ cannot be satisfied due to an interaction between x_1 and x_2.
- The event space does not allow the condition for this transition to be satisfied. Transition r_1 in Figure 3.4 is unrealizable because condition for this transition cannot be satisfied due to restriction on the ranges of parameters x_4 and x_5.

A transition diagram is a graph whose nodes are interaction-states and edges are interaction-state transitions. We define class **TransitionDiagram** using the following members.

- *InitialState* is the name of a special **InteractionState** instance. Every transition diagram must include an initial state, which is the device interaction-state at $T = 0$.

As a special interaction-state, the initial state has all the artifacts including device artifacts and use-environment artifacts. Parameters of these artifacts are initialized in the initial state. However, all the artifacts remain inactive until events trigger the device to leave the initial state.

- *InteractionStateSet* is the set of names of remaining **InteractionState** instances.
- *InteractionStateTransition Set* is the set of names of **InteractionStateTransition** instances.

A transition diagram is considered ***safe*** with respect to an event space E and a set of unsafe world-states U, if there does not exist a sequence of events E_s that results in one of the unsafe world-states. Figure 3.5 graphically shows an example of an ***unsafe*** transition diagram that reaches an unsafe world-state.

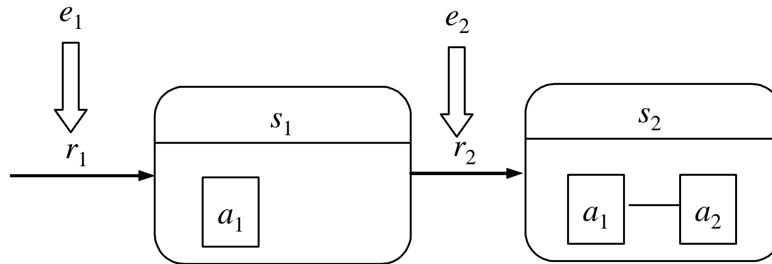


Figure 3.5: Example of unsafe transition diagram

In this example, p_1 is a parameter of artifact a_1 and p_2 is a parameter of artifact a_2. This diagram has four interaction-states including initial state s_0. In each state, the local time variable t is from 0 to some ending time t_e. In state s_1, we have

$s_1::a_1::p_1(t)=s_1::a_1::p_1(t=0) + 1$.

In state s_2, we have

$s_2::a_1::p_1(t)=s_2::a_1::p_1(t=0) + 2t$, $s_2::a_2::p_2(t)=s_2::a_1::p_1(t) + 1$.

The definition of the event space and the unsafe parameter value set are as the following:

***e* : EventSpace**	***w* : ParameterRange**
$ParameterRangeSet = \{w\}$	$Parameter = p_1$ $RangeType = CONTINUOUS$ $ValueSet = NONE$ $ValueLowerLimit = 0$ $ValueUpperLimit = 10$
***u* : UnsafeParameterValueSet**	
$ParameterValueSet = \{(a_1{::}p_1{=}4), (a_2{::}p_2{=}5)\}$	

The transition condition from state s_1 to s_2 is defined using expression $(a_1{::}p_1(t){=}3)$. Thus, when an event $(a_1{::}p_1(t{=}0){:=}2)$ happens, it will result in unsafe parameter value set u, which happens in s_2, when t=0.5.

We define a transition diagram as ***valid*** when the following conditions are met:

- Every state in the transition diagram is consistent.
- Every transition in the transition diagram is realizable.

Given a valid transition diagram and an event space, we can simulate how the transition diagram responds to different events in the event space.

3.3 Elaboration Operators

Primitives are building blocks for modeling design concepts. In our modeling framework, we use operators for constructing and manipulating these primitives. According to the usage, these operators are classified into two categories: constructor operators for constructing primitives and elaboration operators for elaborating primitives.

Each primitive has its own *CONSTRUCT* operator that is similar to the concept of constructor used in object-orientated programming languages. A *CONSTRUCT* operator takes input parameters to perform the initialization of a primitive. When a *CONSTRUCT* operator is called, it first checks if input parameters are sufficient for constructing the primitive. If not, it will return failure. The construction of a transition diagram should be performed in a bottom up manner. That is, first construct the lower level primitives such as parameters, artifacts etc, then construct higher level primitives such as interaction-state and then finally the transition diagram.

In our framework, an initial transition diagram, which represents the specifications of observable behaviors of a device, will be constructed first. After that, the conceptual design is performed by elaborating the initial transition diagram and by creating the internal structures of the mechatronic device being designed. The following operators are used for this purpose.

- **Decompose Artifact**. This operator is called *DECOMPOSE-ARTIFACT* and used to decompose an artifact into a set of artifacts. This operator is defined as the following.
    - Input: artifact a, transition diagram D in which a exists.
    - Output: a set of artifacts A, the artifact mapping M between a and A, and the new transition diagram D' after a is decomposed.
    - Action: decompose a into A by establishing an artifact mapping between a and A. Replace a in D, which leads to D'. The artifact interactions involving a in D will be converted to artifact interactions involving A.

- Precondition: Working principles for the input artifact are not known. Therefore, this artifact has to be treated as a complex artifact and has to be decomposed further.

For example, as shown in Figure 3.6, artifact a_1 has two parameters p_1 and p_2, h is a parameter based artifact interaction between a_1 and a_2 in D as defined in the following:

***h* : ArtifactInteraction**	***c* : ParameterInteraction**
ArtifactSet = $\{a_1, a_2\}$	*InteractionReason* = *ENERGY FLOW*
InteractionInfo = $\{c\}$	*InteractionType* = *CAUSAL INTERACTION*
	ParameterSet = $\{a_1::p_1, a_1::p_2, a_2::p_3\}$
	DependantParameter = $a_2::p_3$
	Equation = $(a_2::p_3 = a_1::p_1 + a_1::p_2)$

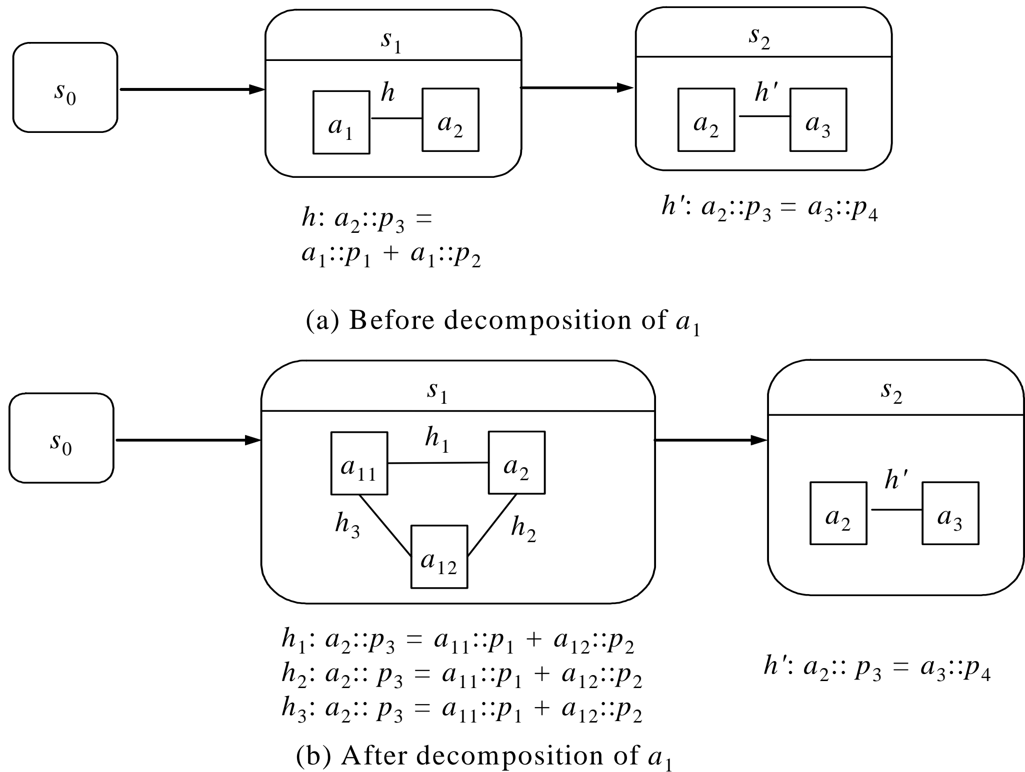


Figure 3.6: Usage of operator *DECOMPOSE-ARTIFACT*

Applying operator *DECOMPOSE-ARTIFACT* will decompose a_1 into two sub-artifacts: a_{11} and a_{12}. Now h is converted into three artifact interactions h_1, h_2 and h_3 in D' as defined in the following:

<u>h_1 : ArtifactInteraction</u> $ArtifactSet = \{a_{11}, a_2\}$ $InteractionInfo = \{c'\}$	<u>h_2 : ArtifactInteraction</u> $ArtifactSet = \{a_{12}, a_2\}$ $InteractionInfo = \{c'\}$
<u>h_3 : ArtifactInteraction</u> $ArtifactSet = \{a_{11}, a_{12}\}$ $InteractionInfo = \{c'\}$	<u>c′ : ParameterInteraction</u> *InteractionReason = ENERGY FLOW* *InteractionType = CAUSAL INTERACTIONS* $ParameterSet = \{a_{11}::p_1, a_{12}::p_2, a_2::p_3\}$ $DependantParameter = a_2::p_3$ $Equation = (a_2::p_3 = a_{11}::p_1 + a_{12}::p_2)$

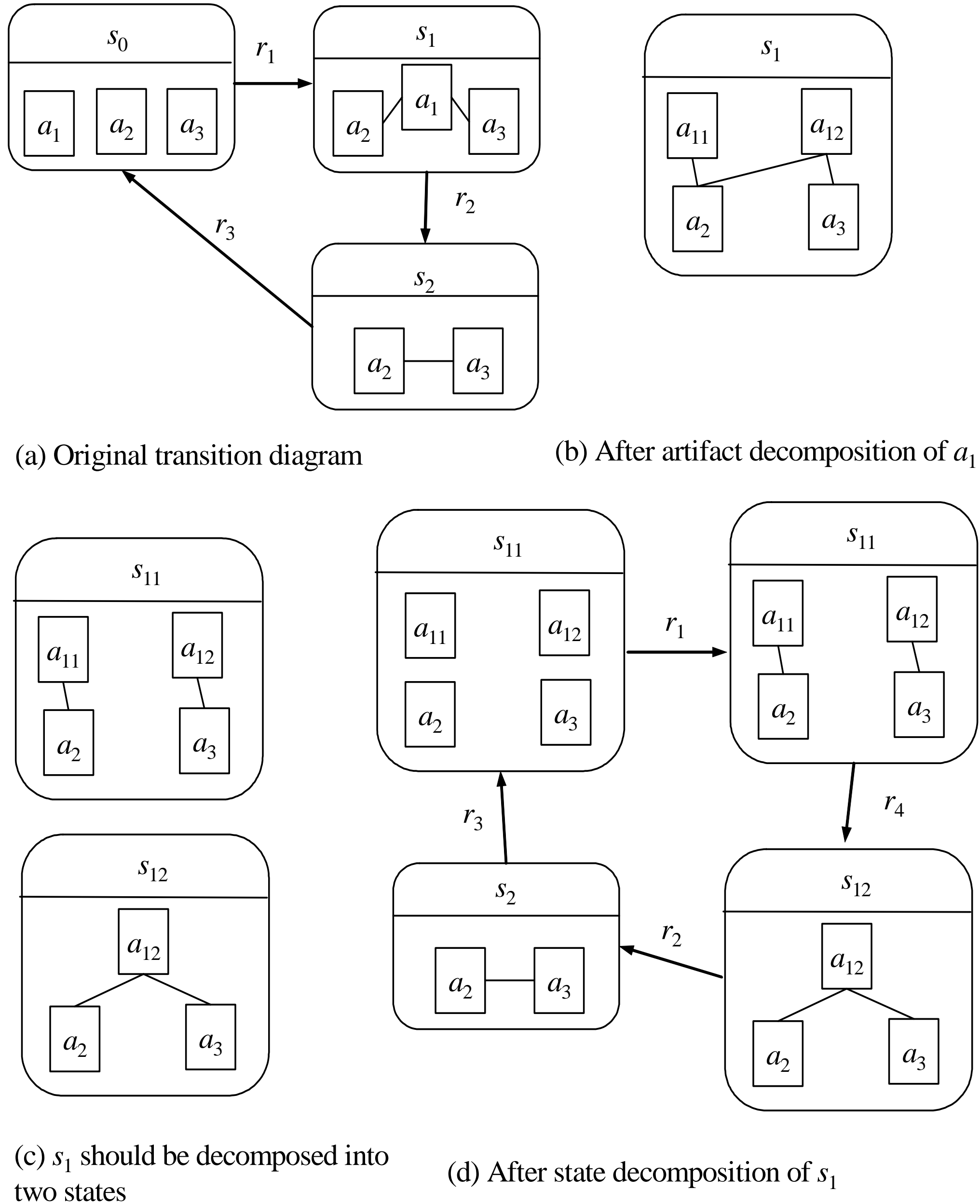
Artifact decomposition should follow the following constraints.

- **Maintain parameter consistency between children artifacts and parent artifact.** Let $\{p_1, \ldots, p_n\}$ be set of parameters belonging to Artifact a and let $\{p_{i1}, \ldots, p_{ij}\}$ be the set of parameters for an artifact $a_i \in A_i$. For every $p_k \in \{p_1, \ldots, p_n\}$, there should exist a mapping of the following type $p_k = f(p_{11}, \ldots, p_{i1}, p_{i2}, \ldots)$. This can be accomplished in the following manner:
  - Parent parameters are inherited directly. For example, if we consider the AC motor as an artifact, and then decompose it into the following artifacts: rotor, electromagnetic stator windings, housing, bearing, and shaft. The power parameter of the AC motor is inherited directly to the power parameter of the electromagnetic stator windings.

        - Parent parameters are mapped to children parameters. For example, the weight parameter of the AC motor artifact is a function of the weight of the rotor, electromagnetic stator windings, housing, bearing, and shaft.
    - **Maintain interaction consistency between children artifacts and parent artifact.** If we replace parent artifact with its children artifacts, then all the interactions between parent artifact and use-environment artifacts must be able to be mapped into the interactions between children artifacts and use-environment artifacts. For example, one of the AC motor's behaviors is to take electrical energy from a power source and convert it into rotational mechanical energy. If we decompose the motor into rotor and electromagnetic stator windings, then rotor and the electromagnetic stator windings must also be able to accept electric energy and carry out the conversion.
- **Decompose State**: This operator is called *DECOMPOSE-STATE* and used to decompose an interaction-state into several sub-states and state transitions among these sub-states. This operator is defined as the following.
    - Input: state s and a transition diagram D that contains s.
    - Output: new state set S, new state transition set R and a new transition diagram D'.
    - Action: Replace the original state by a new state set and a new state transition set. Redirect transitions that involve the original state to the decomposed state.
    - Precondition: Sometimes the artifacts and artifact interactions cannot be satisfied by existing working principles, therefore we need to further

decompose artifacts into finer levels. Sometime artifact decomposition may also require state decomposition to maintain state consistency.

This operation is illustrated in Figure 3.7.



(a) Original transition diagram

(b) After artifact decomposition of a_1

(c) s_1 should be decomposed into two states

(d) After state decomposition of s_1

Figure 3.7: Usage of operator *DECOMPOSE-STATE*

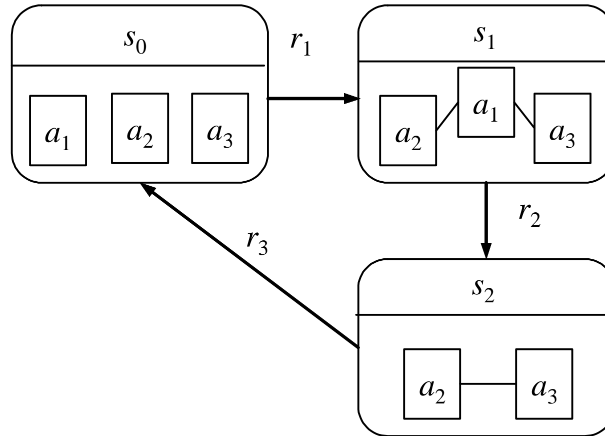
In state s_1, artifact a_1 interacts with a_2 and a_3. Then a_1 is decomposed into a_{11} and a_{12} as shown in Figure 3.7(b). However, it is determined that the interaction

between a_{11} and a_2 and the interaction between a_{12} and a_2 cannot exist at the same time. Thus s_1 should be decomposed as shown in Figure 3.7(c). The original transition diagram will also be changed into a new diagram as shown in Figure 3.7(d).

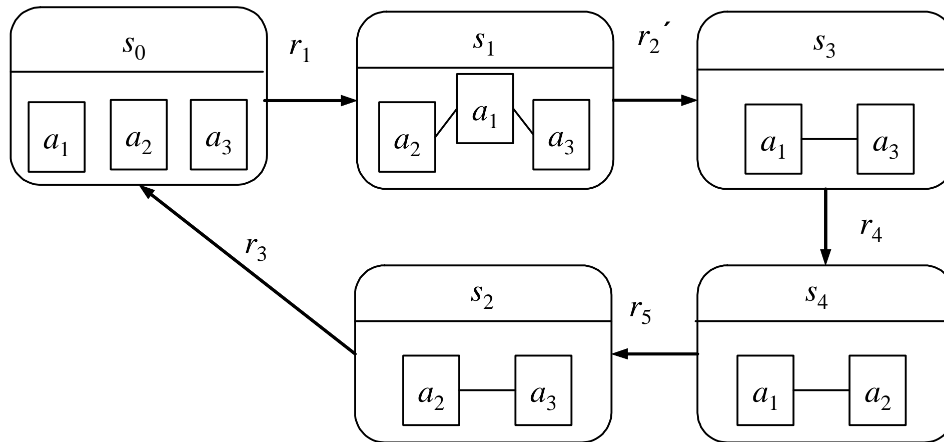
- **Decompose Transition**: This operator is called *DECOMPOSE-TRANSITION* and used to decompose an interaction-state transition into several states and state transitions among these states. This operator is defined as the following.
  - Input: state transition *r*.
  - Output: new state set *S* and new state transition set *R*.
  - Action: Replace the original state transition by a new state set and a new state transition set. In other words, this operator substitutes a state transition with a new transition diagram.
  - Precondition: Designers decide that there are alternative ways for realizing the state transition. Sometimes the transition cannot be satisfied by existing working principles or designers view a better solution by inserting intermediate states and corresponding transitions. In this case, we need to further elaborate the state transition into a finer level. Using the decomposition transition operator must also result in a set of states each with its unique interaction topology.

This is illustrated in Figure 3.8. The transition r_2 between states s_1 and s_2 is decomposed. Figure 3.8(b) shows the result of decomposing r_2. r_2 is replaced by a transition diagram that includes transitions r_2', r_4, and r_5, and states s_3 and s_4.

Design concept generated as a result of applying the elaboration operators described above will not violate the behavioral requirements represented in the initial transition diagram and hence it is referred as an ***elaboration*** of the initial behavior specification. This is due to the following reasons:



(a) Before decomposing transition r_2



(b) After decomposing transition r_2

Figure 3.8: Usage of operator *DECOMPOSE-TRANSITION*

- Applying operator *DECOMPOSE-ARTIFACT* will decompose the artifact into several sub-artifacts. As long as the decomposition follows the parameter and interaction consistency guidelines (refer to the description of the operator

Decompose Artifact), the interaction between the artifact and the use-environment can always be fulfilled by the interactions between its sub-artifacts and the use-environment. These sub-artifacts can also be assembled into the original artifact. Thus the elaborated behavior model will not violate the initial behavior specification.

- Applying operator *DECOMPOSE-STATE* will decompose a state into several sub-states and associated transitions. According to the definition of interaction-state, the interactions in the new states should not happen at the same time. However, after an artifact is decomposed, interaction topology in the original state should be represented by the interactions between sub-artifacts and the use-environment. This reorganization may lead to state decomposition. As stated above, this will not cause any violation. State decomposition without artifact decomposition means that the interactions in the original state actually do not exist at the same time. As long as all the interactions in the original state are preserved in the new states, there is no difference in the behavior.
- Applying operator *DECOMPOSE-TRANSITION* will decompose a transition into several transitions and associated states. Since decomposing transition will not change the starting and ending states of this transition, there is no difference in the behavior either.

3.4 Steps in Conceptual Design

In our framework, a design concept is modeled using primitives and operators defined in Section 3.2 and 3.3 using the following two main steps.

Step 1: Define Behavioral Specifications. This step builds the initial transition diagram from requirements. There are several sub-tasks in this step.

The first task is to define the initial primitives of the design world. The design world usually includes the device to be designed and the artifacts in the use-environment with which the device interacts. For constructing artifacts, we need to construct parameters and parameter-interactions for each artifact. Device artifacts respond to events in the use-environment. Thus, given a design problem, we should also define the event space.

After artifacts are created, we can use these artifacts to construct a set of interaction-states by adding artifact interactions according to the requirements. In creating states, we should check the consistency of each state to make sure that all states are consistent. After states are constructed and their consistencies have been checked, we can define a set of state transitions to construct an initial transition diagram. In creating transitions, we should check if a transition is realizable or not. Unrealizable transitions should be eliminated from the design concept. Also, for a specific design problem, we should know the conditions that result in unsafe operations. These conditions should be defined as unsafe world-states. Thus, we should also be able to check if the created transition diagram is safe or not. If the transition diagram is valid, then we have finished the definition of behavior specifications, i.e., construction of an initial valid transition diagram. To capture design constraints, engineering characteristics should also be defined as a part of the behavior specification.

Step 2: Elaborate Transition Diagrams: This is the main step of the modeling framework. After the initial transition diagram is constructed, the device artifacts may need to be further decomposed such that they can be realized via known working principles. At the same time, interaction-states and transitions may also need to be decomposed because of the following reasons:

1) After the artifact decomposition, the initial artifact interactions should also be replaced by the interactions represented by the sub-artifacts. In this case, new artifact interactions may not happen concurrently and thus they should be broken up into several different interaction-states. New states may also require new transitions to connect these states.
2) Breaking up unrealizable transitions may require state decomposition and transition decomposition. A transition may be unrealizable due to the following reasons: a) no working principles or events could be found that can match the parameter values in two states associated with the transition, b) no working principles or events could be found that can satisfy the transition condition. To solve this, intermediate transitions and states must be introduced. This may also be accompanied by corresponding artifact decompositions.

The elaboration step must ensure that the device's desired behavior is satisfied. Figure 3.9 shows the elaboration process.

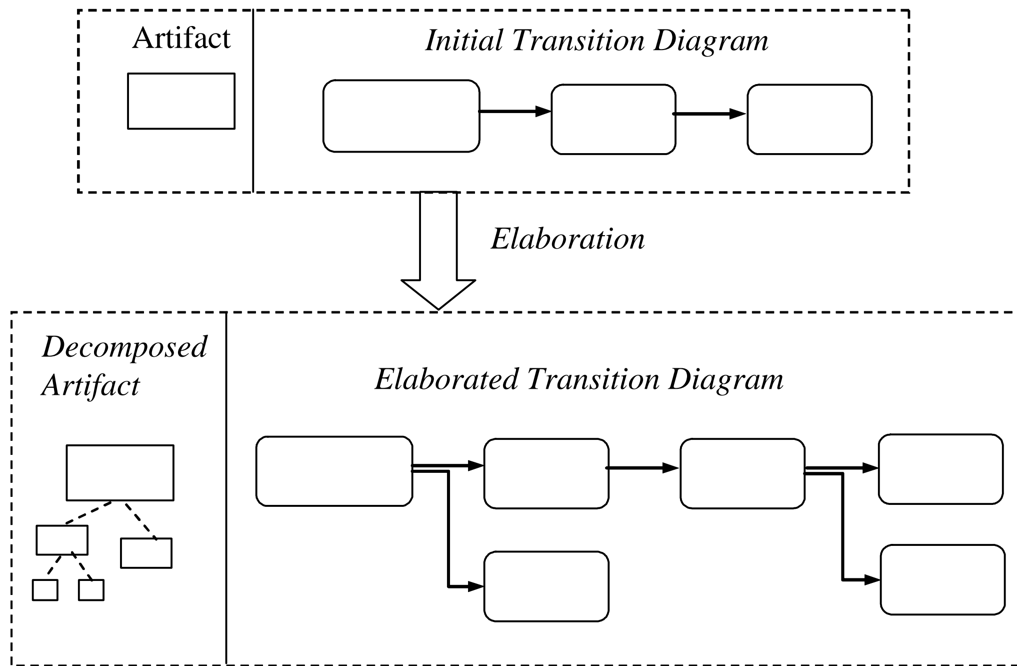


Figure 3.9: Elaboration of interaction transition diagrams

Design concept is formally defined as an ordered set (D_i, D_f, E, U), in which D_i is the initial behavior specification, D_f is the fully elaborated transition diagram, E is the event space, U is the set of unsafe parameter values with respect to E. A design concept is considered ***valid*** if it meets the following conditions:

- D_i and D_f are valid transition diagrams.
- D_i and D_f are safe with respect to E and U.
- Every artifact interaction in every state of D_f can be expressed in terms of parameter interactions. For every parameter interaction and state transition, there exists a known working principle.
- D_f is an elaboration of D_i.

3.5 Simulating Transition Diagrams

This section describes an algorithm for simulating a transition diagram. Transition diagrams in behavior specifications and in design concepts can both be simulated. This would provide capabilities to check the behavior specifications or design concepts as early as possible. The algorithm for this task is described below.

Algorithm SIMULATETRANSITIONDIAGRAM

Input:

- An event sequence *L*
- A transition diagram *D*
- Unsafe parameter value sets *U*

Output:

- A sequence *V* of 5-tuples. Each 5-tuple is defined as (t_1, t_2, *s*, *Q*, *Safety_Status*). Where
  - t_1 is the start time.
  - t_2 is the end time.
  - *s* is the interaction-state in which the device remains between time t_1 and t_2.
  - *Q* is the set of equations that are valid between time t_1 and t_2.
  - *Safety_Status* determines if an unsafe world state is embedded within interaction state *s* or not. *Safety_Status* is set to *TRUE* if *s* does not contain an unsafe parameter value set. It is set to *FALSE* if *s* contains an unsafe parameter value set.

Steps:

1) Assign $T = 0$; *Current_State* = *D.InitialState*; *Incoming_Transition* = *NONE*; and $V = \varnothing$.
2) *Outgoing_Transition* = *NONE*.
3) Find the set of outgoing transitions R_O for *Current_State*. This is done by identifying all transitions *r* in *D.InteractionStateTransitionSet* such that *r.StartState* = *Current_State*.
4) Assign *Transition_Time* = *INFINITY*; *Transition_Event* = *NONE*; and *Safety_Status* = *TRUE*.
5) Initialize parameters in *Current_State* using value assignments in *Current_State.InitialValueSet*.
   i. For every member $i \in$ *Current_State.InitialValueSet*
      a) if *i.InitializationType* is *ASSGIN*, apply the value assignment in *i.Value*.
      b) if *i.InitializationType* is *INHERIT*, then inherit the value from the previous state.
   ii. If *Incoming_Transition* ≠ *NONE*, override the previously defined value assignment using *Incoming_Transition.InitializationActionSet.*
6) Let Q_1 be the set of equations defined in *Current_State.ChangeModeSet*, Q_2 be the set of equations defined in *ParameterInteractionSet* of various artifacts belonging to *Current_State*, Q_3 be the set of equations defined in *Current_State.ArtifactInteractionSet*. $Q=Q_1 \cup Q_2 \cup Q_3$.
7) For every transition *r* in R_O, do the following:
   i. If *r.TransitionCondition* does not involve any event parameter, then

(i). Compute the time *Tmp* when this transition can occur by solving equation set $Q \cup$ (*r.TransitionCondition*) for time *t*.

(ii). *Transition_Time* = *Tmp*; *Outgoing_Transition* = *r*.

ii. Otherwise, do the following:

a) If an event exists in *L* that can satisfy *r.TransitionCondition*, then find the first such event *l* in *L*.

b) *Transition_Time* = *l.GlobalTime*; *Outgoing_Transition* = *r*; *Transition_Event* = *l*.

8) Assign $t_1 = T$; t_2 = *Transition_Time*; t_e = *Transition_Time* – *T*.

9) Check state safety using the approach described below:

i. For every unsafe parameter value set *u* in *U* do the following:

(i). Find the parameter value set *Z* containing those parameters that remain constant during the state. If $u \subseteq Z$, then *Safety_Status* = *FALSE*, go to step 10.

(ii). If equations in *Q* are solvable analytically, then

i) Substitute unsafe values from *u* in *Q* and solve for time *t*.

ii) If $t \leq t_2$, then go to step 10.

iii) If $t > t_2$, then *Safety_Status* = *TRUE*, go to the next *u*.

(iii). Otherwise,

i) Assign $t = 0$;

ii) While $t \leq t_e$ and global time $T \leq$ *Transition_Time*, do the following:

(a) Substitute *t* and *T* in *Q* and compute values of all parameters. Store these values in the value parameter set *Z'*.

(b) For every unsafe parameter value set u in U do the following:

(i). If u involves parameters that belong to artifacts not in this state, go to the next u.

(ii). if $u \subseteq Z'$, then $Safety_Status = FALSE$.

(iii). Assign $T = T + Delta$.

(iv). Assign $t = t + Delta$.

10) Insert (t_1, t_2, *Current_State*, *Q*, *Safety_Status*) into *V*.

11) Assign $T = Transition_Time$.

12) If $T = INFINITY$, then return *V*.

13) Otherwise, $Current_State = Outgoing_Transition.EndState$; $Incoming_Transition = Outgoing_Transition$.

14) If *Transition_Event* is not equal to *NONE*, then remove it from *L*.

15) If *L* is not empty then go to Step 2.

16) Otherwise, return *V*.

3.6 Example of Modeling Autonomous Vacuum Cleaner (*AVC*)

This section describes application of the methodology presented in this chapter to the design of an autonomous vacuum cleaner (*AVC*). The design task is to develop a device that is able to collect the debris on a surface while avoiding collision from obstacles on the surface. The requirements are described in Figure 3.10.

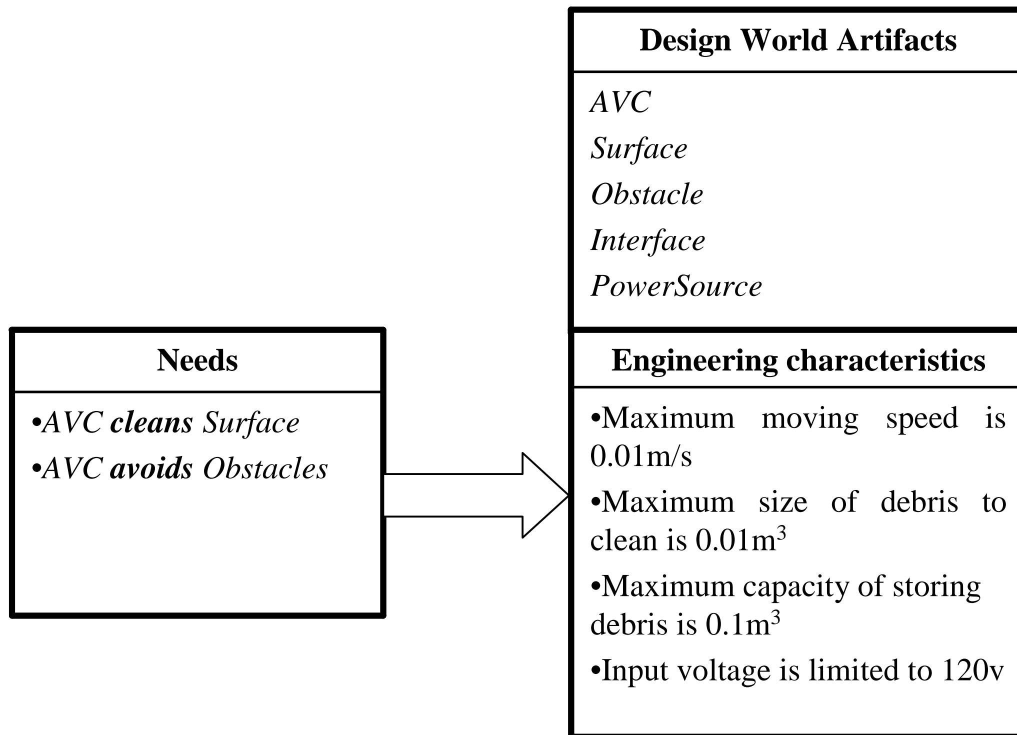


Figure 3.10: Requirements of *AVC*

Customer needs described using the “artifact verb” pairs are shown in the left box (e.g., *AVC* cleans surface). Design world artifacts are extracted from the customer needs in the right box. *AVC* is the device artifact while surface and the obstacles are the use-environment artifacts. Engineering characteristics are also given in the bottom of the right box. The two steps described in Section 3.4 are carried out in the following manner:

1. **Define Behavior Specifications**: Parameters that are used to define behavior specifications are shown in Table 3.2. For example, *AVC* stores the debris thus it has a remaining capacity parameter. The possible interactions between *AVC* and its use-environment are summarized into the event space shown in Table 3.3.

Table 3.2: Artifacts and Parameters used in *AVC* behavior specification

Artifact	**Parameter**	**Type**	**Convention**
AVC	*Speed*	*REAL*	
	RemainingCapacity	*REAL*	0 to 100%
	RemainingEnergy	*REAL*	0 to 100%
	Power	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*
	InputVoltage	*REAL*	
	ObstacleInContact	*BOOLEAN*	*TRUE/FALSE*
Surface	*AreaCovered*	*REAL*	
	LocationVisited	*BOOLEAN*	*TRUE/FALSE*
	MovePossible	*BOOLEAN*	TRUE/FALSE
Power Source	*VoltageOutput*	*REAL*	
Obstacle	*AVCInContact*	*BOOLEAN*	*TRUE/FALSE*
Interface	*Power*	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*

Table 3.3: Event space used in *AVC* behavioral specification

Parameter	**Value**
Surface::*LocationVisited*	{*TRUE*, *FALSE*}
Surface::*MovePossible*	{*TRUE*, *FALSE*}
Interface::*Power*	{*ON*, *OFF*}
Interface::*PauseStatus*	{*ON*, *OFF*}
Obstacle::*AVCInContact*	{*TRUE*, *FALSE*}

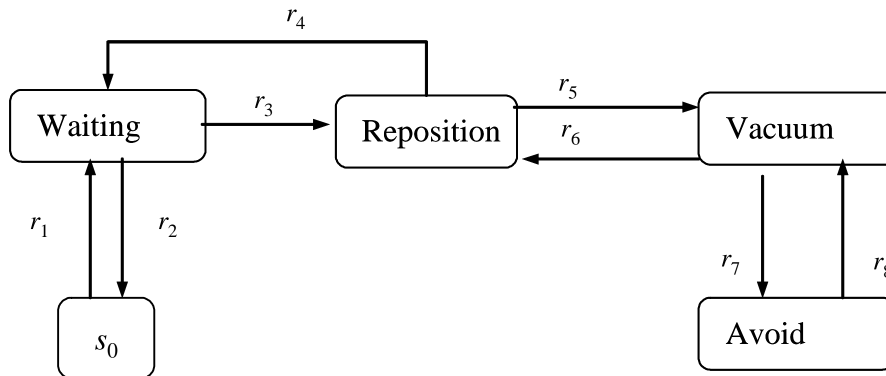
Unsafe parameter value sets are described in Table 3.4. From the requirements, the primary working modes of *AVC* (e.g., interaction-states) are also identified.

Table 3.4: Unsafe state used in *AVC* behavioral specification

$AVC::RemainingEnergy \leq 10\%$
$AVC::RemainingCapacity \leq 2\%$

Figure 3.11 shows proposed behavior specifications for *AVC*.

Transition Diagram



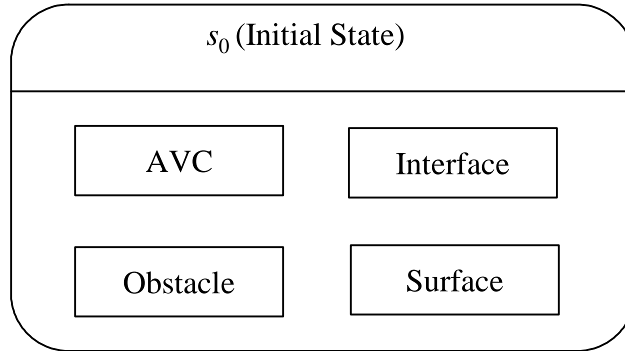
Transition list

Name	Condition
r_1	$Interface::Power = ON$
r_2	$Interface::Power = OFF$
r_3	$Interface::PauseStatus = OFF$
r_4	$Surface::MovePossible = FALSE$
r_5	$Surface::LocationVisited = FALSE$
r_6	$Surface::LocationVisited = TRUE$
r_7	$Obstacle::AVCInContact = TRUE$
r_8	$Obstacle::AVCInContact = FALSE$

Figure 3.11: *AVC* behavior specification #1

Detailed descriptions of each interaction-state are shown in Figures 3.12, 3.13, 3.14, 3.15 and 3.16. In case of only qualitative structures is known for the interactions, we use symbol f to denote there is a relationship between the

parameters involved in the interaction. For example, $x_3 = f(x_1, x_2)$ indicated that parameter x_3 will depend on x_1 and x_2. Parameters x_1, x_2, and x_3 will be related to each other by an equation whose structure is not known at the time of modeling.



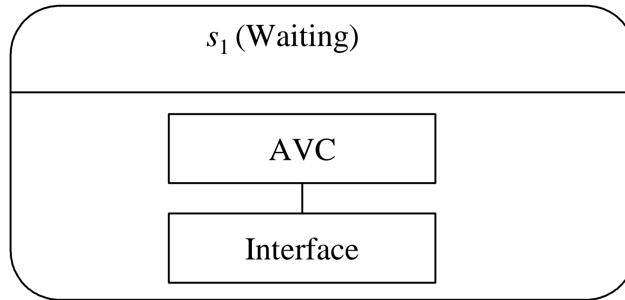
Artifact Interaction Equations

None

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
AVC::Speed	*ASSIGN*	0	*CONSTANT*	*NONE*
AVC::RemainingCapacity	*ASSIGN*	100%	*CONSTANT*	*NONE*
AVC::RemainingEnergy	*ASSIGN*	100%	*CONSTANT*	*NONE*
AVC::Power	*ASSIGN*	*OFF*	*CONSTANT*	*NONE*
AVC::PauseStatus	*ASSIGN*	*ON*	*CONSTANT*	*NONE*
AVC::ObstacleInContact	*ASSIGN*	*FALSE*	*CONSTANT*	*NONE*
AVC::InputVoltage	*ASSIGN*	0	*CONSTANT*	*NONE*

Figure 3.12: Definition of state s_0



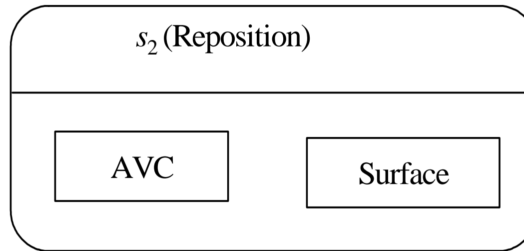
Artifact Interaction Equations

$$AVC::Power = Interface::Power$$
$$AVC::PauseStatus = Interface::PauseStatus$$

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
AVC::Speed	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::RemainingCapacity	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::RemainingEnergy	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::Power	*DERIVE*	*NA*	*CONSTANT*	*NONE*
AVC ::PauseStatus	*DERIVE*	*NA*	*CONSTANT*	*NONE*
AVC::ObstacleInContact	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::InpuVoltage	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 3.13: Definition of state s_1



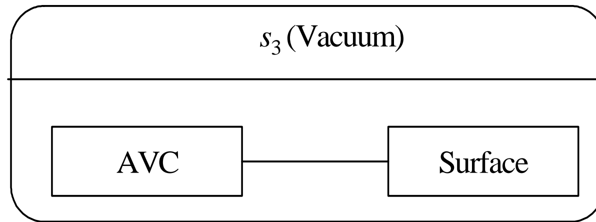
Artifact Interaction Equations

None

Parameters Initialization and Change

Parameter	Initialization Type	Initialization Value	Change Type	Equation
AVC::*Speed*	*ASSIGN*	0.05m/s	*CONSTANT*	*NONE*
AVC::*RemainingCapacity*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	$AVC::RemainingEnergy(t) = AVC::RemainingEnergy(t{=}0) - AVC::Speed \times t / 400$
AVC::*Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*PauseStatus*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*ObstacleInContact*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 3.14: Definition of state s_2



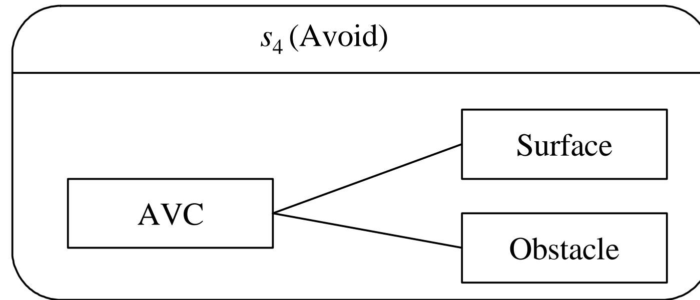
Artifact Interaction Equations

$AVC::RemainingCapacity(t) = AVC::RemainingCapacity(t{=}0) - Surface::AreaCovered / 20$

Parameters Initialization and Change

Parameter	Initialization Type	Initialization Value	Change Type	Equation
AVC::Speed	*ASSIGN*	0.05m/s	*CONSTANT*	*NONE*
AVC::RemainingCapacity	*INHERIT*	*NA*	*EQUATION*	*NONE*
AVC::RemainingEnergy	*INHERIT*	*NA*	*EQUATION*	$AVC::RemainingEnergy(t) = AVC::RemainingEnergy(t{=}0) - AVC::Speed \times t / 400$
AVC::Power	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::PauseStatus	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::ObstacleInContact	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::InputVoltage	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 3.15: Definition of state s_3



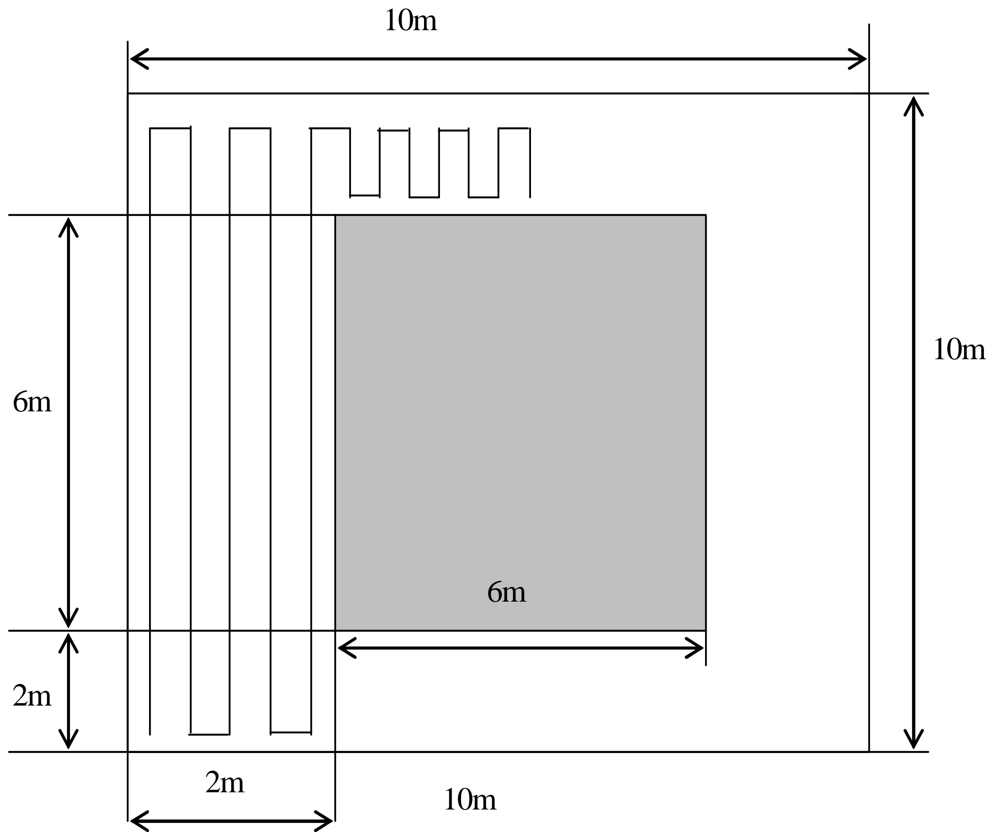
Artifact Interaction Equations

$AVC::RemainingCapacity(t) = AVC::RemainingCapacity(t=0) - Surface::AreaCovered / 80$
$AVC::ObstacleInContact = Obstacle::AVCInContact$

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
AVC::Speed	*ASSIGN*	0.05m/s	*CONSTANT*	*NONE*
AVC::RemainingCapacity	*INHERIT*	*NA*	*EQUATION*	*NONE*
AVC::RemainingEnergy	*INHERIT*	*NA*	*EQUATION*	$AVC::RemainingEnergy(t) = AVC::RemainingEnergy(t=0) - AVC::Speed \times t / 400$
AVC::Power	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::PauseStatus	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::ObstacleInContact	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::InputVoltage	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 3.16: Definition of state s_4



Reposition moving speed is 0.1m/s
Vacuum moving speed is 0.05m/s
Vacuum diameter is 0.4m

Figure 3.17: Illustration of a use-environment for simulation

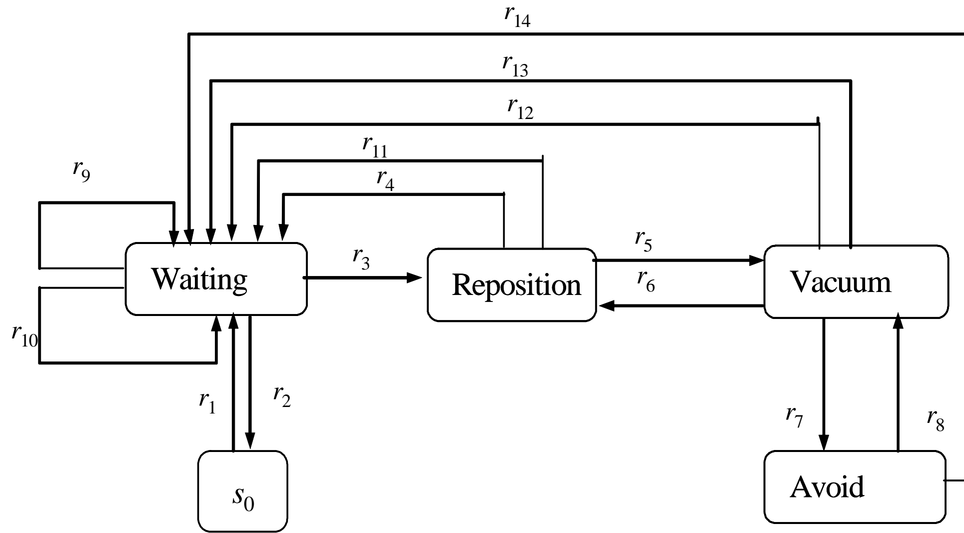
Figure 3.17 illustrates a typical use-environment. Table 3.5 shows an event sequence that results from this use-environment. Simulation shows (see Table 3.6) that this behavior specification is unsafe with respect to the given unsafe parameter value sets. At global time T = 1180s, the value of *AVC*::*RemainingCapacity* is reduced to 2% and thus the device enters an unsafe design-world state. A modified behavior specification is shown in Figure 3.18. This behavior specification is safe. The waiting state is also modified correspondingly, shown in Figure 3.19.

Table 3.5: Event sequence for *AVC* behavior simulation

Time (s)	**Event**
0	None
1	*Interface*::*Power* = *ON*
2	*Interface*::*PauseStatus* = *OFF*
4	*Surface*::*LocationVisited* = *FALSE*
196	*Obstacle*::*AVCInContact* = *TRUE*
204	*Obstacle*::*AVCInContact* = *FALSE*
396	*Obstacle*::*AVCInContact* = *TRUE*
404	*Obstacle*::*AVCInContact* = *FALSE*
596	*Obstacle*::*AVCInContact* = *TRUE*
604	*Obstacle*::*AVCInContact* = *FALSE*
796	*Obstacle*::*AVCInContact* = *TRUE*
804	*Obstacle*::*AVCInContact* = *FALSE*
996	*Obstacle*::*AVCInContact* = *TRUE*
1004	*Obstacle*::*AVCInContact* = *FALSE*
1036	*Obstacle*::*AVCInContact* = *TRUE*
1044	*Obstacle*::*AVCInContact* = *FALSE*
1076	*Obstacle*::*AVCInContact* = *TRUE*
1084	*Obstacle*::*AVCInContact* = *FALSE*
1116	*Obstacle*::*AVCInContact* = *TRUE*
1124	*Obstacle*::*AVCInContact* = *FALSE*
1156	*Obstacle*::*AVCInContact* = *TRUE*
1164	*Obstacle*::*AVCInContact* = *FALSE*
1180	*AVC*::*RemainingCapacity* = 2%

Table 3.6: *AVC* behavior simulation result

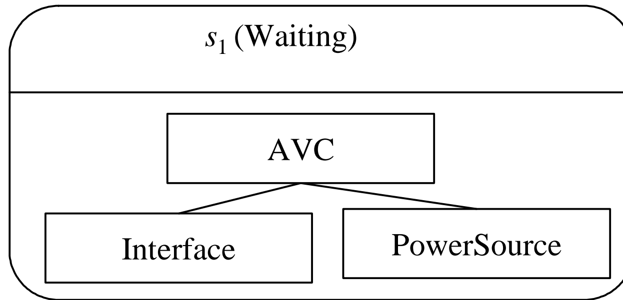
Time (s)	**State**	***RemainingCapacity* at start of the state**	***RemainingEnergy* at start of the state**
0 to 1	Initial	100%	100%
1 to 2	Waiting	100%	100%
2 to 4	Reposition	100%	100%
4 to 196	Vacuum	100%	100%
196 to 204	Avoid	84%	97.6%
204 to 396	Vacuum	83.33%	97.5%
396 to 404	Avoid	67.33%	95.1%
404 to 596	Vacuum	66.67%	95%
596 to 604	Avoid	50.67%	92.6%
604 to 796	Vacuum	50%	92.5%
796 to 804	Avoid	34%	90.1%
804 to 996	Vacuum	33.33%	90%
996 to 1004	Avoid	16.67%	87.6%
1004 to 1036	Vacuum	16%	87.5%
1036 to 1044	Avoid	13.33%	87.1%
1044 to 1076	Vacuum	12.67%	87%
1076 to 1084	Avoid	10%	86.6%
1084 to 1116	Vacuum	9.33%	86.5%
1116 to 1124	Avoid	6.67%	86.1%
1124 to 1156	Vacuum	6%	86%
1156 to 1164	Avoid	3.67%	85.6%
1164 to 1180	Vacuum	3%	85.5%
1180	Unsafe	2%	85.3%
1196	Avoid	0.33%	85.1%



Transition list

Name	Condition
r_1	*Interface*::*Power* = *ON*
r_2	*Interface* ::*Power* = *OFF*
r_3	*Interface*::*PauseStatus* = *OFF*
r_4	*Surface*::*MovePossible* = *FALSE*
r_5	*Surface*::*LocationVisited* = *FALSE*
r_6	*Surface*::*LocationVisited* = *TRUE*
r_7	*Obstacle*:: *AVCInContact* = *TRUE*
r_8	*Obstacle*:: *AVCInContact* = *FALSE*
r_9	*AVC*::*RemainingCapacity* ≤ 2%
r_{10}	*AVC*::*RemainingEnergy* ≤ 10%
r_{11}	*AVC*::*RemainingEnergy* ≤ 10%
r_{12}	*AVC*::*RemainingEnergy* ≤ 10%
r_{13}	*AVC*::*RemainingCapacity* ≤ 2%
r_{14}	*AVC*::*RemainingEnergy* ≤ 10%

Figure 3.18: *AVC* behavior specification #2



Artifact Interaction Equations

$$AVC::Power = Interface::Power$$
$$AVC::PauseStatus = Interface::PauseStatus$$
$$AVC::InputVoltage = PowerSource::VoltageOutput$$

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
AVC::Speed	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::RemainingCapacity	*INHERIT*	*NA*	*EQUATION*	*AVC::RemainingCapacity(t) = AVC::RemainingCapacity(t=0) + t / 200*
AVC::RemainingEnergy	*INHERIT*	*NA*	*EQUATION*	*AVC:: RemainingEnergy(t) = AVC:: RemainingEnergy(t=0) + t / 200*
AVC::Power	*DERIVE*	*NA*	*DERIVE*	*NONE*
AVC ::PauseStatus	*DERIVE*	*NA*	*DERIVE*	*NONE*
AVC::ObstacleInContact	*INHERIT*	*NA*	*CONSTANT*	*NONE*
AVC::InpuVoltage	*INHERIT*	*NA*	*CONSTANT*	*NONE*

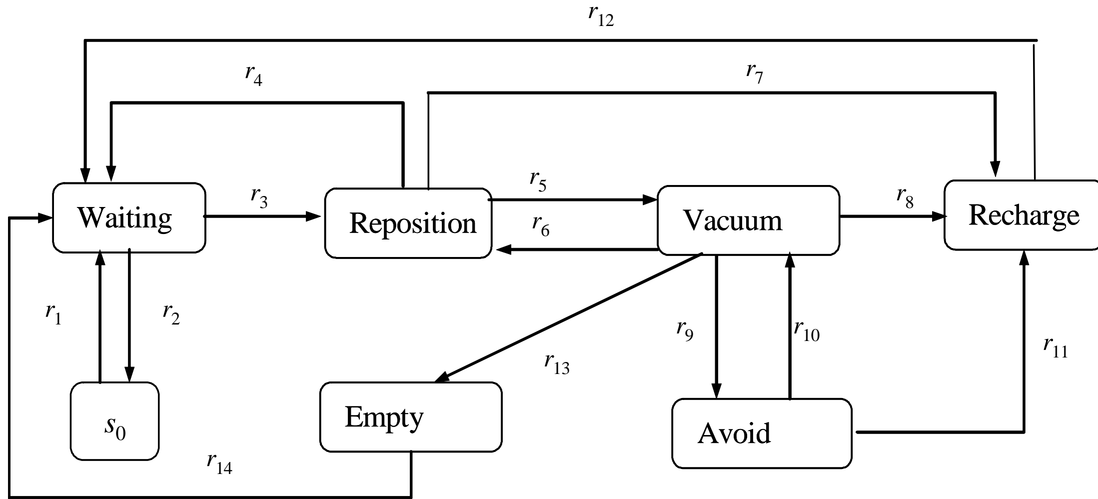
Figure 3.19: Modified “Waiting” state

2. **Elaborate Transition Diagram**: Since there is no known artifact that can fulfill the behavior specification of *AVC* directly, we need to decompose *AVC* into artifacts that can be realized. Starting points of the decomposition are the artifact interactions between *AVC* and use-environment artifacts. The operator *DECOMPOSE-ARTIFACT* is applied to replace the *AVC* in behavior specification with its major component artifacts shown in Table 3.7. *AVC*’s parameters are mapped to the parameters of its children artifacts. In this example the major

parameters of *AVC* are directly mapped to one parameter of one children artifact respectively. Furthermore, the “Waiting” state also needs to be decomposed using operator *DECOMPOSE-STATE*. It is decomposed into “Waiting”, “Recharge” and “Empty” states. The corresponding transitions are also redirected and decomposed. A detailed interaction-state transition diagram for the design concept is shown in Figure 3.20. Detailed descriptions of each interaction-state used in the diagram are shown in Figures 3.21 through 3.27.

Table 3.7: Decomposed Artifacts and Parameters of *AVC*

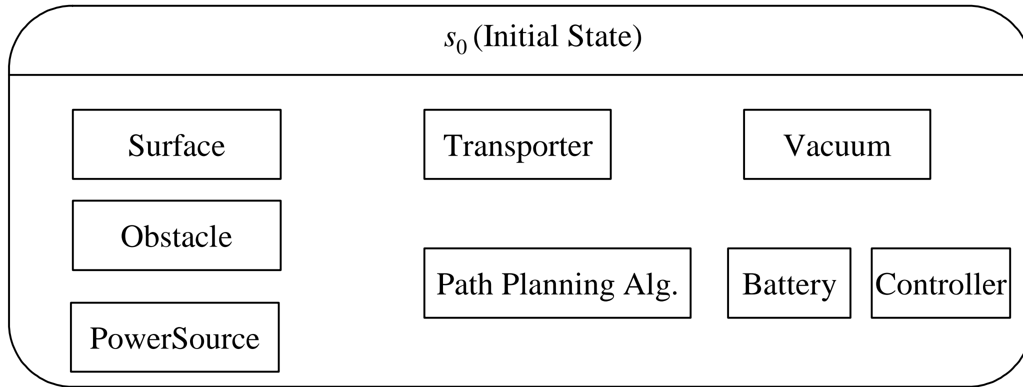
Artifact	**Parameter**	**Type**	**Convention**
Transporter	*Speed*	*REAL*	
	EnergyInput	*REAL*	
Vacuum	*RemainingCapacity*	*REAL*	0 to 100%
Controller	*Power*	*BOOLEAN*	*ON/OFF*
	PauseStatus	*BOOLEAN*	*ON/OFF*
	ObstacleInContact	*BOOLEAN*	*TRUE/FALSE*
Battery	*InputVoltage*	*REAL*	
	RemainingEnergy	*REAL*	0 to 100%
PathPlanningAlg.(PPA)	*SpeedOutput*	*REAL*	



Transition list

Name	Condition
r_1	*Interface*::*Power* = *ON*
r_2	*Interface*::*Power* = *OFF*
r_3	*Interface*::*PauseStatus* = *OFF*
r_4	*Surface*::*MovePossible* = *FALSE*
r_5	*Surface*::*LocationVisited* = *FALSE*
r_6	*Surface*::*LocationVisited* = *TRUE*
r_7	*Battery*::*RemainingEnergy* ≤ 10%
r_8	*Battery*::*RemainingEnergy* ≤ 10%
r_9	*Obstacle*::*AVCInContact* = *TRUE*
r_{10}	*Obstacle*::*AVCInContact* = *FALSE*
r_{11}	*Battery*::*RemainingEnergy* ≤ 10%
r_{12}	*Battery*::*RemainingEnergy* = 100%
r_{13}	*Vacuum*::*RemainingCapacity* ≤ 2%
r_{14}	*Vacuum*::*RemainingCapacity* = 100%

Figure 3.20: *AVC* design concept based on behavior specification #2



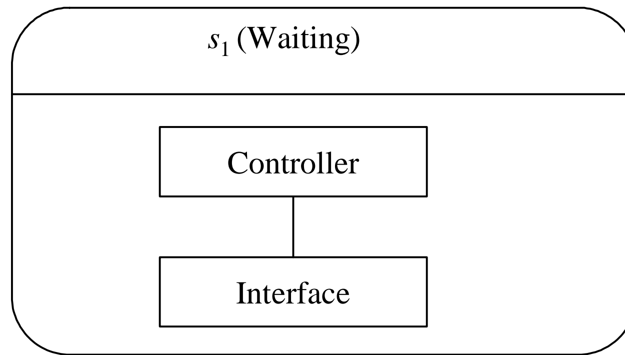
Artifact Interaction Equations

None

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Transporter::*Speed*	*ASSIGN*	0	*CONSTANT*	*NONE*
Transporter::*EnergyInput*	*ASSIGN*	0	*CONSTANT*	*NONE*
PPA::*SpeedOutput*	*ASSIGN*	0	*CONSTANT*	*NONE*
Battery ::*InputVoltage*	*ASSIGN*	0	*CONSTANT*	*NONE*
Vacuum::*RemaininCapacity*	*ASSIGN*	100%	*CONSTANT*	*NONE*
Controller::*PauseStatus*	*ASSIGN*	*ON*	*CONSTANT*	*NONE*
Controller::*ObstacleInContact*	*ASSIGN*	*FALSE*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*ASSIGN*	100%	*CONSTANT*	*NONE*
Controller::*Power*	*ASSIGN*	*OFF*	*CONSTANT*	*NONE*

Figure 3.21: Definition of state s_0



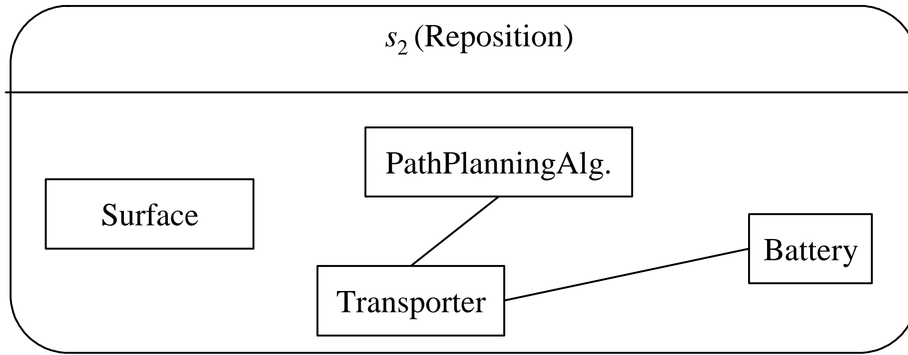
Artifact Interaction Equations

$$Controller::Power = Interface::Power$$
$$Controller::PauseStatus = Interface::PauseStatus$$

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Controller::Power	*DERIVE*	*NA*	*DERIVE*	*NONE*
Controller::ObstacleInContact	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Controller::PauseStatus	*DERIVE*	*NA*	*DERIVE*	*NONE*

Figure 3.22: Definition of state s_1



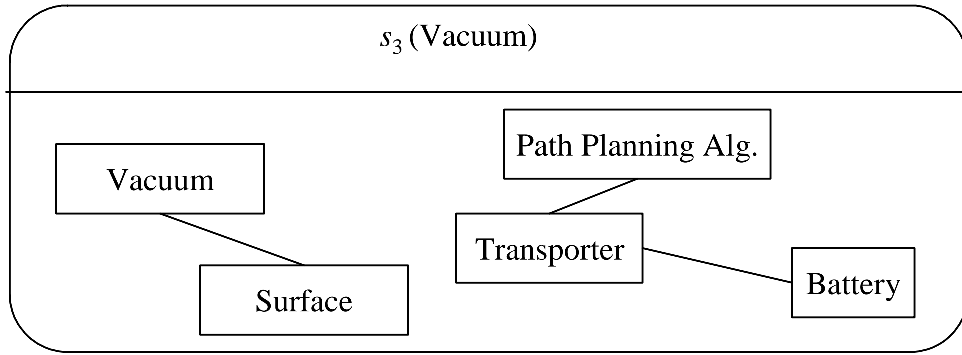
Artifact Interaction Equations

Transporter::*Speed* = *PPA*::*SpeedOutput*
Transporter::*EnergyInput* = *Battery*::*RemainingEnergy*
Battery::*RemainingEnergy*(*t*) = *Battery*(*t*)::*RemainingEnergy*(*t*=0) −
Transporter::*Speed* × *t* / 400

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Transporter::*Speed*	*DERIVE*	*NA*	*DERIVE*	*NONE*
Transporter::*EnergyInput*	*DERIVE*	*NA*	*DERIVE*	*NONE*
PPA::*SpeedOutput*	*ASSIGN*	0.05m/s	*CONSTANT*	*NONE*
Battery::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*NONE*

Figure 3.23: Definition of state s_1



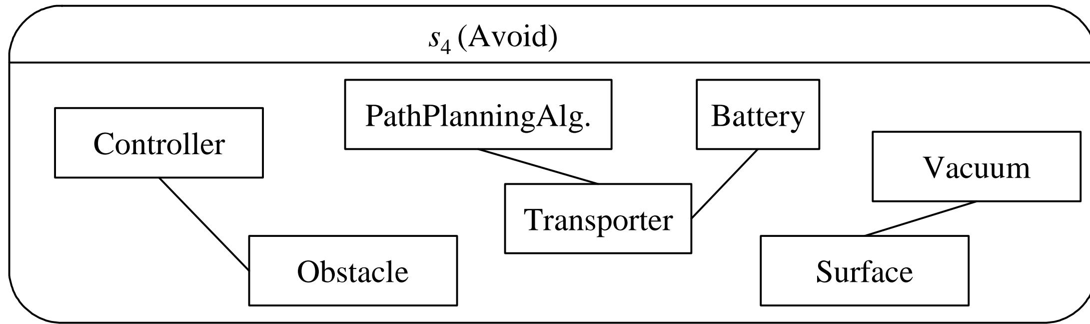
Artifact Interaction Equations

$Transporter::Speed = PPA::SpeedOutput$
$Transporter::EnergyInput = Battery::RemainingEnergy$
$Vacuum::RemainingCapacity(t) = Vacuum::RemainingCapacity(t{=}0) - Surface::AreaCovered / 20$
$Battery::RemainingEnergy(t) = Battery(t)::RemainingEnergy(t{=}0) - Transporter::Speed \times t / 400$

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Transporter::*Speed*	*DERIVE*	*NA*	*DERIVE*	*NONE*
Transporter::*EnergyInput*	*DERIVE*	*NA*	*DERIVE*	*NONE*
PPA::*SpeedOutput*	*ASSIGN*	0.01m/s	*CONSTANT*	*NONE*
Battery::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*NONE*
Vacuum::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*NONE*

Figure 3.24: Definition of state s_3



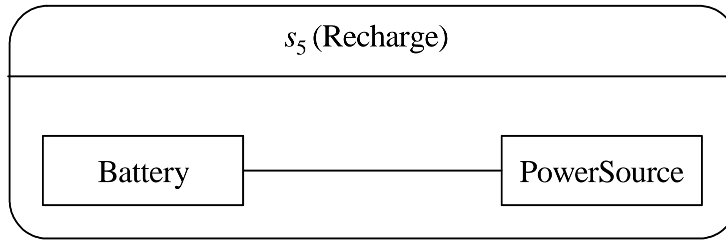
Artifact Interaction Equations

Transporter::*Speed* = *PPA*::*SpeedOutput*
Transporter::*EnergyInput* = *Battery*::*RemainingEnergy*
Vacuum::*RemainingCapacity*(t) = *Vacuum*::*RemainingCapacity*(t=0) – *Surface*::*AreaCovered* / 60
Controller::*ObstacleInContact* = *Obstacle*::*AVCInContact*
Battery::*RemainingEnergy*(t) = *Battery*(t)::*RemainingEnergy*(t=0) – *Transporter*::*Speed* × t / 400

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Transporter::*Speed*	*DERIVE*	*NA*	*DERIVE*	*NONE*
Transporter::*EnergyInput*	*DERIVE*	*NA*	*DERIVE*	*NONE*
PPA::*SpeedOutput*	*ASSIGN*	0.01m/s	*CONSTANT*	*NONE*
Battery ::*InputVoltage*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*NONE*
Controller::*Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Controller::*ObstacleInContact*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Controller::*PauseStatus*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Vacuum::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*NONE*

Figure 3.25: Definition of state s_3



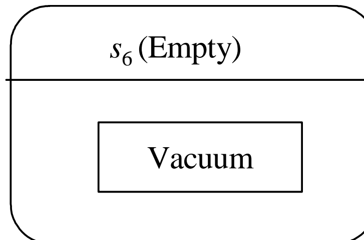
Artifact Interaction Equations

Battery::*InputVoltage* = *PowerSource*::*VoltageOutput*

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Battery::*InputVoltage*	*DERIVE*	*NA*	*CONSTANT*	*NONE*
Battery::*RemainingEnergy*	*INHERIT*	*NA*	*EQUATION*	*Battery*::*RemainingEnergy*(*t*) = Battery::*RemainingEnergy*(*t*=0) + *t* / 200

Figure 3.26: Definition of state s_5



Artifact Interaction Equations

None

Parameters Initialization and Change

Parameter	**Initialization Type**	**Initialization Value**	**Change Type**	**Equation**
Vacuum::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*Vacuum*::*RemainingCapacity*(*t*) = *Vacuum*::*RemainingCapacity*(*t*=0) + *t* / 200

Figure 3.27: Definition of state s_5

3.7 Summary

In this chapter, we describe a new modeling framework for representing design concepts of multiple interaction-state devices. We also provide conditions for ensuring its validity. The distinction between our approach and traditional functional representation approaches for conceptual design is as following:

- We use interactions instead of function flows or input/output flows to describe relationships between artifacts. Interactions are more general than flows. In addition to capturing flows, they can also be used to capture non-flow based relationships such as spatial constraints. Therefore, our approach is more expressive.
- We use interaction-states to capture the operating modes of a device. Hence we can support devices with multiple interaction-states (i.e., devices whose interactions with the use environment change with time). Therefore, design concepts modeled using our framework can be simulated more accurately. For example, events can be used to simulate the behavior of a proposed design concept in response to events in the use-environment.

The main rationale behind developing a new modeling framework in this dissertation was to create a framework that (1) is convenient for mechatronic designers to use, (2) is expressive enough to support conceptual design, and (3) explicitly represents information needed to support evaluation and validation during conceptual design.

General purpose modeling approaches such as UML, extended finite state machine, and hybrid automata are very expressive. However, they are not very convenient for mechatronic designers to use. During conceptual design, most mechatronic designers

focus their attention to identifying the main components, specifying interactions among them, and organizing interactions in a meaningful way. The general purpose modeling approaches are quite capable of capturing all of this information. However, they typically require use of multiple diagrams (e.g. UML) to accomplish this and the designer often needs to customize the environment to create a familiar terminology.

Our modeling framework described in this Chapter is sufficiently expressive and it gives the designers the familiar terminology to carry out the conceptual design. In the background, our modeling framework ensures that sufficient information has been gathered and organized to support automated evaluation and validation without any further manual post-processing of the design information. In summary, our modeling framework has the following distinguishing features to support the conceptual design:

- **Conceptual Design Centric Terminology:** Our modeling framework uses terminology familiar to mechatronic designers for carrying out conceptual design. Familiar notions of parameters, artifact, interactions, and decomposition are used in our framework. In addition, our framework allows for making a distinction among use-environment and device artifacts. This offers the following benefits. First, external behavior and internal behavior of the device are clearly distinguished. External behavior refers to the interactions between the device and use-environment. Internal behavior refers to the interactions between component objects of the device. Designers can focus their energy on developing external behaviors in the early stage of design while internal behaviors in the later stage. Second, device behaviors are clearly shown in different use-environments. Device with multiple interaction-states can usually be used in different use-

environments. Explicit modeling of use-environments helps classify device behaviors in cases of different use-environments. This classification not only simplifies the design problem, but also helps designers explore unexpected behaviors in new environments. Third, it allows designers to investigate hazardous effects device could have in the environment.

- **Interaction Centric Single Modeling Diagrams:** Our modeling framework uses interaction centric single modeling diagrams. This makes it convenient for designers to use our modeling framework. Our approach supports a wide variety of interactions encountered during design tasks. We support classes for interaction. These classes capture relationships between parameters and artifacts. Relationships among parameters are called parameter interactions. We use member InteractionReason to capture different types of interactions in engineering design such as energy flow, material flow, signal flow, spatial constraints, physical law etc. Relationships between artifacts are captured using artifact interactions. We also provide classes for representing interaction-states. As one of the most important primitives in our modeling framework, interaction-states describe the invariant interactions among a set of artifacts. In engineering design, an interaction-state captures a working mode of the device. All interactions in the interaction-state exist at the same time in this working mode. We apply this notion to general modeling techniques in designing this class.
- **Hierarchical Modeling to Support Decomposition Based Engineering Design**: Engineering design is a hierarchical process. Design starts at the top level where the device is modeled as single entity. During subsequent levels the

device is decomposed into its constituent components. This decomposition continues throughout the design process. Our modeling framework has the necessary classes to support both elaboration and refinement encountered in the decomposition process. It also keeps track of all the ownership relationships that result from the decomposition process. Artifact, state and transition can be decomposed further during the design process. Our modeling framework provides decompose-artifact, decompose-state, decompose-transition operators for these operations. Class ArtifactMapping is designed to capture the hierarchical information between artifacts.

- **Support Incomplete Interaction Information:** Often during conceptual design, one has to deal with incomplete interaction information due to missing details in the underlying artifacts. Our framework allows the designers to partially specify interactions. Our system can perform consistency checking with only knowing the structure of the interactions.
- **Providing Information Organization To Supported Automated Validation During Conceptual Design**: Our system organizes the information in such way that the validation can begin during the conceptual design stage. In order to ensure the result of modeling (design concept) is valid in senses of both modeling and engineering, we define validation conditions for a design concept as part of the modeling framework. These conditions ensure that not only design concept is modeled correctly, but also can be validated automatically.
- **Providing Information Organization to Support Automated Evaluation During Conceptual Design:** Engineering design concept, as the result of

modeling using the framework, needs to be evaluated before entering the detailed design stage. The earlier this evaluation could be done, the more time, energy and cost could be saved. Our modeling framework supports engineering related analysis such as determining component sharability and maximum power consumed during the conceptual design stage itself. Our framework facilitates gathering of information to facilitate this evaluation in a seamless manner.

Figure 3.28 shows the connections between this chapter and the following chapters.

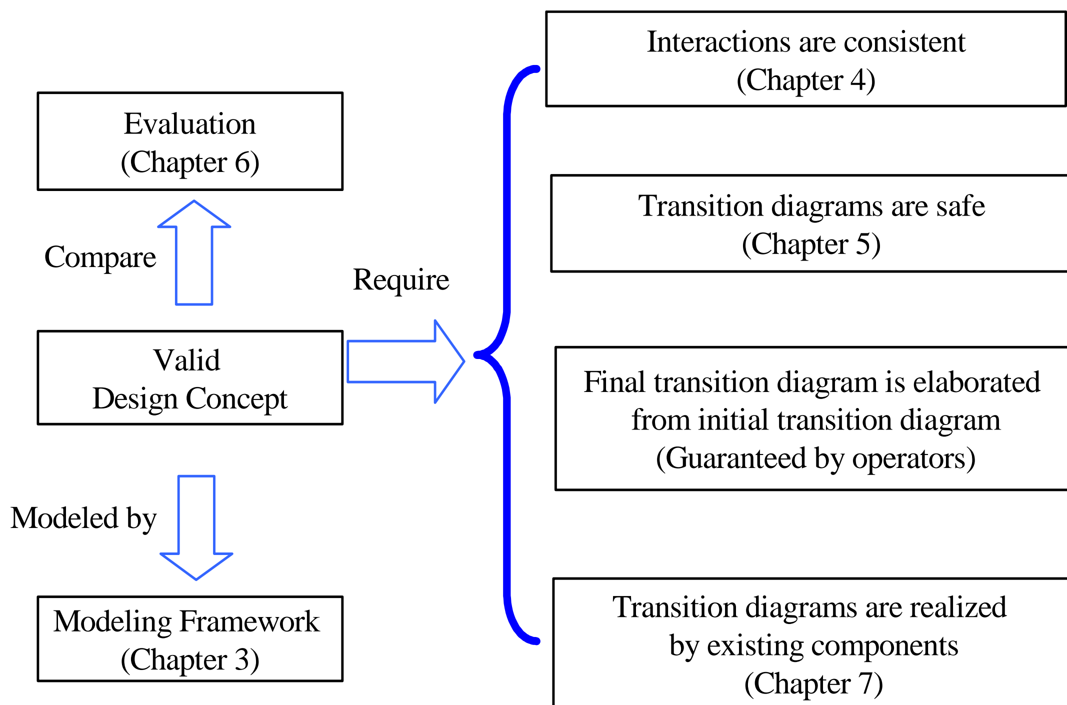


Figure 3.28: Organization of the content of the remaining chapters

Chapter 4: Consistency-Checking of Interaction-states

This chapter defines the problem of consistency-checking of interaction-states and presents a polynomial time algorithm for solving the interaction consistency-checking problem. This chapter also presents an algorithm for analyzing inconsistent interaction-states and identifying the inconsistent interactions.

This chapter has been organized in the following manner. Section 4.1 describes the problem formulation of consistency-checking of interaction-states. Section 4.2 describes the algorithms for mapping consistency checking problem to minimum *s-t* cut problem in an interaction network. Section 4.3 describes the algorithm for finding minimum *s-t* cut and identifying inconsistent interactions. Section 4.4 presents the implementation details and two examples. Finally, Section 4.5 presents concluding remarks.

4.1 Problem Formulation

4.1.1 Problem Statement

Let X be the set of parameters belonging to all the artifacts in an interaction-state s. By examining parameters and artifact interaction in s we can identify the parameter sets that participate in these interactions. The set of all parameters interaction set is denoted as F.

Each f in F is a subset of X and describes an interaction. During the conceptual design stage we are only concerned with the qualitative nature of interaction. For example, consider the hybrid car example. Let us assume we only consider major artifacts: engine, battery, motor, transmission and the wheels. The hybrid car is required to

enter different interaction-states when the road condition changes. When the vehicle travels uphill or accelerates, both the engine and the battery provide power to the wheels through the transmission and motor respectively. The road can be modeled as a use-environment artifact. In this case, the interaction-state consists of these artifacts and their interactions. Figure 4.1 graphically shows the interactions.

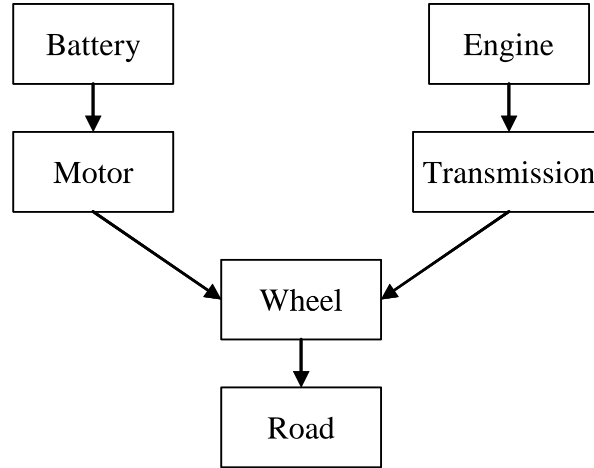


Figure 4.1: Example of an interaction-state for hybrid car

We use the following notations to represent the main parameters participating in the interaction:

x_1 = Battery Power Output, x_2 = Motor Power Output, x_3 = Engine Power Output,

x_4 = Transmission Power Output, x_5 = Wheel Power Input, x_6 = Road Slope.

Then we can list the participating parameters in interactions when the vehicle is going uphill as follows:

$f_1 = \{x_1,\ x_2\}, f_2 = \{x_3,\ x_4\}, f_3 = \{x_2,\ x_4,\ x_5\}, f_4 = \{x_5,\ x_6\},$

Each of the above-described sets of parameters implies that there exists a specific relationship among the parameters in the set and hence all the parameters in the set cannot be assigned values independently. Please note that we are not concerned about

the specific equation that is associated with the interaction. In most cases, such equations are not available at the conceptual design stage. So we are only concerned about the set of parameters that participate in an interaction.

We also model the constraints on the values of individual parameters as sets of participating parameters consisting of only one member. Since there is a maximum power constraint on the engine's power output, we have $f_5 = \{x_3\}$. The slope of the road is determined by the use-environment; therefore we model it as $f_6 = \{x_6\}$. Therefore, in this case we have six variables and six interactions in this interaction-state.

We formulate the interaction consistency problem in the following manner. Given,

- Set $X = \{x_1, x_2, \ldots x_n\}$
- Set $F = \{f_1, f_2, f_3, \ldots f_m\}$, where each $f_i \subseteq X$ and $\cup F = X$
- $n \geq m$

The problem of interaction consistency is to determine if there exists $F' \subset F$ such that cardinality(F') > cardinality($\cup F'$). If such F' exists, then the given set of interactions is considered inconsistent.

Let us consider the following example,

- $X = \{x_1, x_2, x_3, x_4, x_5\}$
- $F = \{f_1 = \{x_3, x_4, x_5\}, f_2 = \{x_1, x_3\}, f_3 = \{x_1, x_2\}, f_4 = \{x_1, x_2, x_3\}, f_5 = \{x_2, x_3\}\}$

Although there is a total of five parameters and only five interactions, the last four interactions (i.e., $f_2 = \{x_1, x_3\}, f_3 = \{x_1, x_2\}, f_4 = \{x_1, x_2, x_3\}, f_5 = \{x_2, x_3\}$) only involve three variables (i.e., x_1, x_2, x_3). Therefore, these interactions are over-constrained. Thus, the interactions in this state are inconsistent and this state is invalid. If $n < m$,

the set of interactions is obviously inconsistent. Thus we only deal with cases in which $n \geq m$.

Here we assume that no redundant equations will be subsequently used in the detailed design stage to realize the set of interactions. A redundant equation can be deduced from a set of other equations. For example, assume that we have the following two equations: $x_1 + x_2 = 3$, $x_2 + x_3 = 5$. Then the equation $x_3 - x_1 = 2$ can be derived from the first two equations and hence it is a redundant equation.

If the set of interactions is inconsistent, a natural problem that arises is identifying the interactions that lead to the inconsistency. Designers need to locate the subset of inconsistent interactions and modify them to ensure that the modified interactions are consistent.

4.1.2 Overview of Our Approach

Given the set of interactions, we use the following approach to solve the problem:

1) Construct an interaction network from the set of interactions. Section 4.2.1 shows how the network is constructed. Then we show that the consistency problem is equivalent to checking the size of the minimum *s-t* cut problem in the interaction network. Section 4.2.2 presents the proof for this equivalence.
2) We use the algorithm FINDMINIMUMSTCUTSIZE to compute the size of the minimum *s-t* cut of the network and find out whether the set of interactions is consistent. Section 4.3.1 presents this algorithm. If the interactions are found to be inconsistent, then we determine the set of interactions that lead to inconsistency. Section 4.3.2 describes the algorithm FINDINCONSISTENTINTERACTIONS defined for this task.

4.1.3 Related Work On Finding Min Cut Of A Graph

The usual approach to solve the minimum cut problem is to use its close relationship to the maximum flow problem. Ford and Fulkerson showed the duality of the maximum flow and the minimum *s*-*t*-cut in their famous Max-Flow-Min-Cut-Theorem [Ford56]. They also gave a simple algorithm for solving the problem. Finding a minimum cut without specifying the vertices to separate can be done by finding minimum *s*-*t*-cuts for a fixed vertex *s* and all $/V/ - 1$ possible choices of $t \in V - \{s\}$ and then selecting the smallest one. Goldberg and Tarjan used push-relabel algorithms to achieve a faster computation. They do not maintain a valid flow during the operation; each node may have a positive “flow excess”, and the algorithm tries to push it to neighboring nodes. Many modifications based on these two types of approaches have been made to achieve faster algorithms.

Algorithms that are not based on flows have also been developed. Nagamochi and Ibaraki gave a procedure that repeatedly identifies and contracts edges that are not in the minimum cut until the minimum cut becomes apparent. It applies only to undirected graphs with non-uniform edge weights [Naga92]. The approach by Gabow is based on a matroid characterization of the minimum cut problem. According to this characterization, the minimum cut in a graph is equal to the maximum number of disjoint directed spanning trees that can be found in it. Gabow’s algorithm finds the minimum cut by finding such trees [Gabo95]. Karger and Stein give a randomized algorithm that finds the minimum cut in an arbitrarily weighted undirected graph [Karg96].

4.2 Mapping Consistency Checking Problem To Minimum S-T Cut Problem In Interaction Network

4.2.1 Construction Of Interaction Network

We build an interaction network G that describes how interactions F and parameters X are related to each other.

There are four kinds of nodes in G:

- s-node: Source node.
- t-node: Sink node.
- x-node: Node corresponding to an parameter in X.
- f-node: Node corresponding to an interaction in F.

There are three types of edges in G:

- sf-edge: Edge connecting the s-node to an f-node. The capacity of this edge is 1 unit.
- fx-edge: Edge connecting an f-node to an x-node. The capacity of this edge is $n + 1$ units.
- xt-edge: Edge connecting an x-node to the t-node. The capacity of this edge is 1 unit.

Now we present the algorithm for constructing the interaction network G.

Algorithm CONSTRUCTINTERACTIONNETWORK

Input: System of interactions F with respect to X. There are n variables in X and m interactions in F.

Output: Interaction network G

Steps:

1) Create an empty network *G*.

2) Insert node *s* into network *G*. Label this node as *s*-node.

3) Insert node *t* into network *G*. Label this node as *t*-node.

4) Insert a node for every $f \in F$ into *G*. Label these nodes as *f*-nodes. Create an edge from the *s*-node to every *f*-node. Label these edges as *sf*-edges. Set the capacity of every *sf*-edge to 1.

5) Insert a node for every $x \in X$ into *G*. Label these nodes as *x*-nodes. Create an edge from every *x-node* to the *t-node*. Label these edges as *xt*-edges. Set the capacity of every *xt*-edge to 1.

6) For every *f*, insert an edge from the *f-node* to an *x-node* if *x* belongs to *f*. Label these edges as *fx*-edges. Set the capacity of every *fx*-edge to $n + 1$.

Figure 4.2 shows network *G* for the following parameters and interactions:

- $X = \{x_1, x_2, x_3, x_4, x_5\}$
- $n = 5$
- $F = \{f_1 = \{x_3, x_4, x_5\},\ f_2 = \{x_1, x_3\}, f_3 = \{x_1, x_2\}, f_4 = \{x_1, x_2, x_3\}, f_5 = \{x_2, x_3\}\}$
- $m = 5$

4.2.2 Mapping Consistency-Checking Problem to Minimum Cut Problem

In this section we will show that the interaction consistency-checking problem can be mapped to the problem of checking the size of the minimum *s-t* cut in network *G*.

Let $G = (V, E)$ be an edge-weighted directed graph (digraph) with a finite set of vertices *V* and a set of ordered pairs of vertices, $E \subseteq V \times V$, called edges. We

typically use e or (u, v) to denote an edge $e = (u, v)$. $c(e)$ is called the capacity of e. A ***network*** is a digraph in which two vertices are distinguished as the source s and the

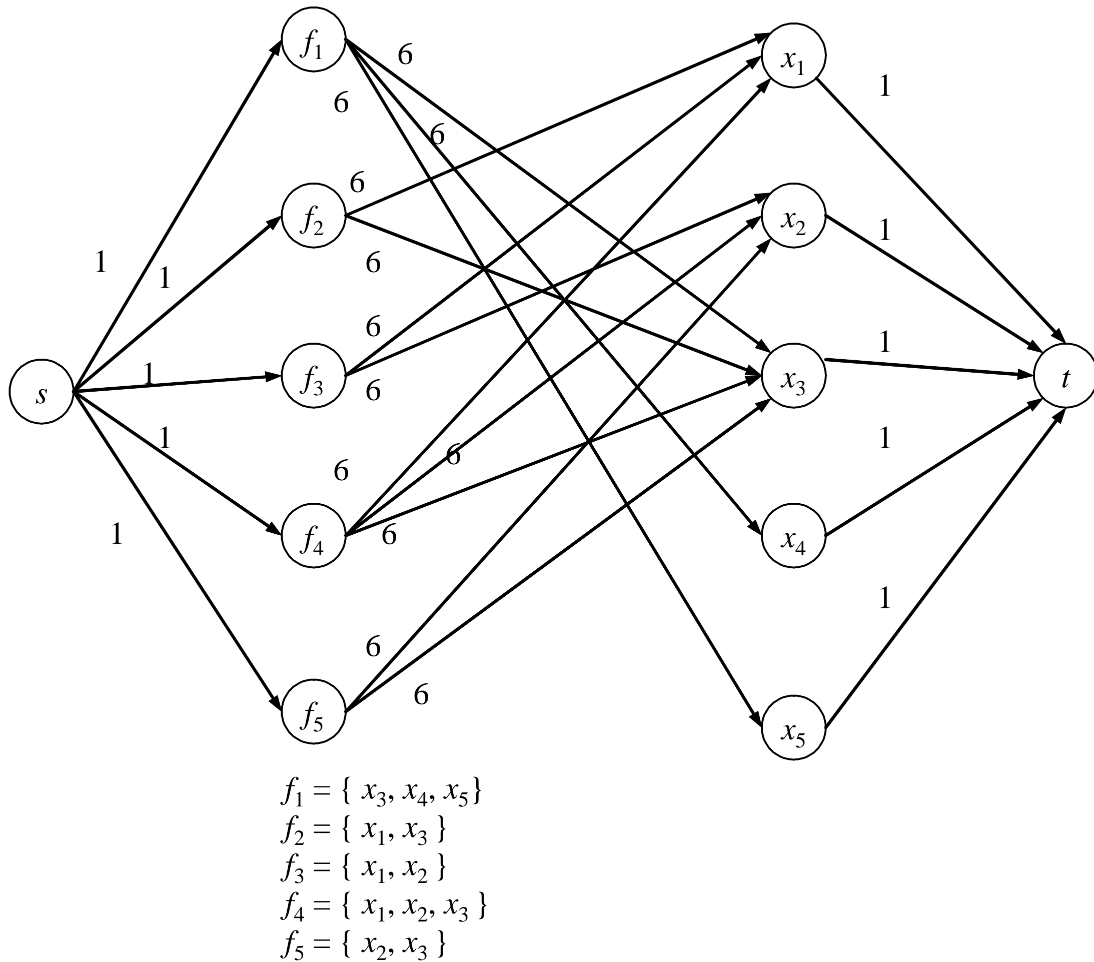


Figure 4.2: Interaction network constructed from the above relationships

target t where $s \neq t$, and in which each edge has a non-negative capacity. A ***flow*** in a network is defined to be a function f that assigns a real number to each edge, subject to two constraints:

- Flow of an edge is non-negative and less than or equal to the capacity;
- For each vertex other than the source and the target, the flow into the vertex equals the flow out of it.

The ***value*** of a flow is the net flow into the sink. Given a network, a flow is a maximum flow provided it has the largest value among all flows. A directed *s-t* path in G is a sequence of vertices and edges of the form $s, (s, v_1), v_1, (v_1, v_2), v_2, \ldots, v_{k-1}, (v_{k-1}, t), t$. An *s-t* cut is a partition of the node set V into two subsets S and $T = V\text{-}S$. Alternatively, we can define a cut as the set of edges whose endpoints belong to the different subsets S and T. A cut is referred to as an *s-t* cut if $s \in S$ and $t \in T$. The size of an *s-t* cut is the sum of the capacities of all the forward edges (edges from S to T) in the cut. An *s-t* cut is a minimum *s-t* cut provided it has the smallest size among all *s-t* cuts.

A path of a network is a sequence $s, e_0, v_1, e_1, \ldots, e_k, t$ with $s, v_1, \ldots, t \in V$, and $e_0, e_1, \ldots, e_k \in E$, such that it starts in s, ends in t and does not contain any vertex twice. The residual capacity of an edge $e_i = (v_i, v_{i+1})$ is given by

$\text{res}(e_i) = c(v_i, v_{i+1}) - f(v_i, v_{i+1})$

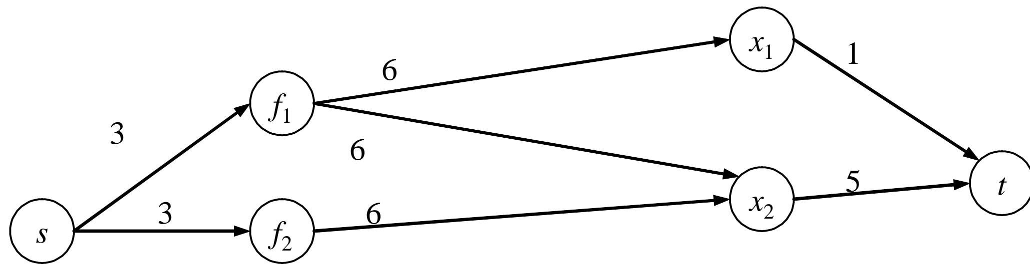
Given a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is $G_r = (V, E_r)$, where

$E_r = \{(u, v) \in V \times V: \text{res}(u, v) > 0\}$.

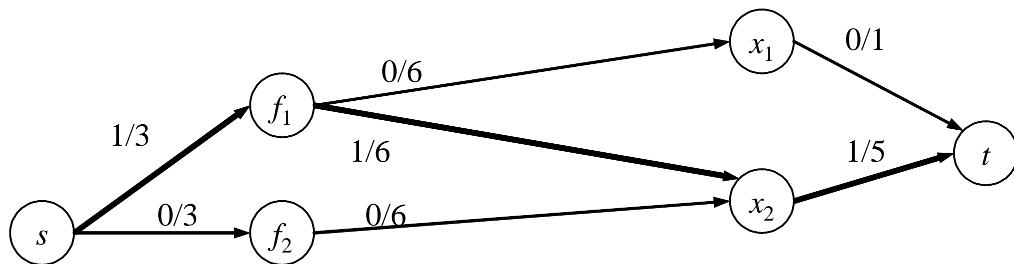
Each edge of the residual network, or residual edge, can admit a strictly positive net flow. A residual edge may not be an edge in E. An augmenting path with respect to a network G and a flow f is a simple path from s to t in the residual network G_r [Corm90].

Figure 4.3(a) and 4.3(b) illustrate a network G and the network with a flow value of 1. An augmenting path P can be formed by $s, (s, f_1), f_1, (f_1, x_2), x_2, (x_2, t), t$. The residual capacity of this path is the minimum $\text{res}(P) = \min\{\text{res}((s, f_1)), \text{res}((f_1, x_2)),$

res((x_2, t))}. Thus res(P) = min{2, 5, 4} = 2. The residual network for the network with a flow value of 1 is shown in Figure 4.3(c).



(a): Original network



(b): Network with flow =1

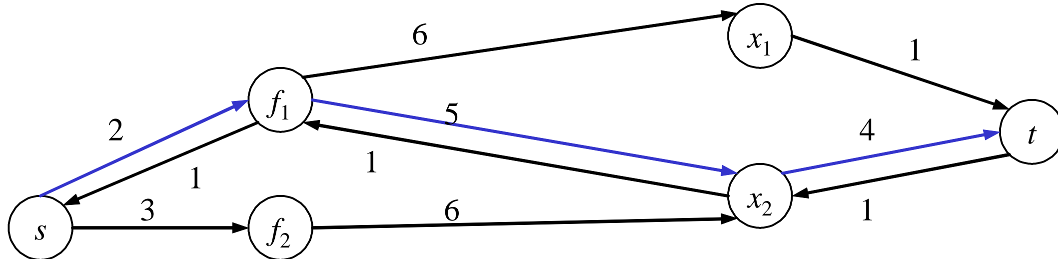


Figure 4.3: Residual network

Now we present mathematical preliminaries that prove that the consistency-checking problem can be mapped to the problem of finding the size of the minimum s-t cut in a network.

Lemma 4.1. The size l^* of the minimum s-t cut in network G is less than or equal to the number of interactions m.

Proof: A cut of G can be created by selecting all edges with an *sf*-edge label from the network (for example, see edges in dotted lines in Figure 4.4). The size of this cut is equal to the sum of the capacities of all edges with an *sf*-edge label. There are m such edges in G and the capacity for each such edge is 1 unit. Therefore, the size of this cut is m. Therefore, we can conclude that the size l^* of a minimum cut in G is less than or equal to m.

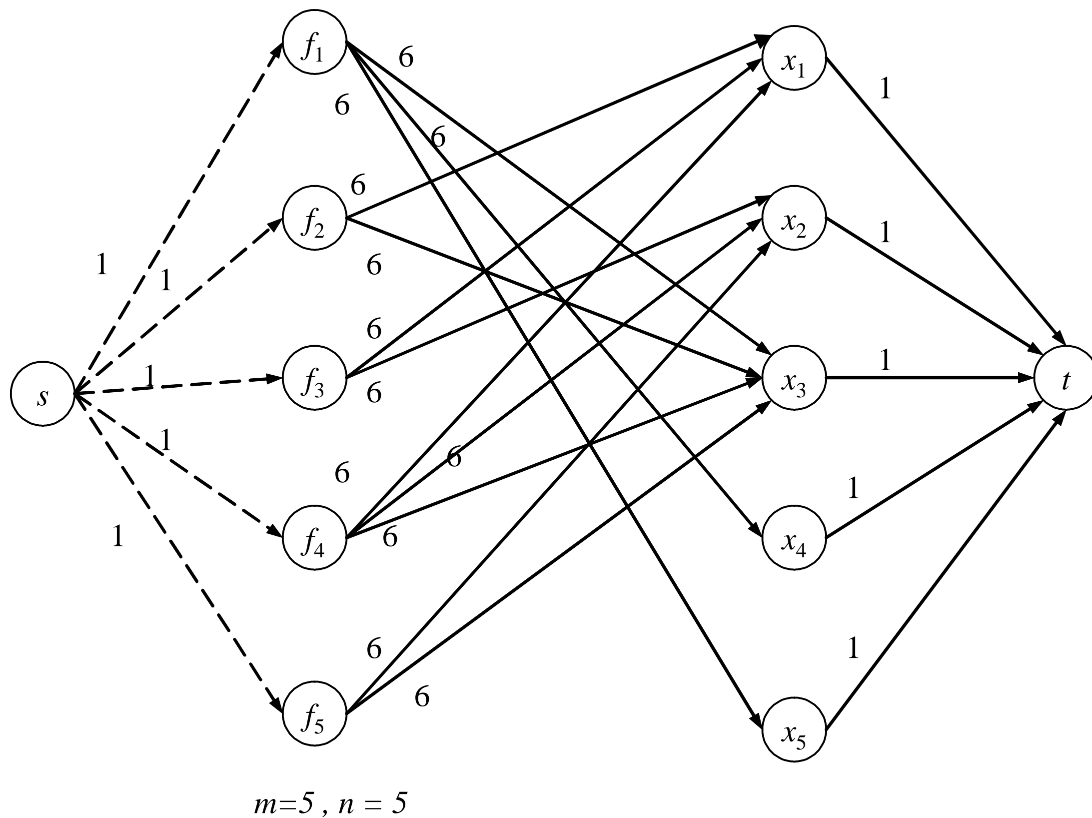


Figure 4.4: A cut of the network

Lemma 4.2. A minimum *s-t* cut of network G cannot contain an edge with an *fx*-edge label.

Proof: According to Lemma 4.1, the size of the minimum *s-t* cut of G is less than or equal to m. Since the capacity of *fx*-edges is $n + 1$, any cut that contains an edge with

an fx-edge label must have a size of at least $n + 1$. Since $n \geq m$, any cut that contains an fx-edge cannot be a minimum s-t cut due to Lemma 4.1.

Lemma 4.3. If the size l^* of the minimum s-t cut of network G is less than m, then the minimum cut must contain at least one sf-edge and one xt-edge.

Proof: According to lemma 4.2, minimum s-t cut C^* does not contain any edges with an fx-edge label. Let C^* be a minimum s-t cut of G such that $l^* < m$. Cut C^* can be of the following three types: 1) all edges in the cut are sf-edges; 2) all edges in the cut are xt-edges; 3) edges in the cut contain both types of edges. In cases 1 and 2, we can find a path from s to t. Therefore, C^* cannot be a cut. Thus only case 3 produces a valid cut.

Theorem 4.1. If there exists a subset of interactions $F' \subseteq F$ such that cardinality(F') > cardinality($\cup F'$) (i.e. the number of interactions is greater then the number of variables in the interactions), then there would exist a minimum s-t cut in network G of a size less than m.

Proof: First let us construct the interaction network as shown in Figure 4.5 according to algorithm CONSTRUCTINTERACTIONNETWORK.

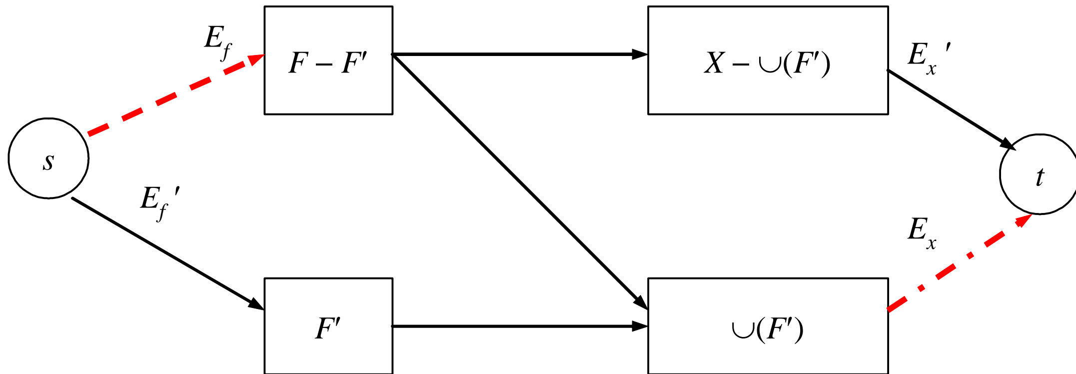


Figure 4.5: A cut illustrating terminology used in Theorem 1

We define E_f as the set of sf-edges that connect the s-node with f-nodes that correspond to $F - F'$, E_x as the set of xt-edges that connect x-nodes that correspond to $\cup F'$ with the t-node, E_f' as the set of sf-edges that connect the s-node with f-nodes that correspond to F', and E_x' as the set of xt-edges that connect x-nodes that correspond to $X - \cup F'$ with the t-node. And we define the cardinalities of these sets of edges as the following:

l_f = cardinality(E_f), l_x' = cardinality(E_x'), l_f' = cardinality(E_f'), l_x = cardinality(E_x)

Since for every f-node$\in F$, there is only one sf-edge that connects it with the s-node, cardinality(F') = cardinality(E_f'), thus l_f' = cardinality(F').

Similarly, since for every x-node$\in F$, there is only one xt-edge that connects it with the t-node, cardinality($\cup F'$) = cardinality(E_x), thus l_x = cardinality($\cup F'$).

Cut $C = E_f \cup E_x$ is an s-t cut (shown in dotted lines in Figure 4.5) based on its construction. We define l = cardinality (C).

According to the construction of the network, we have

$$l_f + l_f' = m \qquad (4\text{-}1)$$

According to the definition of C we have $l = l_f + l_x$

We are given cardinality(F') > cardinality($\cup F'$), that is

$$l_f' > l_x \qquad (4\text{-}2)$$

Hence $l = l_f + l_x$

$$l < l_f + l_f' \qquad \text{(by 4-2)}$$

$$l < m \qquad \text{(by 4-1)}$$

Since, cardinality(C) < m and cardinality(C^*) ≤ cardinality(C), cardinality(C^*) < m.

Figure 4.6 shows an example further illustrating terminology used in this Theorem.

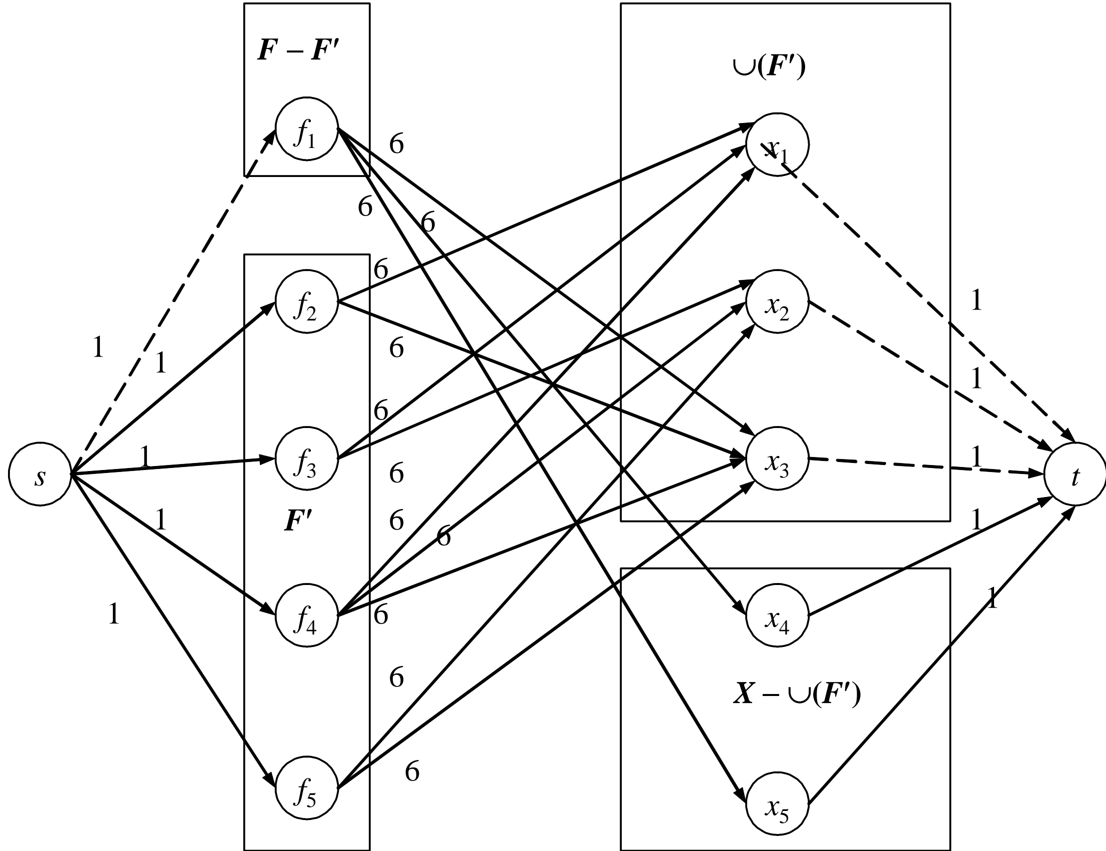


Figure 4.6: An example of a cut for illustrating Theorem 1

Theorem 4.2. Let C^* be a minimum *s*-*t* cut of the interaction network *G*, and the size of the cut l^* be less than *m*. In this case there would exist $F' \subseteq F$ such that cardinality(F') > cardinality($\cup F'$).

Proof:

According to Lemma 4.2 and Lemma 4.3, the cut must be formed in the manner shown in Figure 4.7. We define E_f as the set of *sf*-edges that connect the *s*-node with *f*-nodes that correspond to F'', E_f' as the set of *sf*-edges that connect the *s*-node with *f*-nodes that correspond to $F - F''$, E_x as the set of *xt*-edges that connect *x*-nodes

corresponding to $\cup(F - F'')$ with the t-node, and E_x' as the set of xt-edges that connect x-nodes that correspond to $X - \cup(F - F'')$ with the t-node.

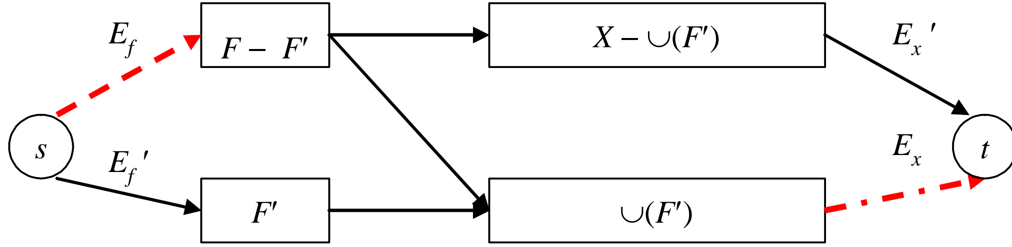


Figure 4.7: A cut illustrating terminology used in Theorem 2

We define the cardinalities of these sets of edges as the following:

l_f = cardinality(E_f), l_x' = cardinality(E_x'), l_f' = cardinality(E_f'), l_x = cardinality(E_x)

According to the construction of the network, we have

$l_f + l_f' = m$ (4-3)

Since cut $C^* = E_f \cup E_x$, cardinality $(C^*) = l_f + l_x$ (4-4)

According to Lemma 4.3, we also have:

$l_f > 0$ and $l_x > 0$

We are given cardinality $(C^*) < m$, thus

$l_f + l_x < m$ (by 4-4)

$l_f + l_x < l_f + l_f'$ (by 4-3)

Then we have $l_x < l_f'$

That states that cardinality(E_x) < cardinality (E_f')

Since for every f-node$\in F$, there is only one sf-edge that connects it with the s-node, cardinality($F - F''$) = cardinality(E_f').

Similarly, since for every x-node$\in F$, there is only one xt-edge that connects it with the t-node, cardinality($\cup(F - F'')$) = cardinality(E_x).

Therefore, cardinality($\cup(F - F'')$) < cardinality ($F - F''$).

We rename ($F - F''$) as F', then we have

cardinality(F') > cardinality($\cup F'$)

Corollary 4.1. Let C^* be a minimum s-t cut of size less than m and E_f' be the set of sf-edges that are not in C^*. The set of inconsistent interactions is represented by the f-nodes that are connected to the s-node by edges in E_f'.

Proof: It directly follows from Theorem 4.2.

Theorem 4.3. Let C^* be a minimum cut of size less than m. Let F' be the set of f-nodes that are connected to s-nodes by edges that are not in C^*. Let F be the set of all f-nodes. Then $\forall\ F'' \subseteq (F - F')$, cardinality($F''$) $\leq$ cardinality ($\cup F''$).

Proof:

We will prove this theorem by contradiction.

Assume there exists $F'' \subseteq (F - F')$ such that cardinality (F'') > cardinality ($\cup F''$).

We define E_f as the set of sf-edges that connect the s-node with f-nodes that correspond to $F - F'$, E_x as the set of xt-edges that connect x-nodes corresponding to $\cup F'$ with the t-node, E_f' as the set of sf-edges that connect the s-node with f-nodes that correspond to F', E_x' as the set of xt-edges that connect x-nodes that correspond to $X - \cup F'$ with the t-node, E_f'' as the set of sf-edges that connect the s-nodes with f-nodes that correspond to F'', and E_x'' as the set of xt-edges that connect x-nodes corresponding to $\cup F''$ with the t-node.

Since for every f-node$\in F$, there is only one sf-edge that connects it with the s-node, cardinality(F'') = cardinality(E_f'').

Similarly, since for every x-node$\in F$, there is only one xt-edge that connects it with the t-node, cardinality($\cup F''$) = cardinality(E_x'').

Then the assumption can also be represented as

cardinality(E_x'') – cardinality(E_f'') < 0 (4-5)

We separate F'' from F-F' as shown in Figure 4.8. Obviously $(E_f - E_f'') \cup E_x'' \cup E_x$ is also a cut C' of the network.

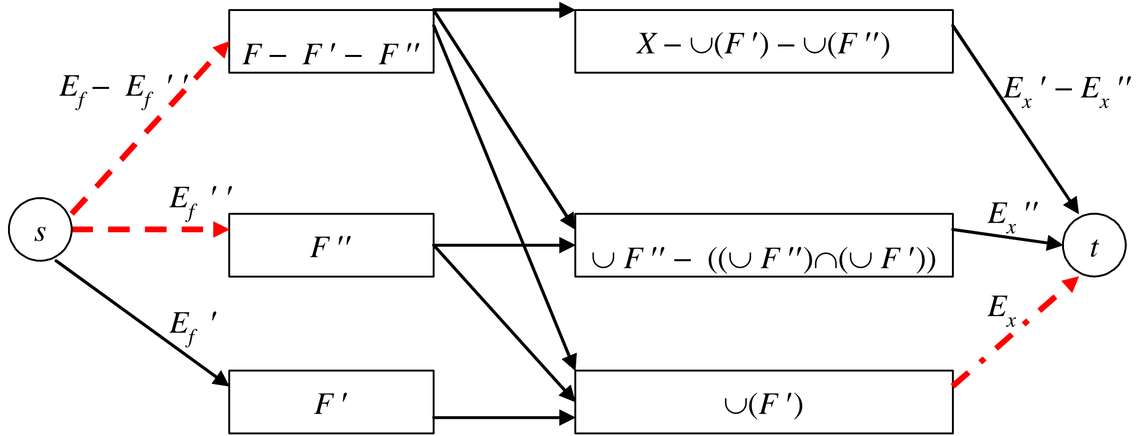


Figure 4.8: A cut illustrating terminology used in Theorem 3

According to the definition of cut,

cardinality(C^*) = cardinality(E_f) + cardinality(E_x) (4-6)

cardinality(C') = cardinality($E_f - E_f''$) + cardinality(E_x'') + cardinality(E_x) (4-7)

cardinality($E_f - E_f''$) = cardinality(E_f) – cardinality(E_f'') (4-8)

Thus, cardinality(C') = cardinality(E_f) – cardinality(E_f'')+ cardinality(E_x'') + cardinality(E_x)

(4-9) (by 4-7 and 4-8)

Then

cardinality(C') – cardinality(C^*) = cardinality(E_x'') – cardinality(E_f'') < 0

(by 4-6 and 4-9)

Therefore, cardinality (C') < cardinality (C^*)

Thus C^* is not a minimum *s-t* cut. This contradicts with the theorem statement.

From the above theorems and corollary, we can conclude that the consistency-checking problem can be solved by finding the size of the minimum *s-t* cut of *G*. Theorem 4.3 helps in ensuring that there are no other inconsistent interactions that are not covered by Corollary 1.

4.3 Algorithms For Finding Minimum S-T Cut And Identifying Inconsistent Interactions

4.3.1 Algorithm for finding minimum s-t cut in network G

According to the duality between maximum flow problems and minimum cut problems, the size of the minimum *s-t* cut can be found by computing the maximum flow between *s* and *t*. Our algorithm is based on Ford and Fulkerson's basic maximum flow algorithm of finding the augmenting path.

Algorithm FINDMINIMUMSTCUTSIZE

Input: A directed network *G*

Output: The size of the minimum cut of *G* and the residual network G_r of *G*

Steps:

1) Set size of minimum cut to 0.
2) Initialize flow of the network, set $f(e) = 0, \forall e \in E$.
3) Set $G_r = G$.
4) Find an augmenting path from the *s-node* to the *t-node* in G_r

i. If a path is found, then

   a. Augment flow along this path.

   b. Increase the size of the minimum cut by 1.

   c. Generate new residual network G_r.

   d. Go to Step 4.

ii. Else, return the size of the minimum cut and residual network.

The working of this algorithm is illustrated in Figure 4.9.

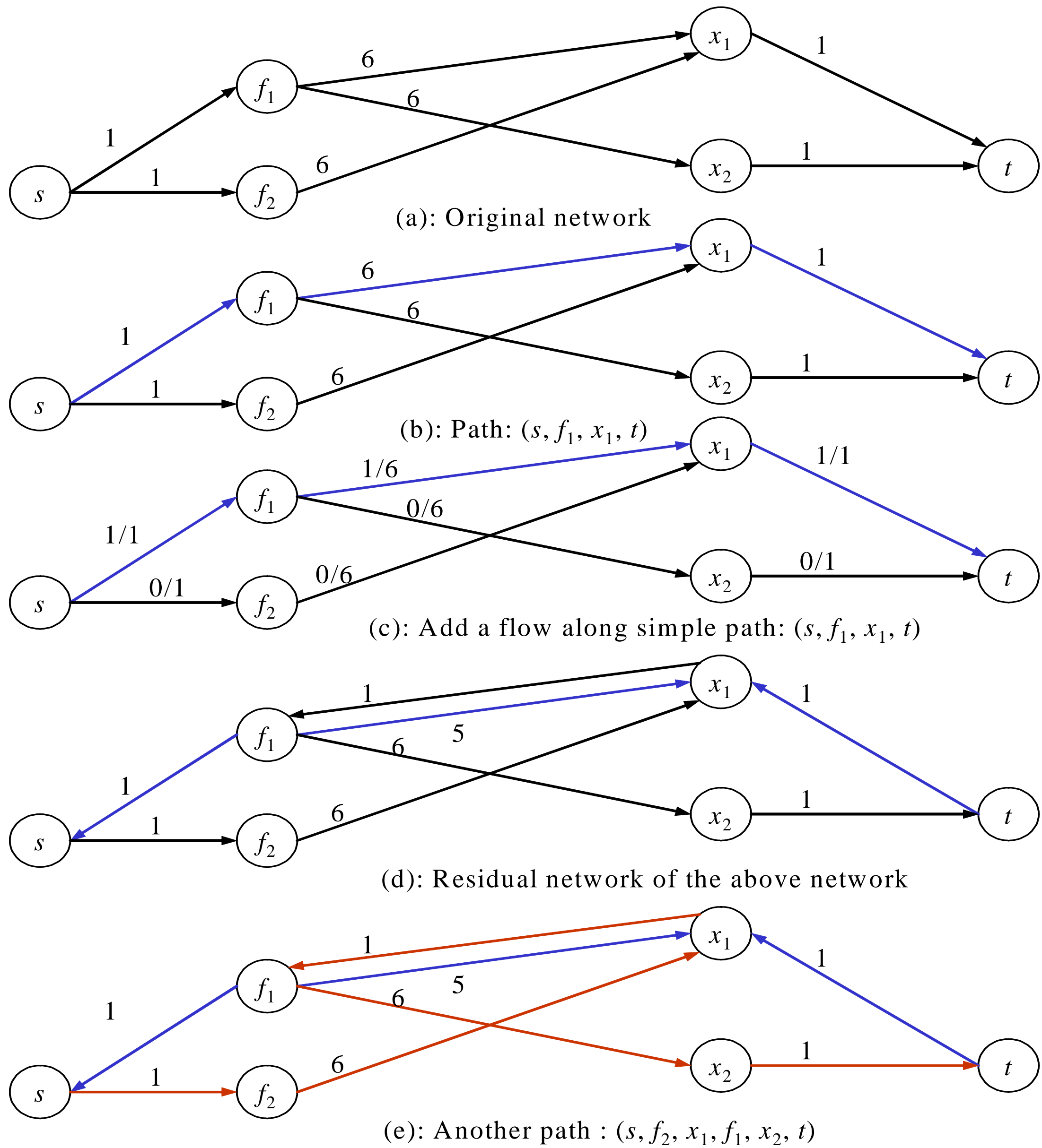


Figure 4.9: Illustration of algorithm FINDMINIMUMSTCUTSIZE

Figure 4.9a shows the original network. Initially, the residual network is the same as this network (see Step 3 of the above algorithm). Figure 4.9b shows an s-t path as s, (s, f_1), f_1, x_1, (x_1, t), t. Sending a unit flow along this path will saturate the flow capacities in edges (s, f_1) and (x_1, t) as shown in Figure 4.9c. The residual network with respect to this flow is shown in Figure 4.9d. A new

path shown in Figure 4.9e is found as s, (s, f_2), f_2, (f_2, x_1), x_1, (x_1, f_1), f_1, (f_1, x_2), x_2, (x_2, t), t.

Now we analyze the complexity of this algorithm. Step 1 can be executed in time O(1). Step 2 can be done in time O(E) where, E = Number of *sf*-edges + Number of *fx*-edges + Number of *xt*-edges. E has an upper bound of $n + nm + m$. Thus, Step 2 takes time O(nm). Step 3 takes O($V + E$). Since O(V) = O(n+m), step 3 takes O(nm). Step 4 will be executed at most m times. For a depth-first search, Step 4a takes time O(E) + O(1) + O(V + 2E) = O(nm). Step 4b takes O(1) time. Thus in the worst case, Step 4 takes O(nm^2). Thus the worst case time complexity for this algorithm is O(nm^2).

For the network shown in Figure 4.2, we find the size of the minimum s-t cut of the network. In this case C^* = 4 as shown in Figure 4.10. The maximum flow of the network is also shown in Figure 4.10. Since m = 5, the set of the interactions is not consistent. The residual network with respect to the maximum flow is shown in Figure 4.11.

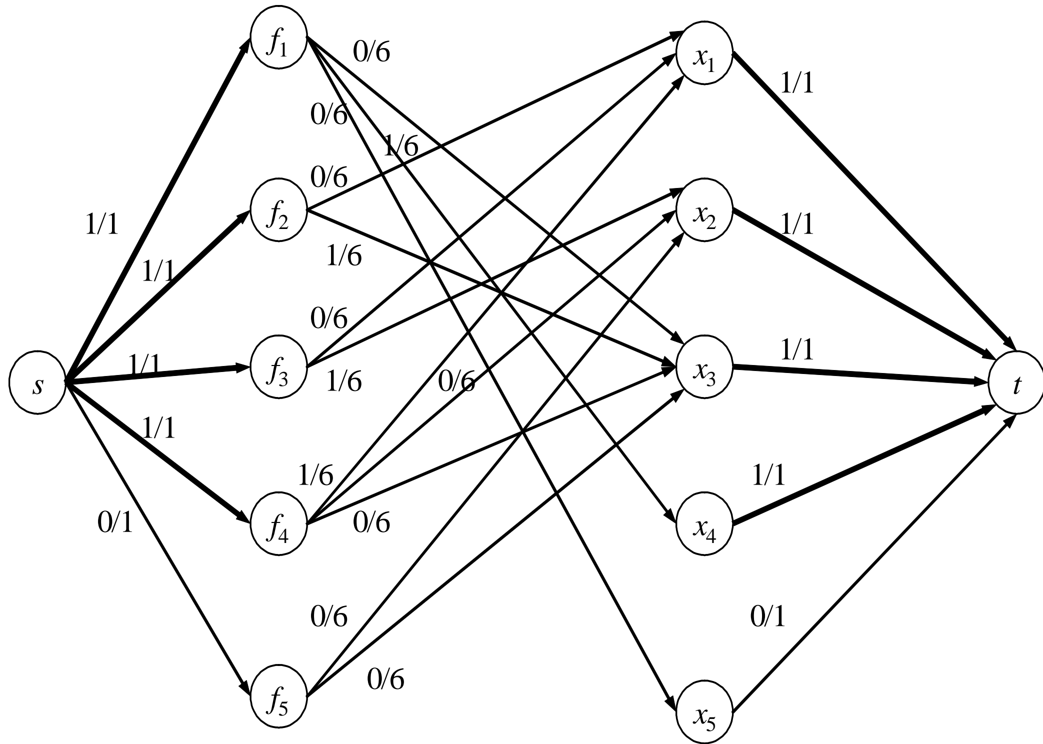


Figure 4.10: Maximum flow of the graph

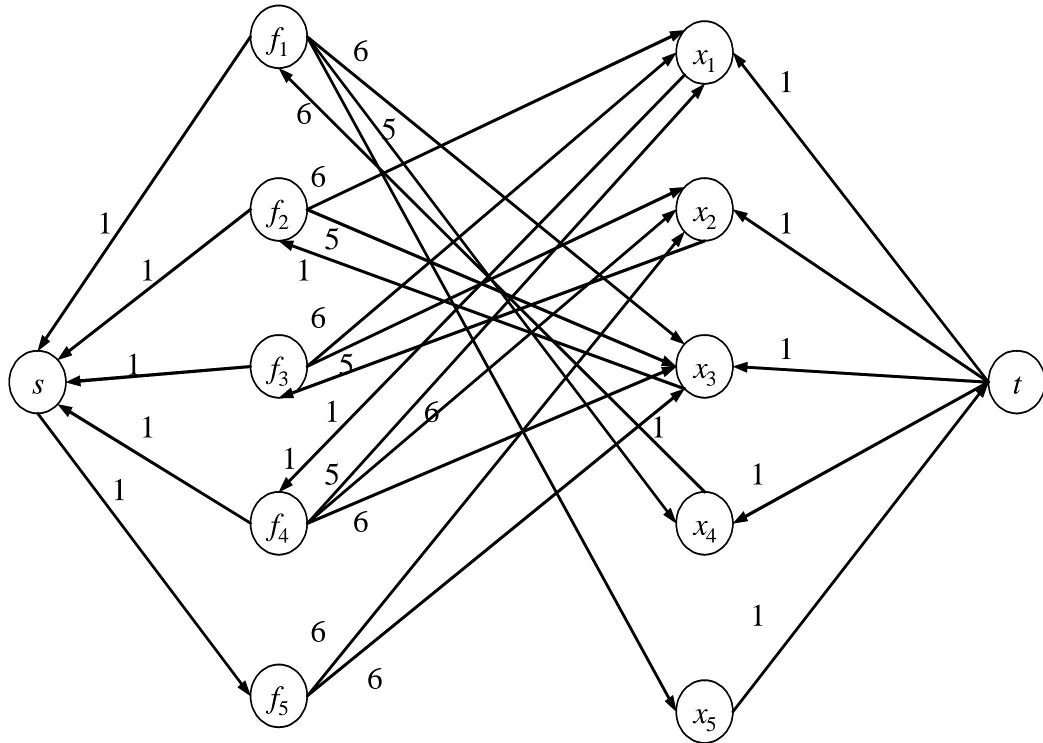


Figure 4.11: Residual network corresponding to maximum flow

Ford and Fulkerson's algorithm finds maximum flow by finding all the augmenting paths in the network from s to t and saturating the flows along the paths. However, there are several characteristics of our problem that can be used to reduce the complexity of the algorithm directly.

1) The network in our problem is actually a special network. Network $G = (V, E)$ has a node set V partitioned into two subsets V_1 and V_2 so that for every edge $e_i=(v_i, v_{i+1})\in E$, either $v_i \in V_1$ and $v_{i+1}\in V_2$ or $v_i \in V_2$ and $v_{i+1}\in V_1$. Thus any s-t path follows the pattern $s, f, x, f, x, f, x, \ldots, t$ in which f-nodes and x-nodes appear in a pair wise manner.
2) Every sf-edge and xt-edge has capacity of 1. That means that once such an edge is used in a path, it won't be used in another path. Meanwhile, an f-node or an x-node also can only be used in one path.

4.3.2 Algorithm For Finding Inconsistent Interactions

Algorithm FINDINCONSISTENTINTERACTIONS

Input: Interaction residual network G_r corresponding to the maximum flow

Output: set of f-nodes corresponding to inconsistent interactions

Steps:

1) Use depth-first search to find all nodes in residual network G_r that are reachable from s-node and put these nodes in set R.
2) Remove x-nodes from R and return R.

Now we will show that R corresponds to the f-nodes that are connected to the s-node by edges in E_f' as stated in Corollary 1. We denote the node set that is reachable from s in G_r as V_1, and the set of the remaining nodes as $V_2 = V - V_1$.

There is no path in the residual network such that the s-node reaches the t-node. Otherwise, an augmenting flow could have been generated and hence flow would have not been maximum. Thus, $s \in V_1$ and $t \in V_2$. Therefore, cut $C = \{V_1, V_2\}$ is an s-t cut. Since the flow is maximum, according to the duality between maximum flow and minimum cut, C is a minimum s-t cut [Ford56]. Therefore, we conclude that inconsistent interactions can be found by finding reachable nodes in the residual network corresponding to the maximum flow. Since we are only concerned about the inconsistent interactions, we remove x-nodes in the reachable node set.

Now we analyze the complexity of this algorithm. For a depth-first search, Step 1 can be executed in time $O(E + V) = O(nm)$. Step 2 takes time $O(n + m)$. Therefore, this algorithm runs in $O(nm)$.

For the network shown in Figure 4.2, the residual network with respect to the maximum flow is shown in Figure 4.11. Now we can find the reachable nodes from s-nodes as $\{f_2, f_3, f_4, f_5, x_1, x_2, x_3\}$ as shown in Figure 4.12. Thus the set of interaction nodes $\{f_2, f_3, f_4, f_5\}$ is inconsistent. One can easily verify that there are only three variables $\{x_1, x_2, x_3\}$ involved in four interactions $\{f_2, f_3, f_4, f_5\}$.

4.4 Implementation And Examples

We have implemented the algorithms described in this chapter using C++. The implementation has been tested on the Windows 2000 platform. We ran the program on a PC with the following configuration: (1) AMD Athlon XP1700+ CPU and (2) 1GB Memory.

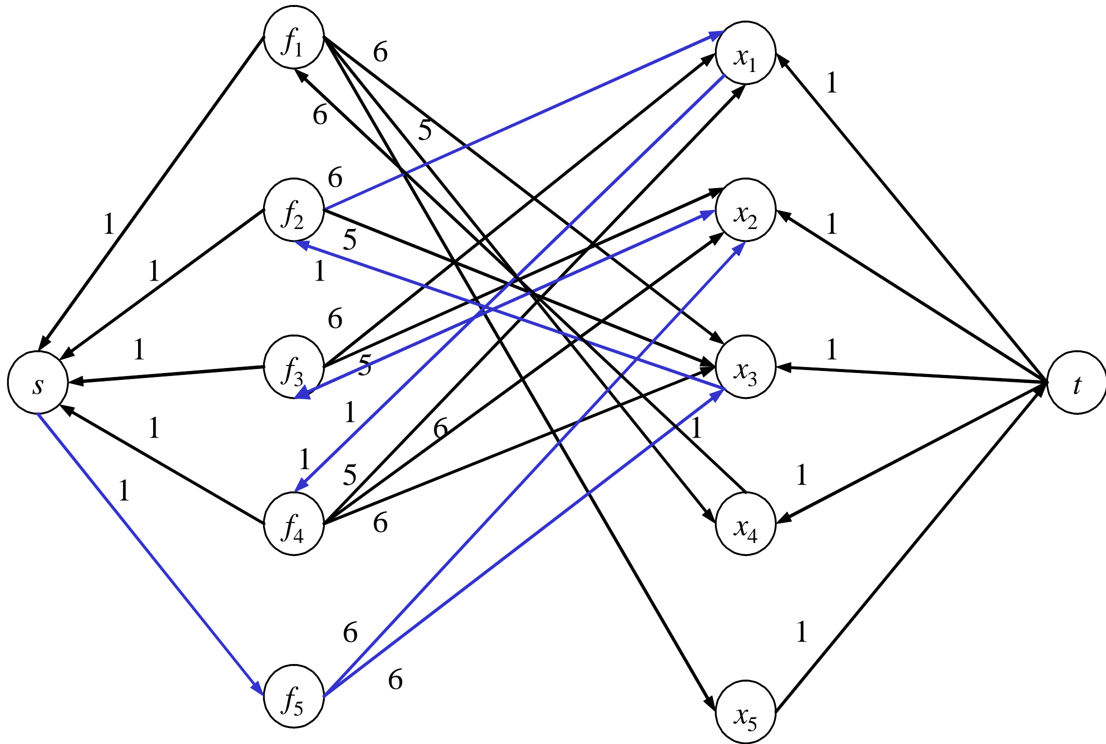


Figure 4.12: Finding inconsistent relationships

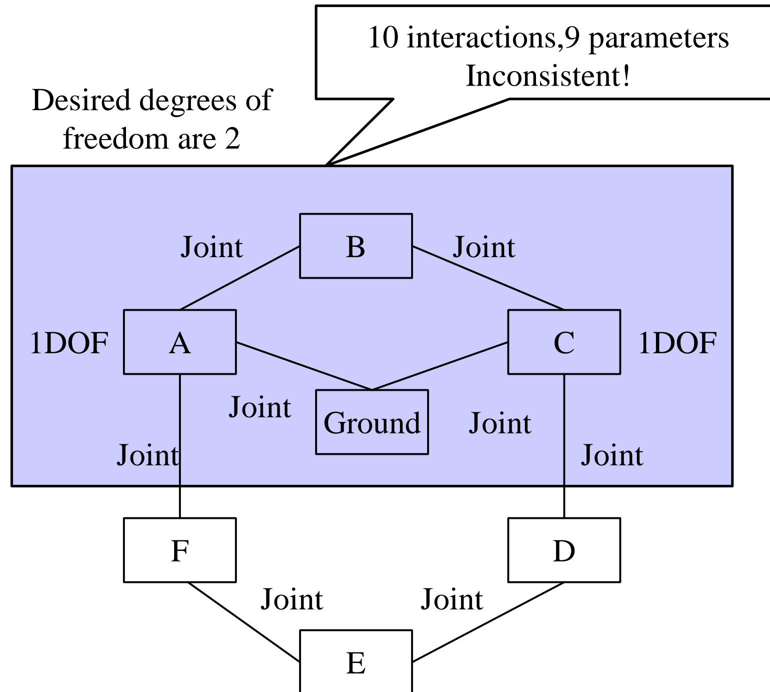


Figure 4.13: Design alternative A of a planar mechanism

Figure 4.13 shows design alterative A behind a device based on a planar mechanism. There are 6 active artifacts that represent various links in the device (the ground artifact is not counted). Every artifact can be described by three parameters (x, y, θ). These parameters present the x and y coordinate of the center of the artifact, and its orientation. In this device, artifacts interact with each other via joints. We assume that all joints in this case are pivot joints. The presence of a pivot joint reduces two degrees of freedom between two links. This means that while (x, y, θ) parameters for one of the links can be assigned independently, only one variable for the second link can be assigned independently. Therefore, as per our terminology, there are two interactions among artifacts due to the presence of the pivot joint. Both of these interactions involve the same set of variables. However, the equations behind each interaction will be different and can only be found after assigning dimensional parameters to the links. As mentioned before, we do not care about the actual equations involved but rather the set of parameters that participate in an interaction.

Therefore, interactions among artifacts due to the presence of joints can be described by the following set of participating parameters:

$f_1 = \{x_A, y_A, \theta_A, x_B, y_B, \theta_B\}$, $f_2 = \{x_A, y_A, \theta_A, x_B, y_B, \theta_B\}$

Similarly, for other joints we get

$f_3 = \{x_B, y_B, \theta_B, x_C, y_C, \theta_C\}$, $f_4 = \{x_B, y_B, \theta_B, x_C, y_C, \theta_C\}$

$f_5 = \{x_C, y_C, \theta_C, x_D, y_D, \theta_D\}$, $f_6 = \{x_C, y_C, \theta_C, x_D, y_D, \theta_D\}$

$f_7 = \{x_D, y_D, \theta_D, x_E, y_E, \theta_E\}$, $f_8 = \{x_D, y_D, \theta_D, x_E, y_E, \theta_E\}$

$f_9 = \{x_E, y_E, \theta_E, x_F, y_F, \theta_F\}$, $f_{10} = \{x_E, y_E, \theta_E, x_F, y_F, \theta_F\}$

$f_{11} = \{x_F, y_F, \theta_F, x_A, y_A, \theta_A\}$, $f_{12} = \{x_F, y_F, \theta_F, x_A, y_A, \theta_A\}$

Artifacts A and C are connected to the ground via pivot joints, so we need to model the following interactions:

$f_{13}= \{x_A, y_A, \theta_A\}$, $f_{14} = \{x_A, y_A, \theta_A\}$

$f_{15}= \{x_C, y_C, \theta_C\}$, $f_{16} = \{x_C, y_C, \theta_C\}$

We want to have two degrees of freedom in this device. These constraints are modeled as interactions as well. However, only one parameter participates in these two interactions. Therefore, we get

$f_{17}= \{\theta_A\}$, $f_{18} = \{\theta_C\}$

Then the interaction consistency problem for this device is formulated as the following:

$X = \{x_A, y_A, \theta_A, x_B, y_B, \theta_B, x_C, y_C, \theta_C, x_D, y_D, \theta_D, x_E, y_E, \theta_E, x_F, y_F, \theta_F\}$

$F = \{f_1, f_2, f_3, \ldots, f_{18}\}$

$n = 18$ and $m = 18$

By running our software, we get the following result:

The size of the minimum *s-t* cut is $17 < m$, thus the interactions are inconsistent.

The set of inconsistent interactions are identified as $\{f_1, f_2, f_3, f_4, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}, f_{18}\}$. These ten interactions only involve nine variables. Hence this design concept is not valid.

Now let us consider another design alternative. This design alternative called alternative B is shown in Figure 4.14. This alternative has the same numbers of artifacts and joints. However, the interactions are different. Interactions in this design can be modeled as the following:

$f_1= \{x_A, y_A, \theta_A, x_B, y_B, \theta_B\}$, $f_2 = \{x_A, y_A, \theta_A, x_B, y_B, \theta_B\}$

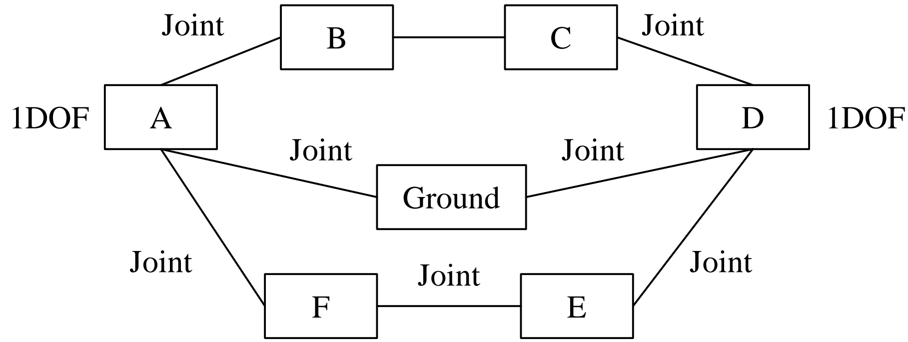


Figure 4.14: Design alternative B of a planar mechanism

$f_3 = \{x_B, y_B, \theta_B, x_C, y_C, \theta_C\}$, $f_4 = \{x_B, y_B, \theta_B, x_C, y_C, \theta_C\}$

$f_5 = \{x_C, y_C, \theta_C, x_D, y_D, \theta_D\}$, $f_6 = \{x_C, y_C, \theta_C, x_D, y_D, \theta_D\}$

$f_7 = \{x_D, y_D, \theta_D, x_E, y_E, \theta_E\}$, $f_8 = \{x_D, y_D, \theta_D, x_E, y_E, \theta_E\}$

$f_9 = \{x_E, y_E, \theta_E, x_F, y_F, \theta_F\}$, $f_{10} = \{x_E, y_E, \theta_E, x_F, y_F, \theta_F\}$

$f_{11} = \{x_F, y_F, \theta_F, x_A, y_A, \theta_A\}$, $f_{12} = \{x_F, y_F, \theta_F, x_A, y_A, \theta_A\}$

Artifact A and artifact D are connected to the ground, so we have the following interactions:

$f_{13} = \{x_A, y_A, \theta_A\}$, $f_{14} = \{x_A, y_A, \theta_A\}$

$f_{15} = \{x_D, y_D, \theta_D\}$, $f_{16} = \{x_D, y_D, \theta_D\}$

We again want to have two degrees of freedom in the system. So we get,

$f_{17} = \{\theta_A\}$, $f_{18} = \{\theta_D\}$

By running our algorithm, we get the following result:

The size of the minimum *s-t* cut is 18 = *m*, thus the interactions are consistent.

This example illustrates that the interactions can have significant influence on the validity of a design concept.

In design of complex spatial mechanisms, it is getting harder to detect redundant links in mechanisms as they are getting more complicated. Figure 4.15 shows design alterative A behind a device based on a spatial mechanism.

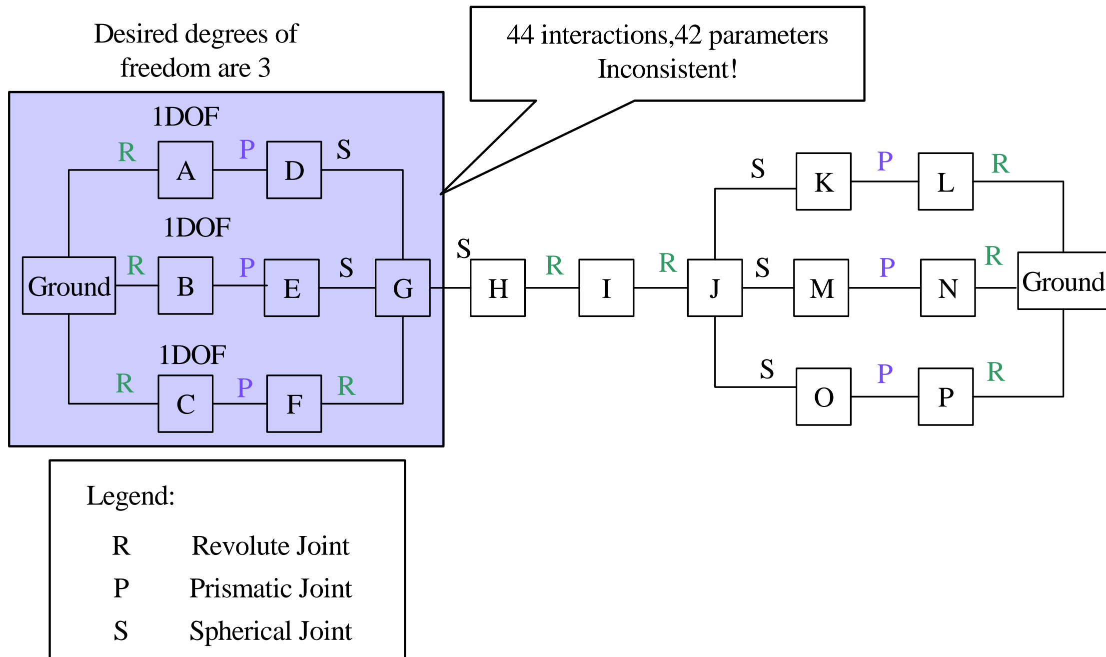


Figure 4.15: Design alternative A of a spatial mechanism

There are 16 active artifacts that represent various links in the device (the ground artifact is not counted). Every artifact can be described by six parameters (x, y, z, θ_x, θ_y, θ_z). These parameters present the x, y and z coordinate of the center of the artifact, and its orientation. In this device, artifacts interact with each other via joints. The joints between artifacts A and Ground, B and Ground, C and Ground, F and G, P and Ground, N and Ground, L and Ground, H and I, I and J, are revolute joints. The joints between A and D, B and E, C and F are prismatic joints. The joints between D and G, E and G, G and H, J and O, J and M, J and K are spherical joints. The presence of a revolute joint reduces five degrees of freedom between two links. This means that while (x, y, z, θ_x, θ_y, θ_z) parameters for one of the links can be assigned

independently, only one variable for the second link can be assigned independently. Therefore, as per our terminology, there are five interactions among artifacts due to the presence of the revolute joint. All of these interactions involve the same set of variables. However, the equations behind each interaction will be different and can only be found after assigning dimensional parameters to the links. As mentioned before, we do not care about the actual equations involved but rather the set of parameters that participate in an interaction. Similarly, the presence of a spherical joint reduces three degrees of freedom between two links and the presence of a prismatic joint reduces five degrees of freedom between two links. Therefore, there are three interactions among artifacts due to the presence of the revolute joint and there are five interactions among artifacts due to the presence of the revolute joint.

Interactions among artifacts due to the presence of joints can be described by the following set of participating parameters:

For the joint that connects A and Ground, we have:

$f_1 = \{x_A, y_A, z_A, \theta_{xA}, \theta_{yA}, \theta_{zA}\}$. f_2, f_3, f_4, f_5, have the same qualitative structure as f_1.

Similarly, for other revolute joints we get

$f_6 = \{x_B, y_B, z_B, \theta_{xB}, \theta_{yB}, \theta_{zB}\}$. f_7, f_8, f_9, f_{10}, have the same qualitative structure as f_6.

$f_{11} = \{x_C, y_C, z_C, \theta_{xC}, \theta_{yC}, \theta_{zC}\}$. $f_{12}, f_{13}, f_{14}, f_{15}$, have the same qualitative structure as f_{11}.

$f_{16} = \{x_F, y_F, z_F, \theta_{xF}, \theta_{yF}, \theta_{zF}, x_G, y_G, z_G, \theta_{xG}, \theta_{yG}, \theta_{zG}\}$. $f_{17}, f_{18}, f_{19}, f_{20}$, have the same qualitative structure as f_{16}.

$f_{21} = \{x_P, y_P, z_P, \theta_{xP}, \theta_{yP}, \theta_{zP}\}$. $f_{22}, f_{23}, f_{24}, f_{25}$, have the same qualitative structure as f_{21}.

$f_{26} = \{x_N, y_N, z_N, \theta_{xN}, \theta_{yN}, \theta_{zN}\}$. $f_{27}, f_{28}, f_{29}, f_{30}$, have the same qualitative structure as f_{26}.

$f_{31} = \{x_L, y_L, z_L, \theta_{xL}, \theta_{yL}, \theta_{zL}\}$. $f_{32}, f_{33}, f_{34}, f_{35}$, have the same qualitative structure as f_{31}.

f_{36}= {x_H, y_H, z_H, θ_{xH}, θ_{yH}, θ_{zH}, x_I, y_I, z_I, θ_{xI}, θ_{yI}, θ_{zI}}. f_{37}, f_{38}, f_{39}, f_{40}, have the same qualitative structure as f_{36}.

f_{41}= {x_I, y_I, z_I, θ_{xI}, θ_{yI}, θ_{zI}, x_J, y_J, z_J, θ_{xJ}, θ_{yJ}, θ_{zJ}}. f_{42}, f_{43}, f_{44}, f_{45} have the same qualitative structure as f_{41}.

For the prismatic joints, we need to model the following interactions:

f_{46}= {x_A, y_A, z_A, θ_{xA}, θ_{yA}, θ_{zA}, x_D, y_D, z_D, θ_{xD}, θ_{yD}, θ_{zD}}. f_{47}, f_{48}, f_{49}, f_{50}, have the same qualitative structure as f_{46}.

f_{51}= {x_B, y_B, z_B, θ_{xB}, θ_{yB}, θ_{zB}, x_E, y_E, z_E, θ_{xE}, θ_{yE}, θ_{zE}}. f_{52}, f_{53}, f_{54}, f_{55}, have the same qualitative structure as f_{51}.

f_{56}= {x_C, y_C, z_C, θ_{xC}, θ_{yC}, θ_{zC}, x_F, y_F, z_F, θ_{xF}, θ_{yF}, θ_{zF}}. f_{57}, f_{58}, f_{59}, f_{60}, have the same qualitative structure as f_{56}.

f_{61}= {x_O, y_O, z_O, θ_{xO}, θ_{yO}, θ_{zO}, x_P, y_P, z_P, θ_{xP}, θ_{yP}, θ_{zP}}. f_{62}, f_{63}, f_{64}, f_{65}, have the same qualitative structure as f_{61}.

f_{66}= {x_K, y_K, z_K, θ_{xK}, θ_{yK}, θ_{zK}, x_L, y_L, z_L, θ_{xL}, θ_{yL}, θ_{zL}}. f_{67}, f_{68}, f_{69}, f_{70}, have the same qualitative structure as f_{66}.

f_{71}= {x_M, y_M, z_M, θ_{xM}, θ_{yM}, θ_{zM}, x_N, y_N, z_N, θ_{xN}, θ_{yN}, θ_{zN}}. f_{72}, f_{73}, f_{74}, f_{75}, have the same qualitative structure as f_{71}.

For the spherical joints, we need to model the following interactions:

f_{76}= {x_D, y_D, z_D, θ_{xD}, θ_{yD}, θ_{zD}, x_G, y_G, z_G, θ_{xG}, θ_{yG}, θ_{zG}}. f_{77}, f_{78}, have the same qualitative structure as f_{76}.

f_{79}= {x_E, y_E, z_E, θ_{xE}, θ_{yE}, θ_{zE}, x_G, y_G, z_G, θ_{xG}, θ_{yG}, θ_{zG}}. f_{80}, f_{81}, have the same qualitative structure as f_{79}.

f_{82}= {x_G, y_G, z_G, θ_{xG}, θ_{yG}, θ_{zG}, x_H, y_H, z_H, θ_{xH}, θ_{yH}, θ_{zH}}. f_{83}, f_{84}, have the same qualitative structure as f_{82}.

f_{85}= {x_J, y_J, z_J, θ_{xJ}, θ_{yJ}, θ_{zJ}, x_O, y_O, z_O, θ_{xO}, θ_{yO}, θ_{zO}}. f_{86}, f_{87}, have the same qualitative structure as f_{85}.

f_{88}= {x_J, y_J, z_J, θ_{xJ}, θ_{yJ}, θ_{zJ}, x_M, y_M, z_M, θ_{xM}, θ_{yM}, θ_{zM}}. f_{89}, f_{90}, have the same qualitative structure as f_{88}.

f_{91}= {x_J, y_J, z_J, θ_{xJ}, θ_{yJ}, θ_{zJ}, x_K, y_K, z_K, θ_{xK}, θ_{yK}, θ_{zK}}. f_{92}, f_{93}, have the same qualitative structure as f_{91}.

Since there are 16 active artifacts excluding ground, there are 96 variables in total. We want to have at least three degrees of freedom in this device. These constraints are modeled as interactions as well. Therefore, we get

f_{94}= {x_A, y_A, z_A, θ_{xA}, θ_{yA}, θ_{zA}}, f_{95}= {x_B, y_B, z_B, θ_{xB}, θ_{yB}, θ_{zB}}, f_{96}= {x_C, y_C, z_C, θ_{xC}, θ_{yC}, θ_{zC}}.

Then the interaction consistency problem for this device is formulated as the following:

X = { x_A, y_A, z_A, θ_{xA}, θ_{yA}, θ_{zA}, x_B, y_B, z_B, θ_{xB}, θ_{yB}, θ_{zB}, …, x_P, y_P, z_P, θ_{xP}, θ_{yP}, θ_{zP}}

$F = \{f_1, f_2, f_3, \ldots, f_{96}\}$

$n = 96$ and $m = 96$

The device seems to work fine according to the analysis of its degree of freedom. However, by running our software, we find that the interactions between artifacts in this device are not consistent and thus the device would not work. The set of inconsistent interactions are identified as {f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}, f_{18}, f_{19}, f_{20}, f_{46}, f_{47}, f_{48}, f_{49}, f_{50}, f_{51}, f_{52}, f_{53}, f_{54}, f_{55}, f_{56}, f_{57}, f_{58}, f_{59}, f_{60}, f_{76}, f_{77} f_{78},

$f_{79}, f_{80}, f_{81}, f_{94}, f_{95}, f_{96},\}$. These 44 interactions only involve 42 variables. Hence this design concept is not valid. We can also determine that the problem happens to artifact A, B, C, D, E, F, G, and Ground.

Now let us consider another design alternative called alternative B that is shown in Figure 4.16.

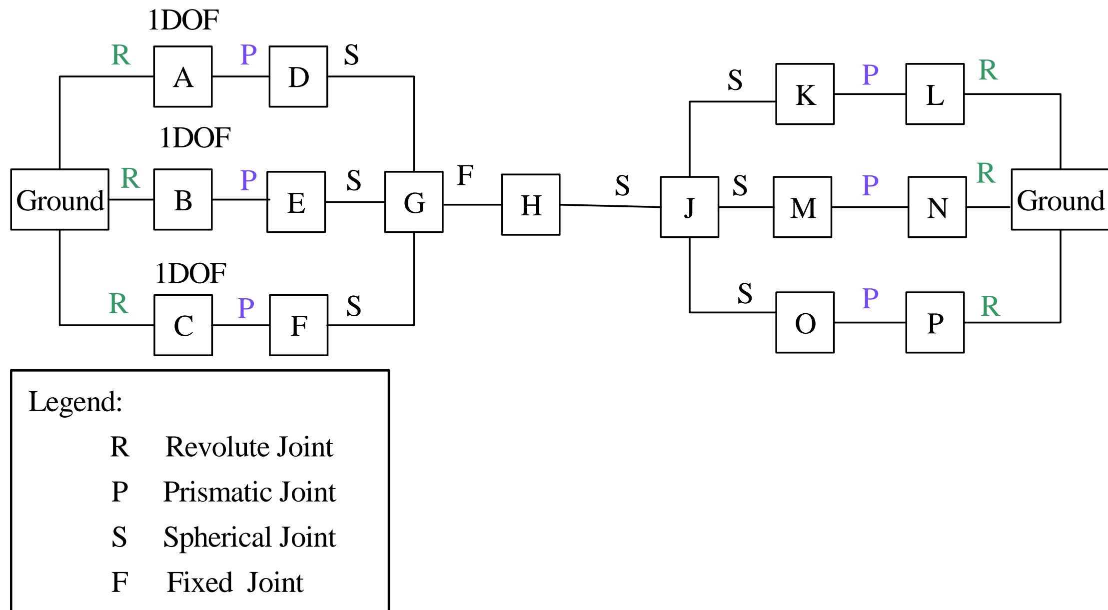


Figure 4.16: Design alternative B of a spatial mechanism

We remove artifact I and use artifact H to connect G and J. We also use a fixed joint between G and H, a spherical joint between H and J, a spherical joint between C and F. This alternative has 15 artifacts and 20 joints. Interactions in this design can be modeled as the following:

$f_1 = \{x_A, y_A, z_A, \theta_{xA}, \theta_{yA}, \theta_{zA}\}, f_2, f_3, f_4, f_5$, have the same qualitative structure as f_1.

$f_6 = \{x_B, y_B, z_B, \theta_{xB}, \theta_{yB}, \theta_{zB}\}, f_7, f_8, f_9, f_{10}$, have the same qualitative structure as f_6.

$f_{11} = \{x_C, y_C, z_C, \theta_{xC}, \theta_{yC}, \theta_{zC}\}, f_{12}, f_{13}, f_{14}, f_{15}$, have the same qualitative structure as f_{11}.

$f_{16} = \{x_P, y_P, z_P, \theta_{xP}, \theta_{yP}, \theta_{zP}\}, f_{17}, f_{18}, f_{19}, f_{20}$, have the same qualitative structure as f_{16}.

f_{21}= $\{x_N, y_N, z_N, \theta_{xN}, \theta_{yN}, \theta_{zN}\}$, $f_{22}, f_{23}, f_{24}, f_{25}$, have the same qualitative structure as f_{21}.

f_{26}= $\{x_L, y_L, z_L, \theta_{xL}, \theta_{yL}, \theta_{zL}\}$, $f_{27}, f_{28}, f_{29}, f_{30}$, have the same qualitative structure as f_{26}.

f_{31}= $\{x_A, y_A, z_A, \theta_{xA}, \theta_{yA}, \theta_{zA}, x_D, y_D, z_D, \theta_{xD}, \theta_{yD}, \theta_{zD}\}$, $f_{32}, f_{33}, f_{34}, f_{35}$ have the same qualitative structure as f_{31}.

f_{36}= $\{x_B, y_B, z_B, \theta_{xB}, \theta_{yB}, \theta_{zB}, x_E, y_E, z_E, \theta_{xE}, \theta_{yE}, \theta_{zE}\}$, $f_{37}, f_{38}, f_{39}, f_{40}$ have the same qualitative structure as f_{36}.

f_{41}= $\{x_C, y_C, z_C, \theta_{xC}, \theta_{yC}, \theta_{zC}, x_F, y_F, z_F, \theta_{xF}, \theta_{yF}, \theta_{zF}\}$, $f_{42}, f_{43}, f_{44}, f_{45}$ have the same qualitative structure as f_{41}.

f_{46}= $\{x_O, y_O, z_O, \theta_{xO}, \theta_{yO}, \theta_{zO}, x_P, y_P, z_P, \theta_{xP}, \theta_{yP}, \theta_{zP}\}$, $f_{47}, f_{48}, f_{49}, f_{50}$, have the same qualitative structure as f_{46}.

f_{51}= $\{x_K, y_K, z_K, \theta_{xK}, \theta_{yK}, \theta_{zK}, x_L, y_L, z_L, \theta_{xL}, \theta_{yL}, \theta_{zL}\}$, $f_{52}, f_{53}, f_{54}, f_{55}$, have the same qualitative structure as f_{51}.

f_{56}= $\{x_M, y_M, z_M, \theta_{xM}, \theta_{yM}, \theta_{zM}, x_N, y_N, z_N, \theta_{xN}, \theta_{yN}, \theta_{zN}\}$, $f_{57}, f_{58}, f_{59}, f_{60}$, have the same qualitative structure as f_{56}.

f_{61}= $\{x_F, y_F, z_F, \theta_{xF}, \theta_{yF}, \theta_{zF}, x_G, y_G, z_G, \theta_{xG}, \theta_{yG}, \theta_{zG}\}$, f_{62}, f_{63}, have the same qualitative structure as f_{61}.

f_{64}= $\{x_D, y_D, z_D, \theta_{xD}, \theta_{yD}, \theta_{zD}, x_G, y_G, z_G, \theta_{xG}, \theta_{yG}, \theta_{zG}\}$, f_{65}, f_{66}, have the same qualitative structure as f_{64}.

f_{67}= $\{x_E, y_E, z_E, \theta_{xE}, \theta_{yE}, \theta_{zE}, x_G, y_G, z_G, \theta_{xG}, \theta_{yG}, \theta_{zG}\}$, f_{68}, f_{69}, have the same qualitative structure as f_{67}.

For the fixed joint that connects G and H, we have six interactions:

f_{70}= $\{x_G, y_G, z_G, \theta_{xG}, \theta_{yG}, \theta_{zG}, x_H, y_H, z_H, \theta_{xH}, \theta_{yH}, \theta_{zH}\}$, $f_{71}, f_{72}, f_{73}, f_{74}, f_{75}$, have the same qualitative structure as f_{70}.

$f_{76}= \{x_H, y_H, z_H, \theta_{xH}, \theta_{yH}, \theta_{zH}, x_J, y_J, z_J, \theta_{xJ}, \theta_{yJ}, \theta_{zJ}\}$, f_{77}, f_{78}, have the same qualitative structure as f_{76}.

$f_{79}= \{x_J, y_J, z_J, \theta_{xJ}, \theta_{yJ}, \theta_{zJ}, x_O, y_O, z_O, \theta_{xO}, \theta_{yO}, \theta_{zO}\}$, f_{80}, f_{81}, have the same qualitative structure as f_{79}.

$f_{82}= \{x_J, y_J, z_J, \theta_{xJ}, \theta_{yJ}, \theta_{zJ}, x_M, y_M, z_M, \theta_{xM}, \theta_{yM}, \theta_{zM}\}$, f_{83}, f_{84}, have the same qualitative structure as f_{82}.

$f_{85}= \{x_J, y_J, z_J, \theta_{xJ}, \theta_{yJ}, \theta_{zJ}, x_K, y_K, z_K, \theta_{xK}, \theta_{yK}, \theta_{zK}\}$, f_{86}, f_{87}, have the same qualitative structure as f_{85}.

We still want to have at least three degrees of freedom:

$f_{88}= \{x_A, y_A, z_A, \theta_{xA}, \theta_{yA}, \theta_{zA}\}$, $f_{89}= \{x_B, y_B, z_B, \theta_{xB}, \theta_{yB}, \theta_{zB}\}$, $f_{90}= \{x_C, y_C, z_C, \theta_{xC}, \theta_{yC}, \theta_{zC}\}$.

Now we have n =90 and m = 90. By running our software, we get the following result:

The size of the minimum s-t cut is 90 = m, thus the interactions are consistent.

This example illustrates that in complex spatial mechanisms, although overall the degree of freedom of the device looks fine, there could be a part of the mechanism that is over-constrained. In this case, it would be very difficult to tell which part of the mechanism is causing the problem. However, our algorithm could solve this problem by finding out the inconsistent interactions.

4.5 Summary

This chapter presents a systematic approach to check the consistency of a set of interactions in an interaction-state of a mechatronic system. We also provide an algorithm to find the set of interactions that cause the inconsistency. During the

conceptual design stage, the actual equations describing the interactions are usually not known. Therefore our algorithm only utilizes the information on participating parameters to carry out its analysis. We have shown both the soundness and completeness of our algorithms. This implies that when our algorithm finds a set of interactions to be inconsistent, they are actually inconsistent. Furthermore, when our algorithm finds a set of interactions to be consistent, they are actually consistent. Even though the consistency-checking problem has an appearance of a combinatorial problem, we have found an algorithm that works in polynomial time and does not require exhaustive enumeration.

The algorithms described in this chapter present a step towards automated validation of a proposed design concept. We believe that the framework described in this chapter will provide the underlying foundations for constructing the next generation software tools for the conceptual design of complex mechatronic systems.

Chapter 5: Detection Of Unsafe Parameter Value Sets Embedded In Interaction-States

This chapter defines the problem of detecting the presence of an unsafe parameter value set inside an interaction-state and presents an algorithm for solving the problem.

This chapter has been organized in the following manner. Section 5.1 describes the problem formulation. Section 5.2 describes the algorithm for detecting the presence of an unsafe parameter value set inside an interaction-state. Section 5.3 presents examples illustrating how the algorithm works. Finally, Section 5.4 presents concluding remarks.

5.1 Problem Formulation

5.1.1 Problem Statement

Let X be the set of parameters belonging to the artifacts in an interaction-state s. Let u be an unsafe parameter value set involving parameters from a subset X' of X. We are concerned whether parameters in X' would take the unsafe values defined in u at some time during the interaction-state. Incoming transitions and outgoing transitions influence whether this can happen. So we need to consider the transitions associated with the state as well.

For example, consider the behavior specification of the AVC example as shown in Figure 3.11. The event space and the unsafe parameter value sets are shown in Table 3.3 and 3.4 respectively for this example. Let us consider the vacuum state. Detailed

description of this state is shown in Figure 3.15. There are two active artifacts in this state: AVC and surface.

We use the following notations to represent the main parameters participating in interactions in this interaction-state:

$x_1 = AVC :: RemainingCapacity$

$x_2 = AVC :: RemainingEnergy$

One of the unsafe parameter value sets is defined as (x_1 = 2%). There are two incoming transitions to the vacuum state: r_5 and r_8. There are two outgoing transitions from this state: r_6 and r_7. If any of the two incoming transitions lead to initial values in the state such that during the state the value of x_1 is equal to 2% during this state, then the state is considered unsafe.

The current framework for checking state safety only deals with discrete parameter value sets. If the unsafe parameter value set involves only one parameter, then we can incorporate value range on the parameter by treating it as two different unsafe discrete values at the upper and lower limits. We formulate the problem of detecting the presence of unsafe parameter value set inside an interaction-state in the following manner.

Given,

- Unsafe parameter value set u identifying a set of parameters X' and defining their values.
- Interaction-state s.
- Set of incoming transitions R^i.
- Set of outgoing transitions R^o.

The problem is to determine if there exists a local time t for s such that parameters in set X' will take values defined in u.

5.1.2 Overview of Our Approach

To find out whether the unsafe parameter value set is embedded in the interaction-state at a valid local time, we need to consider possible values of parameters in X' during the lifetime of the interaction-state. Generally, values of parameters are determined by their initial values and the equations that control the change of values. Parameters may get their initial values by inheriting values from a previous interaction-state. Since current interaction-state may be reached through different paths from different interaction-states, the initial values of parameters may vary according to the history of the state. To enumerate every possible path of reaching the current interaction-state is computationally inefficient.

We believe that for an overall safe device design, we should be able to ensure that an interaction state will be safe irrespective of its history. This obviously leads to a more conservative design. If an interaction-state is deemed to be safe irrespective of which transitions led this state, then there is no possible way for this state to contain an unsafe parameter value set. However, if we discover that it is possible to have initial conditions in the state such that it includes unsafe parameter value set, then it may still not be unsafe. The reason for this is as following. The initial conditions in an interaction state actually depend on the set of transitions by which the interaction-state is reached. So while it may theoretically be possible to initialize the state with the conditions that lead to an unsafe parameter value set, it may not be feasible to initialize the state with those values given the set of transitions and other states in the

system. Considering all possible state histories is computationally almost intractable. Hence, we advocate developing a conservative approach that ensures that states are safe irrespective of their initialization history.

We use the following three-step approach:

- Determine possible initial values from the initialization of parameters in the interaction-state for each incoming transition. Incoming transitions may override the initial values of parameters set by default initialization or inherited from the previous state.
- Analyze the equations that govern the interactions in the interaction-state. Parameters change values according to these interactions.
- Check the influence of outgoing transitions. Even though parameters may have potential to reach the unsafe values at some time, an outgoing transition may transit the state to the next state before the unsafe values are reached.

These steps are described in detail in section 5.2.

5.2 Algorithm for Detecting the Presence of Unsafe Parameter Value Sets

In an interaction state, a state parameter acquires its values by its initialization condition and interacting with other parameters. Initialization types and value-changing modes are used to describe the characteristics of how parameters are initialized and changed. Initialization types are defined as *ASSIGN*, *INHERIT* and *DERIVE*. Value-changing modes are defined as *CONSTANT*, *EQUATION* and *DERIVE*. These are described in detail in Chapter 3. There are some limitations for combining initialization types and value-changing modes as shown in Table 5.1. Since some parameters are set to inherit values from previous state and we don't

consider the state history, their initial values are unknown in this interaction-state. Parameters with derived initial values from other parameters via interactions can be finally determined as having known initial values or unknown initial values according to the following rules:

- If all the parameters from which the value is derived only have known values, then this parameter will have known values. Furthermore, if all the parameters only have known constant values during the interaction-state, then this parameter will have known constant values.
- If any source parameter gets its value from a previous state, then the derived parameter has an unknown value.

The condition that the unsafe parameter value set is embedded in the interaction-state can be classified into several cases described below.

Case 1: If the parameters in X' only take known constant values in s, we can simply compare these values with the unsafe values and determine whether s is unsafe. This can be formulated as following:

Let u be the unsafe parameter value set, s be the interaction-state. If $u \subseteq s(t=0)$ then s is unsafe.

For example, let $X = \{x_1, x_2, x_3, x_4\}$, $u = \{(x_1=5), (x_2=10)\}$, and $s(t=0) = \{(x_1=5), (x_2=10), (x_3=10), (x_4=10)\}$. In this example, $u \subseteq s(t=0)$, therefore s is unsafe.

Case 2: Based on the initial values of the state parameters and the interaction equations, we can determine the values of the parameters at any time during the interaction-state. Then the interaction-state is considered unsafe if at some time, all

the parameters in X' reach their unsafe values simultaneously. This can be formulated as following:

Let u be the unsafe parameter value set, s be the interaction-state. The equations in s are represented as $f_j(X(0), X, t) = 0$, $1 \leq j \leq m$, where $X(0)$ is the set of initial values of X. Then, if there exists t^* such that $u \subseteq s^*$ (where $s^* = s(t=t^*)$), then s can potentially be unsafe if an outgoing transition does not transit the state before time t^*. Otherwise, the state is considered safe. In general, some initial values in the state may be inherited from other states. These initial values can be treated as unknown variables and the system of equations can be solved to determine if there exist initial value assignments that can make the state unsafe.

The basic idea behind determining if a transition will take place before state reaching the unsafe value is as following. Let u be an unsafe parameter value set and s be an interaction-state. Let $X(0)$ be the possible initial values of various parameters in the interaction-state. Let t^* be the time at which parameters in the state reach unsafe values. Let these values be represented by $X(t^*)$. Let there exist an outgoing transition r such that the condition associated with the transition defines a hyper-plane over the values of parameters in state. If $X(0)$ and $X(t^*)$ lie on two different sides of the hyper-plane, then any state that starts with value $X(0)$ will transition to a different state before actually resulting in unsafe values. Therefore this state will be a safe state.

For example, let $X = \{x_1, x_2, x_3, x_4\}$, $u = \{(x_1=5), (x_2=10)\}$, and $X(0) = \{0, 0, 0, 0\}$. Let us assume that the interaction equations in s are $\{(x_1(\mathrm{t}) = x_1(0) + t), (x_2(\mathrm{t}) = x_2(0) + 2x_1)\}$ and an outgoing transition has the condition represented as $(x_1 + x_2 = 6)$. In this case when t is equal to 5, various parameters in s will reach unsafe value defined in u.

However, the outgoing transition happens at $t = 2$, thus s can never actually reach the unsafe value set. Therefore s is safe.

Case 3: If the interaction equations are unknown, then we cannot compute the exact values of the parameters. However, if qualitative structures of the equations are known, we can examine the structure of the equations to determine if by the nature of the equations, the possibility of reaching the unsafe states can be eliminated. In this analysis, we assume that the interaction equations in the state remain irredundant at the unsafe parameter values.

Consider the following example: Let s involve the following interaction equations of known structure and unknown form: $f_1(x_1, x_2), f_2(x_1, x_2, x_3), f_3(x_2, x_3, x_4)$. Let $u = \{(x_2 = 0), (x_3 = 0)\}$. Therefore at u, the state equation structure will become $f_1'(x_1), f_2'(x_1), f_3'(x_4)$, where f_1', f_2', and f_3' have been obtained by substituting values in u. In order for state s to reach u, there need to exist a solution to these equations. A solution to these equations will only exist if $f_1' = f_2'$. In other words, there is at least a redundant equation. As long as f_1 and f_2 have a structure such that substituting $x_2 = 0$ and $x_3 = 0$ in them does not produce an identical equation, s can never reach unsafe value.

The mathematical basis for the analysis in this case is given by Theorem 5.1 described below.

Let F be the set of parameter sets participating in interaction equations in state s. For every $f \in F, f \subseteq Z$, where $Z = X \cup Y \cup \{t\}$. X is the set of parameters in s. Y is the set of auxiliary variables corresponding to the unknown initial conditions in s.

Create F' by eliminating parameters corresponding to X' and X'' from F. X'' is the set of parameters that have known constant value in s. If this leads to an empty member in F', then remove that member from F'.

Theorem 5.1. If there exists a set $F'' \subseteq F'$ such that cardinality(F'') > cardinality($\cup F''$), then s can reach unsafe values defined in u, if at least one of the equations in F'' is redundant at u.

Proof:

In order for variables in s to reach values specified in u, all equations corresponding to F'' will have to be simultaneously satisfied. cardinality(F'') > cardinality($\cup F''$). Therefore, equations corresponding to F can be satisfied at u, if at least one equation associated with F'' is redundant at u. If this is not the case, then s will be safe.

Depending on whether the equations are known or unknown, different algorithms may be applied. The algorithm for solving the problem with the known equations is the following:

Algorithm CHECKSTATESAFETYWITHKNOWNEUQATIONS

Input:

- System of interactions K defined over X. There are n variables in X and m interactions in K.
- Incoming transition r^i.
- Set of outgoing transitions R^o.
- Unsafe parameter value set u involving parameter set X'.

Output:

- State safety status: *SAFE* or *UNSAFE*

Steps:

1) Initialize parameters. If the initialization type is *ASSIGN*, then assign the initial value to the parameter. If initialization type is *INHERIT*, let the parameter value be unknown. If initialization type is *DERIVE*, initialize the independent parameters first. Then use the interaction equations to compute the dependent parameter values. If there is at least one inherited parameter among the independent parameters, then the dependent parameter value is marked as unknown.

2) Override the parameter initialization using the initialization action set in the incoming transition r^i.

3) Find the set of parameters that have known constant value X'' in s.

4) If $X'' \subseteq X'$, then for every parameter $p \in X''$, check if unsafe value of p in u matches with value of p in s. Use the following conditions to determine the sate safety status.

    i. If value of at least one parameter does not match, then the state is safe. Return state safety status as *SAFE* and exit.

    ii. If all values of all parameters match and $X'' = X'$, then the state is unsafe. Return state safety status as *UNSAFE* and exit.

    iii. If all values of all parameters match, $X'' \neq X'$, and all parameters in $X' - X''$ have unknown constant values, then the state is potentially unsafe. Return state safety status as *UNSAFE* and exit.

    iv. If $X'' = \varnothing$ and all parameters in $X' - X''$ have unknown constant values, then the state is potentially unsafe. Return state safety status as *UNSAFE* and exit.

5) Let L be the set of state equations defined in terms of the sets of state parameters X and auxiliary variables Y. Y is the set of auxiliary variables that correspond to the unknown initial conditions. If all initial conditions are known, then Y will be empty. The state equations will be represented in the following form: $L(X, Y, t) = \{l_1(X, Y, t) = 0, \ldots, l_m(X, Y, t) = 0\}$. Substitute unsafe values of parameters from X''in L. This reduces L to $L'(Z, Y, t)$ where $Z = X - X''$. Substitute value of parameters from $X' - X''$ in L. This reduces L' to $L''(Z', Y, t)$ where $Z' = (X - X'') - X'$. Solve the above equations and compute values of variables Z', Y and t such that $0 \leq t < \infty$. If such a solution does not exist then the state is considered safe. Return state safety status as *SAFE* and exit.

6) If a solution has been found in Step 5, then for every outgoing transition r^o, do the following:

   i. If the outgoing transition is of the following form: $l_t(X_t)=0$, $X_t \supseteq Z$ and $X' \supseteq X_t$, then do the following:

      - Substitute value of parameters from u into l_t, let value of l_t be l_1.
      - Substitute values of parameters from the initial values of state (either known as a part of the state definition or taken from set Y computed as a part of Step 5) into l_t, let value of l_t be l_2.
      - If l_1 and l_2 have different signs, then the state is safe. In this case initial values for the state and unsafe values for the state are on different sides of hyper plane defined by the outgoing transition. Hence it's impossible to go from initial value to unsafe value. Return state safety status as *SAFE* and exit.

7) Return state safety status as *UNSAFE* and exit.

If multiple solutions are found is Step 5, then we need to check all the solutions as a part of Step 6. If all solutions cannot be checked then one should directly proceed to Step 7. Moreover, the above algorithm needs to be used on all incoming transitions associated with the state.

Many different types of mathematical techniques can be used in Step 5 to solve a given set of equations. Usually the choice of technique being used will depend upon the nature of equations. Since the execution of Step 5 depends on the mathematical techniques that are used to solve the equations, the complexity of the algorithm is difficult to estimate.

Whenever the algorithm CHECKSTATESAFETYWITHKNOWNEUQATIONS returns state safety status as *UNSAFE*, a caution should be exercised in interpreting these results. This result in most case only implies that this state has a potential of reaching unsafe values. Whether or not the state will actually reach unsafe value will depend upon the state history.

The algorithm for solving the problem with the unknown equations is the following:

Algorithm CHECKSTATESAFETYWITHUNKNOWNEUQATIONS

Input:

- Set of parameter sets F for interactions defined over X. There are n variables in X and m parameter sets in F.
- Incoming transition r^i.
- Unsafe parameter value set u involving parameter set X'.

Output:

- State safety status: *SAFE*, *UNSAFE*, or *UNKNOWN*

Steps:

Step 1, 2, 3, and 4 are identical to the ones used in algorithm CHECKSTATESAFETYWITHKNOWNEUQATIONS.

Step 5: Find $V = X \cup Y \cup \{t\}$. Y is the set of auxiliary variables corresponding to the unknown initial conditions in s.

Create F' by eliminating parameters corresponding to X' and X'' from F. X'' is the set of parameters that have known constant value in s. If this leads to an empty member in F', then remove that member from F'. Let $V' = (V - X'') - X'$. Let the cardinality of V' be n' and the cardinality of F' be m'.

Call CONSTRUCTINTERACTIONNETWORK using F' and V' as inputs to create a flow network (this algorithm is defined in Chapter 4).

Call FINDMINIMUMSTCUTSIZE on the flow network created in the previous step (this algorithm is also defined in Chapter 4).

If a min-cut has been found such that the size of the min-cut is less than m', then s is considered safe. Return state safety status as *SAFE* and exit.

Step 6: Return state safety status as *UNKNOWN*.

The above algorithm can return three results. If the algorithm returns state safety status as *SAFE*, then it means that unless one chooses redundant state equations, the state will be safe. If the algorithm returns state safety status as *UNSAFE*, then it means that there is possibility of the state parameters reaching the unsafe value depending upon how the state is initialized. If the algorithm returns state safety status as *UNKOWN*, then we cannot reach any conclusions based on the structure of the

equations. Hence the user should run algorithm CHECKSTATESAFETYWITHKNOWNEUQATIONS when the forms of the equations are known.

5.3 Examples

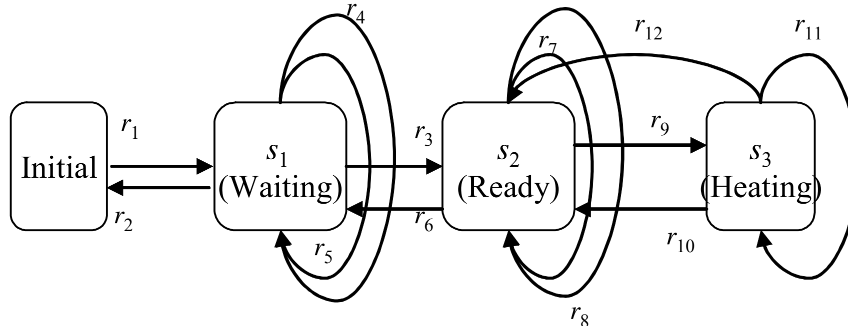
The algorithm described above can be illustrated by the following examples.

First, let us consider the microwave oven design. The behavior specification of the microwave oven is shown in Figure 5.1. Unsafe parameter value set is defined as (*Microwave*::*DoorStatus*=*OPEN* AND *Microwave*::*Heater* = *ON*). We are concerned about whether this unsafe parameter value set is embedded in the interaction-state of heating. Description of the Heating state is shown in Figure 5.2.

Now we follow the algorithm described above to check the safety of the "heating" state. First, we initialize the state parameters. As all of them are inheriting values from the previous state, their values are unknown at this time. Then in step 2, we identify the incoming transitions. Let us take r_9 as the example. This transition will set the value of *Microwave*::*Heater* to *ON*. In step 4, the only parameter with known constant value is *Microwave*::*Heater* and its value matches the unsafe value. The other parameter *Microwave*::*DoorStatus* has unknown constant value depending on the previous state. Thus we consider the unsafe parameter value set embedded in the heating state.

The second example comes from the AVC example described in Chapter 3. The behavior specification of AVC is shown in Figure 3.11. The unsafe parameter value set is defined as (*AVC*::*RemainingCapacity* ≤ 2%). We are concerned whether this

unsafe parameter value set is embedded in the interaction-state of vacuum. Description of the vacuum state is shown in Figure 3.15.

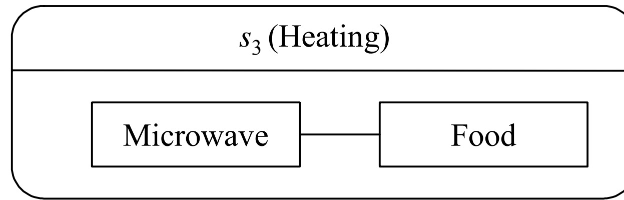


Transition list in the microwave behavior specification

Name	Closure Action
r_1	*Microwave*::*Power* = *ON*
r_2	*Microwave*::*Power* = *OFF*
r_3	*Food*::*InMicrowave* = *TRUE*
r_4	*Microwave*::*DoorClosed* = *TRUE*
r_5	*Microwave*::*DoorClosed* = *FALSE*
r_6	*Food*:: *InMicrowave* = *FALSE*
r_7	*Microwave*::*DoorClosed* = *TRUE*
r_8	*Microwave*::*DoorClosed* = *FALSE*
r_9	*Microwave*::*HeaterStatus* = *ON*
r_{10}	*Microwave*::*HeaterStatus* = *OFF*
r_{11}	*Microwave*::*DoorClosed* = *TRUE*
r_{12}	*Microwave*::*DoorClosed* = *FALSE*

Figure 5.1: Transition diagrams for behavior specification of microwave

Artifact Set and Interaction Topology



Artifact Interaction Set

Food::*Temperature* = *f*(*Microwave*::*HeatingPower*, *t*)

Parameters Initialization and Change

parameter	Initialization Type	Initialization Value	Change Type	Equation
Microwave:: *HeaterStatus*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Microwave:: *HeaterPower*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Microwave:: *DoorClosed*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Microwave:: *Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*

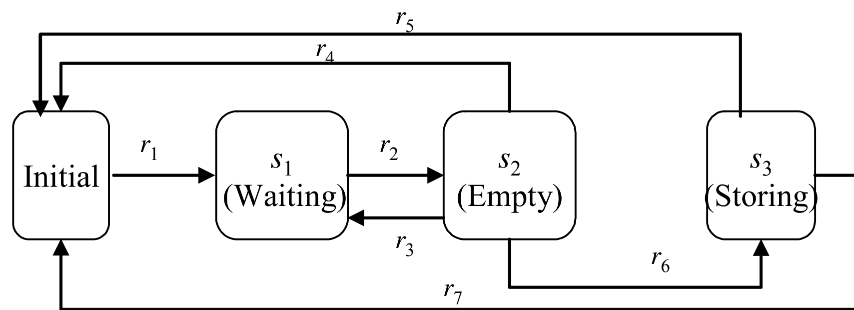
Figure 5.2: Definition of state s_3

Now we follow the algorithm described above to check the safety of the "vacuum" state. First, we initialize the state parameters. As all of them except *AVC*::*Speed* are inheriting values from the previous state, their values are unknown at this time. Then in step 2 we identify the incoming transitions as r_5 and r_8. However, these two transitions do not influence the initialization of the state variables. We also know the equation that describes the change of the parameter in the unsafe parameter value set as:

AVC::*RemainingCapacity*(*t*)= *AVC*::*RemainingCapacity*(*t*=0) – *t*/20000

In step 4, although we don't know the exact initial value of *AVC*::*RemainingCapacity*, we know there exist an initial value and a time t such that the value of *AVC*::*RemainingCapacity* would be below 2%. In step 6, the outgoing transitions are r_6 and r_7. They have no influence on state safety. Thus this state is considered unsafe.

The third example is a device for storing and draining liquid. The behavior specification of this device is shown in Figure 5.3. Unsafe parameter value set is defined as (*Reservoir*::*RemainingCapacity* = 10). We are concerned whether this unsafe parameter value set is embedded in the interaction-state of Empty. Description of the Empty state is shown in Figure 5.4.

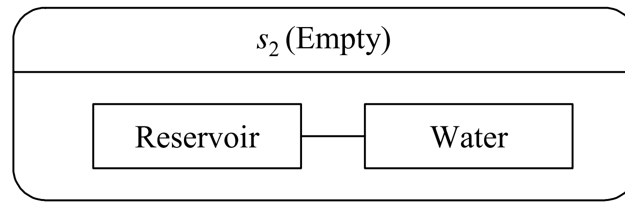


Transition list in the Reservoir behavior specification

Name	Closure Action
r_1	*Reservoir*::*Power* = *ON*
r_2	*Reservoir*::*Drain* = *TRUE*
r_3	*Reservoir*::*Drain* = *FALSE*
r_4	*Reservoir*::*Power* = *OFF*
r_5	*Reservoir*::*RemainingCapacity* = 100
r_6	*Reservoir*::*RemainingCapacity* = 20
r_7	*Reservoir*::*Power* = *OFF*

Figure 5.3: Transition diagrams for behavior specification of reservoir

Artifact Set and Interaction Topology



Artifact Interaction Set

Parameters Initialization and Change

parameter	Initialization Type	Initialization Value	Change Type	Equation
Reservoir::*RemainingCapacity*	*INHERIT*	*NA*	*EQUATION*	*Reservoir*::*RemainingCapacity*(t) = *Reservoir*::*RemainingCapacity*(t=0) – 5t
Reservoir::*Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
Reservoir::*Drain*	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 5.4: Definition of state s_2

Now we follow the algorithm described above to check the safety of the Empty state. First, we initialize the state parameters. As all of them are inheriting values from the previous state, their values are unknown at this time. Then in step 2, we identify the incoming transition as r_2. r_2 does not influence the initialization of the state variables. We also know the equation that describes the change of the parameter in the unsafe parameter value set as:

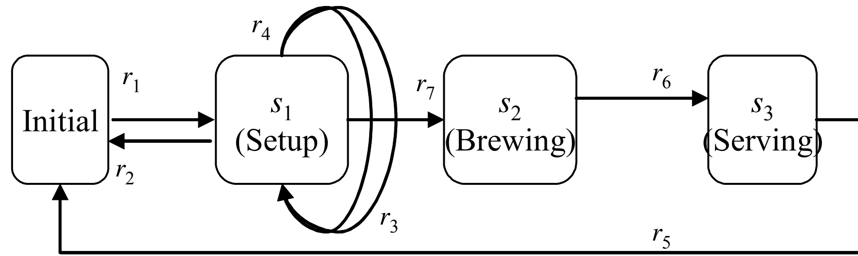
Reservoir::*RemainingCapacity*(t) = *Reservoir*::*RemainingCapacity*(t=0) – 5t

In step 4, although we don't know the exact initial value of *Reservoir*::*RemainingCapacity*, we know there exist an initial value and a time t such that the value of *Reservoir*::*RemainingCapacity* would be below 10. Thus the state is potentially unsafe. However, in step 6, the outgoing transition r_6 will transit the state

to state s_3 before Reservoir::Remaining Capacity decreases to 20. The transition condition is:

f_t: *Reservoir*::*RemainingCapacity* = 20. That is: f_t: *Reservoir*::*RemainingCapacity* – 20 = 0. We can find an initial value such that l_2 = *Reservoir*::*RemainingCapacity*(t=0) – 20 > 0. If we substitute unsafe value of parameters into f_t, we get l_1 =10 – 20 < 0. Since l_1 and l_2 have different signs, we conclude that this interaction-state is safe.

The fourth example is a coffee maker. The behavior specification of the coffee maker is shown in Figure 5.5. Unsafe parameter value set is defined as (*CoffeeMaker*::*PotPresent* = *FALSE* AND *CoffeeMaker*::::*Brewer* = *ON*). We are concerned whether this unsafe parameter value set is embedded in the interaction-state of Brewing. Description of the Brewing state is shown in Figure 5.6.

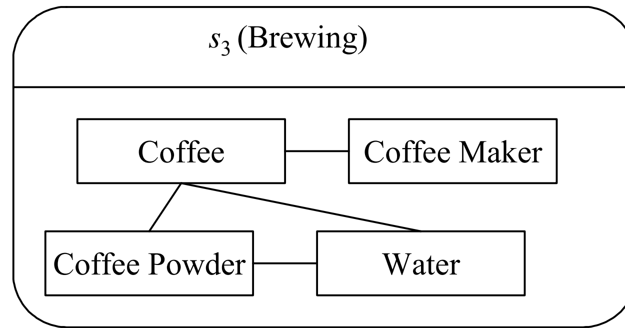


Transition list in the coffee maker behavior specification

Name	Closure Action
r_1	*CoffeeMaker*::*Power* = *ON*
r_2	*CoffeeMaker*::*Power* = *OFF*
r_3	*CoffeePowder*::*InCoffeeMaker* = *TRUE*
r_4	*CoffeeMaker*::*PotPresent* = *TRUE*
r_5	*CoffeeMaker*::*Power* = *OFF*
r_6	*CoffeeMaker*::*Brewer* = *OFF*
r_7	*CoffeeMaker*::*Brewer* = *ON*

Figure 5.5: Transition diagrams for behavior specification of coffee maker

Artifact Set and Interaction Topology



Artifact Interaction Set

Coffee::*Temperature*(*t*) = *f*(CoffeeMaker::*HeatPower, t*)
Coffee::*Weight*(*t*) = *Coffee*::*Weight*(*t*)/30 + *Water*::*Weight*(*t*)

Parameters Initialization and Change

parameter	Initialization Type	Initialization Value	Change Type	Equation
CoffeeMaker:: *Heater*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
CoffeeMaker:: *HeaterPower*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
CoffeeMaker:: *PotPresent*	*INHERIT*	*NA*	*CONSTANT*	*NONE*
CoffeeMaker:: *Power*	*INHERIT*	*NA*	*CONSTANT*	*NONE*

Figure 5.6: Definition of state s_3

Now we follow the algorithm described above to check the safety of the “brewing” state. First, we initialize the state parameters. As all of them are inheriting values from a previous state, their values are unknown at this time. Then in step 2, we identify the incoming transitions. Let us take r_7 as the example. This transition will set the value of *CoffeeMaker*::*Brewer* to *ON*. In step 3, the only parameter with known constant value is *CoffeeMaker*::*Brewer* and its value matches the unsafe value.

The other parameter *CoffeeMaker*::*PotPresent* has unknown constant value depending on the previous state. Thus we consider the unsafe parameter value set embedded in the Brewing state.

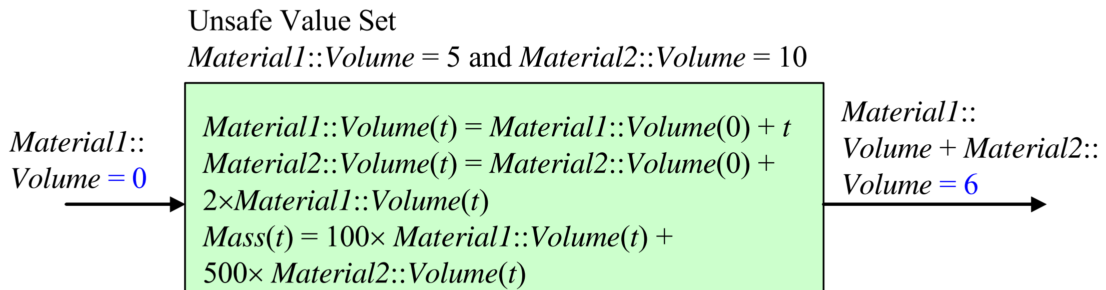
The fifth example is a mixer used in the manufacturing of composite materials. This machines works by mixing different types of materials and casting them into desired parts. We only show the mixing state here. In this state, two nozzles are used to lead in two different materials. A controller is used to control the total mass of the materials according to requirements. Figure 5.7 shows the mixing state. Unsafe parameter value set is defined as (*Material1*::*Volume* = 5 and *Material2*::*Volume* = 10). We are concerned whether this value set is embedded in the mixing state. In this state, parameters have zero initial values. If we simply look at the interactions and parameter initialization, we would arrive at the conclusion that this state is unsafe. However, if we consider the outgoing transitions, state in figure 5.7(a) will be safe because the state is exited before unsafe values are reached. On the other hand state in figure 5.7(b) will reach unsafe values.

5.4 Summary

This chapter presents a systematic approach to check whether a predefined unsafe parameter value set is embedded in an interaction-state. The conceptual design stage lacks complete design details, hence we analyze different cases in which unsafe parameter value sets can be embedded in an interaction state and provide an algorithm to determine whether the interaction-state is safe based on these cases. This algorithm is not dependent on the state history. Hence, it can be applied to each interaction-state independently. During the conceptual design stage, the actual

equations describing the interactions may not be known. Therefore we present algorithms for handling both cases when interaction equations are known and when they are not known. We have shown that our algorithms are conservative in nature.

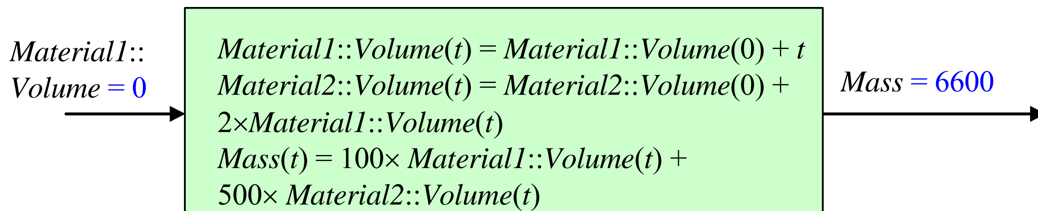
We believe that the framework described in this chapter will provide the underlying foundations for constructing the next generation software tools for conceptual design of complex mechatronic systems.



• This state is safe

• Values *Material1*::Volume = 5 and Material2::Volume = 10 are reached at $t = 5$, but transition happens before that time (at $t = 2$)

(a) Unsafe value set not embedded in state



• This state will reaches unsafe values

• Values *Material1*::*Volume* = 5 and *Material2*::*Volume* = 10 are reached at $t = 5$

• Outgoing transition will be reached at $t = 6$

(b) Unsafe value set embedded in state

Figure 5.7: Mixing state

Chapter 6: Design Concept Evaluation

Given a behavior specification, designers could generate many design concepts. An important design step is to evaluate these design concepts and select the best concepts and develop them into detailed designs. This chapter describes methods for performing new types of evaluations that are facilitated by the modeling framework described in Chapter 3. Specifically, it discusses two different types of evaluations that can be performed: determination of maximum power consumption and determination of optimal component sharing.

This chapter has been organized in the following manner. Section 6.1 describes the optimal component-sharing problem. It first describes the optimal component-sharing problem and shows that this problem is NP-hard. It also presents a branch and bound algorithm for solving this problem. Section 6.2 describes the maximum power consumption problem and presents an algorithm for solving it. Finally, Section 6.3 summarizes this chapter.

6.1 Optimal Component Sharing

6.1.1 Problem Statement

Artifacts in design concepts will be realized by selecting components from the component library to implement the design concept. For examples, actuator artifacts will be mapped to suitable physical motors. Consider a situation in which a design concept needs two different actuators-- one for elevating a platform and one for tilting the platform. Now assume that these two actuators are used in two different states and hence never need to be used simultaneously. In such a situation, one might consider

the possibility of using a single physical motor that can play the role of elevating the platform in one state and tilting the platform in the other state. In this case we will say that the two artifacts in the design concept are sharing the physical component motor.

Component sharing becomes an important design strategy in applications where weight or space is very tight. In such situations, a design concept that maximizes number of sharable components may be preferred over the design that does not allow sharing components. Examples of such applications include medical devices used in minimally invasive surgery and satellites. In both of these applications it becomes necessary to use a single actuator or sensor to play multiple different roles.

In order for a component to play multiple roles, it typically needs to be disconnected from one component and be connected to some other component. This in turn makes the connector a lot more complex because they need to incorporate elaborate switching mechanisms. If the switching mechanism becomes too complex, then it defeats the purpose of sharing components. Hence a tradeoff needs to be made between sharing components and deploying complex switching mechanisms.

State transition diagrams carry the information about the artifacts that are not being used simultaneously. Hence they enable us to determine which artifacts can share physical components. It is difficult to assess the actual complexity of switching mechanism during the conceptual design. Therefore, in formulating optimal component sharing problem, we do not explicitly consider the switching mechanism complexity. We instead account for it implicitly by requiring that if a component has been selected to play the role of an artifact, then it should play the role of that artifact in every state. This restriction ensures that the same artifact is not being realized by

different components in different states and hence unnecessarily increases the number of switching mechanisms.

We use the following notation to describe this problem. Let S be the set of interaction-states. Let A be the set of artifacts used in the design concept. Each member of A describes an artifact and its type. Each member of S can be viewed as a subset of A. Let T be the set of artifact types used in the design concept. The optimal component sharing problem can be formulated as the following:

Given:

- $T = \{t_1, \ldots, t_l\}$
- $A = \{(a_1, t(a_1)), \ldots, (a_m, t(a_m))\}$, where $t(a_i) \in T$
- $S = \{s_1, \ldots, s_n\}$, where $s_i \subseteq A$

We are interested in finding a set

$B = \{B_1, \ldots, B_b\}$ satisfying:

- $B_i \subseteq A$, such that every member of B_i has the same type.
- Cardinality of B is minimum.
- For each $s \in S$, B_i has at most one element common with s.
- No two elements of B intersect with each other.

Basically, every member of set B represents a set of artifacts that can be realized by the same physical component.

Two elements with different types cannot be shared. Hence we need to solve this problem for each artifact type separately. Therefore, by eliminating the type we can significantly simplify this problem. The simplified problem can be stated as following:

Given:

- $A = \{a_1, \ldots, a_m\}$
- $S = \{s_1, \ldots, s_n\}$, where $s_i \subseteq A$

We are interested in finding a set

$B = \{B_1, \ldots, B_b\}$ satisfying:

- $B_i \subseteq A$.
- Cardinality of B is minimum.
- For each $s \in S$, B_i has at most one element common with s.
- No two elements of B intersect with each other.

Let us consider the following example.

We are given:

- $A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$
- $S = \{s_1, s_2, s_3, s_4\} = \{\{a_1, a_2, a_4, a_6\}, \{a_3, a_5, a_7\}, \{a_1, a_5, a_7\}, \{a_1, a_3, a_6\}\}$

In this example, seven artifacts of the same type are used. However, some of them can share physical components since they are not used in the same state. For example, a_2 can share a physical component with a_3. a_4 can share a physical component with a_5. a_6 can share a physical component with a_7. A possible solution for component-sharing is $B = \{\{a_1\}, \{a_2, a_3\}, \{a_4, a_5\}, \{a_6, a_7\}\}$. Then we only need four physical components to realize seven artifacts in the design concept. This is enabled by the fact that all artifacts are not being used simultaneously.

6.1.2 Complexity Analysis of Optimal Component Sharing Problem

Before attempting to develop an algorithm for this problem, we will first analyze the complexity of the problem.

We want to prove that the optimal component sharing problem is NP-hard and we base this assertion by comparing it to the graph coloring problem.

Graph Coloring Problem (GCP) is defined as following:

Input: An undirected graph G.

Problem: Assign colors to vertices of the graph such that adjacent vertices are not assigned the same color and the number of colors is minimized.

Theorem 6.1. Optimal Component Sharing Problem is NP-Hard.

Proof: To prove a problem C is NP-Hard we must show that it is at least as hard as a known NP-Hard problem, say D. Specifically this requires,

1. A reduction, i.e., an algorithm to turn any instance of D into an instance of C.
2. An argument that the reduction takes only polynomial-time.
3. An argument that the reduction works, i.e., answer to the instance of C can be used to create the answer for the instance of D.

We shall show that there is a natural reduction from the graph coloring problem to the optimal component sharing problem. It is well known that the graph coloring problem is NP-Hard [Corm90].

Given any instance of graph coloring, we construct an instance of OCSP by the following transformation. For every vertex $v \in V$ in the graph, insert an element a into A. For every edge $e = (v_1, v_2) \in E$ in the graph, insert an element $s = \{a_1, a_2\}$ into S. Here a_1 and a_2 are corresponding elements to v_1 and v_2. This transformation can be realized for any instance of graph coloring and it is done in linear time with respect to the size of the graph. Therefore requirements 1 and 2 have been met.

Solution to OCSP can be mapped to graph coloring solution in the following manner (an illustration is shown in Figure 6.1). For each group of sharable components, generate a distinct color. Vertices in the graph that correspond to the components assigned to the same group are marked with the corresponding color. This mapping ensures that no two adjacent vertices have the same color. Because an element has been inserted into set *S* for every pair of adjacent vertices, this ensures that elements corresponding to the adjacent vertices will not belong to the same member of *B*.

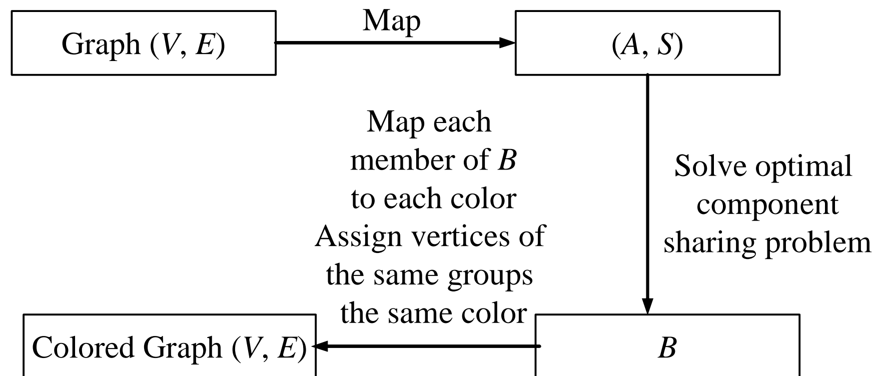


Figure 6.1: Converting GCP to OCSP

Now we need to show that the minimum number of groups of sharable components also leads to the minimum number of colors in the graph. This can be shown by a simple contradiction. Let us assume that the optimal solution to the OCSP is not an optimal solution for the graph coloring problem. In that case, let us find the optimal solution to the graph coloring problem. Using this solution, we can generate a solution for the OCSP that will have the same cardinality as the optimal solution to the graph coloring problem. Now based on our assumption, the optimal solution to the graph coloring problem is better than the optimal solution found for OCSP. Hence, we have just found a solution to OSCP that is better than the optimal solution to OCSP. This leads to a direct contradiction and hence we conclude that the optimal

solution to OCSP is also an optimal solution for the graph coloring problem. Thus the requirement 3 has also been met. This proves that OCSP is NP-Hard.

6.1.3 Branch And Bound Algorithm For Solving The Problem

In real life situations, very few artifacts are actually sharable. Hence, this leads to problem instances of relatively small size consisting of 10 or fewer elements in set A. Therefore, we believe that branch and bound algorithm is a good candidate for solving this problem. We expect that due to the pruning, it will work fast for many problem instances. Even if the truly worst case is encountered, since the problem size is small, it will still be able to find the optimal solution. So it will do better than simple enumeration and yet ensure optimal solution.

Graph coloring problems are notoriously difficult [Corm90] to find greedy algorithms with good approximation bounds. OCSP appears to be very similar in structure to graph coloring problem, hence we did not attempt to look for a greedy algorithm.

The branch and bound algorithm developed as a part of this dissertation is given below:

Algorithm FINDSHARABLECOMPONENTS

Input:

- $A = \{a_1, \ldots, a_m\}$
- $S = \{s_1, \ldots, s_n\}, s_i \subseteq A$

Output:

- Optimal set $B = \{B_1, \ldots, B_b\}$

Steps:

1. $B = \{\{a_1\}, \ldots, \{a_m\}\}$.

2. Assign *Current_Best* = Cardinality(B).

3. Call MERGEPAIRS(S, B).

4. Return B.

Algorithm MERGEPAIRS used in the above algorithm is given below.

Algorithm MERGEPAIRS(S, B)

Input:

- $S = \{s_1, \ldots, s_n\}$, $s_i \subseteq A$

Output:

- *Current_Best* and the current best solution B

Steps:

1) Find all pairs M_p in B that can be merged. A pair can be merged if the merged pair does not share two or more elements with any members of S.

2) If M_p is empty then

   i. if Cardinality(B) < *Current_Best*, then *Current_Best* = Cardinality(B)

   ii. Return.

3) Otherwise, if LOWERBOUND(B) ≥ *Current_Best*, return.

4) Sort members of M_p by increasing values of *Filled_Count*. *Filled_Count* is defined on a pair (b, b') as the number of elements in S with which $b \cup b'$ will have an intersection.

5) For every pair (b, b') in M_p, perform the following:

   i. $b'' = b \cup b'$

   ii. MERGEPAIRS(S, ((B-$\{b, b'\}$)$\cup\{b''\}$))

Algorithm LOWERBOUND used in the above algorithm is given below.

Algorithm LOWERBOUND(*B*)

Input:

- *B*

Output:

- Lower bound on solution resulting from *B*

Steps:

1) *Assign* n_1 = Number of elements in *B* that cannot be merged with any other member; $n_2 = 0$; $n_3 = 0$; $C = B$
2) Remove those elements from *C* that cannot be merged with any elements of *B*.
3) Until there exists *c* in *C* such that *c* can be merged with at least one element of *C*, do the following:
   i. Remove *c* from *C*.
   ii. Remove the members from *C* that can be merged with *c*
   iii. $n_2 = n_2 + 1$
4) If *C* is not empty, then $n_3 = 1$
5) Return $(n_1 + n_2 + n_3)$

The algorithm FINDSHARABLECOMPONENTS uses the following two heuristics:

- Function LOWERBOUND computes the lower bound on the solution that can result from performing future merging on *B*. Thus we can prune the solutions with larger lower bounds.
- Examining members of M_p after sorting it by *Filled_Count* helps in ensuring that we examine promising solutions first. This heuristic first examines those options that appear to have more merging choices in future.

Function LOWERBOUND guarantees that only unpromising solutions will be pruned.

Theorem 6.2. For any B, the cardinality of B after merging sharable components is larger or equal to LOWERBOUND(B).

Proof:

For any element in B that cannot be merged with any other element, it cannot be merged in any solution. Number of these elements corresponds to n_1. Let us assume that any element in B could merge with at least one element and one of these solutions B' will have cardinality less than $n_2 + n_3$ after the merging. The merge will lead to two groups: group B_2 includes members of B' that have at least two elements and group B_3 include members that have only one element.

Let m_2 be the cardinality of B_2 and m_3 be the cardinality of B_3. Each time we remove an element from any member of B_2, we can at least remove the rest of the elements from the same member because they can be merged. Thus we have $m_2 \geq n_2$. If there are members in B' that have only one element, then n_3=1 and $m_3 \geq 1$. Otherwise, n_3=0 and m_3=0. Thus we have $m_3 \geq n_3$. We conclude that

$$\begin{aligned} \text{Cardinality}(B') &= m_2 + m_3 \\ &\geq n_2 + n_3 \end{aligned}$$

This contradicts our assumption. Therefore any component sharing solution will have cardinality larger or equal to the return of the function LOWERBOUND(B)

6.1.4 Example

This algorithm can be illustrated using the following example.

Given:

- $A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$
- $S = \{s_1, s_2, s_3, s_4\} = \{\{a_1, a_2, a_4, a_6\}, \{a_3, a_5, a_7\}, \{a_1, a_5, a_7\}, \{a_1, a_3, a_6\}\}$

Figure 6.2 shows a transition diagram illustrating this example. Initial state has been omitted from this figure because it has no bearing on this example. Figure 6.2(a) shows the diagram before components are shared. Following the above branch and bound algorithm, we find an optimal solution to be $B = \{\{a_1\}, \{a_2, a_3\}, \{a_4, a_5\}, \{a_6, a_7\}\}$. Figure 6.2(b) shows the solution graphically.

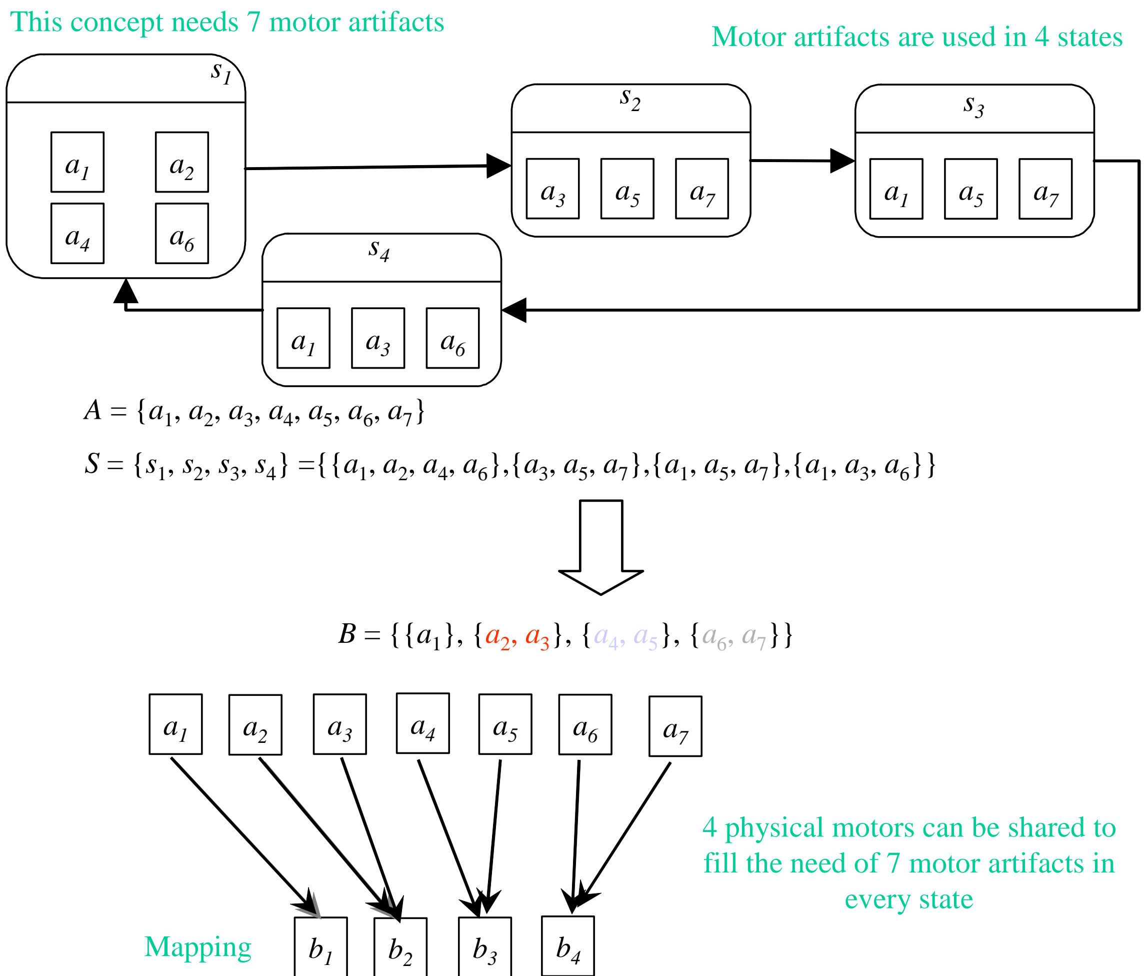


Figure 6.2: An example illustrating the branch and bound algorithm

6.2 Evaluating Design Concept Based On Maximum Power Consumption

The new representation of multiple interaction-state mechatronic design concepts makes it possible for us to determine which components are active in which states. This characteristic can be used to find out the maximum power consumed by a mechatronic device. The maximum power consumed cannot be simply computed by summing up the power requirements for all components. Instead, we need to figure out when components are active and when they are not active. We also need to determine the state where the maximum power is being consumed by active components.

In a given valid interaction-state transition diagram, each interaction-state represents a runtime working status of the device and use-environment. By definition, these working statuses are not concurrent. Thus the power consumption in each interaction-state can be compared and the maximum value is also the maximum power requirement for the device.

The following algorithm describes how to estimate maximum power consumption for a given design concept.

Algorithm FINDMAXIMUMPOWERCONSUMPTION

Input:

- A valid interaction-state transition diagram, where s_0 is the marked initial state.

Output:

- Maximum power consumption P_{max} and interaction-state s^*.

Steps:

1. Assign maximum power consumption $P_{max} = 0$.

2. For each state s except the initial state, do the following:

   a. Find active power consuming artifacts a_j in s.

   b. Assign $P = \Sigma\{\text{power consumption of } a_j\}$.

   c. If $P > P_{max}$, assign $P_{max} = P$; $s^* = s$.

3. Return P_{max}.

Figure 6.3 depicts a simplified interaction-state diagram.

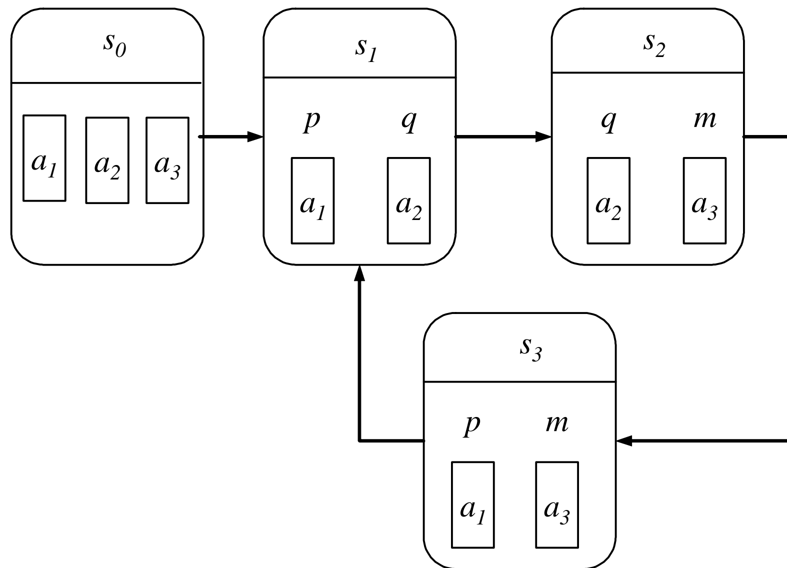


Figure 6.3: An example of estimating maximum power consumption

There are three artifacts in the design world, a_1, a_2, a_3, and different states have different active artifacts. Let us assume that a_1, a_2, a_3 consume power p, q, m respectively when they are active. Then according to the above algorithm, the maximum power required by the designed device is $\max\{p + q, p + m, q + m\}$. It is worth noting that simply summing up the power requirement of the three artifacts will yield the power consumption estimate of $p+q+m$, which will unnecessarily lead to the selection of a bigger power supply. This example illustrates that modeling the interaction-states can produce more accurate estimate of the power consumption in

case of multiple interaction-state devices. Similar approach can followed for the estimation of noise level etc.

6.3 Summary

This chapter presents algorithms to evaluate design concepts based on two criteria: maximum power consumption and optimal number of sharable components. For maximum power consumption problem we provide a simple algorithm to generate the solution. For the optimal component sharing problem we prove that it is NP-hard by comparing it to the graph coloring problem. We also provide a branch and bound algorithm to find the solution.

Chapter 7: Transition Diagram Synthesis

This chapter introduces the transition diagram synthesis problem based on the modeling framework introduced in Chapter 3. It presents the structure for describing the basic elements for synthesizing a transition diagram behind a design concept and provides an algorithm for synthesizing transition diagrams.

This chapter has been organized in the following manner. Section 7.1 describes the formulation of the problem based on the modeling framework introduced in Chapter 3. Section 7.2 describes the structure of the component library used during the synthesis process. Section 7.3 describes algorithms for synthesizing transition diagrams. Section 7.4 describes theorems showing soundness of the algorithms. Section 7.5 presents an example. Finally, Section 7.6 presents concluding remarks.

7.1 Problem Formulation

7.1.1 Preliminaries

Let D_i be the transition diagram describing the desired behavior specifications of a device. D_i is defined using the device artifact a_d and a set of use-environment artifacts A_u. Let C be the set of components from which the device artifact will be composed. All parameters used in D_i and C will be selected from a standard parameter list P. For every component $c \in C$, c is defined by a transition diagram $D_i(c)$ describing its behavior specifications, a detailed transition diagram $D_f(c)$ describing the concept behind it, and elaboration operators that describe how the initial transition diagram is mapped into the detailed transition diagrams.

We classify components into the following two categories. Basic components are the components that are not further decomposed. Complex components are components that are further decomposed into basic components. For a basic component, its initial transition diagram and final transition diagram will only consist of the component itself and its use-environment artifacts. We would like to make the following observations:

- If c is a basic component, then $D_i(c) = D_f(c)$. In other words, if c is a basic component, then the transition diagram corresponding to the behavior specifications cannot be further elaborated.
- If c is a complex component, then $D_i(c) \neq D_f(c)$. In other words, if c is a complex component, then the transition diagram corresponding to the behavior specifications will need to be further elaborated. Such elaboration will typically introduce basic components in the definition of $D_f(c)$ and hence c is realized by connecting other basic components together.

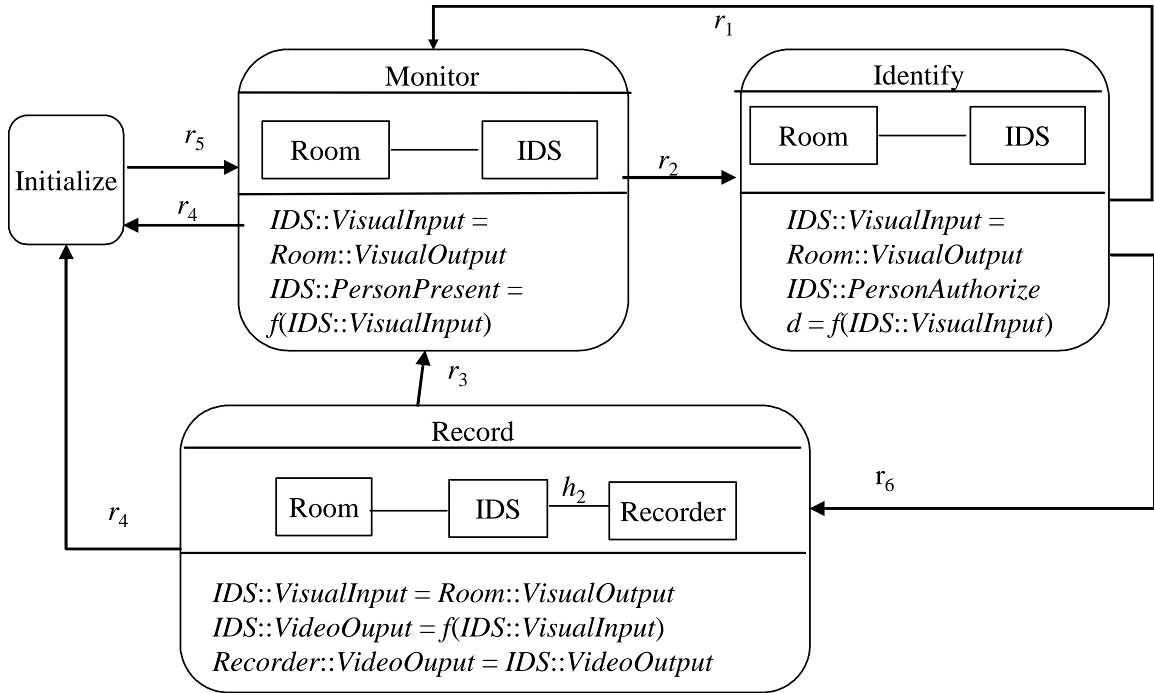
We are interested in modeling complex components because availability of complex components in the component library significantly reduces the combinatorial complexity of the synthesis problem by exploiting proven complex components. Once a complex component has been synthesized it can be reused in future design synthesis problems.

Now we will provide few examples to further clarify definitions given above. Few representative standard parameters are shown in Table 7.1. The desired behavior specification of an intruder detection system is shown in Figure 7.1. The artifact definitions used in this behavior specification are shown in Table 7.2. An example of

a component library is shown in Figure 7.2. Definitions of artifacts in the component library include both basic components and complex components (shown in Table 7.3). The behavior specifications for each component are shown in Figures 7.3 to 7.6.

Table 7.1: Standard parameters used in *IDS* example

Parameter	**Domain**	**Type**
OpticalSignal	Optical	*REAL*
DigitalSignal	Digital	*REAL*
Illumination	Optical	*REAL*
Focus	Optical	*REAL*
Voltage	Electrical	*REAL*
Energy	Electrical	*REAL*
Power	Electrical	*REAL*
Volume	Mechanical	*REAL*
Speed	Mechanical	*REAL*
Weight	Mechanical	*REAL*
StoringCapacity	Mechanical	*REAL*
Dimension	Mechanical	(*REAL*, *REAL*, *REAL*)
Position	Mechanical	(*REAL*, *REAL*, *REAL*)
Area	Mechanical	*REAL*
Time	Electrical	*REAL*



Name	Condition
r_1	*IDS*::*PersonAuthorized* = *TRUE*
r_2	*Room*::*PersonPresent* = *TRUE*
r_3	(*IDS*::*Timer*≥1200s) OR (*IDS*::*PersonPresent* = *FALSE*)
r_4	*IDS*::*Power* = *OFF*
r_5	*IDS*::*Power* = *ON*
r_6	*IDS*::*PersonAuthorized* = *FALSE*

Attribute	**Value**
Room::*PersonPresent*	*TRUE* / *FALSE*
IDS::*PersonAuthorized*	*TRUE* / *FALSE*
IDS::*Timer*	0 to 1200s
IDS::*Power*	*ON* / *OFF*

Figure 7.1: Transition diagram and event space used in *IDS* behavioral specification

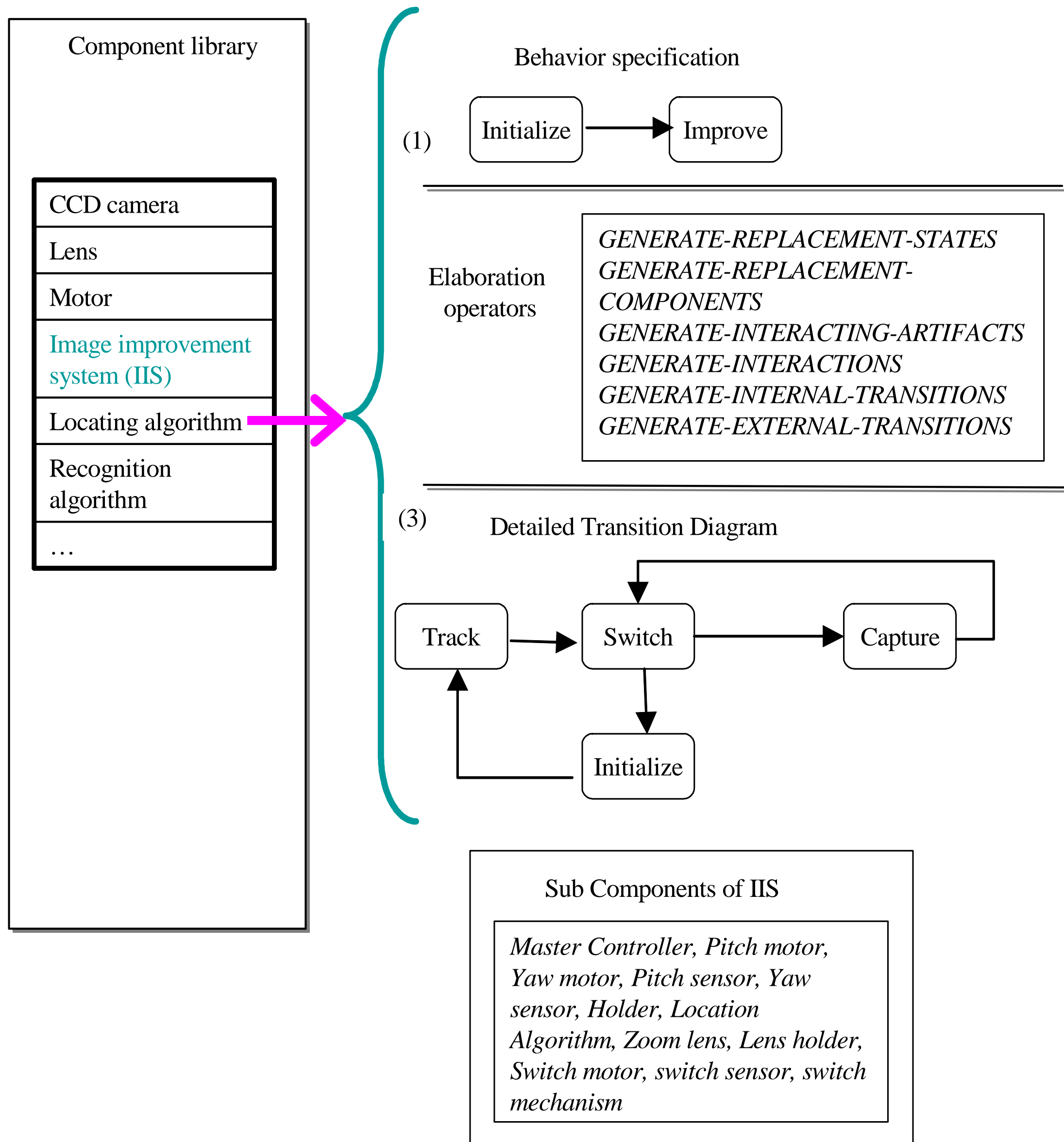
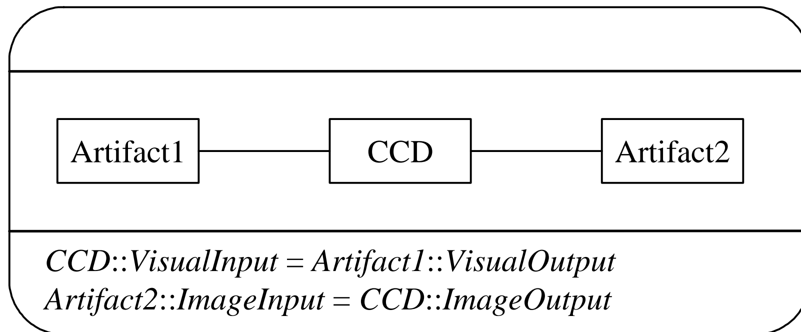


Figure 7.2: Example of a component library

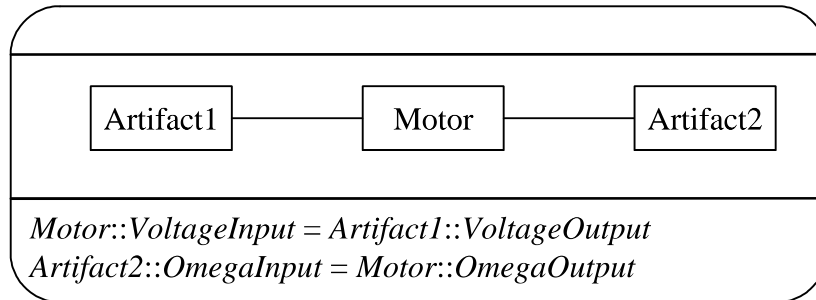
Table 7.2: Parameters selection for artifact definition

Artifact	Parameter	Alias	Type	Convention
IDS	*OpticalSignal*	*VisualInput*		
	DigitalSignal	*VideoOutput*	*IMAGE*	
	Information	*PersonAuthorized*	*BOOLEAN*	*TRUE/FALSE*
	Timer	*Timer*		
	Information	*PersonPresent*	*BOOLEAN*	*TRUE/FALSE*
	Power	*Power*	*BOOLEAN*	*ON/OFF*
Room	*Information*	*PersonPresent*	*BOOLEAN*	*TRUE/FALSE*
	OpticalSignal	*VisualOutput*		
Recorder	*DigitalSignal*	*VideoInput*	*IMAGE*	



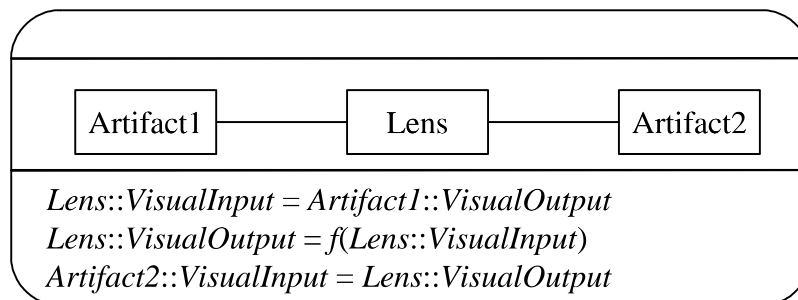
Object	Parameter	Alias
CCD Camera	*OpticalSignal*	*VisualInput*
	DigitalSignal	*ImageOutput*
	Power	*Power*
Artifact1	*OpticalSignal*	*VisualOutput*
Artifact2	DigitalSignal	*ImageInput*

Figure 7.3: Working state for *CCD* in behavior specification



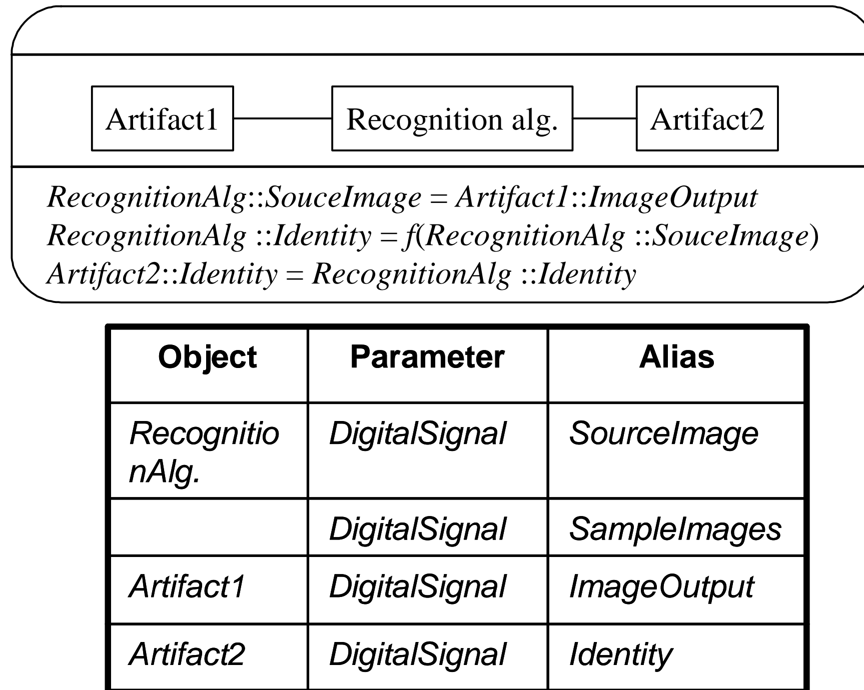
Object	Parameter	Alias
Motor	*Voltage*	*VoltageInput*
	AngularSpeed	*OmegaOutput*
Artifact1	*Voltage*	*VoltageOutput*
Artifact2	*Omega*	*OmegaInput*

Figure 7.4: Working state for motor in behavior specification



Object	Parameter	Alias
Lens	*Focus*	*Focus*
	ViewAngle	*ViewAngle*
	OpticalSignal	*VisualInput*
	OpticalSignal	*VisualOutput*
Artifact1	*OpticalSignal*	*VisualOutput*
Artifact2	*OpticalSignal*	*VisualInput*

Figure 7.5: Working state for lens in behavior specification



Object	Parameter	Alias
RecognitionAlg.	*DigitalSignal*	*SourceImage*
	DigitalSignal	*SampleImages*
Artifact1	*DigitalSignal*	*ImageOutput*
Artifact2	*DigitalSignal*	*Identity*

Figure 7.6: Working state for recognition algorithm in behavior specification

When generating a possible transition diagram for a design concept by connecting different components together, there are two possible representations. The first representation is called *compact* transition diagram D_c. This transition diagram represents complex components as single components and only utilizes their behavior specifications (i.e., $D_i(c)$). The second representation is called *elaborated* transition diagram D_f. This representation represents complex components by their constituent basic components and utilizes their elaborate transition diagrams (i.e., $D_f(c)$).

In our framework, synthesis problem is solved using a two-step approach. In the first step, given the behavior specifications D_i for the device, we generate compact transition diagram D_c behind the design concept by utilizing components in the component library. This step only uses behavior specifications of the components. Once we have successfully generated the compact transition diagram D_c, we map it

into elaborated transition diagram D_f by mapping each complex component in D_c into its constituents.

The following definitions of equivalence will be used in the subsequent sections of this chapter.

- **Equivalence of parameters**: Two parameters p_1 and p_2 are considered equivalent if their names are identical. This also implies ($p_1.DataType = p_2.DataType$) AND ($p_1.Unit = p_2.Unit$).
- **Equivalence of parameter interactions:** Two parameter interactions are equivalent if their corresponding members are equivalent.
- **Equivalence of artifacts:** Two artifacts are equivalent if their corresponding parameter sets are equivalent and parameter interactions are equivalent. However, *ArtifactType* of the artifacts could be different because it is context related. An artifact can serve as device in one context and use-environment artifact in another context.
- **Equivalence of interaction-states**: Two interaction-states are equivalent if their corresponding members are equivalent. Since we only deal with synthesis related to component connectivity, we consider two equivalent states as having equivalent parameters and parameter interactions.
- **Equivalence of transitions:** Two transitions are equivalent if their corresponding members are equivalent.
- **Equivalence of transition diagrams:** two transition diagrams are equivalent if their corresponding members are equivalent.

7.1.2 Problem Statement

In this dissertation, we impose the following restrictions on the synthesis problems being considered:

- All parameters used will be only selected from a standard parameter list *P*.
- We do not consider checking unsafe parameter value sets and validating interaction-state consistency as a part of the synthesis process. Transition diagram synthesis process will be limited to the selection of components only based on the interactions of the components with use-environment artifacts. Additional checking can be done as a post-processing step.
- We only consider interactions that are defined between two artifacts. Interactions that simultaneously involve more than two artifacts will not be considered.
- We only handle those components whose behavior specifications have only one working interaction-state (a component may have one additional initial state). However, complex components can have multiple interaction-states in their elaborated transition diagrams.

We formulate the design concept synthesis problem in the following manner.

Given,

- Standard parameter list *P*
- Transition diagram behind behavior specification D_i (all parameters used in D_i belong to *P*)
- Component library *C*

We are interested in finding the following two transition diagrams:

1. Compact transition diagram D_c, satisfying the following conditions:

a) For every artifact a that is used in D_c, there exists either a basic or a complex component c in C such that a and c have the equivalent behaviors. This condition is described as the following:

i. For every parameter interaction in the working state of $D_i(c)$ (please recall that $D_i(c)$ has only one working state), we can find an equivalent parameter interaction in those states of D_c that involve a.

ii. Let S^* be those states in D_c that involve a. Every state s in S^* meets the following condition. For every parameter interaction in s, we can find an equivalent parameter interaction in the working state of $D_i(c)$.

b) D_c is consistent with D_i. This condition requires that D_c is equivalent to D_i after the following transformations:

i. For every state in D_c, remove those interactions that only involve components (i.e., use environment artifacts are not involved). Replace components in all the remaining interactions with the device artifact.

ii. For every transition in D_c, replace the components and their parameters with the device artifact and its parameters.

2. Elaborated transition diagram D_f, generated by applying elaboration operators for every complex component in D_c.

7.2 Structure of the Component library

The component library is a set of available components that can be used during the synthesis process. As explained in Section 7.1, each component is defined by two transition diagrams $D_i(c)$ and $D_f(c)$. In addition to these two transition diagrams, for

each complex component we also have a sequence of elaboration operators that describe how to transform $D_i(c)$ into $D_f(c)$. These operators are described below:

- **Generate Replacement States**. This operator is called *GENERATE-REPLACEMENT-STATES* and used to replace the working state in the behavior specification of the complex component with the working states in its final transition diagram. This operator is defined as the following.
    - Input: Transition diagram that includes the working state *s* of the complex component in its behavior specification.
    - Action: Remove *s* from the transition diagram. Generate a set of empty states *S* that has the same number of working states as in the final transition diagram of the complex component.
- **Generate Replacement Components**. This operator is called *GENERATE-REPLACEMENT-COMPONENTS* and used to replace the complex component with its constituent basic components in the empty states generated above. This operator is defined as the following.
    - Input: Transition diagram produced by *GENERATE-REPLACEMENT-STATES*.
    - Action: Insert participating components of the complex component for each working state in its final transition diagram into each corresponding state in *S*.
- **Generate Interacting Artifacts**. This operator is called *GENERATE-INTERACTING-ARTIFACTS* and used to generate use-environment artifacts for each states generated above. This operator is defined as the following.

- Input: Transition diagram produced by *GENERATE-REPLACEMENT-COMPONENTS*.
- Action: Insert interacting use-environment components of the complex component into each state in *S* according to each corresponding working state in its final transition diagram.

- **Generate Interactions**. This operator is called *GENERATE-INTERACTIONS* and used to generate interactions for each state generated above. This operator is defined as the following.
  - Input: Transition diagram produced by *GENERATE-INTERACTING-ARTIFACTS*.
  - Action: Insert interactions between artifacts in each state of *S* based on the final transition diagram for the complex component.
- **Generate Internal Transitions**. This operator is called *GENERATE-INTERNAL-TRANSITIONS* and used to generate transitions between states generated above. This operator is defined as the following.
  - Input: Transition diagram produced by *GENERATE-INTERACTIONS*.
  - Action: Insert transitions between states of *S* according to the corresponding transitions in the final transition diagram of the complex component.
- **Generate External Transitions**. This operator is called *GENERATE-EXTERNAL-TRANSITIONS* and used to generate the transitions between states generated by the previous operator and other states in the compact transition diagram. This operator is defined as the following.

- Input: Transition diagram produced by *GENERATE-INTERNAL-TRANSITIONS*.
- Action: Apply incoming transitions for *s* to the states in *S* that correspond to the states that take incoming transitions from the initial state in the final transition diagram of the complex component. Apply outgoing transitions for *s* to the states in *S* that correspond to the states that take outgoing transitions to the initial state in the final transition diagram of the complex component. Remove transitions that do not involve any artifact in the starting and ending states of the transitions.

The above described operator sequences are communicative in nature across complex components. For example, consider a state that involves two complex components *c* and *c'*. Applying the operator sequence for *c* first and then applying the operator sequence for *c'* produces the same result as applying the operator sequence for *c'* first and then applying the operator sequence for *c*.

Parameters can be organized according to different levels of abstraction. For example, signals can be specialized into optical signals and digital signals. Usually desired parameters of a device being designed are expressed with a higher level of abstraction (i.e., more general). Parameters of components in the library are expressed using lower levels of abstraction (i.e., more specialized). When we look for matching parameters in the component library for desired parameters in the behavior specifications, specialized parameters can always be used in the place of more general parameters. For example, if we want a signal parameter in the desired artifact, we can choose a component with either optical signal or digital signal parameters in the

component library. On the other hand, parameters can use wildcard values in transition conditions. That means if any value in the range of values represented by the wildcard, the condition is satisfied. For example, transition condition (*Signal* = *ANY*) means any signal with any value could satisfy the condition. A satisfying condition could be (*DigitalSignal* = 5). These types of wildcards are mainly used to enable matching of transition conditions.

Compatibility of parameters: Parameters can be organized hierarchically. For example, the parameter signal has optical signal and digital signal as its children. A child parameter can be used in any place where its parent parameter is used. We say the children parameter is one of the compatible parameters of the parent parameter.

Similarly, a parameter interaction is compatible with another parameter interaction if the parameters in the second interaction are equivalent or children of the parameters in the first interaction.

For the convenience of modeling, designers could use non-standard parameter names. These non-standard names serve as alias of standard parameters. We consider two parameters to be equivalent if their standard names are the same.

Compatibility of transitions: a transition r_1 is compatible with a transition r_2 if the parameters used in r_1 are equivalent or compatible with the parameters used in r_2.

7.3 Synthesis Algorithms

Developing the transition diagram behind a design concept not only requires one to select the right components from the library, but also requires one to connect them in a consistent manner to generate the compact and the elaborated transition diagrams. Complex components can be treated as basic components while generating compact

transition diagrams. Once compact transition diagrams have been developed, they can be mapped to fully elaborated transition diagrams by including details about the complex components. Once fully elaborated transition diagrams are generated, they could be added into the component library for future reuse.

Our synthesis algorithm starts from the initial transition diagram and uses breadth first search method for exploring various component combinations and generating the new compact transition diagrams.

The basic ideas behind generating compact transition diagrams are as following. First we look for a state which still has unknown artifacts. Unknown artifact could be implemented by a component if the set of the component's input/output interactions in its behavior specifications and the interactions associated with the unknown artifact have common members. The more the number of matching interactions is, the higher the possibility is that the component can be used as a part of the unknown artifact. If multiple components are possible, then these components are tried in the decreasing cardinality of matching interactions. The intent behind this heuristics is to converge to a solution quickly. After, identifying promising components we integrate the selected component into the current transition diagram. Since a component has to be used in its proper use-environment, we have to either find equivalent interactions in the state for the substituting component's parameter interactions or we need to insert extra interactions that are not present in the state into the state. Interactions of device artifact that are equivalent to the interactions of the component are realized by the component. Thus we replace the device artifact with the component in those interactions. For those interactions of the device artifact that cannot be fulfilled, we

leave them as they are and they would be resolved by another component later. The extra interactions between the component and its use-environment artifact are considered as the interactions between the component and a new device artifact. We also need to replace the parameters in the transition that are equivalent between device artifact and the component just selected. This step inserts a new component and updates the interactions between the device artifact and its use-environment artifacts. Each introduction of component will realize some parameter interactions of the device artifact. Inserting a component may also introduce new interactions. The selection of a set of components will be able to finally realize all the desired interactions if there exists a solution.

After selecting a component, we update transition diagrams for each applicable component. Components will be added one by one until there is no unknown artifact in any state, then the device artifact has been realized and we will get a compact transition diagram with the desired characteristics and the solution is considered complete.

The algorithm for generating compact transition diagram is described below.

Algorithm GENERATECOMPACTTRANSITIONDIAGRAM

Input:

- Standard parameter list *P*.
- Initial behavior specification D_i.
- Component library *C*.

Output:

- Compact transition diagram D_c

Steps:

1. Initialize the queue *W* with D_i.
2. Select the first element *D* from *W* and do the following:
   a Remove *D* from *W*.
   b Find the set of components C^* from the component library *C* that are applicable to *D* in the following manner.
      i. Initialize C^* as an empty sequence.
      ii. For each state $s \in D$, if there exit an artifact in *s.ArtifactSet* that is not a known component or known use-environment artifact, do the following:
         1) For every component $c \in C$, do the following:
            a) Let s' be the working state of $D_i(c)$.
            b) if $s'.InteractionInfo \cap s.InteractionInfo \neq \emptyset$, insert *c* into C^*, m = cardinality($s'.InteractionInfo \cap s.InteractionInfo$).
      iii. Sort elements of C^* by decreasing value of *m* (ties will be broken randomly).
   c Examine every *c* in C^* sequentially and do the following.
      i. Copy *D* to D'.
      ii. For each state $s \in D'$:
         1) Let s' be the working state of $D_i(c)$.
         2) If $s'.InteractionInfo \cap s.InteractionInfo \neq \emptyset$, insert *c* into *s.ArtifactSet*, and replace a_D with *c* in $s'.InteractionInfo \cap s.InteractionInfo$.

3) Insert $H' = (s'.InteractionInfo - (s'.InteractionInfo \cap s.InteractionInfo))$ into $s.InteractionInfo$.

iii. For every transition $r \in D'$, if r involves parameters of a_D that are equivalent to that of c, replace a_D with c.

iv. If there is no device artifact in any state of D, then the solution is complete, return D' and exit.

v. Insert D' into W.

3. If the time limit has exceeded then exit with failure. Otherwise, go to Step 2.

The basic idea behind generating elaborated transition diagram from the compact transition diagrams is as following. Since complex components have multiple states in their final transition diagram $D_f(c)$, these states need to replace the state s where the complex component is used. We do the replacement for each complex component one by one. We use the operators associated with the complex component in the component library to decompose the initial transition diagram into a detailed transition diagram. The elaborated transition diagram is obtained after all complex components have been decomposed and the transition diagram has been updated accordingly.

The algorithm for generating elaborated transition diagram is described below.

Algorithm GENERATEELABORATEDTRANSITIONDIAGRAM

Input:

- Compact transition diagram D_c.
- Standard parameter list P.
- Component library C.

Output:

- Elaborated transition diagram D_f.

Steps:

1. For every complex component c used in the compact transition diagram D_c, do the following:
   a Use the operators associated with c in the component library to elaborate D_c.
2. Return the elaborated transition diagram as D_f.

7.4 Characteristics of Algorithms

The following two theorems highlight the main characteristics of the above-described algorithms.

Theorem 7.1. Compact transition diagram D_c generated by **Algorithm** GENERATECOMPACTTRANSITIONDIAGRAM meets the following conditions:

1. For every artifact a that is used in D_c, there exists either a basic or a complex component c in C such that a and c have equivalent behaviors. This condition is described as the following:
   a) For every parameter interaction in the working state of $D_i(c)$ (please recall that $D_i(c)$ has only one working state), we can find an equivalent parameter interaction in those states of D_c that involve a.
   b) Let S^* be those states in D_c that involves a. Every state s in S^* meets the following condition. For every parameter interaction in s, we can find an equivalent parameter interaction in the working state of $D_i(c)$.
2. D_c is consistent with D_i. This condition requires that D_c is equivalent to D_i after the following transformations:

c) For every state s in D_c, remove those interactions that only involve components from the component library (i.e., use environment artifacts are not involved). Replace components in all the remaining interactions with the device artifact.

d) For every transition in D_c, replace the components and their attributes with the device artifact and its attributes.

Proof:

Condition 1a is satisfied by Step 2c of the algorithm. Each time we select a component into each state of D_c, we keep the interactions that are identical to that in $D_i(c)$ and add the interactions that are only in $D_i(c)$ into D_c. Thus all parameter interactions in $D_i(c)$ are kept in D_c. Since $D_i(c)$ only involves one working state, Step 2c also ensures that condition 1b is satisfied.

Since behavior specifications of all components selected will have only one working state, there will be the same number of states and transitions in D_c and D_i. Step 2c ensures each time a component is added, some interactions of the device artifact will be realized by the interactions of the component. When the algorithm exits successfully, all interactions of the device artifact in each state must have been fulfilled. The interactions between the device artifact and use-environment artifacts must have been fulfilled too. Thus condition 2a is satisfied by D_c. For each transition, the algorithm only replaces the device artifact when equivalent parameters are found in the components selected. All component parameters in D_c must be realized by this kind of replacement. Thus if we reverse the replacement, we will get the same transitions in D_i. Thus condition 2b is satisfied by D_c.

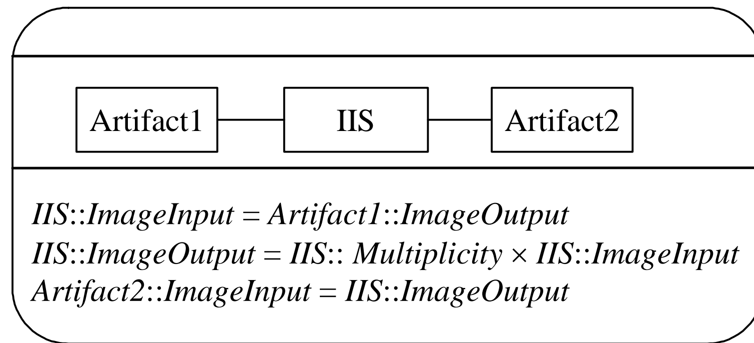
Theorem 7.2. If there exists a compact transition diagram that satisfies the given behavior specifications and involves finite number of components from the component library, then **Algorithm** GENERATECOMPACTTRANSITIONDIAGRAM will find it.

Proof:

Each state in the compact transition diagram is a connected graph. Nodes in the graph are the components and use-environment artifacts, and edges in the graph are interactions among components. The use-environment artifacts only interact with components. Behavior specifications of a state identify the use-environment artifacts and their interactions with the desired artifact. Therefore, if a possible graph exists for a state, then it can be discovered by starting from a use-environment artifact and adding a component that corresponds to a node in the graph one at a time. **Algorithm** GENERATECOMPACTTRANSITIONDIAGRAM considers all possible sequences of introducing components in the graph. Hence if a solution exists with a finite number of components, then the algorithm will find it.

7.5 Example

Let us take the design of intruder detection system (*IDS*) as an example. We are given a list of standard parameters shown in Table 7.1, desired behavior specification of an intruder detection system shown in Figure 7.1. The artifact definitions for the behavior specification are shown in Table 7.2. The artifact definitions of a component library that shows both basic components and complex components are shown in Table 7.3. The behavior specifications for each component are shown in Figures 7.3 to 7.11. Now we need to generate the compact transition diagram for *IDS*.

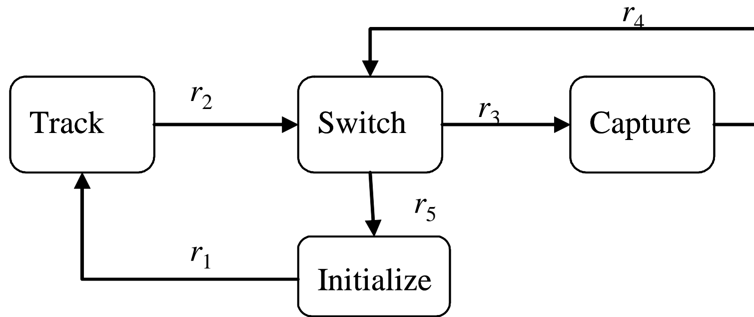


Object	Parameter	Alias
Image Improvement System (IIS)	*DigitalSignal*	*ImageInput*
	DigitalSignal	*ImageOutput*
	Multiplicity	*Multiplicity*
Artifact1	*DigitalSignal*	*ImageOutput*
Artifact2	*DigitalSignal*	*ImageInput*

Elaboration Operators

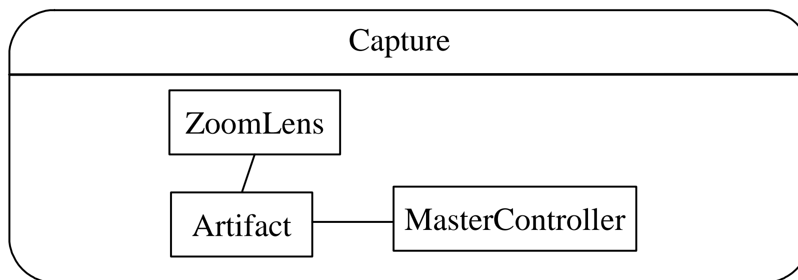
GENERATE-REPLACEMENT-STATES *GENERATE-REPLACEMENT-COMPONENTS* *GENERATE-INTERACTING-ARTIFACTS* *GENERATE-INTERACTIONS* *GENERATE-INTERNAL-TRANSITIONS* *GENERATE-EXTERNAL-TRANSITIONS*

Figure 7.7: Working state for image improvement system in behavior specification



Name	Condition
r_1	$Artifact::Singal = ANY$
r_2	$(Holder::Theta = MasterController::Theta)$ AND $(Holder::Phi = MasterController::Phi)$
r_3	$MasterController::LensinPosition = TRUE$
r_4	$MasterController::ImageOut \neq NONE$
r_5	$MasterController::LensinPosition = FALSE$

Figure 7.8: Final transition diagram for the Image Improvement System

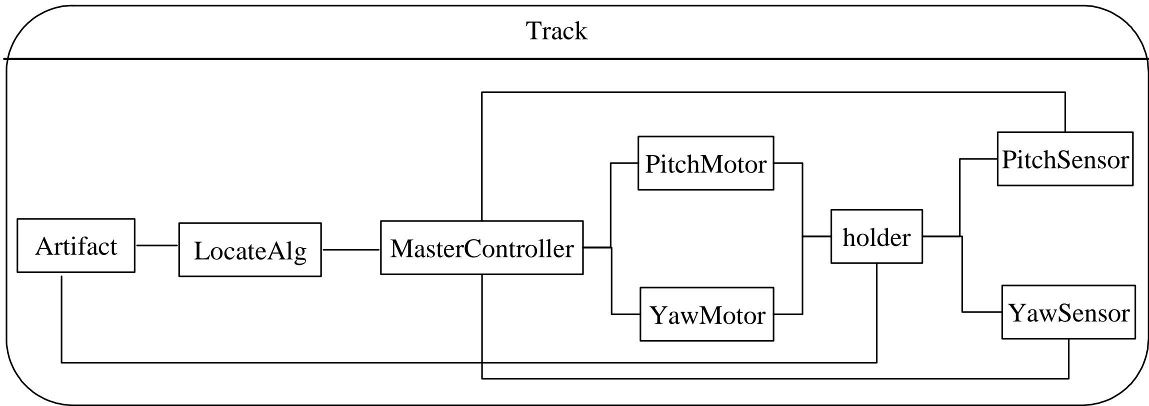


Artifact Interaction Equations

$Artifact::VisualInput = ZoomLens::VisualOutput$
$Artifact::ImageOutput = LocateAlg::ImageInput$
$MasterController::\text{ImageInput} = Artifact::ImageOutput$

Object	Parameter	Alias
Artifact	*DigitalSignal*	*ImageOutput*

Figure 7.9: State description of “Capture”

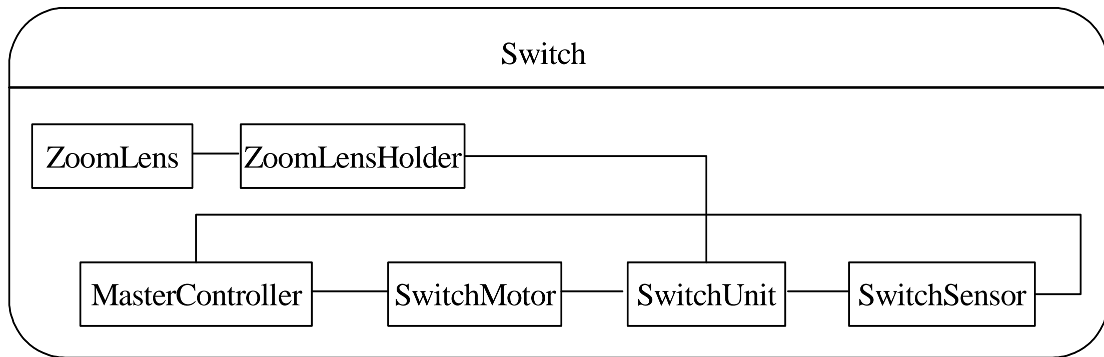


Artifact Interaction Equations

LocateAlg:: *ImageInput* = *Artifact*::*ImageOutput*
MasterController:: *Theta* = *LocateAlg*:: *Theta*
MasterController:: *Phi* = *LocateAlg*:: *Phi*
PitchMotor:: *VoltageInput* = *MasterController*:: *Theta*
YawMotor:: *VoltageInput* = *MasterController*:: *Phi*
Holder:: *Theta* = *Pitchmotor*::*OmegaOutput*
Holder:: *Phi* = *YawMotor*:: *OmegaOutput*
PitchSensor:: *Theta* = *Holder*:: *Theta*
YawSensor:: *Phi* = *Holder*:: *Phi*
MasterController:: *Theta* = *PitchSensor*:: *Theta*
MasterController:: *Phi* = *YawSensor*:: *Phi*

Object	**Parameter**	**Alias**
Artifact	*DigitalSignal*	*ImageOutput*

Figure 7.10: State description of “Track”



Artifact Interaction Equations

SwitchMotor:: *VoltageInput* = *MasterController*:: *LensPosition*
SwitchUnit:: *Position* = *SwitchMotor*:: *OmegaOutput*
SwitchSensor:: *AngularInput* = *SwitchUnit*:: *Position*
MasterController:: *LensPosition* = *SwitchSensor*:: *AngularOutput*
ZoomLensHolder:: *Position* = *SwitchUnit*:: *Position*
ZoomLens:: *Position* = *ZoomLensHolder*:: *Position*

Figure 7.11: State description of "Switch"

In order to find applicable components from the library, we look at the interactions in each state of *IDS*. First, *CCD camera* is selected because its interactions are compatible with that of *IDS* in the monitor state. Then we incorporate *CCD* into the initial transition diagram of *IDS* and requirements for the new device artifact are generated. Second, image improvement system (*IIS*) is selected to magnify the digital image obtained from the *CCD camera*. IIS is only applied to the identify state. Third, we select recognition algorithms to produce the signal from the image. Two recognition algorithms are needed for detecting person's presence and determining person's identity. Since all the interactions have been fulfilled, a solution is reached. This process is illustrated in Figure 7.12. After we have a component, we try to incorporate the component into the current transition diagram. This will limit the behavior of the device and provide clues for selecting the next component. Figures

7.13 to 7.15 illustrate steps in incorporating different components after they are selected.

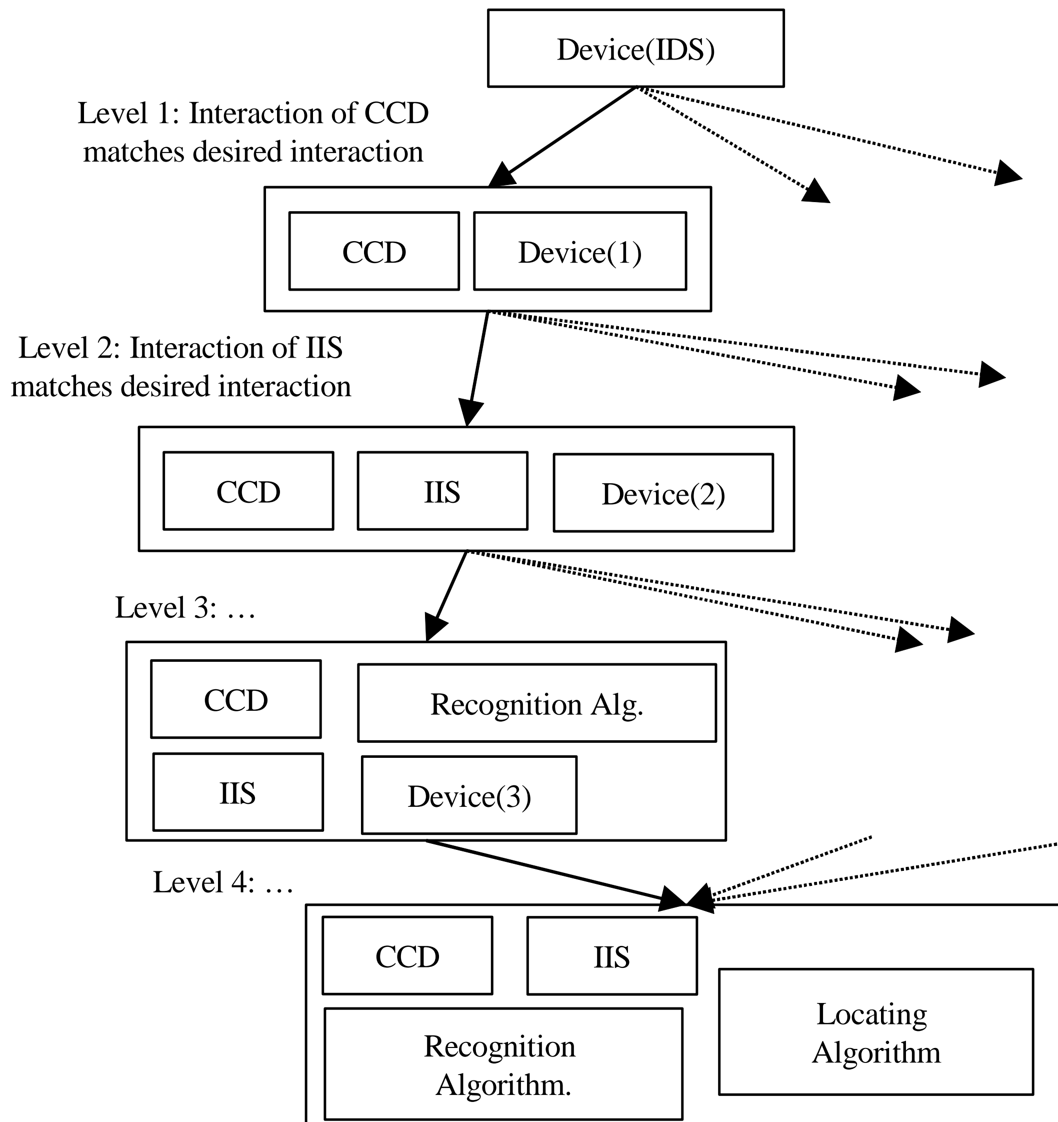


Figure 7.12: Illustration of searching for components of *IDS*

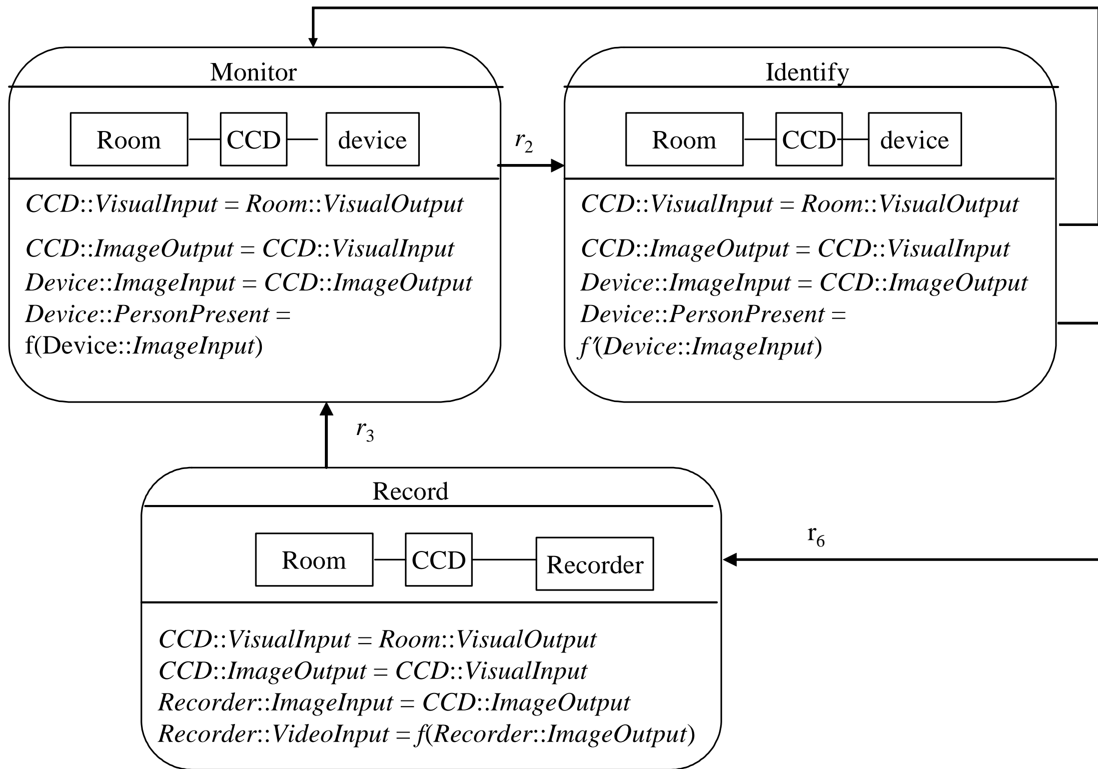


Figure 7.13: Incorporate *CCD* into *IDS*

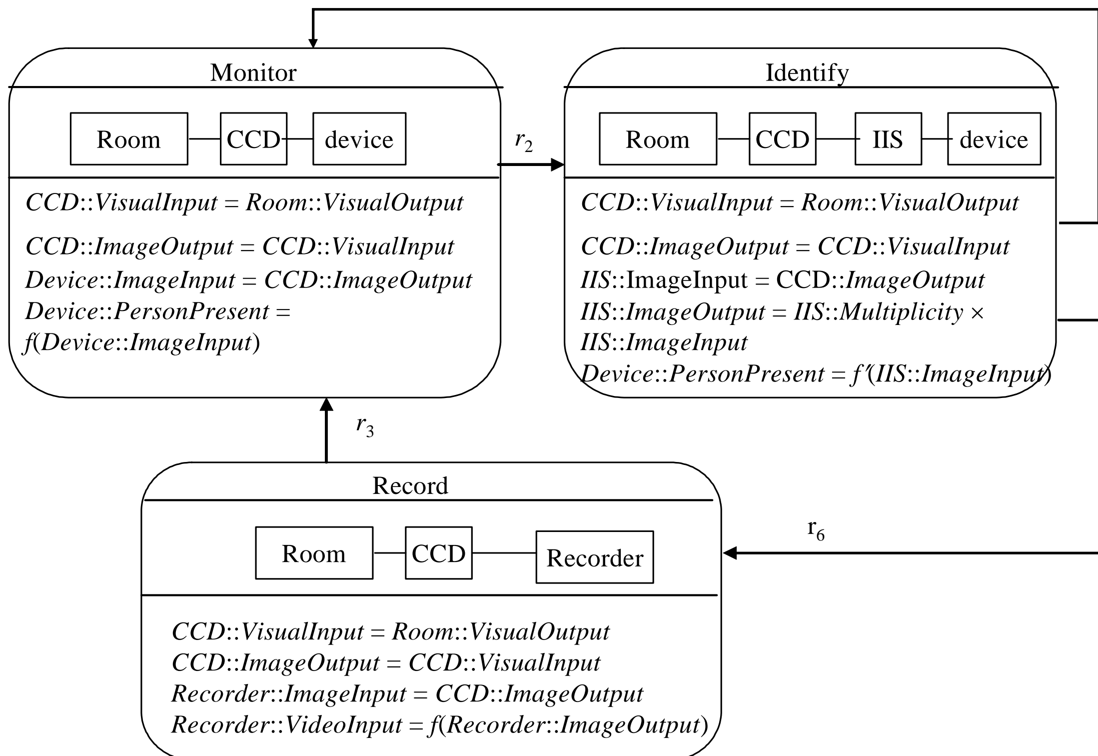


Figure 7.14: Incorporate IIS into *IDS*

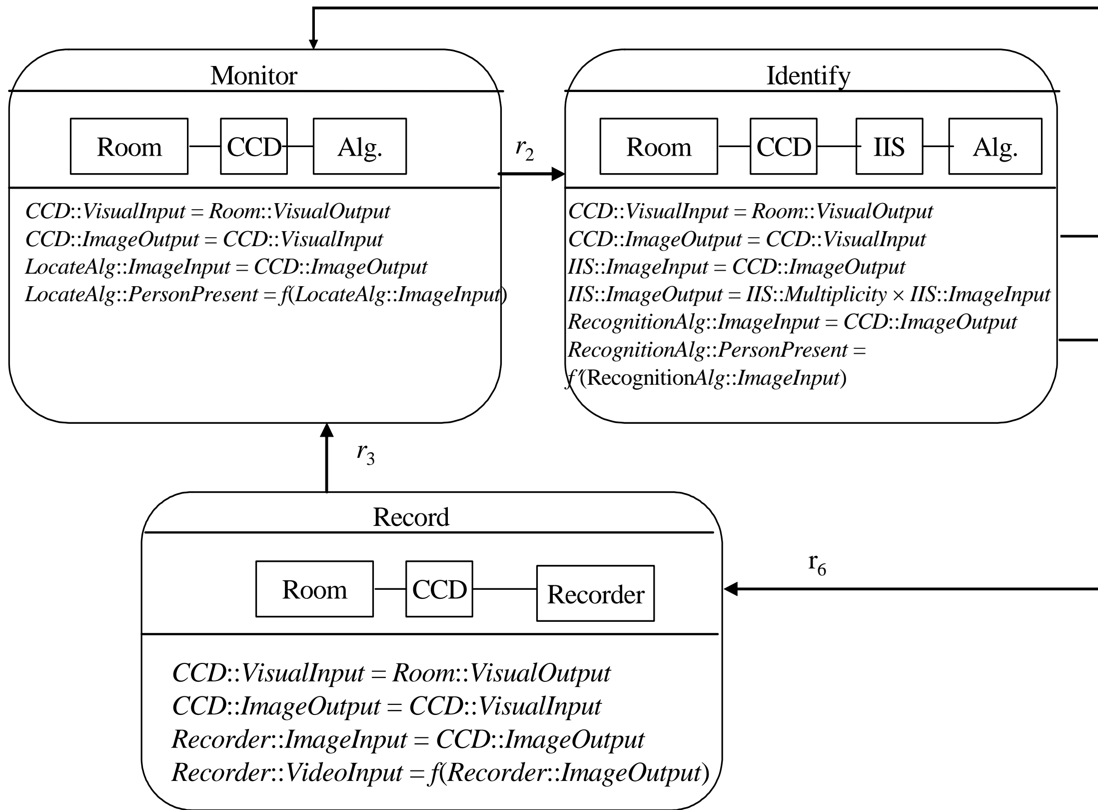


Figure 7.15: Incorporate recognition algorithm and locate algorithm into *IDS*

IIS has four states in its final transition diagram, of which three are working states. In order to get an elaborated transition diagram from the compact transition diagram, we use the six elaboration operators sequentially. First we generate three empty states corresponding to the three working states of *IIS*. Second we insert the components of *IIS* into each state. Third, we incorporate artifacts interacting with IIS into the three states. Fourth, interactions are added for the three states. Fifth, transitions between the three states are inherited from the final transition diagram of IIS. Sixth, we identify that Track and Switch are the two states that are connected to the initial state in IIS's final transition diagram. Since Track state takes incoming transition from the initial state in IIS's final transition diagram, it will take the incoming transition from Monitor state in the elaborated transition diagram. Since Switch state takes outgoing

transition to the initial state in *IIS*'s final transition diagram, it will take the outgoing transition to Monitor state and Record state in the elaborated transition diagram. Since there is only one complex component used, the elaborated design concept is generated after applying elaboration operators for *IIS*. Figures 7.16 to 7.25 show various steps.

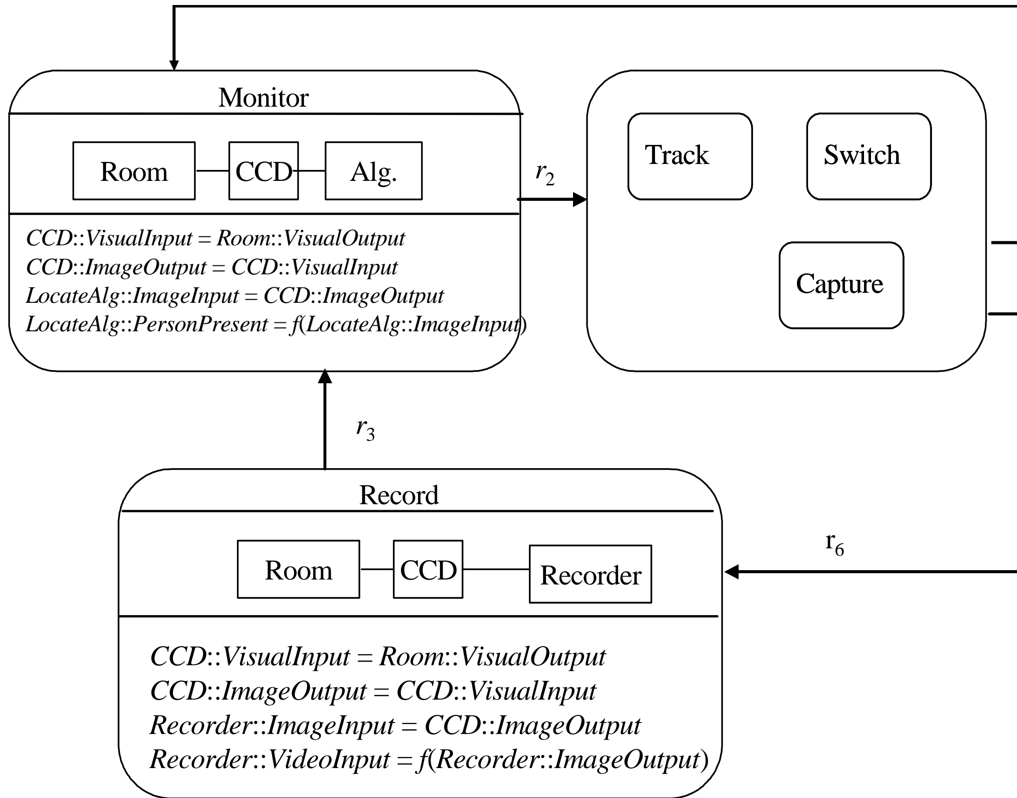


Figure 7.16: Applying operator: generate replacement states

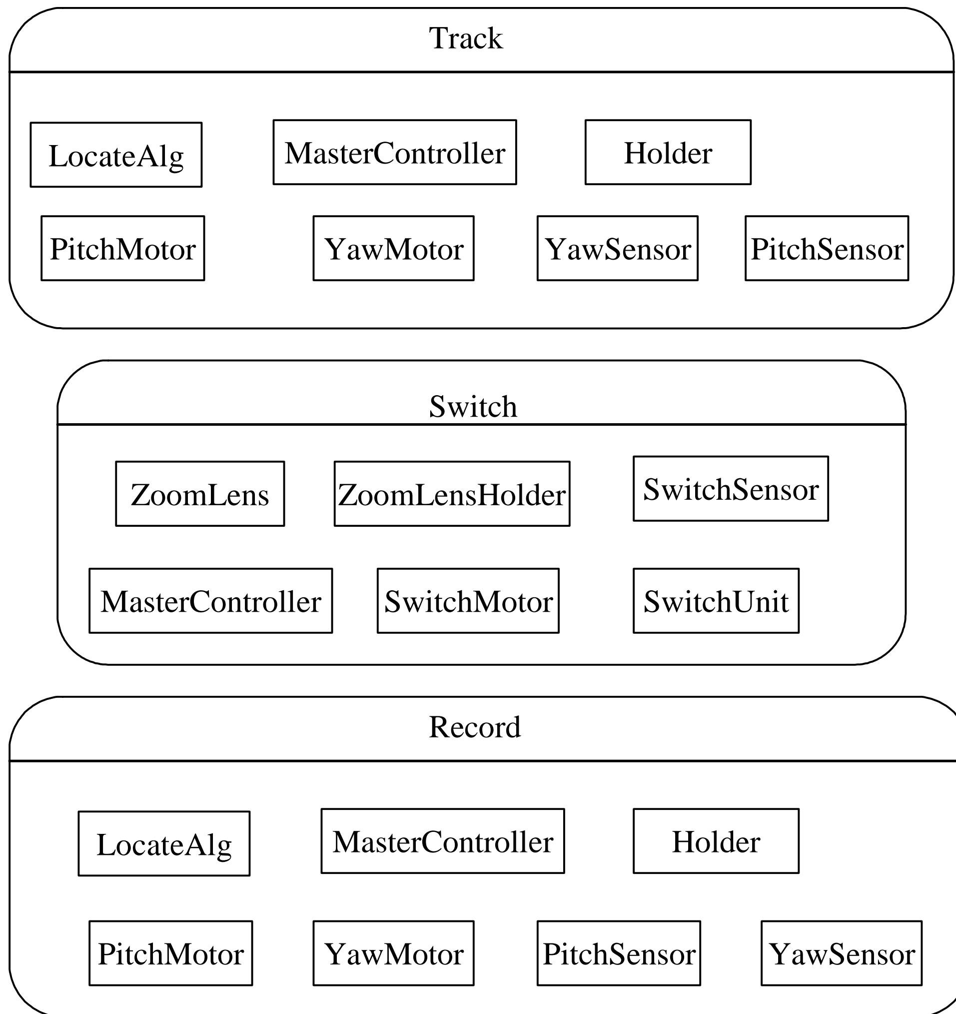


Figure 7.17: Applying operator: generate replacement components

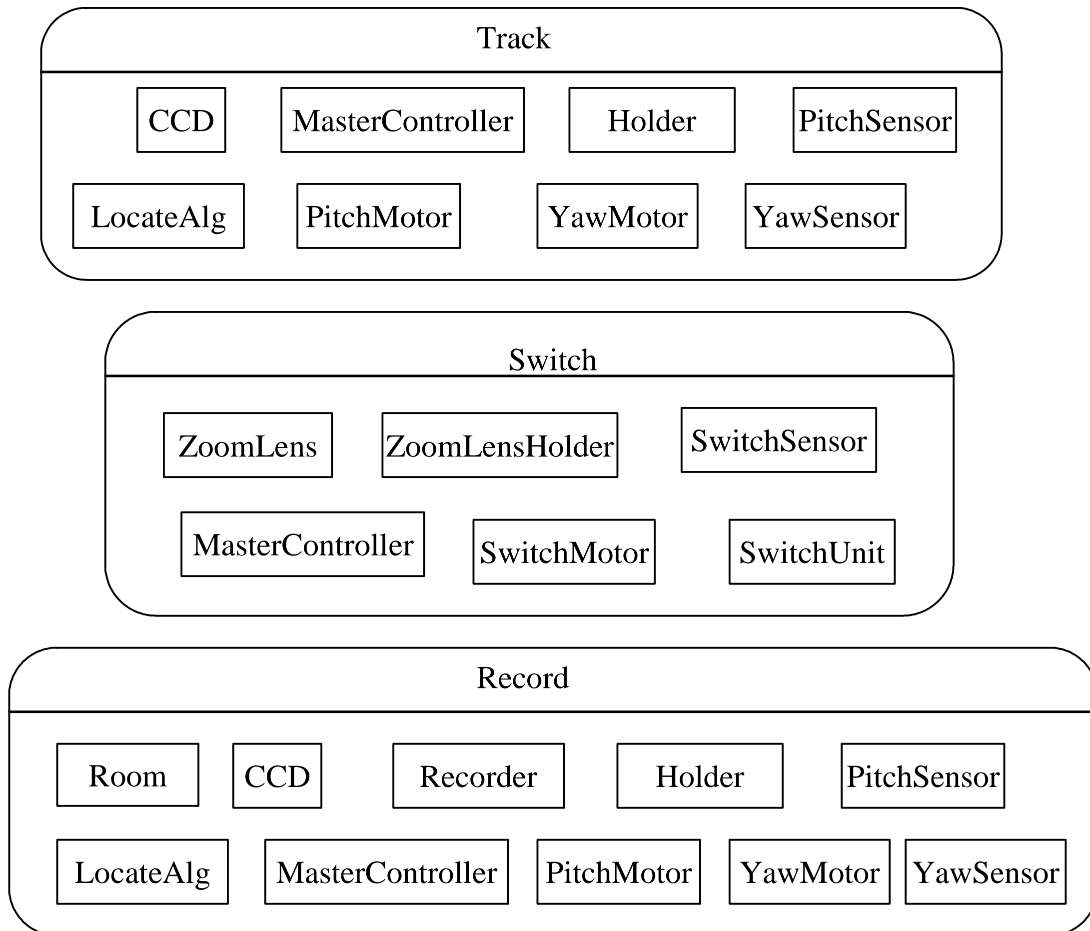
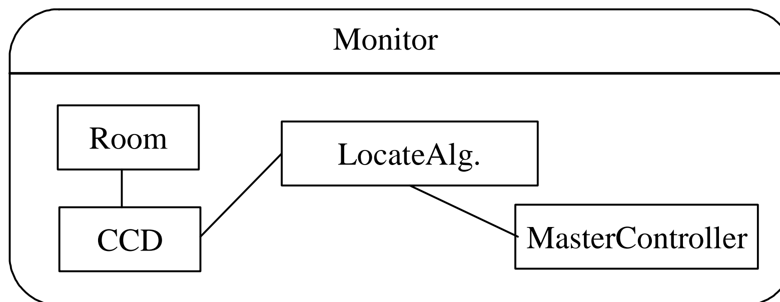


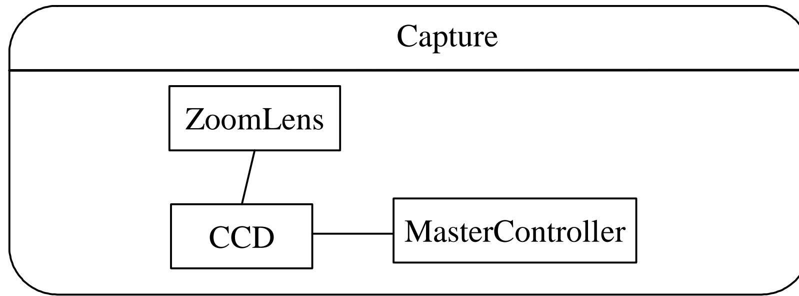
Figure 7.18: Applying operator: generate use-environment components



Artifact Interaction Equations

CCD::*VisualInput* = *Room*::*Visualoutput*
LocateAlg:: *ImageInput* = *CCD*::*Imageoutput*
MasterController::*Theta* =*LocateAlg*:: *Theta*
MasterController:: *Phi* = LocateAlg:: *Phi*

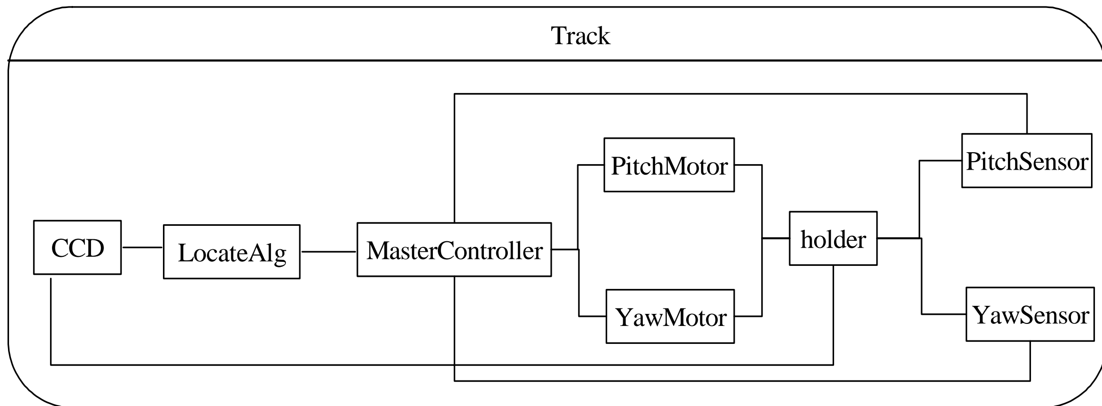
Figure 7.19: State description of "Monitor"



Artifact Interaction Equations

CCD::*VisualInput* = *ZoomLens*::*VisualOutput*
CCD::*ImageOutput* = *LocateAlg*:: *ImageInput*
MasterController::ImageInput = *CCD*::*ImageOutput*

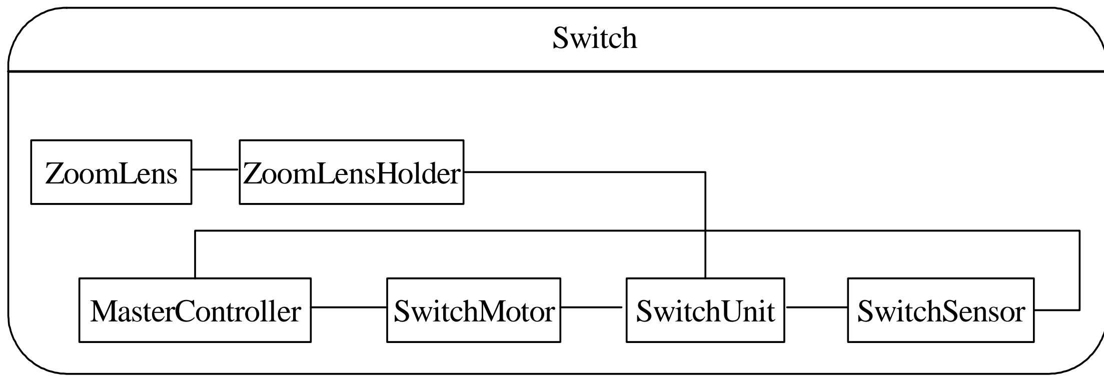
Figure 7.20: State description of "Capture"



Artifact Interaction Equations

LocateAlg:: *ImageInput* = *CCD*::*ImageOutput*
MasterController:: *Theta* = *LocateAlg*:: *Theta*
MasterController:: *Phi* = *LocateAlg*:: *Phi*
PitchMotor:: *VoltageInput* = *MasterController*:: *Theta*
YawMotor:: *VoltageInput* = *MasterController*:: *Phi*
Holder:: *Theta* = *Pitchmotor*::*OmegaOutput*
Holder:: *Phi* = *YawMotor*:: *OmegaOutput*
PitchSensor:: *Theta* = *Holder*:: *Theta*
YawSensor:: *Phi* = *Holder*:: *Phi*
MasterController:: *Theta* = *PitchSensor*:: *Theta*
MasterController:: *Phi* = *YawSensor*:: *Phi*

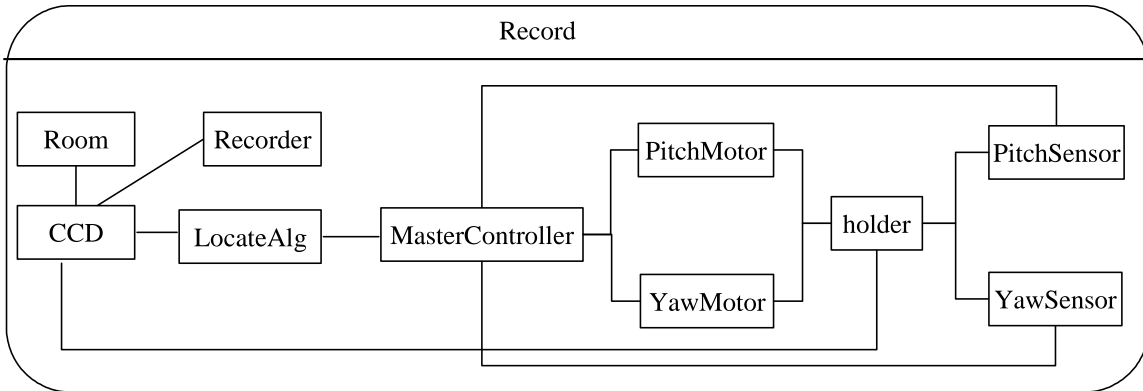
Figure 7.21: State description of "Track"



Artifact Interaction Equations

SwitchMotor:: *VoltageInput* = *MasterController*:: *LensPosition*
SwitchUnit:: *Position* = *SwitchMotor*:: *OmegaOutput*
SwitchSensor:: *AngularInput* = *SwitchUnit*:: *Position*
MasterController:: *LensPosition* = *SwitchSensor*:: *AngularOutput*
ZoomLensHolder:: *Position* = *SwitchUnit*:: *Position*
ZoomLens:: *Position* = *ZoomLensHolder*:: *Position*

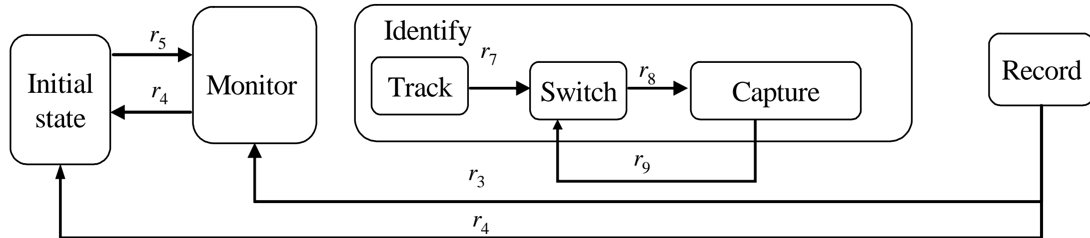
Figure 7.22: State description of “Switch”



Artifact Interaction Equations

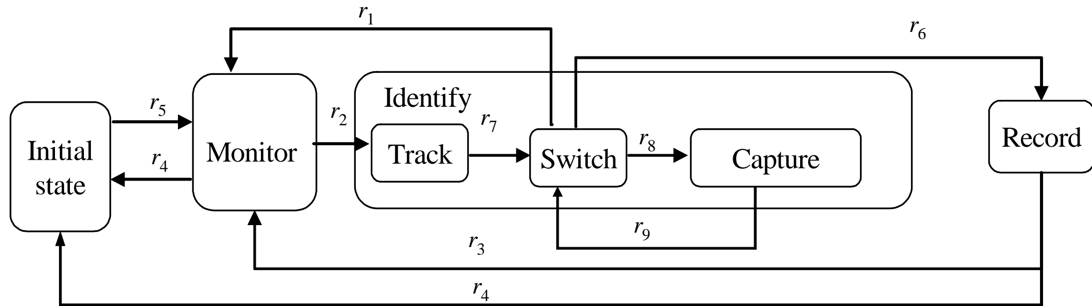
CCD::*ImageInput* = *Room*::*VisualOutput*
Recorder::*ImageInput* = *CCD*::*ImageOutput*
LocateAlg:: *ImageInput* = *CCD*::*ImageOutput*
MasterController:: *Theta* = *LocateAlg*:: *Theta*
MasterController:: *Phi* = *LocateAlg*:: *Phi*
PitchMotor:: *VoltageInput* = *MasterController*:: *Theta*
YawMotor:: *VoltageInput* = *MasterController*:: *Phi*
Holder:: *Theta* = *Pitchmotor*::*OmegaOutput*
Holder:: *Phi* = *YawMotor*:: *OmegaOutput*
PitchSensor:: *Theta* = *Holder*:: *Theta*
YawSensor:: *Phi* = *Holder*:: *Phi*
MasterController:: *Theta* = *PitchSensor*:: *Theta*
MasterController:: *Phi* = *YawSensor*:: *Phi*

Figure 7.23: State description of "Record"



Name	Condition
r_3	(*IDS*::*Timer*≥1200s) OR (*IDS*::*PersonPresent* = *FALSE*)
r_4	*IDS*::*Power*=*OFF*
r_5	*IDS*::*Power*=*ON*
r_7	(*Holder*::*Theta* = *MasterController*::*Theta*) AND (*Holder*::*Phi* = *MasterController*::*Phi*)
r_8	*MasterController*::*LensinPosition* = *TRUE*
r_9	*MasterController*::*ImageInput* ≠ *NONE*

Figure 7.24: Applying operator: generate internal transitions



Name	Condition
r_1	*IDS*::*PersonAuthorized* = *TRUE*
r_2	*Room*::*PersonPresent* = *TRUE*
r_3	(*IDS*::*Timer*≥1200s) OR (*IDS*::*PersonPresent* = *FALSE*)
r_4	*IDS*::*Power*=*OFF*
r_5	*IDS*::*Power*=*ON*
r_6	*IDS*::*PersonAuthorized* = *FALSE*
r_7	(*Holder*::*Theta* = *MasterController*::*Theta*) AND (*Holder*::*Phi* = *MasterController*::*Phi*)
r_8	*MasterController*::*LensinPosition* = *TRUE*
r_9	*MasterController*::*ImageInput* ≠ *NONE*

Figure 7.25: Applying operator: generate external transitions

7.6 Summary

This chapter presents a systematic approach to synthesizing design concepts based on the modeling framework described in Chapter 3.

This chapter describes how to represent the components library based on which design concepts are constructed. It then provides an algorithm to synthesize design concepts from the components. Stored complex components are used to simplify the synthesis process. Generated design concepts can be stored as a new complex component in order to be reused for a more complex design concept. We also show the soundness of the algorithm.

The limitations of the approach described in this chapter lie in the assumptions that have been made. Although the final design concept can have multiple states, the synthesis algorithm cannot handle components with multiple states in their behavior specifications. Future work needs to be done to relax this assumption.

Chapter 8: Transition Diagram Synthesis

This chapter has been organized in the following manner. Section 8.1 describes the main research contributions of this dissertation. Section 8.2 identifies the anticipated industrial benefits resulting from the research described in this dissertation. Section 8.3 discusses the limitations of the methods and approach described in this dissertation and provides future research directions.

8.1 Intellectual Contributions

This dissertation makes intellectual contributions in the following areas:

- **A Modeling and Simulation Framework:** We have developed a new modeling framework for representing design concepts of multiple interaction-state devices. We also describe conditions for ensuring its validity. The distinction between our approach and traditional functional representation approaches for conceptual design is as following. First, we use interactions instead of function flows or input/output flows to describe relationships between artifacts. Interactions are more general than flows. Therefore, our approach is more expressive than existing approaches. Second, we use interaction-states to capture the operating modes of a device. Hence we can support devices with multiple interaction-states. Therefore, design concepts modeled using our framework can be simulated more accurately.
- **Validation Algorithms**: We have developed a systematic approach to check the consistency of a set of interactions in an interaction-state of a mechatronic system. We also provide an algorithm to find the set of interactions that cause the inconsistency. During the conceptual design stage, the actual equations describing

the interactions are usually not known. Therefore, our algorithm utilizes the information on participating parameters to carry out its analysis. We have shown both the soundness and completeness of our algorithms. This implies that when our algorithm finds a set of interactions to be inconsistent, they are actually inconsistent. Furthermore, when our algorithm finds a set of interactions to be consistent, they are actually consistent. Even though the consistency-checking problem appears to be combinatorial, we have developed an algorithm that works in polynomial time and does not require exhaustive enumeration.

We have also developed a systematic approach to check whether a predefined unsafe parameter value set is embedded in an interaction-state. We analyze different cases in which unsafe parameter value sets can be embedded in an interaction-state and provide an algorithm to determine whether the given interaction-state is safe. This algorithm is not based on the state history and hence it can be applied to each interaction-state separately. We have shown that this approach results in a conservative analysis, i.e., when we conclude that a state is safe, it is actually safe.

- **Evaluation Algorithms**: We have developed algorithms for evaluating design concepts based on maximum power consumption and optimal component sharing. Our approach utilizes the characteristics of the new modeling framework that makes it possible for us to determine which artifacts are active in which states, and which artifacts play what roles. Therefore we can evaluate maximum power consumption more accurately and make the components sharable that play different roles but not used concurrently.

For maximum power consumption estimation we have developed a simple algorithm to generate the solution. We have proved that the optimal component sharing problem is NP-hard. We have also developed a branch and bound algorithm to find the solution for the optimal component sharing problem.

- **Synthesis Algorithms**: We utilize our modeling framework for representing known components. We utilize interaction-states transition diagram to represent behavior of complex components. Ability to model complex components allows us to utilize them in synthesizing new design concepts. We have developed a new synthesis algorithm for synthesizing transition diagrams given the desired behavior specifications and a component library. We have also shown soundness of the algorithm.

8.2 Anticipated Benefits

Conceptual design stage currently lacks computer-supported engineering design tools when compared to the detailed design stage. The problem lies in the lack of formal representation, evaluation and synthesis methods to be used during the conceptual design stage. We expect that the research reported in this dissertation will facilitate the development of computer aided design tools for the conceptual design stage, thus streamlining the design process. Specific benefits of the research reported in this dissertation include:

- **Improved support for design information archival and reuse:** Not all of the design activities require development of new designs from scratch. Actually, many "new" product designs are developed by adopting existing designs. Thus it is very important to archive design information in a computer interpretable and

formal scheme for reuse purposes. Indexed design information also facilitates quick and efficient searching for reuse. Our modeling framework supports the computer interpretable representation of multi-state mechatronic device concepts that cannot be conveniently captured by traditional approaches. Therefore, new product design could benefit from the archived design.

- **Improved support for design concept evaluation and selection:** Evaluation is important for selecting the most appropriate design option. Eliminating infeasible design alternatives in the design process as early as possible could save a significant amount of development time and money. By simulating and validating the generated design concept, we could avoid spending time and energy on developing infeasible design concepts. By comparing design concepts based on the evaluation criteria, we can identify promising design alternatives, thus reducing the search space for further exploration.
- **Design automation:** Computer aided design tools are helping designers in many ways. Computer aided design tools for conceptual design will greatly help designers in generating and selecting promising design concepts. Automated design synthesis techniques could generate design alternatives much faster. In a given amount of product development time, it allows designer to explore larger design space. Therefore it also improves the chances of finding better design solutions.

8.3 Directions for Future Work

The methods and approach described in this dissertation work have the following limitations and therefore future work is needed to extend it in those areas:

1. **Extended modeling framework:** Our modeling framework uses flat state descriptions to depict the state transition diagrams. However, when the device has hundreds of components, the flat states may not be the most efficient modeling primitives. Extensions of the state structure may be needed to handle this situation by extending the states to utilize a hierarchical structure.
2. **Design suggestion based on validation results:** Our interactions consistency checking algorithm only identifies the set of inconsistent interactions. It would be much useful if redesign suggestions were automatically generated based on the inconsistency of interactions. The representation of interactions in a graph may be utilized to provide design improvement suggestions to rearrange interactions.
3. **Richer evaluation schemes:** Current evaluation schemes only include evaluation based on maximum power consumption and optimal components sharing. Other evaluation schemes are needed such as device life estimation and device failure diagnosis. New evaluation algorithms will need to be developed for these new criteria.
4. **Synthesis using complex components with multiple interaction-state behavior specification:** Our current synthesis algorithm assumes that complex component only has one working state in its behavior specification. Extensions are needed to utilize complex components with multiple states in their behavior specifications.

Bibliography

[Akiy91] K. Akiyama. *Function Analysis: Systematic Improvement of Quality Performance*. Productivity Press, 1991.

[Arms00] J.R. Armstrong and F.G. Gray. *VHDL Design Representation and Synthesis*. Prentice Hall, 2000.

[Booc98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Press, 1998.

[Bohm04] M.R. Bohm and R.B. Stone. Representing functionality to support reuse: conceptual and supporting functions. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.

[Brac96] R.H. Bracewell and J.E. Sharpe. Functional descriptions used in computer support for qualitative scheme generation–schemebuilder. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing,* 10(4):333-345, 1996.

[Broo04] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng (eds.). HyVisual: A Hybrid System Visual Modeler. *Technical Memorandum UCB/ERL M04/18*, June 28, 2004, University of California, Berkeley, CA 94720.

[Caga01] J. Cagan. Engineering shape grammars. *Formal Engineering Design Synthesis*. Antonsson, E. K., and J. Cagan, eds., Cambridge University Press, 2001.

[Camp00] M.I. Campbell, J. Cagan and K. Kotovsky. Agent-based synthesis of electromechanical design configurations. *Journal of Mechanical Design, Transactions of the ASME*, 122(1):61-69, March 2000.

[Chak02] A. Chakrabarti, P. Langdon, Y.C. Liu, and T.P. Bligh. An approach to compositional synthesis of mechanical design concepts using computers. In *Engineering Design Synthesis: Understanding, Approaches and Tools*, Springer, 2002.

[Chan90] B. Chandrasekaran. Design problem solving: a task analysis. *AI Magazine*, 11(4):59-71, 1990.

[Chan93] B. Chandrasekaran, A. Goel, and Y. Iwasaki. Functional representation as a basis for design rationale. *IEEE Computer*, 26(1):48-56, January 1993.

[Chan94] B. Chandrasekaran. Functional representations: a brief historical perspective. *Applied artificial intelligence*, special issue on functional reasoning, 8(2):173-197, 1994.

[Chen02] L. Chen, M. Jayaram and J.F. Xi. A new functional representation scheme for conceptual modeling of mechatronic systems. In Proceedings of the *ASME Design Engineering Technical Conferences*, Montreal, Canada, September 2002.

[Corm90] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Deng99] Y.M. Deng, S.B. Tor and G.A. Britton. A computerized design environment for functional modeling of mechanical products. In Proceedings of the *Fifth ACM Symposium on Solid Modeling,* Ann Arbor, Michigan, USA, 1999.

[Deng00] Y.M. Deng, G.A. Britton and S.B. Tor. Constraint-based functional design verification for conceptual design. *Computer Aided Design*, 32(14):889-899, December 2000.

[Deng04a] Y.M. Deng and W. Liu. From function to structure and material: a conceptual design framework. In Proceedings of the *Fifth International Symposium on Tools and Methods of Competitive Engineering*, Lausanne, Switzerland, April 13-17, 2004.

[Deng04b] Y.-M. Deng and W.F. Lu. A conceptual design synthesis framework for micro-electro-mechanical systems (MEMS). In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.

[Devi94] Y. Deville and K.K. Lau. Logic Program Synthesis. *Journal of Logic Programming*, 20: 321-350, May-July, 1994.

[Diaz99] A. Diaz-Calderon, C. Paredis, and P. Khosla. Combining information technology components and symbolic equation manipulation in modeling and simulation of mechatronic systems. In Proceedings of the *1999 IEEE International Symposium on Computer Aided Control System Design*, August 1999.

[Doeb82] E.O. Doebelin. *Measurement Systems: Application and Design*. McGraw-Hill, NY, 1983.

[Dori03] D. Dori and E. Crawley. Towards a common computational synthesis framework with object-process methodology. In Proceedings of *Computational Synthesis, AAAI Spring Symposium*, Stanford, CA, March 2003.

[Dym94] C.L. Dym. *Engineering Design: A Synthesis of Views*. Cambridge University Press, 1994.

[Erdm95] A.G. Erdmann. Computer-aided mechanism design: now and the future. *Journal of Mechanical Design*, 117: 93-100, 1995.

[Erde03] Z. Erden, A. Erden and A.M. Erkmen. Petri net approach to behavioral simulation of design artifacts with application to mechatronic design. *Research in Engineering Design*, 14(1):34-46 , February 2003.

[Fenv01] S.J. Fenves. A core product model for representing design information. Technical Report, Number NISTIR6736, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2001.

[Ford56] L.R. Ford Jr., and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*. 8:399 – 404, 1956.

[Fu93] Z. Fu, A. De Pennington, and A. Saia. A graph grammar approach to feature representation and transformation. *International Journal of Computer Integrated Manufacturing*. 6(1-2):137-151, 1993.

[Gabo95] H.N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259 –273, 1995.

[Gaus01] J. Gausemeier, M. Flath and S. Mohringer. Conceptual design of mechatronic systems supported by semi-formal specification. In Proceedings of the *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, AIM*, v 2, 2001, p 888-892.

[Good02] E.D. Goodman, K. Seo, R.C. Rosenberg, Z. Fan, J. Hu, and B. Zhang. Automated design methodology for mechatronic systems using bond graphs and genetic programming. In Proceedings of the *NSF Design, Service and Manufacturing Grantees and Research Conference*, San Juan, Puerto Rico, January 2002.

[Gold88] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[Gort98] S.R. Gorti, G.J. Gupta, G.J. Kim, R.D. Sriram, and A. Wong. An object-oriented representation for product and design process. *Computer-Aided Design*, 30(7):489-501, 1998.

[Grab99] H. Grabowski, S. Rude and M. Huang. Supporting early phase of mechatronic product design with layered function models. In Proceedings of the *IEEE International Symposium on Industrial Electronics*, v 2, 1999, p 914-918.

[Grab04] H. Grabowski, R.-S. Lossack and C. Bruch. Requirements development in product design - A state and state transition-based approach In Proceedings of the *Fifth International Symposium on Tools and Methods of Competitive Engineering*, Lausanne, Switzerland, April, 2004.

[Grun00] M. Gruninger, K. Atefi, and M.S. Fox. Ontologies to support process integration in enterprise engineering. *Computational and Mathematical Organization Theory*, 6(4):381-394, 2000.

[Gurn03] A.P. Gurnani, T.K. See and K. Lewis. An approach to robust multiattribute concept selection. In Proceedings of the *ASME Design Engineering Technical Conferences*, Chicago, Illinois, USA, September 2003.

[Hare87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231-274, June 1987.

[Haze96] G.A. Hazelrigg. *System Engineering: An Approach to Information-Based Design*. Prentice hall Inc. 1996.

[Hirt01] J. Hirtz, R. Stone, D. McAdams, S. Szykman, and K. Wood. A functional basis for engineering design: reconciling and evolving previous efforts. *Research In Engineering Design,* 13(2):65-82, March 2002.

[Huan00] Y.M. Huang. On evaluation method during conceptual design. In Proceedings of the *ASME Design Engineering Technical Conferences*, Baltimore, Maryland, USA, September 2000.

[Hull02] E. Hull, K. Jackson and J. Dick. *Requirement Engineering*. Springer, London, 2002.

[Iwas93] Y. Iwasaki et al. How things are intended to work: capturing functional knowledge in device design. In Proceedings of the *Thirteenth International Joint Conference on Artificial Intelligence,* AAAI Press, San Mateo, California, USA, 1993.

[Iwas95] Y. Iwasaki, M. Vescovi, R. Fikes, and B. Chandrasekaran. Casual functional representation language with behavior-based semantics. *Applied Artificial Intelligence*, 9:5-31, 1995.

[Jaya03] M. Jayaram and L. Chen. Functional modeling of complex mechatronic systems. In Proceedings of the *ASME Design Engineering Technical Conferences*, Chicago, Illinois, USA, September 2003.

[Karg96] D.R. Karger and C. Stein. A new approach to minimum cut problem. *Journal of the ACM*, 43(4): 601–640, July 1996.

[Karn00] D. Karnopp, D. Margolis and R. Rosenberg. *System Dynamics - Modeling and Simulation of Mechatronic Systems*. John Wiley & Sons Inc., New York, 2000.

[Kont98] G. Kontonya and I. Sommerville. *Requirements Engineering: Processes And Techniques*. John Wiley & Sons Inc., New York, 1998.

[Kuma96] R. Kumar, C. Blumenrohr, D. Eisenbiegler and D. Schmid. Formal synthesis in circuit design - a classification and survey. *Lecture Notes in Computer Science*, v 1166, p 294, 1996.

[Li01] X. Li, L. Schmidt, W. He, L. Li, Y. Qian. Transformation of an EGT grammar: new grammar, new designs. In Proceedings of *ASME 2001 Design Engineering Technical Conferences*, Pittsburgh, PA, USA, 2001.

[Madh98] T. N. Madhusudan. *On Synthesis Of Electromechanical Assemblies - Automated Generation And Evaluation Of Design Alternatives*. Ph. D. Dissertation, Carnegie-Mellon University, June 1998.

[Mage99] J. Magee and J. Kramer. Concurrency: state models and java program. John Willey & Sons Ltd, 1999

[Magr97] E. B. Magrab. *Integrated Product and Process Design and Development*. CRC Press, 1997.

[Mcgo98] A. McGown, G. Green and P. A. Rodgers. Visible ideas: information patterns of conceptual sketch activity. *Design Studies*, 19 (4):431-453, 1998.

[Mile72] L. Miles. *Techniques of Value Analysis Engineering*. McGraw-Hill, 1972.

[Mroz01] Z. Mrozek. UML as integration tool for design of the mechatronic system. *Second workshop on robot motion and control*. Oct. 18-20, 2001.

[Naga92] H. Nagamochi, and T. Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *Siam Journal On Discrete Mathematics*, *5*(1):54-66, February 1992.

[Navi91] D. Navin-Chandra, K. P. Sycara, and S. Narasimhan. A transformational approach to case based synthesis. *Artificial Intelligence in Engineering, Manufacturing and Design*. 5(1): 31-45, May, 1991.

[Navi92] D. Navin-Chandra, S. Narasimhan, and K. P. Sycara. Qualitative reasoning methods in design. *Intelligent Design and Manufacturing*, A. Kusiak, Editiors, John Wiley and Sons, January, 1992.

[Pahl96] G. Pahl and W. Beitz. *Engineering Design: A Systematic Approach*. Springer-Verlag, 1996.

[Purc98] A.T. Purcell and G.S. Gero, G.S. Drawings and the design process. *Design Studies*, 19 (4):389-430, 1998.

[Qian96] L. Qian, and J.S. Gero. Function-behavior-structure paths and their role in analogy-based design. *AI EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10(5):289-312, 1996.

[Qian02] L. Qian. Creative Design by Analogy. *Engineering Design Synthesis: Understanding, Approaches and Tools*. Springer, 2002.

[Uric88] K.T. Ulrich. Computation and pre-parametric design. *Technical Report* 1043, AI Lab, MIT, Cambridge, MA.

[Saat90] T.L. Saaty. How to make a decision: the analytic hierarchy process. *European Journal of Operational Research*. 48(1):9-26, 1990.

[Sasa96] M. Sasajima, Y. Kitamura, M. Ikeda, and M. Mizoguchi. Representation language for behavior and function: FBRL. *Expert Systems with Applications*, 10(3/4):471-479, 1996.

[Schm97] L.C. Schmidt and, J. Cagan. GGREADA: A graph grammar-based machine design algorithm. *Research In Engineering Design*, 9 (4): 195-213, 1997.

[See02] T.K. See and K. Lewis. Multiattribte decision making using hypothetical equivalents. In Proceedings of the *ASME Design Engineering Technical Conferences*, Montreal, Canada, September 2002

[Shoo00] S.B. Shooter, W.T. Keirouz, S. Szykman and S. Fenves. A model for information flow in design. In Proceedings of the *ASME Design Engineering Technical Conferences*, Baltimore, Maryland, USA, September 2000.

[Srid04] P. Sridharan and M.I. Campbell. A grammar for function structures. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.

[Stac96] M. Stacey, H. Sharp, M. Petre, G. Rzevski, and R. Buckland. A representation schema to support conceptual design of mechatronic systems. *Artificial Intelligence in Design*, J. S. Gero and F. Sudweeks Editors, 603-622, 1996.

[Stah98] T.F. Stahovich, R. Davis and H. Shrobe. Generating multiple new designs from a sketch. *Artificial Intelligence*, 104 (1-2): 211-264, Sep 1998.

[Ston00] R.B. Stone and K.L. Wood. Development of a functional basis for design. *Journal of Mechanical Design*, 122(4):359-370, December 2000.

[Subr95] D. Subramanian D and C.S.E. Wand. Kinematic synthesis with configuration-spaces. *Research In Engineering Design*, 7 (3): 193-213, 1995.

[Suh90] N. Suh. *The Principles of Design*. Oxford University Press, New York. 1990.

[Szyk99] S. Szykman, J.W. Racz, and R.D. Sriram. The representation of function in computer-based design. In Proceedings of the *ASME Design Engineering Technical Conferences* (11th International Conference on Design Theory and Methodology), Las Vegas, NV, September 1999.

[Szyk01] S. Szykman, R.D. Sriram, and W.C. Regli. The role of knowledge in next-generation product development systems. *ASME Journal of Computation and Information Science in Engineering*, 1(1):3-11, 2001.

[Taka04] S. Takai and K. Ishii. Modifying Pugh's concept evaluation methods. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.

[Thom93] B. Thome (editor). *Systems Engineering: Principles and Practice of Computer-based Systems Engineering*. John Wiley & Sons, 1993.

[Tove04] M. Tovey and C. Richards. Computer representation for concept design and maintenance instruction. In Proceedings of the *Fifth International Symposium on Tools and Methods of Competitive Engineering*, Lausanne, Switzerland, April 2004.

[Ullm97] G.D. Ullman. *The Mechanical Design Process*. McGraw-Hill, New York, 1995.

[Ulri88] K. Ulrich and W. Seering. Computation and conceptual design. *Robotics and Computer-Integrated Manufacturing*, 4(3/4):309-315, 1988.

[Ulri95] K.T. Ulrich and S. Eppinger. *Product Design and Development*. McGraw-Hill, New York, 1995

[Ulri02] K.T. Ulrich and W.P. Seering. Synthesis of schematic descriptions in mechanical design. In *Engineering Design Synthesis: Understanding, Approaches and Tools*, Springer, 2002.

[Umed96] Y. Umeda et al. Supporting conceptual design based on the function-behavior-state modeler. *AIEDAM*, 10(4):275-288, September 1996.

[Umed97] Y. Umeda and T. Tomiyama. Function reasoning in design. *IEEE Expert*, 2(12):42-48 1997.

[Varg04] N. Vargas-Hernandez and J.J. Shah. 2nd-CAD: a tool for conceptual systems design in electromechanical domain. *Journal of Computing and Information Science in Engineering*, 4(3):28-36, 2004.

[Verm03] M. Verma and W.H. Wood. Functional modeling: toward a common language for design and reverse engineering. In Proceedings of the *ASME Design Engineering Technical Conferences*, Chicago, Illinois, USA, September 2003.

[Walt01] R.M. Walters. Overview of the design and development of mechatronic systems. In Proceedings of the *5th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, July 2001.

[Ward93] A.C. Ward and W.P. Seering. Quantitative inference in a mechanical design compiler. *Journal Of Mechanical Design*, 115 (1): 28-35, Mar 1993.

[Welc91] R. Welch and J.R. Dixon. Conceptual design of mechanical systems. *Design Theory and Methodology*, *American Society of Mechanical Engineers, Design Engineering Division (Publications)*, DE.31: 61-68, 1991.

[Welc94] R.V. Welch and J.R. Dixon. Guiding conceptual design through behavioral reasoning. *Research in Engineering Design*. 6(3):169-188, 1994.

[Will89] B. Williams. Invention from First Principles via Topologies of Interaction. PhD thesis, Massachusetts Institute of Technology Artifical Intelligence Lab, June 1989.

[Will92] B.C. Williams. Interaction-based design: constructing novel devices from first principles. In *Intelligent Computer Aided Design*, edited by D.C. Brown, M. Waldron and H. Yoshikawa, pages 255-274, Elsevier Science Publishers, 1992.

[Wood01] K.L. Wood and J.L. Greer. Function-based synthesis methods in engineering design: state of the art, methods analysis, and visions for the future. *Formal Engineering Design Synthesis* (edited by E.K. Antonsson and J. Cagan), Cambridge University Press, 2001.

[Yang03] M.C. Yang. Concept generation and sketching: correlations with design outcome. In Proceedings of the *ASME Design Engineering Technical Conference*, Chicago, IL, USA, September 2003.

[Zein04] A. Zeiny. Computable dynamic design repository for product data representation. In Proceedings of the *ASME Design Engineering Technical Conference*, Salt Lake City, Utah, USA, September 2004.