A Design Tool for Reusing Integration Knowledge in Simulation Models

by

Sangmok Han

B.S. Mechanical Engineering
Seoul National University, 2000

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN MECHANICAL ENGINEERING
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2006

Signature of Author . . . .
Department of Mechanical Engineering
May 12, 2006

Certified by. . . . .
David Wallace
Esther and Harold E. Edgerton Associate Professor
Thesis Supervisor

Accepted by. . . . .
Lallit Anand
Chairman, Department Committee on Graduate Students

A Design Tool for Reusing Integration Knowledge in Simulation Models

By

Sangmok Han

Submitted to the Department of Mechanical Engineering
on 12 May 2004, in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering

Abstract

In the academic field of computer-aided product development, the role of the design tool is to support engineering designers to develop and integrate simulation models. Used to save time and costs in product development process, the simulation model, however, introduces additional costs for its development and integration, which often become considerably large due to the fact that many, complex simulation models need to be integrated. Moreover, the result of integration and the effort taken during the integration process are often not reused for other product development projects. In this paper, we attempt to develop a design tool that can capture integration knowledge and make the knowledge reusable for other design tasks. More specifically, we are interested in the two kinds of integration knowledge: the first captured in the form of a graph structure associating simulation models, called the integration structure, and the second generalized from script codes into rule-based patterns, called the integration code pattern. An integration mechanism and a pattern generalization algorithm have been developed and incorporated into a design tool utilizing a new integration model called *catalog model*, a model that enables us to reuse the integration structure and code patterns of one model to quickly build another. Application scenarios have demonstrated the effectiveness of the design tool: The same integration task could be performed in less time, and repetitive and error-prone elements in the task were substantially reduced as a result of reusing integration knowledge in the simulation models.

Thesis Supervisor: David Wallace
Title: Esther and Harold E. Edgerton Associate Professor

Contents

List of Figures

Chapter 1 Introduction

1.1. Motivation

As foreseen by social scientists, both aging population and unemployment are likely to be growing sources of social problems in many countries. Interestingly, technological development and free-market competition are causal forces that lead to improvement in quality of life while simultaneously giving rise to an aging population and reduced labor needs. Even though no one yet has come up with a clear solution for this dilemma, it can be generally agreed upon that we need to develop more job opportunities that create greater value by taking advantage of exclusively human capabilities, which are not subject to replacement by automation.

Design, be it of software, machines, or organizations, is one of the novel tasks that rely heavily on human creativity and experience, so I conjecture that designers and inventors will be increasingly important in the future. However, design is often thought to be a task best suited to talented, younger people — yet future design problems are likely to become increasingly complex, combining diversified customer needs, converging product functionalities, and rising environmental concerns. A challenging mission for design tool technologies of the future will be to allow a wider range of designers to creatively tackle increasingly complex design problems.

To address this challenge, I believe that a high priority should be placed on advancing two key functional attributes of design tools: flexibility and usability. Both functionalities aim at efficiently creating complex models, but take different approaches. The role of flexibility is to make a design tool manage changes in requirements and configurations of simulation models efficiently. The more flexible a tool is, the more complex problems designers can tackle, without being overwhelmed by redundant, repetitive tasks introduced by design requirement changes or design exploration for a better product configuration. Meanwhile, usability seeks for compatibility with the human cognitive system. Based on the assumption that the interaction between a computer and a human is most efficient when the computer has an interface compatible with the underlying mechanism of the brain and thereby can utilize most capabilities of the brain, the role of usability is to deliver complex information in an easily digestible form, preferably customized for each designer. That way, the thought processes of designers will not distracted by procedural, tool-operation issues.

In this thesis, we are interested in design tools whose role in the product development process is to help engineering designers develop and integrate simulation models. While most design tools have competent at creating simulation models, few tools address the issue of integration effectively because they does not provide an adequate level of flexibility to meet variability in modern product development process. Once a simulation model is made from several simulation model components, the components are tied to each other and cannot flexibly adapt to subsequence modification, which is often required to reflect customer requirement changes or to explore alternative design options like purchasing a standardized machine part from different vendors. Such a modification often result in budget overruns or schedule delays because a change in one simulation model has far-reaching effects on other models and making them all consistent requires considerable amount time and effort. Moreover, it is a tedious task requiring no creative thinking and thus avoided by most designers.

A usable design tool supporting the reuse of integration knowledge in simulation models is promising solution to the current problem. We assume that the problem, the lack of flexibility in simulation models, originates from the missing support of design tools. Because current design tools do not allow designers to reuse integration knowledge in previously built simulation models, much of integration effort taken for one simulation model needs to be repeated when some components need to be replaced or modified. We hope a design tool to be able to capture the integration knowledge in simulation models in a reusable form, so that it can be applied for building another integration model. Because time and cost wasted in redundant, repetitive tasks can be greatly saved, the product development process can employ more design iterations as well as extensive exploration of design alternatives. Reusing integration knowledge in simulation models also improves the usability of the design tool because it can provide valuable information that can save typing and prevent mistakes when combined with appropriate visualization and interaction techniques. Consequently, this thesis aims at developing mechanism, algorithm, and user interface for a design tool that enables us to capture integration knowledge in simulation models for another use.

1.2. Concepts and Terminology

Before we begin our discussion, this section is to establish a common ground on terminology. Some terms used to describe our goal and contributions need to be defined in a narrower sense than their general use because they can have various meanings. They include simulation model, integration model, integration knowledge, relation and mapping.

Simulation model refers to a computer program that attempts to simulate behaviors of a particular system. All simulation models in our discussion are assumed to be parametric models and therefore their behaviors are exhibited through output parameters whose values are determined by submitted values of input parameters. When the behavior of a simulation model is implemented by subscribed behaviors of other simulation models, we call the implemented simulation model as the *integration model.*

As the title of the thesis – reusing integration knowledge in simulation knowledge – implies, we use integration knowledge to refer to something that is inside a simulation model that can be reused for building another. *Integration knowledge* is defined as information extracted from data in an integration model, where the information is assumed to reflect knowledge used by human during the integration process. In other word, we define integration knowledge as tangible information such as data structures or patterns that can be extracted from the model data. It is a practical definition because such tangible information not only serves sufficiently to solve reuse problems of our interest but it also eliminates the need for solving unnecessarily complicated problems of representing knowledge in a computer understandable way.

Relation is a modeling element of the simulation model to define how input parameters are transformed to output parameters. We have two kinds of relation: *local relation* and *remote relation.* The local relation is a relation which includes script code, called *relation script*, and transforms parameters using the script code. The remote relation is a relation whose transformation is defined by subscription of a model interface. *Model interface* is a port, represented by a set of input and output parameters, through which a simulation model interacts with other modeling elements. When a simulation model is executed, values submitted to the input parameters are transformed into those of the output parameters.

Mapping is another modeling element of the simulation model to describing how a parameter in one relation, called *mapping target parameter*, is related to parameters in other relations, called *mapping source parameters.* Script code called *mapping script* is used to define transformation between the mapping target parameter and the mapping source parameters. The mapping script is different from the relation script in the local relation in that it is a line of script code whose evaluation result is assigned to the mapping target parameter.

1.3. Goal

The goal of our new design tool is to reuse two kinds of integration knowledge: *integration structure* and *integration code pattern.*

The integration structure is a graph structure in the integration model describing how parameters are mapped to each other. The graph structure is used to coordinate local and remote relations so that output parameters in the model interface of the integration model give the intended transformation of input parameters in the model interface. We suppose that the graph structure reflects a designer's knowledge on how to organize a given set of local and remote relations to achieve the intended transformation between input and output parameters in the model interface. It is because we expect the designer to apply the same knowledge when he or she is asked to integrate a similar set of local and remote relations, which have the same interface as the previous set but exhibit different behaviors. Therefore, the integration structure is considered as integration knowledge: its graph structure is extracted from modeling data, and the structure reflects a designer's knowledge used in the integration process.

Integration code pattern refers to a pattern generalized from mapping scripts, which is represented as a set of rules describing regularity found in the scripts. We assume that the regularity results from applying the same knowledge on naming and indexing convention over several mapping scripts; thus the set of rules is considered to reflect a designer's knowledge used in the integration process. For example, in a situation in which an engineering designer has a set of parameters that needs a similar mapping such as *free_voltage* mapped to *freeVolt*, and *stall_voltage*mapped to *stallVolt*, we assume that the person will perform those mappings thinking they have a consistent pattern among parameter names. As a result, when the person is asked to create another mapping for *average_voltage*, the answer *averageVolt* can be created by re-applying his or her knowledge on the regular naming pattern, which is often called naming convention.

The use case scenario in Chapter 2 will explain how these two aspects of integration knowledge are reused in the simulation building process, highlighting the benefit from reusing them.

1.4. Contributions

We categorize contributions of our research in two areas: intellectual and implementational contributions. Intellectual contributions include new ideas and algorithms, while implementational contributions include mechanisms and designs, which have been adapted from other disciplines to serve our needs.

- Intellectual contribution
  - □ Identification of two reusable integration knowledge: Integration structure and integration code pattern
  - □ Dependency solving algorithm: The algorithm generates an execution sequence of relations and mappings: The algorithm works with a new set of relational operators: the relation and the mapping with transformation.
  - □ Pattern generalization algorithm: The algorithm identifies reusable integration code patterns and generalizes the patterns into a rule set
- Implementational contribution
  - □ Implementation dispatch mechanism for the catalog model: This mechanism is used to decouple implementation from interface and to dynamically associate them in the run-mode. Conceptually, it is similar to a mechanism used for implementing polymorphism in object-oriented languages. While the object-oriented languages use a dispatch table approach, our mechanism uses a script generation approach.
  - □ User interface design for the catalog model builder: The graphical user interface of our design tool is specialized in reusing integration knowledge in simulation models. It utilizes techniques developed in the user interface design discipline to provide an

effective user interface.

1.5. Overview

Chapter 2 describes how our design tool achieves two goals of reusing integration knowledge. In the first scenario, the integration structure is reused to explore alternative design options. The scenario is based on an integration model for estimating the performance of power window system. The second scenario describes what kinds of integration code patterns can be captured and how we can reuse them to save integration efforts.

Chapter 3 reviews research efforts taken in other engineering disciplines to find useful concepts and techniques that can be employed to achieve our goal of reusing integration knowledge. We investigate research works done in the areas of software engineering, source code mining, and user interface design.

Chapter 4 describes how we build our design tool that can reuse two kinds of integration knowledge. The first section of the chapter provides an overview of software components in the design tool, and the proceeding sections give details on the four major software components: catalog model definition, dependency solving algorithm, pattern generalization algorithm, and catalog model builder.

Chapter 5 describes several integration scenarios, through which we validate the effectiveness of reuse features of our design tool. The integration scenarios include the door seal catalog browser, the evaluation of a power window system with various configurations, and the finite element analysis (FEA) of the door seal with multiple levels of fidelity.

Chapter 6 presents a concluding remark of this thesis. Future directions of research are discussed.

Chapter 2 Reuse Scenarios

The use case scenarios in this chapter illustrate how our design tool achieves two goals of reusing integration knowledge. The first scenario demonstrates how the integration structure is reused in a step-by-step way. The second scenario presents several integration codes containing exemplary textual patterns and shows how they can be reused.

2.1. Reuse of the Integration Structure

2.1.1. Scenario terminology

In this section, we introduce several new terms that will be used in the integration structure reuse scenario as follows: *implementation*, *implementation switch*, *mapping with transformation*, *run-mode*, and *build-mode*. The implementation refers to a collection of parameters, relations, and mappings, which are coordinated to realize a specific behavior of a simulation model. In a new integration model supporting the reuse of integration structures, a model can have multiple implementations for one model interface. The implementation switch is a parameter placed in an integration model that diverts the source of model behaviors from one implementation to another. While typical mapping just copies the value of a source parameter to a target parameter, the mapping with transformation is enhanced in that it can perform mathematical transformation of source parameters and assign the result value of transformation to a target parameter. The run-mode refers to a state of a simulation model after the model is deployed in the model container of an integrated design framework. The run-mode is used in contrast with the build-mode, indicating a state of a simulation model before deployment and possibly under modification.

2.1.2. Scenario of reusing the integration structure

A company develops an integration model that analyzes the performance of a power window system used in a car door. The overall configuration of the integration model is shown in Figure 2-1. The integration model is aimed at finding the stall force – a force imposed by a window when its movement is blocked by an object – and the maximum speed of the window. It subscribes a window geometry model to find geometry data such as pillar lengths and glass widths. It also subscribes a motor performance model giving torque and rotational speed and a window guide model calculating the stall force and the maximum speed. Each subscribed model interface has three to twenty parameters. Thos parameters are mapped to each other in the integration model, and as a result a fairly complex integration model having four relations and forty seven mappings is created.

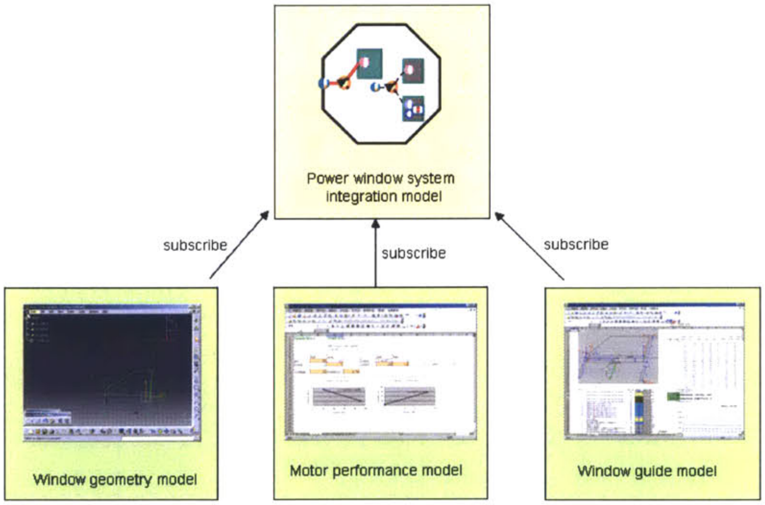
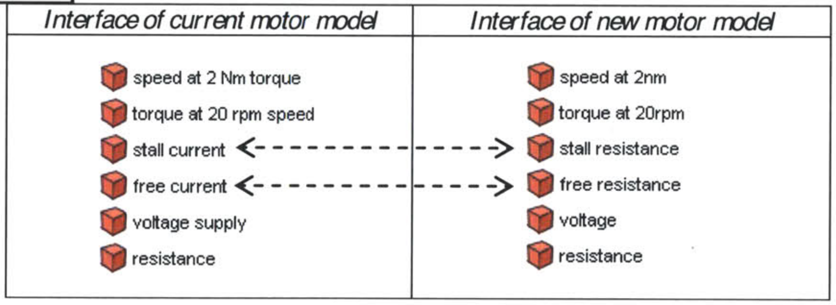


Figure 2-1: Simulation models for a power window system

After the company evaluates the performance of the current power window system, they are interested in exploring design alternatives. More specifically, they are interested in a motor from a new supplier that provides maximum torque and maximum rotation speed similar to the current motor but has different characteristics in terms of acceleration and electric resistance. They want to know how the system's performance will change if they replace the current motor with the new motor. Assuming that the company can obtain the new supplier's motor simulation model, two challenging aspects of the replacement task are observed. First, they want to replace the motor simulation model without having to rebuild, meaning to create a new integration model and define all the mappings, the power window system integration model. They want to reuse the integration structure they have defined in the model: the integration structure refers to a graph composed of all the mappings and relations defined in an integration model. Second, the simulation model provided by a new supplier has a slightly different model interface such as it gives resistance instead of current (Figure 2-2) so mathematical transformation on those parameters will be needed.



Interface of current motor model	*Interface of new motor model*
speed at 2 Nm torque	speed at 2nm
torque at 20 rpm speed	torque at 20rpm
stall current <- - - - - - - - - - - - - ->	stall resistance
free current <- - - - - - - - - - - - - ->	free resistance
voltage supply	voltage
resistance	resistance

Figure 2-2: Comparison between two motor simulation models

As a first step, the company creates an integration model called *motor catalog model*, which has the same model interface definition as the current motor model, and maps all parameters in the current motor model to the corresponding parameters in the model interface. Now the motor catalog model has the current motor simulation as one of its implementation. The second step is to add the new motor simulation model as another implementation of the motor catalog model (Figure 2-3) and to set up necessary mappings between parameters in the new motor simulation model and those in the catalog model interface. The last step is to replace the motor simulation model in the power window system integration model with the new motor catalog model and set its implementation switch to the implementation added in the second step. Since both of the replaced and replacing models have the same interface definition, this step can be done without affecting any mappings or relations in the power window system integration model. In other words, the integration structure of the power window system integration model is reused for another configuration of the integration model.

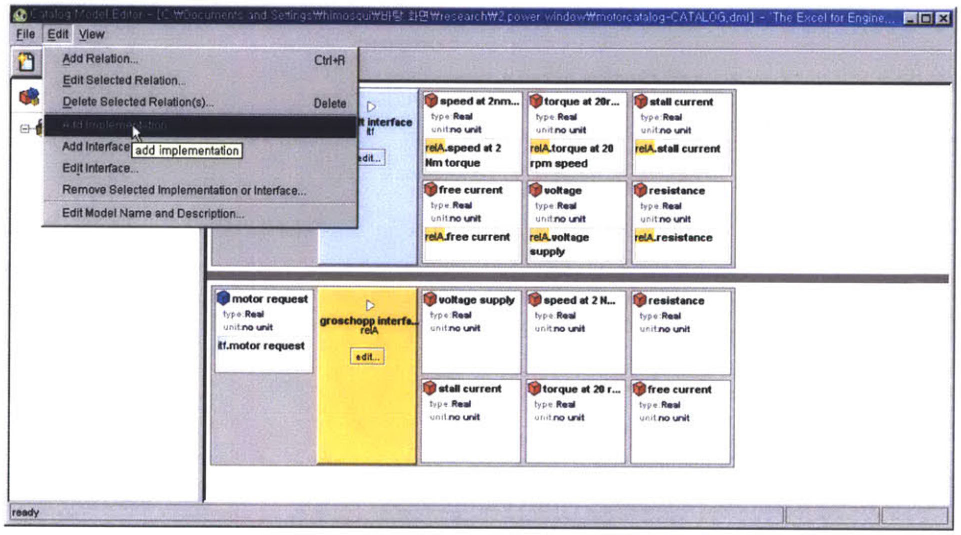
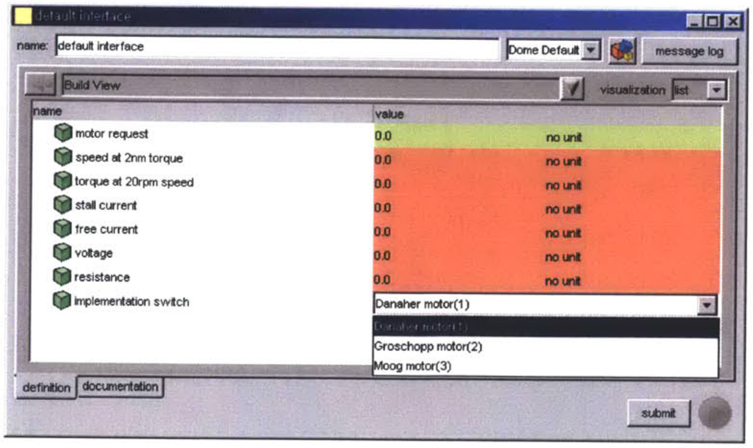


Figure 2-3: Adding a new implementation to motor simulation model

Another benefit from using the motor catalog model instead of the previous typical motor simulation model is that we can switch and execute different implementations in the run-mode. While we had to go back to the build-mode to change the configuration of the power window system and re-deploy the modified integration model, now just changing the value of the implementation switch in the run-mode will activate the selected implementation (Figure 2-4 and Figure 2-5).



default interface

name: default interface

Dome Default

message log

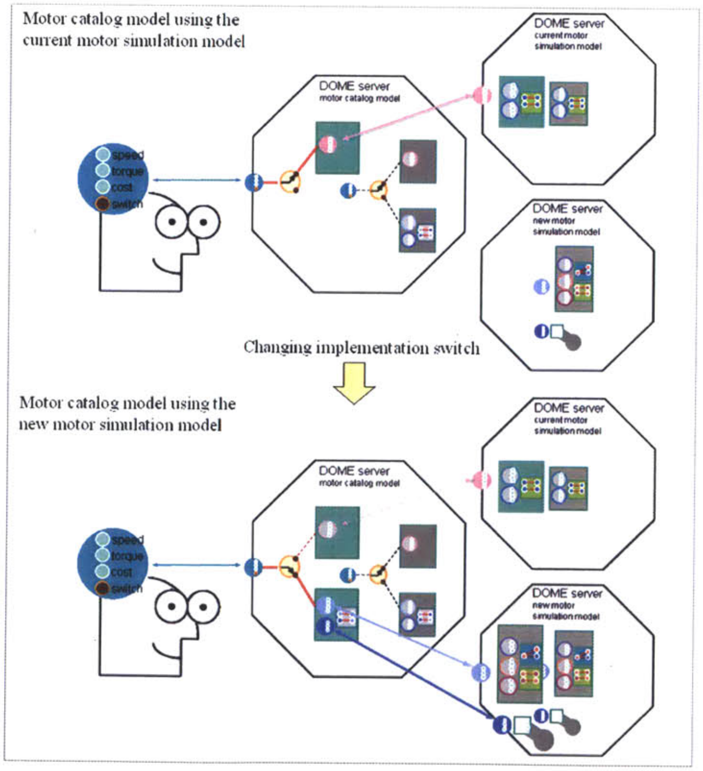
Build View

visualization: list

name	value	
motor request	0.0	no unit
speed at 2nm torque	0.0	no unit
torque at 20rpm speed	0.0	no unit
stall current	0.0	no unit
free current	0.0	no unit
voltage	0.0	no unit
resistance	0.0	no unit
implementation switch	Danaher motor(1)	

Danaher motor(1)
Groschopp motor(2)
Moog motor(3)

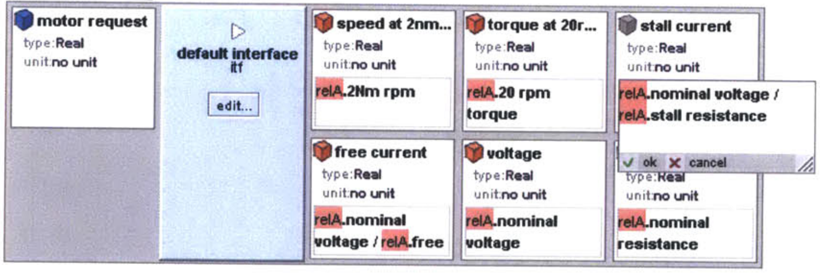
Figure 2-4: Implementation can be switched to another in run-mode



Motor catalog model using the current motor simulation model
DOME server
motor catalog model
DOME server
current motor simulation model
DOME server
new motor simulation model
speed
torque
cost
switch
Changing implementation switch
Motor catalog model using the new motor simulation model
DOME server
motor catalog model
DOME server
current motor simulation model
DOME server
new motor simulation model
speed
torque
cost
switch

Figure 2-5: By changing the implementation switch, the configuration of the power window system can be changed from using the current motor simulation model to using the new motor simulation model.

The difference between two subscribed model interfaces is handled by the mapping with transformation. The new motor model provides *nominal voltage* and *stall resistance*, and therefore *stall current* can be calculated from them by dividing voltage with resistance. One method of doing such a transformation is to write a procedural relation that transforms voltage and resistance into current. It requires us to create several modeling elements – one relation, three parameters and three mappings – as well as to specify the causality information. While we are defining a simple transformation, this method increases the complexity of the integration model considerably; therefore it is not considered very efficient. However, with the new integration model, we can do the same task in a simpler way because it supports the mapping with transformation. The mapping with transformation allows us to write a line of mathematical equation in the mapping script editor, and the equation is used to transform source parameter values into a target parameter value. Therefore, a task that used to require several modeling elements can be done with one mapping with transformation (Figure 2-6).



motor request
type:Real
unit:no unit

default interface
itf

speed at 2nm...
type:Real
unit:no unit
relA.2Nm rpm

torque at 20r...
type:Real
unit:no unit
relA.20 rpm torque

stall current
type:Real
unit:no unit
relA.nominal voltage / relA.stall resistance

free current
type:Real
unit:no unit
relA.nominal voltage / relA.free

voltage
type:Real
unit:no unit
relA.nominal voltage

type:Real
unit:no unit
relA.nominal resistance

Figure 2-6: Using mapping with transformation to handling differences in interface definition

As a result, we can handle the difference in model interface definition in an efficient, manageable way. Note that a parser embedded in the mapping script editor generates causality information for the equation, and users can save time and effort spent on specifying trivial causality information.

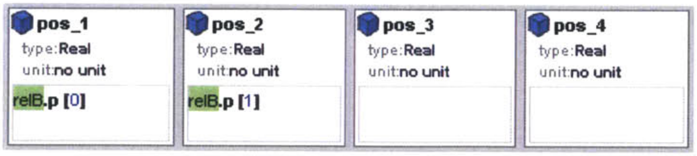
2.2. Reuse of the Integration Code Pattern

2.2.1. Exemplary reuse problems

Before we describe the reuse scenario of integration code patterns, several exemplary integration code reuse problems will be presented to clarify the focus of our interest in reuse, and they will be solved by examining regularity existing in the integration code.

In Problem 1, we have four parameters, two of which have their mapping scripts completed (Figure 2-7). What would be the mapping scripts for *pos_3* and *pos_4* then? The answer would be *relB.p[2]* and *relB.p[3]*. One justification is for this intuitive reasoning is the regularity found in two completed mapping scripts, which is called the integration code pattern or the code pattern in short. The code pattern for this problem is expressed using the following set of rules: Text part of the script, *relB.p[* and *]*, is the same for both mapping scripts, and the number inside brackets is one smaller than the number comes after *pos_*.

Builder GUI represetnation



pos_1
type:Real
unit:no unit
relB.p [0]

pos_2
type:Real
unit:no unit
relB.p [1]

pos_3
type:Real
unit:no unit

pos_4
type:Real
unit:no unit

Text representation

pos_1 = relB.p[0]
pos_2 = relB.p[1]
pos_3 = ?
pos_4 = ?

Figure 2-7: The first example of code pattern generalization problem

Problem 2 has four parameters, three of which have mapping scripts (Figure 2-8). What would be the mapping script for *a_3_5*? Again it is not very difficult to infer that the answer would be *relB.q[3][5]*. We see regularity in three mapping scripts that two numbers inside brackets are the same as the first and second number in the parameter name.

Builder GUI represetnation	*Text representation*

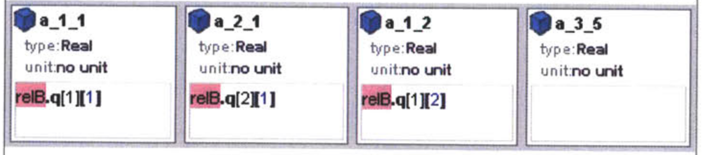

a_1_1 type:Real unit:no unit relB.q[1][1]
a_2_1 type:Real unit:no unit relB.q[2][1]
a_1_2 type:Real unit:no unit relB.q[1][2]
a_3_5 type:Real unit:no unit
	a_1_1 = relB.q[1][1]
	a_2_1 = relB.q[2][1]
	a_1_2 = relB.q[1][2]
	a_3_5 = ?

Figure 2-8: The second example of code pattern generalization problem

Problem 3 have three parameters with one mapping scripts not completed (Figure 2-9). We are interested to know what would be the mapping script for *newDepth*. Because two mapping scripts show that a string coming after *new* in the parameter name is the same as a string that comes after *old*, the answer would be *oldDepth * 0.5*.

Builder GUI represetnation	*Text representation*

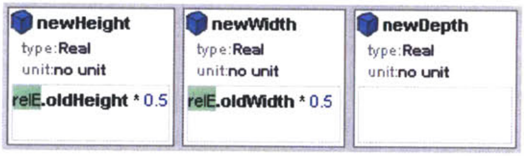

newHeight type:Real unit:no unit relE.oldHeight * 0.5
newWidth type:Real unit:no unit relE.oldWidth * 0.5
newDepth type:Real unit:no unit
	newHeight = originHeight * 0.5
	newWidth = originWidth * 0.5
	newDepth=?

Figure 2-9: The third example of code pattern generalization problem

In Problem 4, ten mapping scripts are given, and we are asked to fill a mapping script for *B_7* (Figure 2-10). While previous problems could be explained by one set of rules that applies to whole examples codes, we can't find such a rule in this case. This problem is not as intuitive as the last three problems, and it may seem unnecessarily complicated. However, this is a more realistic case of the integration code pattern reuse. It reflects the complexity of the environment in which the code pattern reuse should operate: A user provides no explicit information other than which parameter needs code completion. Also, among many mapping scripts in a simulation model, only some of them can be generalized into a rule set that can generate valid candidate, while others can be generalized but their candidate may not be compatible with a given code completion – as we will see in the last solving step of Problem 4, candidates generated by the first and the third rule sets begin with A_7, which are not acceptable because it is not consistent with the given parameter name B_7 – still others may not even be generalized into a rule set. After several trial-and-error to generalize rules, B_7=B[7]*2+B[8]*2 is found to be an answer based on justification as follows.

Builder GUI represetnation	Text representation

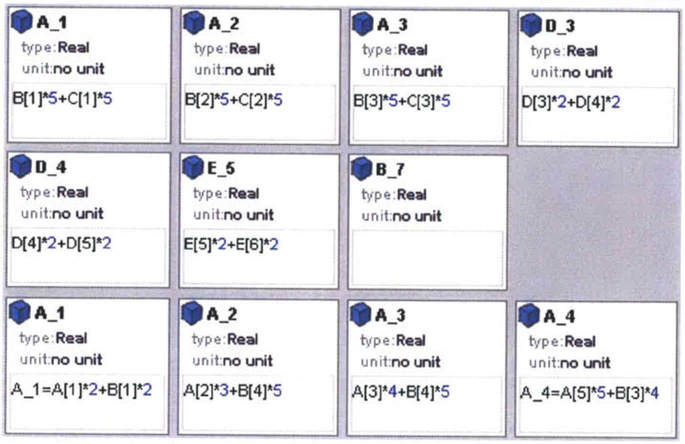

A_1 type:Real unit:no unit B[1]*5+C[1]*5
A_2 type:Real unit:no unit B[2]*5+C[2]*5
A_3 type:Real unit:no unit B[3]*5+C[3]*5
D_3 type:Real unit:no unit D[3]*2+D[4]*2
D_4 type:Real unit:no unit D[4]*2+D[5]*2
E_5 type:Real unit:no unit E[5]*2+E[6]*2
B_7 type:Real unit:no unit
A_1 type:Real unit:no unit A_1=A[1]*2+B[1]*2
A_2 type:Real unit:no unit A[2]*3+B[4]*5
A_3 type:Real unit:no unit A[3]*4+B[4]*5
A_4 type:Real unit:no unit A_4=A[5]*5+B[3]*4
	A_1=B[1]*5+C[1]*5
	A_2=B[2]*5+C[2]*5
	A_3=B[3]*5+C[3]*5
	D_3=D[3]*2+D[4]*2
	D_4=D[4]*2+D[5]*2
	E_5=E[5]*2+E[6]*2
	A_1=A[1]*2+B[1]*2
	A_2=A[2]*3+B[4]*5
	A_3=A[3]*4+B[4]*5
	A_4=A[5]*5+B[3]*4
	
	B_7=?

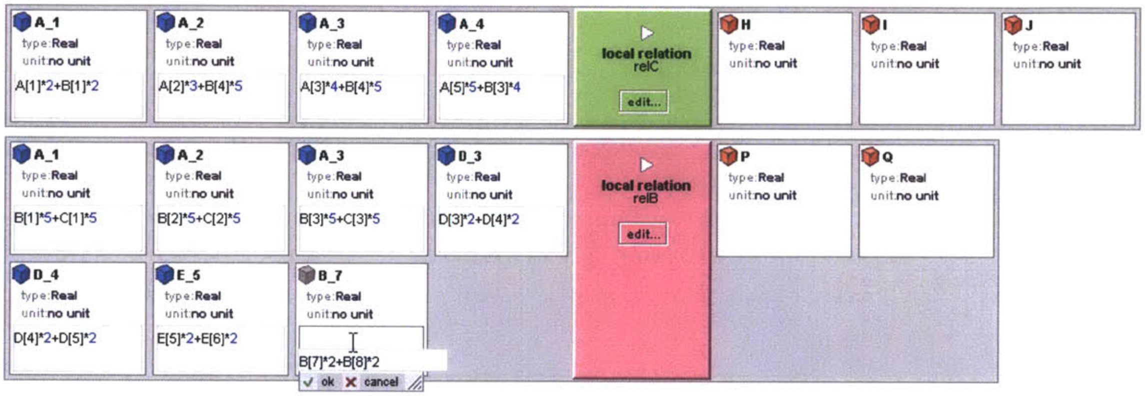
Figure 2-10: The fourth example of code pattern generalization problem

We need three sets of rules to explain all mapping scripts, or all lines in text representation. The first three lines are explained by a set of rules: "A, B, C, 5, _, *, [, and] are the same for each line" and "The number after _ is repeated for two numbers in the brackets." The second set of rules explains the next three lines: "The same text string such as D or E is repeated three times – one at the beginning and two before two [", "The number inside the first brackets is the same as the first number after _, while the number inside the second brackets is one bigger than it," and "other numbers are the same as 2 for each line." Similarly, another set of rules explains the remaining four lines. Given these three set of rules, now we can create three possible answers. The first and the third rule sets give A_7=B[7]*2+B[8]*2 and A_7=A[7]*8+B[${N+0}]*${N+1}, in which ${N+0} is a placeholder for an undetermined number that should be one smaller than the number at ${N+1}. Although the two rule sets have created candidates, their candidates are not acceptable because they suggest A_7 to be the parameter name, not B_7. Only one candidate generated from the second rule set is consistent with the given parameter name *B_7*, and finally *B_7=B[7]*2+B[8]*2* is displayed as a candidate for the mapping script.

2.2.2. Scenario of reusing the integration code pattern

Assuming a pattern generalization algorithm that can systematically solve problems described in Section 2.2.1 has been developed, a user will go through the following steps to reuse the integration code pattern. We suppose that a user is working on the same simulation model used in Problem 4. The user just have finished writing mapping scripts for parameter *D_4* and *E_5* and is about to write a mapping script for *B_7*. If the design tool did not support automatic code completion based on code pattern reuse, she would copy the mapping script of *D_4*, *D[4]*2+D[5]*2* and modify it to *B[7]*2+B[8]*2*. Even though changing characters and index numbers seems to be a trivial task to do, people often miss to modify some indexes and characters, creating a bug in the simulation model. The bug not only generates unexpected, wrong results, but it also requires considerable time and effort to be fixed, especially when it is an index-related bug.

Now that the design tool offers code completion feature, enabling the user to reuse the integration code pattern, the user decides to use it. After having applied the same pattern for the two mapping scripts of *D_4* and *E_5*, the user activates the code completion feature to generate a mapping script for *B_7*. The code completion feature is accessible either by hitting a space key with control down or clicking a mapping script editor with control down. The list of code completion candidates pops up in the mapping script editor (Figure 2-11). The user finds that *B[7]*2+B[8]*2* is the only code completion candidate. After selecting the candidate, the user accepts it either by double-clicking the candidate or by hitting an enter key. Note that the user does not have to provide any extra information to the computer other than which parameter needs code completion. Information such as which mapping scripts should be used as a reference of the code pattern generalization might be helpful for the code pattern generalization algorithm to solve the problem easily, but requiring such information each time of code completion would make the code pattern reuse process inconvenient and inefficient, possibly damaging usability of the integration code pattern reusing feature. When the code completion finds many candidates, the user has an option to narrow down the candidates by typing more literals, which match the beginning of the desired mapping script.



A_1
type:Real
unit:no unit
A[1]*2+B[1]*2

A_2
type:Real
unit:no unit
A[2]*3+B[4]*5

A_3
type:Real
unit:no unit
A[3]*4+B[4]*5

A_4
type:Real
unit:no unit
A[5]*5+B[3]*4

local relation
relC

H
type:Real
unit:no unit

I
type:Real
unit:no unit

J
type:Real
unit:no unit

A_1
type:Real
unit:no unit
B[1]*5+C[1]*5

A_2
type:Real
unit:no unit
B[2]*5+C[2]*5

A_3
type:Real
unit:no unit
B[3]*5+C[3]*5

D_3
type:Real
unit:no unit
D[3]*2+D[4]*2

local relation
relB

P
type:Real
unit:no unit

Q
type:Real
unit:no unit

D_4
type:Real
unit:no unit
D[4]*2+D[5]*2

E_5
type:Real
unit:no unit
E[5]*2+E[6]*2

B_7
type:Real
unit:no unit
B[7]*2+B[8]*2

Figure 2-11: Reusing integration code pattern: after activating code completion

Chapter 3 Related Work

This chapter reviews most relevant branches of research that address the issues of software reusability. These include software engineering, user interface design, and data mining. Section 3.1 describes concepts and techniques developed in those disciplines. Section 3.2 discusses the current support of reusability in major integrated design frameworks, identifying what kind of features are missing or need improvement.

3.1. Approaches

3.1.1. Software engineering approach to reusability

Simulation models being a kind of software, software engineering concepts developed for software reusability is helpful to solve our problem of reusing simulation models. Some of the useful concepts are found in object-oriented programming, which is the most popular programming paradigm to date. Given the list of concepts that characterize the object-oriented programming [1], we have chosen the following concepts since they are closely related to the issue of reusability: instantiation, polymorphism, and inheritance

Instantiation is a concept on the way how an object is created. In a language supporting instantiation, multiple objects can be created from a class. Because behaviors and data structures defined for one class can be reused to create many objects, it improves reusability [2]. Polymorphism is a mechanism that allows multiple implementations to be associated with a class [3]. It decouples the external representation of a class, called the interface, from its internal implementations; therefore, programmers can modify implementations without affecting other classes using the class. To implement polymorphism, behaviors of a class need to be chosen based on the target of the invocation. The implementation dispatching mechanism is used to achieve this in most object-oriented languages [4]. Inheritance is also related to software reusability because it suggests that specialized behaviors of a sub-class be built on the implementation of super-classes.

In addition to concepts derived from object-oriented programming, we have other software engineering concepts, which are also essential to achieve software reusability: composition and interoperability. Composition a concept that allows simple software modules to be combined to build up a complex one [5]. Because it provides a simple, yet powerful way of reusing software components, it is supported by most programming languages. Interoperability is also an important concern for reusability since a software component cannot be reused if it is not accessible to other software components. Some of existing solutions addressing the interoperability issue include COM [6], CORBA [7], and SOAP [8].

3.1.2. Source code mining approach to reusability

Source code mining is an application of data mining techniques to find relationships and correlations hidden in a large amount of source code [9]. While software engineering approaches described in Section 3.1.1 try to invent new mechanism for reusability and embed it in a programming language, code mining approach focus on how we can use a given programming language in a better way promoting reusability. For this reason, code mining approaches have developed a range of tools that help programmers produce reusable code.

Two categories of the code mining tool are reviewed in detail because they address the similar technical issues we have to deal with when implementing our feature for reusing integration code patterns. The first category is a tool that detects undesirable, not-easily-reusable code fragments, called *code clones*. The code clone is a code fragment the same or equivalent copies of which are found in another part of source code [10]. It has been reported that code clones degrade the reusability and maintainability of the software [10-14]. The second category is a tool that helps programmers to find reusable code from a repository of source code. Because the tool automatically completes the remaining part of source code a programmer is working on, it is called *example-based code completion tool.*

Research on the code clone detection tool has focused on developing fast and reliable algorithm that can locate the same or equivalent code fragments. We identify two broad approaches, namely, lexical-analyzer-based and textual-pattern-based. The lexical-analyzer-based approach employs a parser to transform a source code into a specialized data structure for clone detection [12, 14]. Such an approach is computationally more expensive than the textual-pattern-based, but has an advantage in detecting non-contiguous code clones because lexical information, such as program dependence graph [14] depicting a relationship among non-contiguous code fragments, can be used. Some of the initial works done in the textual-pattern-based approach suffer from not being able to detect code clones that contain slight modification or span over multiple lines [11, 15]. Later works address this problem by utilizing token-based representation [10, 13, 16]. Because the textual pattern-based approach does not require a parser, which needs to be developed for each programming language, it can be applied in language-independent manner [17], an advantageous aspect over the lexical-analyzer-based approach.

Several example-based code completion tools have been developed based on the code mining technique [18-21]. Code completion is achieved by a two-step process: building a code example repository and querying relevant code from the repository. To perform the first step, each code completion tool has been found to be using different data representation of source code depending on the search algorithms they employ. A code completion tool described in [20] represents source codes with a graph of method signatures because it generates code completion candidates based on a graph search algorithm, while another presented in [21] stores vectors because its search algorithm is based on structural similarity measured by the vector distance. Regarding the second step, most tools address the issue of automatic generation of search queries [18-21] because it is an essential feature for code completion feature to be usable: Few programmers would use code completion feature if they have to learn new query syntax and write a query each time they use it [20].

3.1.3. User interface design approach to reusability

User interface design is a technique to improve usability of an artifact. Based on the assumption that the actual performance of a tool is determined by both the functionality and the usability the tool provides, user interface design is another important technique for reusability. A design tool's user interface that employs appropriate visualization techniques not only improves an engineering designer's understanding of simulation models, but also it increases the chance of applying reuse features of the design tool. Moreover, it reduces the number of mistakes during model editing and reusing. For example, an engineering designer is editing a simulation model having a number of parameters connected by complicated causalities. Because the designer has limited knowledge of the model, he or she may define an erroneous mapping that will cause a loop in causality. Such an erroneous manipulation of simulation models can be prevented by improving the design tool's user interface so that it gives visual feedbacks on causality information. Extensive studies on information visualization techniques along with justification for them based on the recent founding on human vision and cognitive can be found in [22]. Useful guidelines provided by user interface specialists are also available in [23, 24].

3.2. Challenges

The integrated design framework is a computational environment where simulation models are built, integrated, and reused for engineering analysis. This section aims at identifying reusability problems in current integrated design frameworks. We begin our discussion with a survey result showing how software engineering concepts for reusability, which have been described in Section 3.1.1, are supported in three integrated design environments [25-27] and a simulation modeling languages [28]. What we observe in Table 3.1 is that even major integrated design frameworks do not support some of the key concepts, while object-oriented simulation languages [28-30], such as Modelica, have a support for them.

	Integrated design framework			Simulation modeling language
	DOME	Fiper	Model Center	Modelica
Instantiation: Creating multiple objects from one simulation model	O	O	O	O
Polymorphism: Decoupling interface from implementation	X	X	X	O
Inheritance: Implementing a model using inherited implementation of other model	X	X	X	O
Composition: Creating a simulation model by aggregation of multiple simulation models	O (declarative approach)	△ (master model approach)	△ (master model approach)	O
Interoperability: Subscribing simulation models in a remote server	O	O	O	X

Table 3.1: Comparison of the reuse support in major integrated design frameworks

One way of explaining this difference in support is that some of the object-oriented programming concepts are not as effective as they promise for reusability. Interestingly, critics on the object-oriented programming have pointed out that the gain from inheritance is often outweighed by the inflexibility it introduces to a system [31, 32]. Any change to a super-class has far-reaching effects on all its sub-classes [33], and as a result the system gets less flexible in adapting itself for subsequent changes in requirements [31].

In contrast to inheritance, polymorphism is considered as a missing, needed feature for improving reusability. The integration structure reuse, whose benefits have been demonstrated in the use case scenario in Section 2.2, is one application of the polymorphism concept to a simulation model because both – integration structure reuse and polymorphism – share the same key idea of decoupling the implementation from the interface. Thus, by developing integration mechanism implementing the integration structure reuse, we complement the missing support of the key concept in integrated design frameworks. As a platform for developing the integration mechanism, *DOME* (Distributed Object-oriented Modeling Environment) will be used. The DOME is an integrated design framework which has addressed several major reusability issues such as instantiation, composition, interoperability, portability, and user interface design, making it the most reuse-supporting platform among compared integrated design frameworks [25, 26].

We decompose our problem of the integration code pattern reuse into two sub-problems so that we can identify how and which of source code mining techniques can be utilized for our work. The first sub-problem is to find relevant integration code from the whole integration code, and the second is to find rules that capture the regularity within the relevant codes. Based on our discussion on the clone code detection and the example-based code completion, we need to decide which approach will be taken for the first sub-problem: lexical-analyzer-based or textual-pattern-based. The textual-pattern-based approach, more specifically the token-based representation, has been chosen because we aim to make our pattern reuse mechanism be language-independent and widely applicable to other text editing software components. Some researchers have also found that a relatively simple textual-pattern-based approach for code detection performs effectively when compared to sophisticated lexical analyzer approaches [34]. Other issues such as the code repository organization and automatic query generation can be addressed accordingly based on the code representation scheme, and the solution will be presented in Section 4.4. For the second sub-problem of generalizing rules from the relevant codes, we could not find relevant previous work, so a new mechanism addressing this problem will be developed in Section 4.4.

As for the user interface design of the design tool reusing two kinds of integration knowledge, most of visualization and user interaction techniques discussed in [22-24, 35] are found relevant, and they will be selectively adopted and adapted to meet the design tool's needs. Issues to be addressed by the user interface of the design tool include 2-D structures for parameters, relations, mappings visualization; interaction for model definition and reuse; focus and context management during model navigation; and data mapping for model understanding. Considering a number of useful user interface design techniques available to solve our design problem, the focus of our research, in terms of user interface design, will be exploiting available knowledge to build an effective user interface, specialized in reusing integration knowledge in simulation models. Further details on the application will be explored in Section 4.5.

Chapter 4 Implementation

This chapter describes how we build the design tool for reusing integration knowledge. The first section of the chapter provides an overview of software components. Instead of just showing the end result, the section explains the way how our software components are derived and the reason why they are suitable for our needs. The proceeding sections give details on major software components: *catalog model definition*, *dependency solving algorithm*, *pattern generalization algorithm*, and *catalog model builder*.

4.1. System Overview

Section 4.1.1 and 4.1.2 formulate and analyze the design of the design tool based on a design methodology called *axiomatic design*. Further description on the design methodology with extensive application examples can be found in [36].

4.1.1. Functional requirements

We find the functional requirements for the design tool by sequentially following what the system performs and writing down what conditions should be met at each step to proceed. In order to reuse the integration structure, we suppose that the design tool should have a system that performs the following steps. First, the system has a data structure defining modeling elements in an integration model. When the system receives a request for executing a model interface, it looks up the data structure to create an execution plan. The execution plan is translated into an executable form such as script code, which is executed by a script engine. After executing the executable form, the execution result is collected and sent out to the DOME runtime environment. In addition to these run-mode steps, we suppose that following build-time steps are needed. First, the system provides a programmable interface to create and modify an integration model. Next, based on the programmable interface, the system provides a graphical user interface so that users can easily build integration models.

In order to reuse the integration code pattern, we assume that the system should perform the followings. First, the system needs to store the whole collection of mapping scripts in an integration model; also the collection needs to be queried to find mapping scripts relevant to the parameter name to which code completion is requested. Next, the system is expected to employ an algorithm to generalize patterns from the relevant mapping scripts; the generalized patterns are used to generate code completion candidates. The system should interact with the user interface of the design tool. It provides a programmable interface to access the code completion feature. A user interface component, such as a code completion popup, is integrated with the programmable interface and displays the code completion candidates. The functional requirements for the design tool are summarized as follows (Figure 4-1):

FR1: Reuse integration structure

FR1.1: Manage data structures defining the new integration model
FR1.2: Generate execution plans for the currently selected implementation
FR1.3: Provide programmable interface for building integration models
FR1.4: Provide user-friendly interface for building integration models
FR1.5: Run the execution plan for the selected implementation
FR1.6: Interface with DOME runtime environment to send out the results

FR2: Reuse integration code pattern

FR2.1: Retrieve relevant code from the whole collection of code
FR2.2: Generalize patterns, each which is expressed as a rule set, from the retrieved code
FR2.3: Generate code completion candidates by applying rule sets
FR2.4: Provide programmable interface for executing the algorithm
FR2.5: Integrate the algorithm into design tool user interface

Figure 4-1: Function requirements identified for the design tool

4.1.2. Design parameters

Given the functional requirements, finding design parameters is to find a set of software components that can satisfy all the functional requirements. Because there can be multiple sets of software components satisfying the requirements, we try to come up with a design that is close to an ideal design, characterized by having no coupling within design parameters. Minimizing coupling is an important issue for software components since it significantly affects their reusability. Finally, Figure 4-2 summarizes the design parameters of our design tool.

DP1: Software components for integration structure reuse

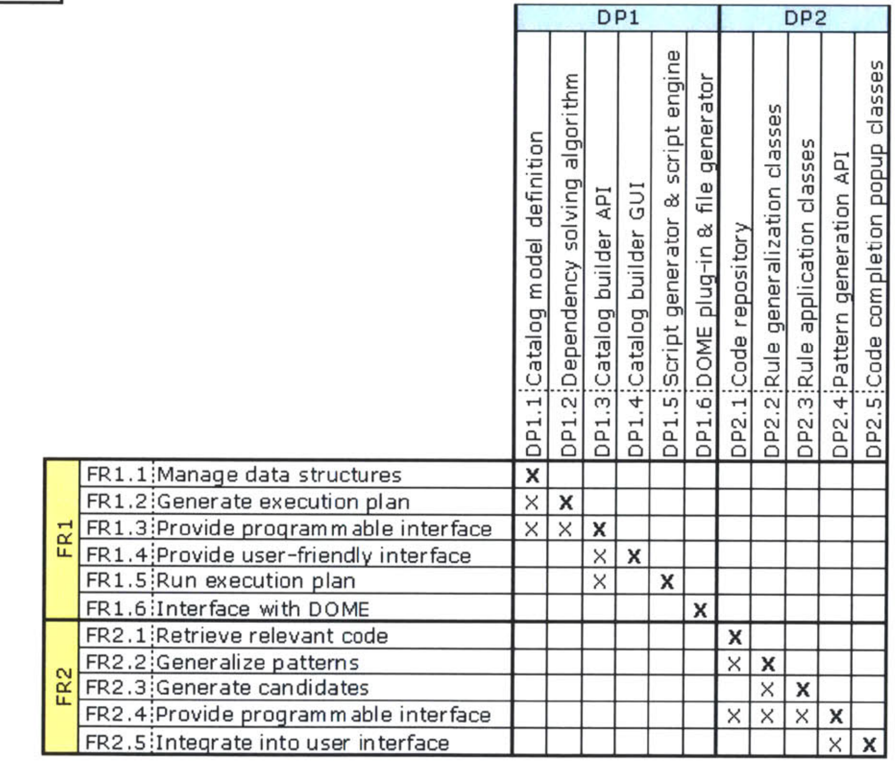
- DP1.1: Catalog model definition classes
  (mit.cadlab.dome3.plugin.catalog.core)
- DP1.2: Dependency solver classes
  (mit.cadlab.dome3.plugin.catalog.core)
- DP1.3: Catalog model builder API classes
  (mit.cadlab.dome3.plugin.catalog.core)
- DP1.4: Catalog model builder GUI classes
  (mit.cadlab.dome3.plugin.catalog.ui)
- DP1.5: Groovy Script generator, Groovy script engine, and DOME API
  (mit.cadlab.dome3.plugin.catalog, mit.cadlab.dome3.api)
- DP1.6: Catalog model plug-in and DOME-specific file generator classes
  (mit.cadlab.dome3.plugin.catalog, mit.cadlab.dome3.plugin.catalog.serialization)

DP2: Software components for integration code pattern reuse

- DP2.1: Code repository classes
  (mit.cadlab.dome3.plugin.catalog.pcc)
- DP2.2: Rule generalization classes
  (mit.cadlab.dome3.plugin.catalog.pcc)
- DP2.3: Rule application classes
  (mit.cadlab.dome3.plugin.catalog.pcc)
- DP2.4: Pattern generalziation API classes
  (mit.cadlab.dome3.plugin.catalog.pcc)
- DP2.5: Code completion popup classes
  (mit.cadlab.dome3.plugin.catalog.ui)

Figure 4-2: Design parameters identified for the design tool

We analyze the design matrix shown in Figure 4-3 to evaluate how desirable our selection of design parameters is. Design matrix visualizes which design parameters are used to satisfy a certain functional requirement; for example, to satisfy *FR1.2 Generate execution plan*, we use two components for the catalog model definition and the dependency solving algorithm. Generally, uncoupled or decoupled design is considered as an acceptable design because all *FRs* can be satisfied by choosing *DPs* in a certain sequential order, which can be solved from the design matrix. As Figure 4-3 shows, our design matrix is a decoupled one, having only lower triangular elements.



		DP1						DP2				
		DP1.1 Catalog model definition	DP1.2 Dependency solving algorithm	DP1.3 Catalog builder API	DP1.4 Catalog builder GUI	DP1.5 Script generator & script engine	DP1.6 DOME plug-in & file generator	DP2.1 Code repository	DP2.2 Rule generalization classes	DP2.3 Rule application classes	DP2.4 Pattern generation API	DP2.5 Code completion popup classes
FR1	FR1.1 Manage data structures	**X**										
	FR1.2 Generate execution plan	X	**X**									
	FR1.3 Provide programmable interface	X	X	**X**								
	FR1.4 Provide user-friendly interface			X	**X**							
	FR1.5 Run execution plan			X		**X**						
	FR1.6 Interface with DOME						**X**					
FR2	FR2.1 Retrieve relevant code							**X**				
	FR2.2 Generalize patterns							X	**X**			
	FR2.3 Generate candidates								X	**X**		
	FR2.4 Provide programmable interface							X	X	X	**X**	
	FR2.5 Integrate into user interface										X	**X**

Figure 4-3: Design matrix showing *FRs-DPs* relationship of our new design tool

Modularity analysis on the design matrix reveals that our concern on the reusability of software components has been effectively addressed. Some of our software components such as the dependency solving algorithm and the pattern generalization algorithm are elaborated implementation works, and thus we want to organize the software components so that those expensive components can be used for other applications. *FR1.5 Run the execution plan*, which is decoupled from other *DPs* but *DP1.3* and *DP1.5*, is one example of effectively decoupled *FRs*. The current implementation retrieves an execution plan through *DP1.3 Catalog model builder API*, and translates the execution plan into a Groovy script, which is finally run by Groovy script engine. Because *DP1.5* has no interaction with complicated implementation in *DP1.1* and *DP1.2*, the script engine can be easily replaced and even multiple script engines can be supported. We also notice that *FR1.6 Interface with DOME* is decoupled from other *DPs* but *DP1.6*. This implies that we can make the integration model interact with other type of computational environments by implementing additional *DP1.6*. This decoupling provides considerable flexibility to enhance the interoperability of the integration model. Possible applications include deploying the integration model, supporting integration knowledge reuse, in other integrated design frameworks and subscribing simulation services from standard web service containers. Lastly, the design matrix of *FR2.5 Integrate pattern generalization algorithm into user interface* suggests that the pattern generalization algorithm can be integrated into different kinds of user interfaces through *DP2.4 pattern generalization API*; therefore, text editors or software development tools embedding the pattern generalization algorithm can be easily developed.

4.1.3. Component map

The following component map illustrates interactions between software components. Figure 4-4 depicts build-time interactions among participating software components. In the build-time, the model builder GUI (Graphic User Interface) component initiates interactions. Upon a user's request, it modifies model definition and queries code completion from the pattern generalization algorithm component. When the user invokes a save command, the integration model will be serialized into a DOME model file.

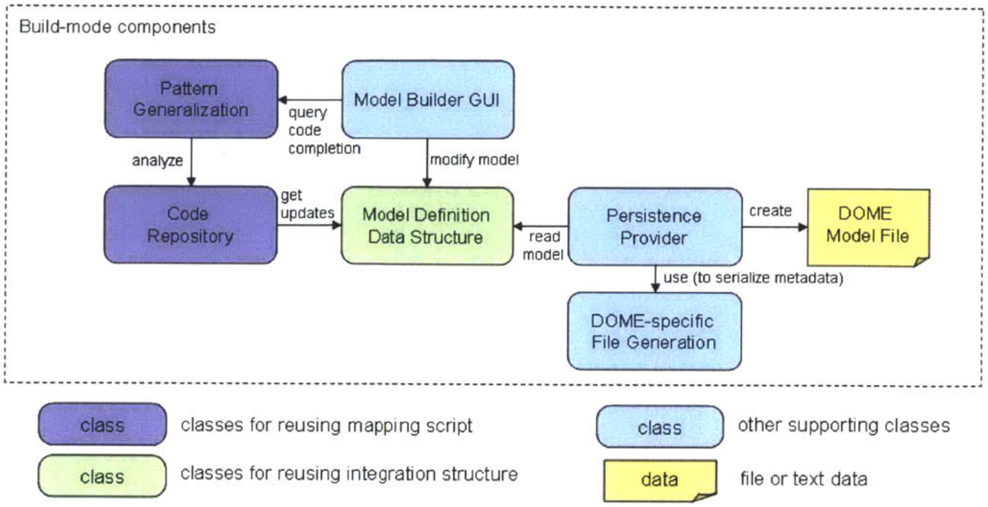


Figure 4-4: A component map showing interactions among software components in the build-mode

Figure 4-5 shows interactions in the run-time. In the run-time, a model interface execution request drives the interaction. When a DOME server receives a model interface execution request, it instantiates a catalog model runtime using DOME Plug-in API. The catalog model runtime first invokes the script generator component to create script code, which will be evaluated by the script engine. The script generator component relies on the model definition component and the dependency solver component to generate script code. When evaluating the script code, the script engine uses DOME run-mode API to access and execute simulation models in other DOME servers.

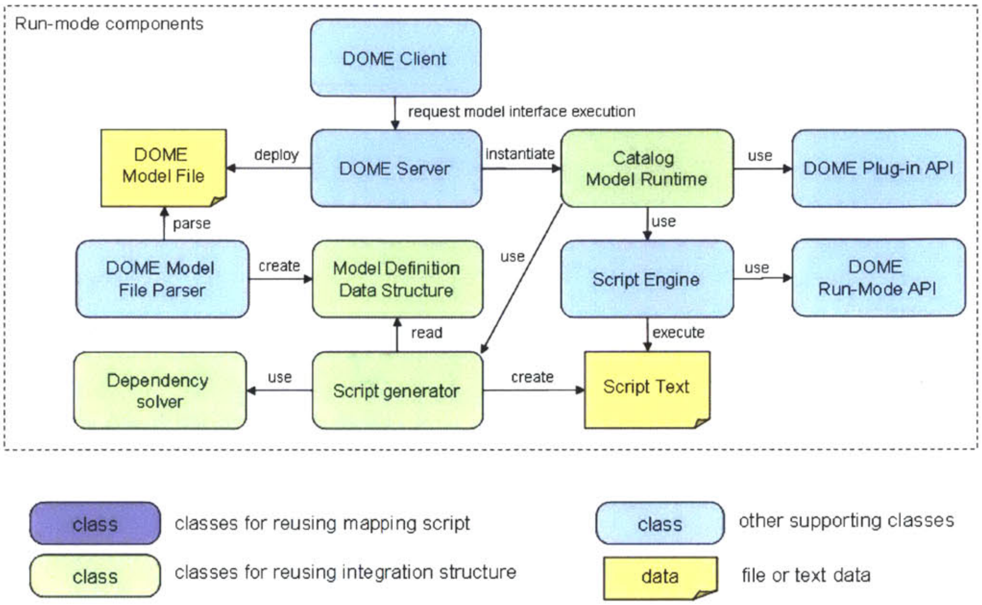


Figure 4-5: A component map showing interactions among software components in the run-mode

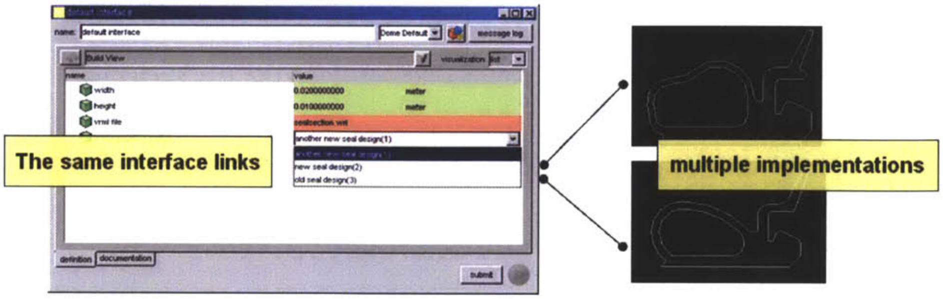
4.2. Catalog Model Definition

4.2.1. Data structure

Catalog model is a name for our new integration model supporting the reuse of the integration structure. This section illustrates how the catalog model organizes its model definition data. The model definition data are created by *catalog model builder*, which refers to GUI for creating and modifying the catalog model. The script generation component uses the model definition data during the run-mode to convert an execution plan into script code. The catalog model is defined by five modeling elements: *parameter*, *relation*, *mapping*, *interface*, and *implementation*. The parameter represents an atomic unit of data in a simulation model with data types of integer, real, Boolean, vector, matrix, enumeration, string, and file. It contains unit information, which allows the system to perform data conversion automatically between different units and to check dimensional conformity when assigning a value to a certain parameter. The parameter further decomposed into four categories based on its parent entity and causality: interface input/output parameters and relation input/output parameters.

The relation is used to define a mathematical relationship among parameters within a catalog model. Depending on the source of the mathematical relationship, we have two implementations of relations: *local relation* defined by Groovy script code [37] inside a model and *remote relation* defined by a subscription of a model interface outside a model. Mapping in the catalog model is called *mapping with transformation.* It not only copies the values of source parameters to a target parameter, but it can also perform mathematical transformation before it assigns a value to the target parameter. In implementation point of view, the mapping with transformation is similar to the relation, but it does not require users to specify causality information when defining it because it has mechanism to generate causality information from the mapping script.

Interface and *implementation* are the modeling elements that distinguish the catalog model from other integration models that do not support the reuse of the integration structure. The interface is defined by a set of input and output parameters and causality information between parameters. *Implementation* is defined by a collection of relations and mappings, which transforms the values of input parameters into those of output parameters. Because the definition of a certain interface is shared by all implementations associated with the interface, one-to-many relationship exists between interface and implementations (Figure 4-6). The interface defines a kind of communication protocol between a catalog model and other simulation models subscribing the catalog model, and therefore we gain flexibility to vary the internal implementation without affecting the overall integration structure.



name: default interface

Dome Default

name	value	
width	0.0200000000	meter
height	0.0100000000	meter
vrml file	sealsection.wrl	
	another new seal design(1)	
	another new seal design(1)	
	new seal design(2)	
	old seal design(3)	

Figure 4-6: One-to-many relationship between the interface and the implementations

4.2.2. Addressing mechanism

The catalog model introduces new mechanism for addressing modeling elements, which is implemented by a software component called *naming service*. It is based on a namespace concept, which has been applied to many languages such as XML, Java, or C#. Using the naming service, any modeling element can be accessed by a reference string: *relB* for a relation or *relA.max speed* for a parameter in the relation. We assign a separate namespace for each relation, and the namespace is accessed using an alias of the relation, which is unique in an implementation of the integration model: For example, *relA* and *relB* are aliases for two local relations. Because we may have multiple subscriptions of the same model interface in an implementation, such a unique alias is needed to address them efficiently. A dot (.) has a special meaning in the reference string and is called the scope-resolution operator. It is used to navigate between namespace and sub-namespace, namespace and a model element, or sub-namespace and a model element. A parameter named *max speed* in a relation aliased as *relB* can be addressed using a reference string, *relB.max speed*. Therefore, we can use the same parameter name multiple times in a simulation model as long as they belong to different namespaces. The namespace-based addressing mechanism is more user-friendly than GUID (Globally Unique Identifier)-based reference systems because it allows a short and human-understandable reference string such as *relB.max speed* when compared to a GUID-based reference string such as *33158ded-bf9d-1004-8e0f-20d36b52f98a*. Table 4.1 gives a brief overview of classes used to define the catalog model.

Class Name	Description
CParameter CInterfaceInputParameter CInterfaceOutputParameter CRelationInputParameter CRelationOutputParameter	Defines name, data type, unit, and solving state. CInterfaceOutputParameter and CRelationInputParameter can have a mapping associated with the parameters.
CRelation CLocalRelation CRemoteRelation	Defines a set of relation parameters. CLocalRelation defines a script and causality. CRemoteRelation defines subscription and causality.
CMapping	Defines a mapping script and causality
CImplementation	Defines a collection of relations and mappings transforming inputs into outputs
CInterface	Defines a set of interface parameters and causality
CNamingService	Implements namespace-based naming service for accessing model elements

Table 4.1: Description of Catalog model definition classes

4.3. Dependency Solving Algorithm

4.3.1. Overview of the algorithm

Dependency solving algorithm finds a sequence by which mappings and relations should be executed. It solves a graph search problem where the goal is propagating value changes in some nodes to all other nodes affected by them, under the constraint that the navigation between nodes is restricted by dependencies imposed by mapping and relations. The algorithm first creates a directional graph using causality information in mappings and relations; in the graph, a node is a parameter and an arc is either a relation or a mapping. The algorithm searches for a acceptable navigation sequence, called *execution plan*, by following steps described in Section 4.3.2. The output of the algorithm, the execution plan, is a list of mappings and relations; for example, a execution plan of *[mapping: relA.A, mapping: relA.B, relation: relA, mapping: itf.Z]* indicates that we can make all parameters consistent by executing mappings and relations in the given order. Therefore, the model execution will start with executing the mapping for parameter A in relA and finish with executing mapping for parameter Z in the interface. The algorithm uses a queue called *green queue* to keep track of parameters which are consistent and thereby available as input of mappings or relations. The key idea of pseudo code presented in the next section is as follows:

- A graph connected by directional arcs is created.
- The modification in input parameters makes some nodes green and others red.
- The algorithm propagates green color from a green node to a red node along directional arcs.
- Propagation is allowed when all input parameters of a mapping or a relation are green.
- The algorithm keeps on propagating the green color until it meets the following exit condition:
  - □ Success when there is no red node in the graph.
  - □ Failure when there are red nodes in the graph, but there is no relation or mapping that can propagate the green color.

4.3.2. Pseudo code

Nomenclature

[GQ] : a queue containing green parameters.

[EP] : a list containing mappings and relations in an order to be executed

{mi-param}: a whole set of modified interface input parameters

{io-param}: a whole set of interface output parameters

{ri-param}: a whole set of relation input parameters

{param | query condition} : a set of parameters queried by a condition

{mapping | query condition} : a set of mappings queried by a condition

{mapping node | query condition} : a set of mapping nodes queried by a condition

→ green or red : turns parameter or mapping node into green or red

➔ [EP] or [GQ] : put an item into [EP] or [GQ]

Each {mapping A} : $mapping : each element in {mapping A} will called $mapping

Algorithm

1. The system is given a set of modified input parameters denoted as {mi-param}

2. Turn {mi-param} into green, indicating consistent status, because the input parameters are consistent.

```
{mi-param} → green
```

3. Push modified interface input parameters into green queue [GQ].

```
{mi-param} ➔ [GQ]
```

4. Find all parameters driven by {mi-param}, and turn them into red, indicating inconsistent status. Each time such a parameter turns red, all mapping nodes mapped to the parameters turns red, too. When it is the first time executing this implementation, {mi-param} should be

given the same as {ii-param}

```
{param | driven by {mi-param}}→ red
{mapping node | mapped to {param | driven by {mi-param} } } → red
```

5. When it is the first time executing this implementation, add relation input parameters and interface output parameters having no driver to [GQ]. For later execution, skip this step because those parameters will not change their value.

```
{param | driven by {empty} among {ri-param} } → [GQ]
{param | driven by {empty} among {io-param} } → [GQ]
```

6. Start a loop popping a green parameter from [GQ].

   6.1. For each popped green parameter, turn all mapping nodes mapped to it into green

   ```
   Each popped parameter : $popped
   {mapping node | mapped to $popped} → green
   {mapping node | mapped to $popped} : {mapping node A}
   ```

      6.1.1 For each mapping node turning green, find the mapping that is made executable by the iterated mapping node (A mapping becomes executable when all input mapping nodes of the mapping are green)

      ```
      Each {mapping node A}: $mapping_node
      {mapping | made executable by $mapping_node} → {mapping A}
      ```

         6.1.1.1. For each executable mapping, put the mapping into [EP], turn its output mapping node into green, turn a parameter mapped by the mapping into green, and put the parameter into [GQ].

         ```
         Each {mapping A} : $mapping
         $mapping → [EP]
         Output mapping node of $mapping. → green
         Parameter mapped to the output node → green
         ```

```
Parameter mapped to the output node ➔ [GQ]
```

6.2. If [GQ] has remaining green parameters, repeat A.

6.3. If [GQ] has no remaining green parameter, do the followings.

6.3.1 Check if all interface output parameters are in a consistent status. If so, the dependency is solved successfully. Exit the popping loop of Step 6 and return [EP].

6.3.2 Find executable relations – A relation becomes executable when all input parameters are green. If no executable relation is found, the dependency solving has failed. Exit the popping loop of Step 6, and report an error message containing which parameters remains inconsistent.

6.3.3 If several executable relations are found, select one that creates the most number of new green parameters

```
a relation selected from {relation | executable}: $relation
```

6.3.4 Turn output parameters of the picked relation into green, and push the parameters into [GQ]. (If there is a derived parameter associated with an output parameter, it will need the same treatment.)

```
$relation ➔ [ES]
output parameters of $relation → green
output parameters of $relation ➔ [GQ]
```

7. This execution point is not reached because the popping loop exits either at 6.3.1 or at 6.3.2.

4.4. Pattern Generalization Algorithm

4.4.1. Overview of the algorithm

Pattern generalization algorithm finds code completion candidates based on the given hint literals, which consist of a parameter name and a partially completed mapping script in our case. It first retrieves relevant code lines from the whole collection of script code and generalizes patterns from them. A pattern is expressed as a rule set describing regularity discovered from the code lines. The algorithm then applies using rule sets to the hint literals to generate code completion candidates. Generalizing the rule sets from a given set of code line is the most challenging part of the algorithm for the following reasons:

First, more than one rule set can be generalized from a given set of code lines. Some of the code lines are generalized into one rule set, and some others are generated into another rule set. Still others may not be generalized and dismissed as not generalizable. For example, when we have ten code lines, eight lines can be generalized into three rule sets, while two code lines are left as not generalizable. We call such a dismissed code line as an unexplained code lines because they are not explained by any rules sets generalized from the set of code lines.

Second, we don't have information on which code lines are grouped and generalized into a rule set. Let us assume that, among the eight code lines that could be generalized into three rule sets, row 1, row 2, and row 3 are grouped together and generalized into a rule set, while row 4 and row 6 are generalized into another rule set. Unfortunately, such grouping information is not provided to the algorithm. The algorithm needs mechanism to group rows efficiently so that each group of code lines can be generalized into a certain rule set.

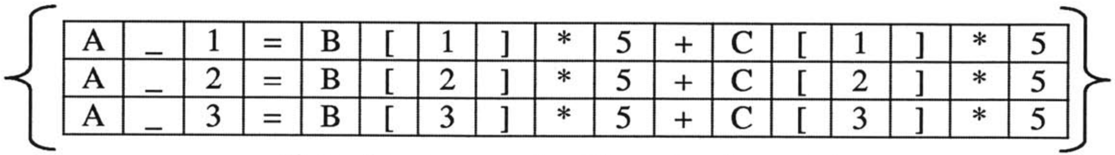
Third, there can be more than one way of generalizing rule sets, so we need to quantitatively differentiate them. We can state the goal of the generalization algorithm quantitatively that it aims at explaining the most numbers of rows using the least number of rule*s*. A method we suggest to quantify the closeness to the goal is a score calculated from weighted sum of the number of unexplained rows and the number of rule sets: The smaller score a generalization has, the more desirable way of generalization it is.

The algorithm follows the steps described in Section 4.4.3 to address the challenges. The first and second challenges are solved by two techniques called *hasty rule set generalization from pairs* and *rule set merging*, which are described in Step 4.3.1 and Step 4.3.3, respectively. The third challenge is addressed by a technique called *rule set permutation*, described in Step 4.5.2 and 4.5.3.

4.4.2. Algorithm terminology

Followings are new terms introduced to describe our algorithm: *code line*, *code token*, *code token signature*, *code base*, *code hint*, and *code token matrix*. The code line is a piece of source code, which is used as a unit of pattern generalization. Logically, a code line may span over multiple lines. However, in most case, one line of source code is served as a code line because we assume that users are interested in a pattern discovered when source code is examined in a line-by-line manner. The code token is an atomic element of a code line, created by the code tokenizer. Four types of code token are defined: delimiter token, string token, integer token, and double token. Code tokens that belong to one code line form an array of code tokens, called the token row. The code tokenizer splits a code line into a set of code tokens based on regular expression rules; it not only splits a code line at a symbolic character like + or [but also splits it at a point where numeric character begins or ends and alphabet character changes capitalization, which helps splitting code lines in camel case. The code token signature is a simplified representation of code tokens. It is created by following transformation rule: delimiter tokens are kept as they are, string tokens are replaced with S, and integer and real tokens replaced with N. For example, *A_1=B[1]*5+C[1]*5* is transformed into *S_N=S[N]*N+S[N]*N*.

The code base is a repository of code lines. It allows us to index code lines into the repository and to query code lines relevant to a code hint. The code hint is a string used to generate code completion candidates. When a user request code completion, the user specifies a parameter whose parameter needs to be completed, and the parameter name is passed as a code hint to the pattern generalization algorithm. If a mapping script editor has a string in it, the string will be combined with the parameter name and served as a code hint. The code token matrix is a a data structure representing multiple code lines. All code lines in the code token matrix have the same code token signature, and thus it can be organized in a matrix-like form. For example, if we create a code token matrix from *A_1=B[1]*5+C[1]*5*, *A_2=B[2]*5+C[2]*5*, and *A_3=B[3]*5+C[3]*5*, the matrix will be as follows (Figure 4-7):



A	_	1	=	B	[	1	]	*	5	+	C	[	1	]	*	5
A	_	2	=	B	[	2	]	*	5	+	C	[	2	]	*	5
A	_	3	=	B	[	3	]	*	5	+	C	[	3	]	*	5

Figure 4-7: An example of the code token matrix

4.4.3. Pseudo code

1. Create a code base, and index code lines. For each code line indexed to the code base, the code base indexes a code token signature, a code line string, and a source id. For example, if it indexes mapping script *B[1]*5+C[1]*5* from parameter *A_1* in a relation aliased as *relC* in implementation *ExcelImpl*, a record of code token signature *S_N=S[N]*N+S[N]*N*, code line string *A_1=B[1]*5+C[1]*5*, and source id *ExcelImpl/relC/A_1* is stored in the code base. We assume that the code base has following records (Table 4.2):

Code token signature	*Code line string*	*Source ID*
S_N=S[N]*N+S[N]*N	A_1=B[1]*5+C[1]*5	ExcelImpl/relC/A_1
	A_2=B[2]*5+C[2]*5	ExcelImpl/relC/A_2
	A_3=B[3]*5+C[3]*5	ExcelImpl/relC/A_3
	D_3=D[3]*2+D[4]*2	ExcelImpl/reC/D_3
	D_4=D[4]*2+D[5]*2	ExcelImpl/reC/D_4
	E_5=E[5]*2+E[6]*2	ExcelImpl/relB/E_2
	A_1=A[1]*2+B[1]*2	ExcelImpl/reB/A_1
	A_2=A[2]*3+B[4]*5	ExcelImpl/relB/A_2
	A_3=A[3]*4+B[4]*5	ExcelImpl/relB/A_3
	A_4=A[5]*5+B[3]*4	ExcelImpl/relB/A_4
S_N=S_N*N	G_1=Z[0]*2+W[0]*2	ExcelImpl/relD/G_1

	G_2=Z[1]*2+W[1]*2
	G_3=Z[2]*2+W[2]*2
		ExcelImpl/relD/G_2
		ExcelImpl/relD/G_3

S_N=N	A_1=1.0
	A_2=1.0
		ExcelImpl/relE/A_1
		ExcelImpl/relE/A_2

Table 4.2: Records in the code base

2. Code completion is requested for a parameter named B_7. A code hint, *B_7*, is created, and a code token signature for the code hint, *S_N=*, is created.

3. Query the code base to get code token signatures which starts with the code token signature of the code hint. The code base will look up the first column code token signature and return all code token signatures satisfying the query condition.

   {signature | starts with the signature of a code hint} : $relevant_signature_list

4. Prepare a list to collect code completion candidates : $candidate_list.

   For each queried code token signature, which will be denoted as $relevant_signature, do the followings:

   For each item in $relevant_signature_list : $relevant_signature

   4.1. Query the code base to get code lines whose code token signatures are the same as $relevant_signature. The query result, a list of code lines, will be called $relevant_codelines, and the number of relevant code lines will be denoted as N.

   {codeline | whose code token signature is $relevant_signature } : $relevant_codeline_list

   The size of $relevant_codelines : N

   4.2. Create a code token matrix from the queried relevant code lines. For example, if we make a code token matrix from code lines in the second row of Table 4.2, it will be the matrix shown in Figure 4-8

A	_	1	=	B	[	1	]	*	5	+	C	[	1	]	*	5
A	_	2	=	B	[	2	]	*	5	+	C	[	2	]	*	5
A	_	3	=	B	[	3	]	*	5	+	C	[	3	]	*	5
D	_	3	=	D	[	3	]	*	2	+	D	[	4	]	*	2
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
E	_	5	=	E	[	5	]	*	2	+	E	[	6	]	*	2
A	_	1	=	A	[	1	]	*	2	+	B	[	1	]	*	2
A	_	2	=	A	[	2	]	*	3	+	B	[	4	]	*	5
A	_	3	=	A	[	3	]	*	4	+	B	[	4	]	*	5
A	_	4	=	A	[	5	]	*	5	+	B	[	3	]	*	4

Figure 4-8: Code token matrix for the second row

4.3. For each pair of rows in the matrix, do the followings:

For each pair of rows in the code token matrix: $first_row, $second_row

4.3.1 For each columns of the selected two rows, do the followings – this step is called the hasty rule set generalization from pairs because we try to generalize a rule set based on just two rows):

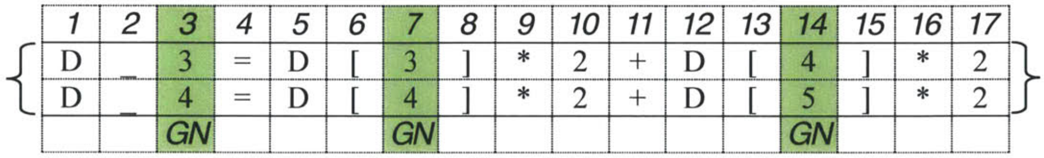
4.3.1.1. If two tokens in the column have the same number, assign the same number rule for the column. For the following pair of the fourth and fifth rows in Figure 4-8, column 10 and 17 will be assigned the same number rule (SN) as shown in Figure 4-9.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
D	_	3	=	D	[	3	]	*	2	+	D	[	4	]	*	2
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
SS				SS					SN		SS					SN

Figure 4-9: The same number rule (SN) is assigned to column 10 and 17, denoted by yellow. The same string rule (SS) is assigned to column 1, 5, and 12, denoted by blue.

4.3.1.2. If two tokens in the column have the same string, assign the same string rule (SS) for the column. For the pair of the fourth and fifth row, column 1, 5, and 12 will be assigned the same number rule as shown in Figure 4-9.

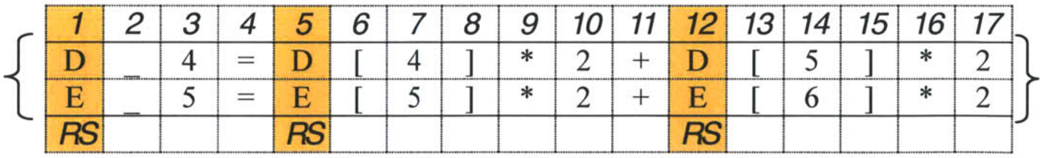
4.3.1.3. If a certain group of columns in the first row have the same gap with a certain group from columns, which should be the same group we have chosen for the first row, in the second row, assign the gapped number rule (GN) for those columns. For the pair of the fourth and fifth row, column 3, 7, and 14 are gapped with zero for both rows, so they are assigned a gapped number rule with gap information of (N, N+0, N+0), which means if the value of first column is 2 (N=2), the values for the second column and the third column is 2 (N+0), and 2 (N+0).



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
D	_	3	=	D	[	3	]	*	2	+	D	[	4	]	*	2
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
		GN				GN							GN			

Figure 4-10: The gapped number rule (GN) is assigned to column 3, 7 and 14, denoted by green.

4.3.1.4. If a certain group of column has a repeated string in the first row and if a certain group of column, which should be the same group we have chosen for the first row, has a repeated string in the second row, assign the repeated string rule (RS) for those columns. For the pair of the fifth and sixth row in Figure 4-8, column 1, 5, and 12 have a repeated string for both rows, so they are assigned a repeated string rule.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
E	_	5	=	E	[	5	]	*	2	+	E	[	6	]	*	2
RS				RS							RS					

Figure 4-11: The repeated string rule (RS) is assigned to column 1, 5 and 12, denoted by orange.

4.3.2 After iterating all columns, check if we could find a rule set that can explain all columns. If so, the rule set will be accepted for the next step of rule set merging and denoted as $accepted_rule_set. If not, the rule set will be abandoned.

4.3.3 When a rule set accepted for rule merging reaches this point, do the followings:

4.3.3.1. At this point, we have a map of rule sets called the rule set map, which is initialized as an empty map at the beginning of loop at Step 4.3 and later populated by a process called *rule set merging*, described in Step 4.3.3.2 to Step 4.3.3.4. The rule set map stores a rule set as its key and supporting rows as its value. For example, if we have generalized a rule set from row 4 and 5 and if the rule set has been added as an entry in the rule set map, the resulting rule set map will be as follows:

Key (rule set)	*Value (supporting rows)*
[same number rule: column 10, 17] [same string rule: column 1, 5, 12] [gapped number rule: column 3,7,14 with (N, N+0, N+0)]	row 3, 4

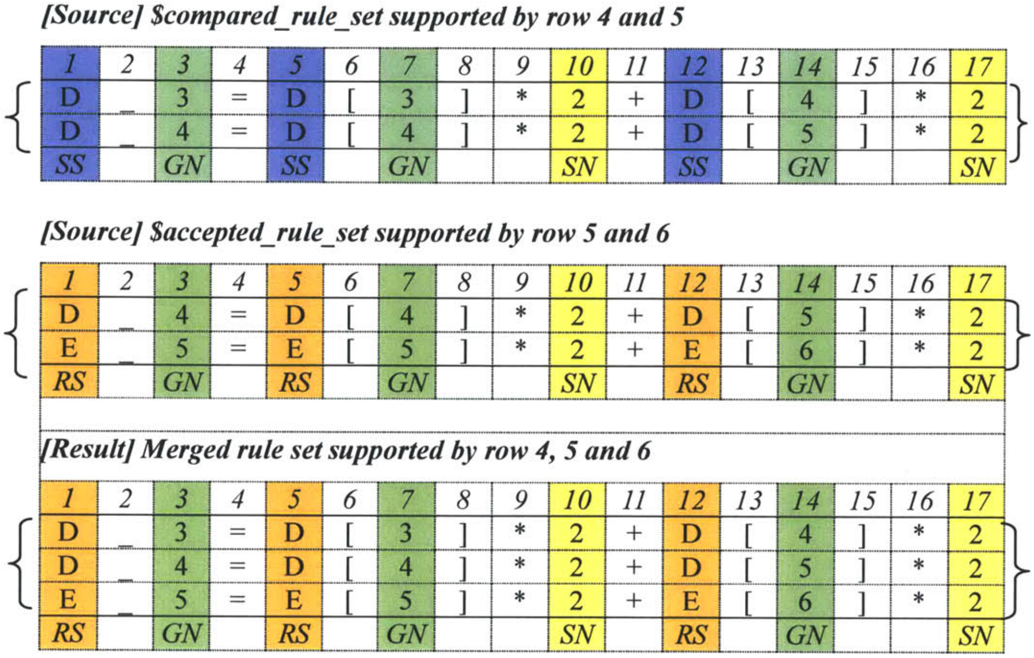
Table 4.3: A rule set map having a rule set generated from row 1 and row 2

4.3.3.2. Once a new $accepted_rule_set is generalized from a row pair, the $accepted_rule_set is compared with each of the rule sets in the rule set map, which denoted as $compared_rule_set.

A. If $compared_rule_set is the same as $accepted_rule_set, the row pair that have generated the $accepted_rule_set is added to supporting rows of $ compared_rule_set.

B. Even though $compared_rule_set and $accepted_rule_set are not exactly the same, there are cases when they can be merged into one. It is because some columns with the same number rule can be re-assigned as the gapped number rule and also because some columns with the same string rule can be re-assigned as the repeated string rule. In that case, two rule sets are merged into one rule set, and their supporting rows are also merged. The following example shows a case when $compared_rule_set found in Table 4.3 is merged with

$accepted_rule_set generalized from row 5 and 6 in Figure 4-8.



[Source] $compared_rule_set supported by row 4 and 5

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
D	_	3	=	D	[	3	]	*	2	+	D	[	4	]	*	2
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
SS		*GN*		*SS*		*GN*			*SN*		*SS*		*GN*			*SN*

[Source] $accepted_rule_set supported by row 5 and 6

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
E	_	5	=	E	[	5	]	*	2	+	E	[	6	]	*	2
RS		*GN*		*RS*		*GN*			*SN*		*RS*		*GN*			*SN*

[Result] Merged rule set supported by row 4, 5 and 6

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
D	_	3	=	D	[	3	]	*	2	+	D	[	4	]	*	2
D	_	4	=	D	[	4	]	*	2	+	D	[	5	]	*	2
E	_	5	=	E	[	5	]	*	2	+	E	[	6	]	*	2
RS		*GN*		*RS*		*GN*			*SN*		*RS*		*GN*			*SN*

Figure 4-12: Rule set merging of $compared_rule_set and $accepted_rule_set

C. If $compared_rule_set and $accepted_rule_set are different and if they cannot be merged into one, $accepted_rule_set is added as a new entry in the rule set map.

4.4. Now the loop of Step 4.3 has finished. The final rule set map will have several rule sets supported by two or more than two rows as shown in Table 4.4.

Key (rule set)	*Value (supporting rows)*
[same string rule: column 1, 5, 12] [gapped number rule: column 3,7,14 with (N, N+0, N+0)] [same number rule: column 10, 17]	row 1, 2, 3
[repeated string rule: column 1, 5, 12] [gapped number rule: column 3,7,14 with (N, N+0, N+0)] [same number rule: column 10, 17]	row 3, 4, 5

[same string rule: column 1, 5, 12] [gapped number rule: column 3,7,10 with (N, N+0, N+1)] [gapped number rule: column 14,17 with (N, N+1)]	row 6, 7, 8, 9

Table 4.4: A rule set map having a rule set generated from row 1 to row 9

4.5. The resulting rule set map can have rows that support more than one rule sets, the rule set permutation is performed to adjust the rule set map so that the most number of rows can be explained by the least number of rule sets. The algorithm creates all possible sequences of lining up the rule sets in the rule set map. For each sequence, we do the followings:

4.5.1 We have two lists: one for storing explained rows, denoted as $explained_row_list and the other for storing unexplained rows, denoted as #unexplained_row_list. We also have a list containing active rule sets, denoted as $active_rule_set_list. All lists are created empty.

4.5.2 As iterating through each permutated sequence of rule sets, we do the followings:

4.5.2.1. Count how many of the supporters of the rule set are not in $explained_row_list

4.5.2.2. If the count is equal or more than two, the rule set is added to $active_rule_set_list, and the supporting rows of the rule set are merged into $explained_row_list.

4.5.2.3. If the count is less than two, the rule set will not be added to $active_rule_set_list. If the counter is one, the counted supporting row is added to $unexplained_row_list.

4.5.3 Compute a score for this sequence based on the size of $active_rule_set_list and the size of $unexplained_row_list. The score is computed by the equation of (weight_1 * the size of $active_rule_set_list + weight_2 * the size of $unexplained_row_list). Because we want to have a small size of $active_rule_set_list and a small size of

$unexplained_row_list, a smaller score indicates that this sequence is more desirable.

4.6. The rule set permutation has finished, giving scores for all the permutated sequences. We sort it to find the best sequence with the lowest score, and the sequence's $active_rule_set_list will be returned.

4.7. Combine the code hint with $active_rule_set_list to create a code completion candidate. Stored it in the $candidate_list

5. Return a list of code completion candidates stored in $candidate_list.

4.5. Catalog Model Builder

In this section, we present how user interface of the catalog model builder has been built. The first section describes the layout of the catalog model builder to illustrate how it interacts with a user to create a catalog model. The second section addresses the issue of model representation. Questions such as what kind of graphical structure is used to represent a relation in the model builder will be answered. The next part explains how we visualize dependency among parameters to help users in the mapping process. The remaining three sections describe features related to the mapping script editor: *reference-by-click*, *color-coding*, and *code completion popup*

4.5.1. Layout of the catalog model builder

The catalog model builder has three panels for navigation, interface definition, and implementation definition (Figure 4-13). After creating or opening a catalog model, a user chooses one of model implementations from the navigation panel, and it makes the selected model implementation displayed in the implementation definition panel. The user can add local and remote relations into the implementation definition panel. After adding all necessary relations, the user defines mappings. *Mapping script editor* is used to define mappings between parameters; it is placed on the input parameters of relations and the output parameters of an interface.

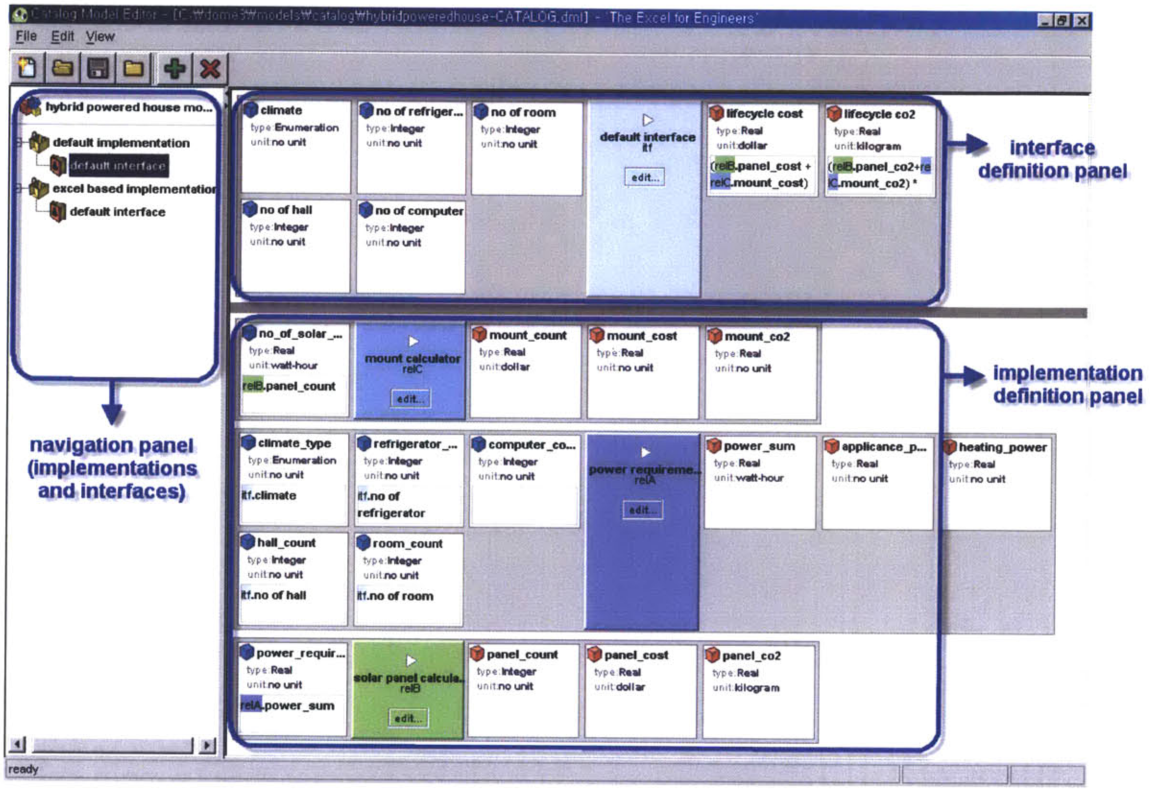
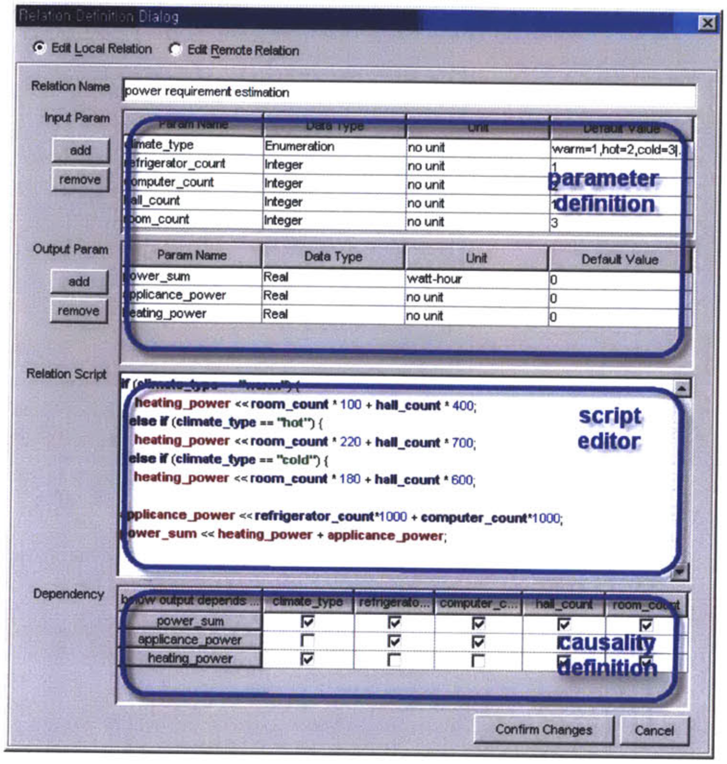

Catalog Model Editor - [C:₩dome3₩models₩catalog₩hybridpoweredhouse-CATALOG.dml] - 'The Excel for Engineers'
File Edit View
hybrid powered house mo...
default implementation
default interface
excel based implementation
default interface
climate type:Enumeration unit:no unit
no of refriger... type:Integer unit:no unit
no of room type:Integer unit:no unit
default interface itf
edit...
lifecycle cost type:Real unit:dollar
(relB.panel_cost + relC.mount_cost)
lifecycle co2 type:Real unit:kilogram
(relB.panel_co2+relC.mount_co2) *
no of hall type:Integer unit:no unit
no of computer type:Integer unit:no unit
interface definition panel
no_of_solar_... type:Real unit:watt-hour
relB.panel_count
mount calculator relC
edit...
mount_count type:Real unit:dollar
mount_cost type:Real unit:no unit
mount_co2 type:Real unit:no unit
implementation definition panel
climate_type type:Enumeration unit:no unit
itf.climate
refrigerator_... type:Integer unit:no unit
itf.no of refrigerator
computer_co... type:Integer unit:no unit
power requireme... relA
edit...
power_sum type:Real unit:watt-hour
applicance_p... type:Real unit:no unit
heating_power type:Real unit:no unit
hall_count type:Integer unit:no unit
itf.no of hall
room_count type:Integer unit:no unit
itf.no of room
navigation panel (implementations and interfaces)
power_requir... type:Real unit:no unit
relA.power_sum
solar panel calcula... relB
edit...
panel_count type:Integer unit:no unit
panel_cost type:Real unit:dollar
panel_co2 type:Real unit:kilogram
ready

Figure 4-13: Catalog model editor layout

The local relation definition dialog is used to add or edit a local relation. It is accessed by *add(+) button* in the tool bar or *edit... button* on the center block of the relation and interface bar. When adding a new local relation, a user first populates a list of parameters using *add* or *remove button* on the left side of the parameter definition section in Figure 4-14. Next, using the relation script editor in the middle of Figure 4-14, the user writes script code, which will be evaluated by the Groovy script engine, to define a transformation from input parameters to output parameters. The script editor help users write the script code efficiently by providing parameter name completion and parameter name highlighting: the parameter name completion shows a popup containing parameter names, and the parameter name highlighting set the color of input parameter names blue while setting the color of output parameter names red. In the causality definition section, the user specifies dependency among parameters, and clicks ***confirm button*** to add a local relation into the implementation definition panel.



● Edit Local Relation ○ Edit Remote Relation

Relation Name: power requirement estimation

Input Param

Param Name	Data Type	Unit	Default Value
imate_type	Enumeration	no unit	warm=1,hot=2,cold=3
efrigerator_count	Integer	no unit	1
omputer_count	Integer	no unit	2
all_count	Integer	no unit	1
oom_count	Integer	no unit	3

Output Param

Param Name	Data Type	Unit	Default Value
ower_sum	Real	watt-hour	0
pplicance_power	Real	no unit	0
eating_power	Real	no unit	0

Relation Script

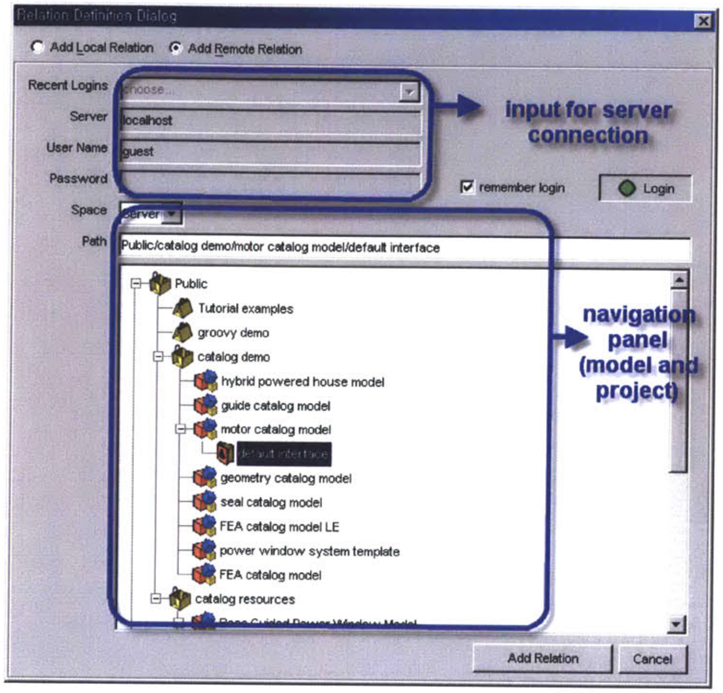
```
if (climate_type == "warm") {
 heating_power <<room_count * 100 + hall_count * 400;
else if (climate_type == "hot") {
 heating_power <<room_count * 220 + hall_count * 700;
else if (climate_type == "cold") {
 heating_power <<room_count * 180 + hall_count * 600;

pplicance_power << refrigerator_count*1000 + computer_count*1000;
ower_sum << heating_power + applicance_power;
```

Dependency

below output depends ...	climate_type	refrigerato...	computer_c...	hall_count	room_count
power_sum	☑	☑	☑	☑	☑
applicance_power	☐	☑	☑		
heating_power	☑	☐	☐	☑	☑

Figure 4-14: Local relation definition dialog layout

The remote relation definition dialog and the local relation definition can be switched by a radio button located at the top of both dialogs. The remote relation definition dialog is used to add or replace a subscription to a model interface. After switching to the remote relation definition dialog, a user specifies server URL, user name, and password to establish a connection to a DOME server as shown in Figure 4-15. Once connected to a DOME server, the user uses the navigation panel to navigate through simulation models in the server. The user clicks ***add button*** to add a remote relation into the implementation definition panel. This dialog is also used to replace the subscription of a model interface with the subscription of another model interface. Because the mapping script of the replaced subscription is copied to the replacing subscription if they have the same parameter names, we can save the effort of re-defining mapping scripts.



Relation Definition Dialog

○ Add Local Relation ● Add Remote Relation

Recent Logins: choose...

Server: localhost

User Name: guest

Password:

☑ remember login

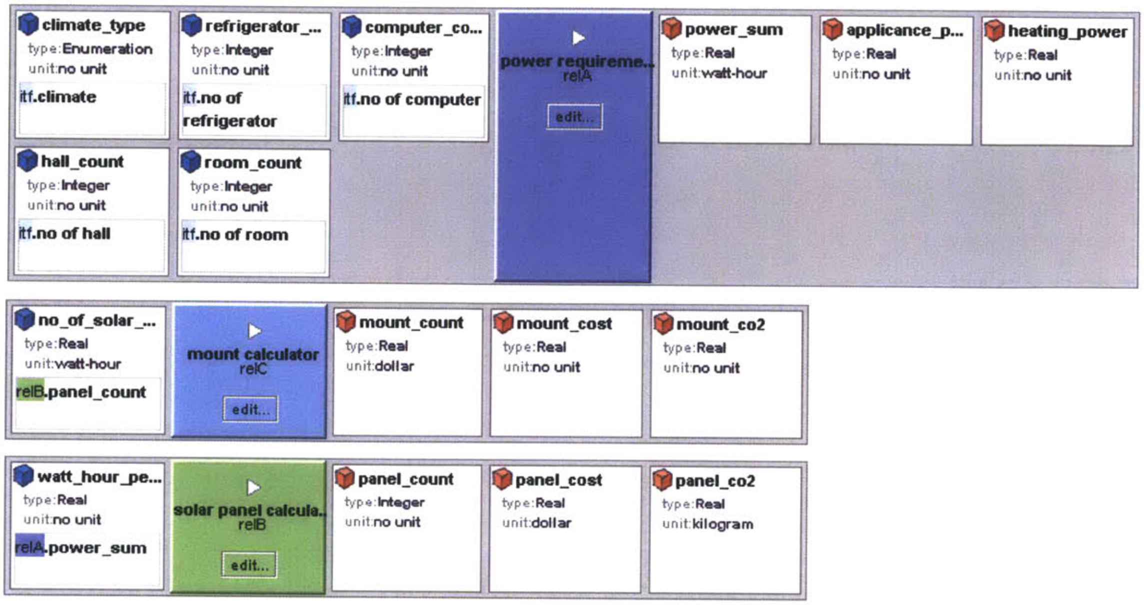
Space: server

Path: Public/catalog demo/motor catalog model/default interface

Figure 4-15: Remote relation subscription dialog layout

4.5.2. Model representation

This section will describe what kinds of graphical structures have been used to represent parameters, relations, interface, and mappings in the catalog model builder. A parameter is represented by a small square, called ***parameter cell*** (Figure 4-16). Four types of parameter cell are provided corresponding to four types of parameter: interface input parameter, interface output parameter, relation input parameter, and relation output parameter.



climate_type
type:Enumeration
unit:no unit
itf.climate

refrigerator_...
type:Integer
unit:no unit
itf.no of refrigerator

computer_co...
type:Integer
unit:no unit
itf.no of computer

power requireme...
relA
edit...

power_sum
type:Real
unit:watt-hour

applicance_p...
type:Real
unit:no unit

heating_power
type:Real
unit:no unit

hall_count
type:Integer
unit:no unit
itf.no of hall

room_count
type:Integer
unit:no unit
itf.no of room

no_of_solar_...
type:Real
unit:watt-hour
relB.panel_count

mount calculator
relC
edit...

mount_count
type:Real
unit:dollar

mount_cost
type:Real
unit:no unit

mount_co2
type:Real
unit:no unit

watt_hour_pe...
type:Real
unit:no unit
relA.power_sum

solar panel calcula..
relB
edit...

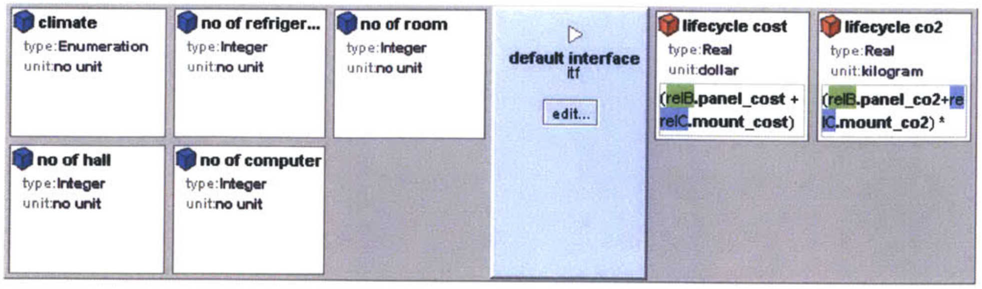
panel_count
type:Integer
unit:no unit

panel_cost
type:Real
unit:dollar

panel_co2
type:Real
unit:kilogram

Figure 4-16: Graphical representation of parameters and relations in the model editor

All parameter cells commonly have fields for the parameter name, the data type, and the unit, while the mapping script editor is only placed on the interface output parameter cells and the relation input parameter cells as shown in Figure 4-16 and Figure 4-17. Each parameter cell has a blue or red cube icon at the top left corner of a parameter cell, which is used to provide causality information: the blue color means input causality and the red color output. Further description on dependency visualization will be covered in Section 4.5.3. A relation is represented by a bar consisting of three blocks: a left block for input parameters cells, a right block for output parameter cells, and a center block for the relation name and the relation alias.



climate
type:Enumeration
unit:no unit

no of refriger...
type:Integer
unit:no unit

no of room
type:Integer
unit:no unit

default interface
itf
edit...

lifecycle cost
type:Real
unit:dollar
(relB.panel_cost + relC.mount_cost)

lifecycle co2
type:Real
unit:kilogram
(relB.panel_co2+relC.mount_co2) *

no of hall
type:Integer
unit:no unit

no of computer
type:Integer
unit:no unit

Figure 4-17: Graphical representation of the model interface in the model builder

4.5.3. Dependency visualization

We have developed a visualization technique to inform users of relevant dependency information and help them avoid erroneous mappings causing a loop in dependency. When we edit a mapping script of a certain parameter denoted, the loop in dependency is created if the mapping script refers other parameter that is affected by the parameter. Therefore, visualizing which parameters are affected by the current edited parameter would prevent users from creating such an erroneous mapping.

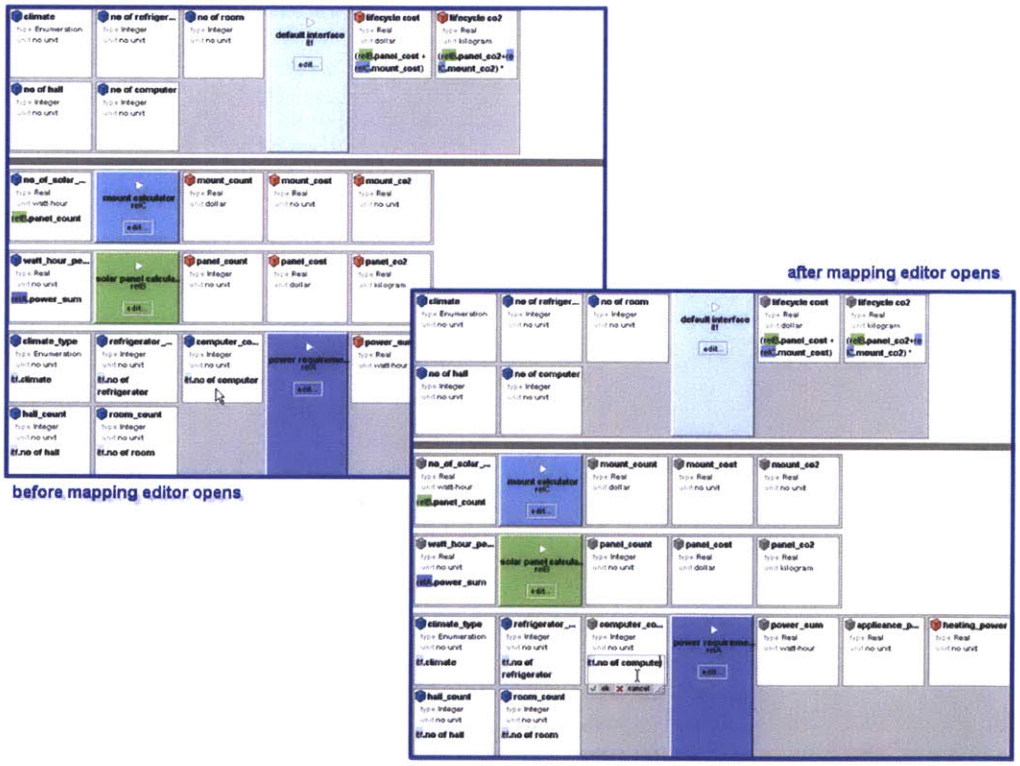
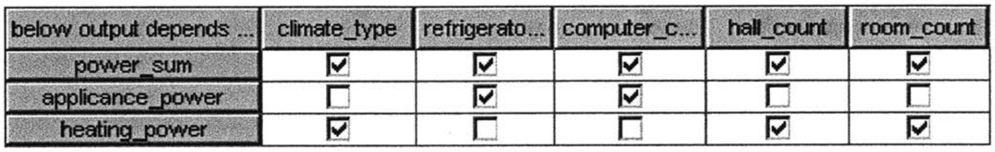


Figure 4-18: Dependency visualization with colors: Parameters affected by the edited parameter turn gray, indicating that they are not valid targets of reference-by-click.

To visualize the information, we decided to use the color change of cubes on the parameter cells: In the initial status, cubes of input parameter cells are blue, and cubes of output parameter cells are red, but, when the mapping script editor of a certain parameter opens, all parameters affected by it will change the color of their cubes to grey. An example of the color change is shown in Figure 4-18: When a user opens the mapping script editor for an input parameter, all parameters affected turns grey – some are directly affected, while others are affected through a chain of dependency starting from the parameter. When this colorized visualization is compared to the matrix-based dependency visualization used for defining local relation shown in Figure 4-19, it shows that the same dependency information can be utilized more effectively for mapping depending on the way how the information is delivered.



below output depends ...	climate_type	refrigerato...	computer_c...	hall_count	room_count
power_sum	☑	☑	☑	☑	☑
applicance_power	□	☑	☑	□	□
heating_power	☑	□	□	☑	☑

Figure 4-19: Matrix-based dependency visualization: efficient for editing, but not so for mapping

4.5.4. Reference-by-click

The reference-by-click feature has been invented to ease the burden of writing reference strings in the mapping script editor. Instead of typing a reference string, clicking on a parameter name will add a reference string to the parameter in the mapping script editor. As shown in Figure 4-20, when a user wants to map *power_required* in *solar panel calculator* relation to *power_sum* in *power requirement* relation, it can be accomplished by clicking on the parameter name of *power sum*.

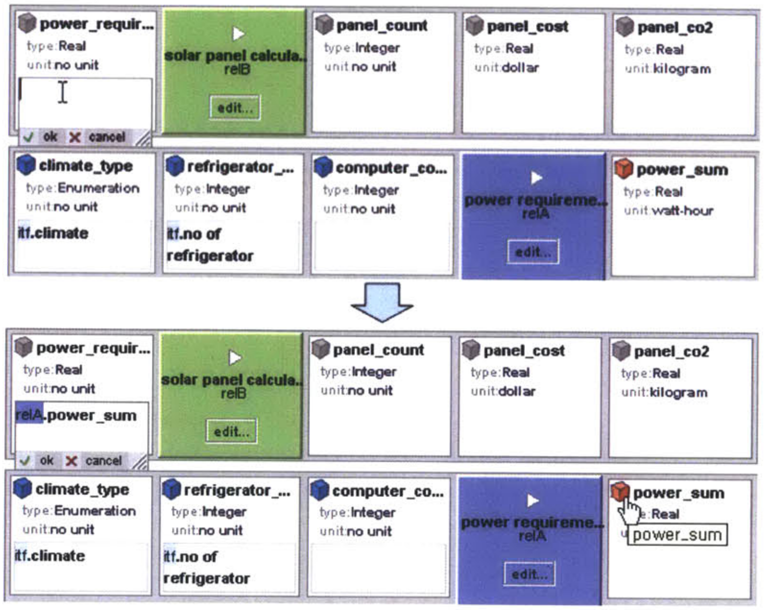


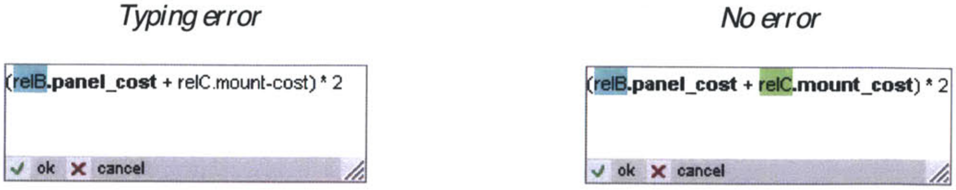
Figure 4-20: A click on parameter name allows user to insert a reference string to the parameter.

However, there is a case when reference-by-click becomes less usable. When mapping source parameters are far from a mapping target parameter, because of other relations placed between them, users may need to scroll down the model editor pane until they can click the name of the source parameter. Moreover, if other parameters next to the target parameter need similar mappings to other parameters in the remote relation, they may have to scroll up and down laboriously.

One way to solve this problem is to allow users to reorder relations in the editor pane and put two relations of their interest next to each other. Such flexibility in visualization is accomplished because the dependency solving mechanism decouples the execution sequence of relations from the spatial sequence of relations and generates the proper execution plan from the causality information. Using code completion feature is another way of solving it. Once we have defined mapping scripts for the first a few of the mapping target parameters with reference-by-click, mapping scripts for others can be generated by reusing the code patterns.

4.5.5. Color-coding

The mapping editor utilizes color-coding technique to give a colorized visual feedback of the typed string; as a result, it not only prevents typing errors but also ease the visual search of the source parameter. When a typed string is not valid, having no match with parameter names in the simulation model, the string will be displayed with no decoration as a thin black font. For a valid string having a match to one of the parameter names, the relation alias part of the string will have the same background color as the center block of the referred relation; also the parameter name part will be made bold. Figure 4-22 and Figure 4-21 show how this color-coding strategy has been implemented in the editor. One issue to be addressed in the future is to provide secondary cues that convey the information to those have color blindness; the differentiation in other graphical properties such as brightness or texture pattern can be used as secondary cues. Also, because several forms of color blindness such as red/green blindness are much more common than others, the current randomized color selector could be improved so that it avoids picking specific combination of colors in a simulation model.



Typing error

(relB.**panel_cost** + relC.mount-cost) * 2

No error

(relB.**panel_cost** + relC.**mount_cost**) * 2

Figure 4-21: Color-coding helps detecting typing errors

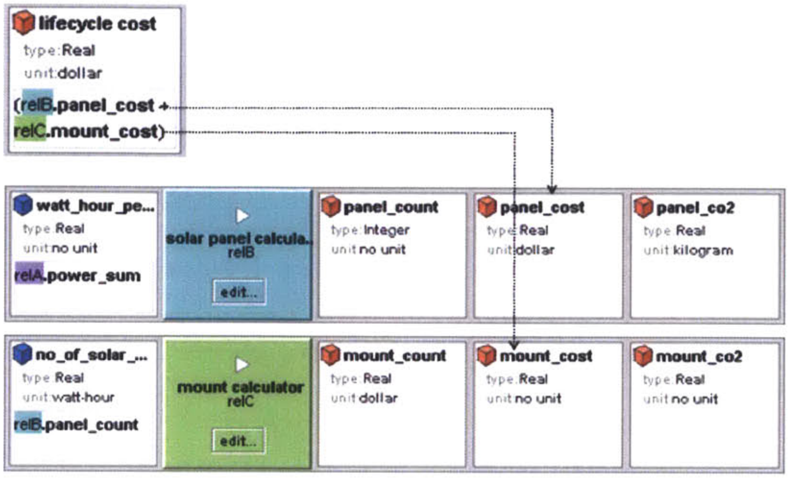


Figure 4-22: Color-coding helps ease visual search of source parameters.

4.5.6. Code completion popup

Pattern generalization algorithm generates candidates for the mapping script completion. Code completion popup is used to display, narrow down, and finally select from the candidates. The usability of the code completion popup has a significant effect on the productivity gain we can achieve from the algorithm, and therefore it should be designed as efficient as it can be. We approach this problem in two steps. First, we learn interaction conventions from code completion features of software development tools; users who have an experience of using such tools are expected to try to use our code completion feature in the same manner. This kind of interaction conventions includes "code completion feature is accessed by hitting a space key with control down," "users can iterate through code completion candidates using the arrow-up key and the arrow-down key," "hitting enter-key inserts the selected candidate into the code editor," and "the list of code completion candidates can be narrowed down by typing more literals" (Figure 4-23).

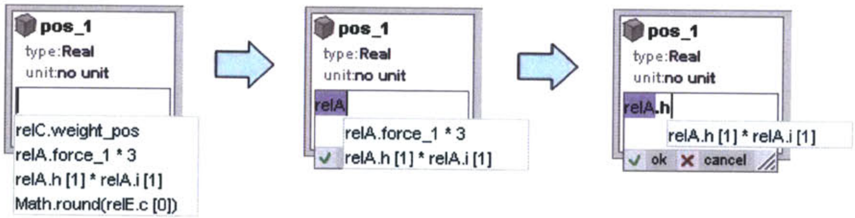


Figure 4-23: A user can narrow down the code completion candidates by typing more literals.

As a second step, we identify the difference between user behaviors assumed by the current interaction conventions and those observed from the catalog model builder; it leads us to improve the interaction specialized for the catalog model builder. As we can infer from the fact that the current interaction conventions for the code completion popup are mostly based on keyboard-based interactions, the current conventions assume that both hands are on the keyboard when the code completion is requested. However, in the case of the catalog model builder, we observe that users' hands stay on the mouse most of the time. It is because when users define mapping scripts, a task that accounts for a significant portion of the time spent for the model integration, most mapping scripts can be done with mouse control: Many mapping scripts can be completed by one or two reference-by-clicks bridged with mathematical symbols such as +, *, and /. Therefore, the conventional interaction of pressing a space with control down and using arrow keys for navigation is not considered as the best way to use our code completion feature. We suggest that the code completion feature of the catalog model builder be easily accessible with mouse control, without having to move a hand to the keyboard. Clicking with control down is a proposed solution, whose combination is expected to be easy to remember using the analogy of pressing a space key with control down.

The final implementation of code completion popup supports three modes of activation (Figure 4-24). It can be activated by pressing the space key with control down, clicking on the mapping script editor with control down, and clicking on the mapping script editor with both control and shift down. The last mode is the same as the second mode except for the fact that it clears the existing mapping script before it generates code completion candidates. The default behavior of code completion, which is activated by the first and second mode, consumes all written literals in the mapping script and generates candidates based on that. However, in some cases when a user thinks the existing mapping script is not relevant and wants to replace it, the default behavior will supply too many unrelated literals and leave no room for generating fresh candidates. Introducing the third mode solves this problem by giving us an option to ignore the existing mapping script; thereby it accommodates the case when a user needs to replace the existing mapping script with one of the code completion candidates.

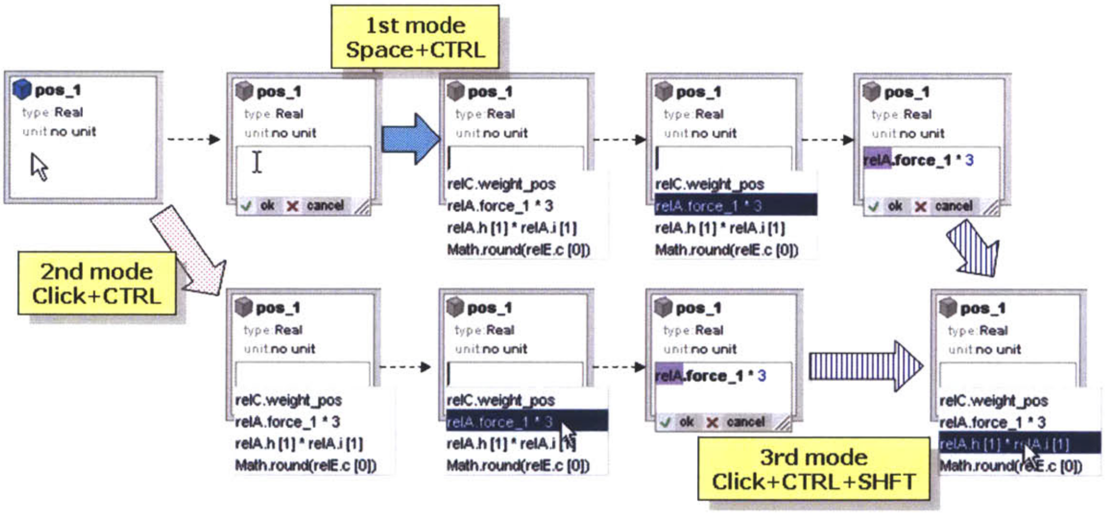


Figure 4-24: Code completion popup supports three modes of activation

Chapter 5 Evaluation

In the first part of this chapter, we describe several integration scenarios, through which we validate the effectiveness of reuse features of our design tool. The integration scenarios include the door seal catalog browser, the evaluation of a power window system with various configurations, and the finite element analysis (FEA) of the door seal with multiple levels of fidelity.

5.1. Door Seal Catalog Model

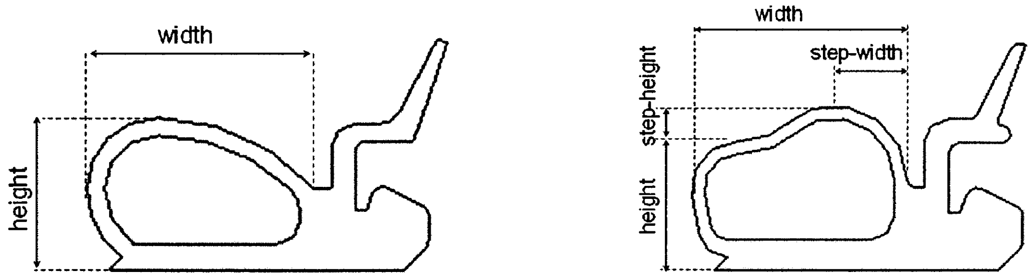
5.1.1. Evaluation goal

The first demo application called door seal catalog model is built to evaluate if the catalog model satisfies the basic requirement of decoupling the implementation from the interface. The requirement can be further decomposed into two sub-level requirements: allowing multiple implementations to be associated with one interface and supporting switching between implementations in the run-mode. An implication of this decoupling goal is that we can utilize a parametric model that allows substantially larger variations than typical parametric models. From a user point of view, the catalog model is seen as a typical parametric model having an implementation switch as one of its model parameters. However, the catalog model can exhibit a much wider range of model behaviors, when compared to typical parametric models, because its variation in behavior is not based on parameterization of one simulation model, but on many different simulation models.

5.1.2. Configuration of the door seal catalog model

The variation in geometry originating from different design concepts is one of the tricky variations that are not easily handled by parameterization. Our test application, the door seal catalog model, will deal with that kind of geometric variation. We have two door seal designs which are derived from two significantly different design concepts. The first is the current door seal design with a round-shape bulb, while the second is a new door seal design inspired by an idea that a convex-shaped bulb, having the shape of a deformed round bulb, may reduce door closing force and energy while providing a comparable level of wind noise blockage. Because the variation between two door seal designs cannot be handled by parametric changes to the existing round-shaped model, a new door seal geometry model has been built for the test application. Two door seal geometry models are created using CATIA [38], a computer aided modeling tool, and they are named as *round-seal.CATPart* and *convex-seal.CATPart* (Figure 5-1).

This door seal geometry model will be used as a part of another integration model, and we assume that the role of this door seal geometry model is to provide other simulation models with an IGES file and a VRML file so that they can perform further engineering analysis, such as estimating door seal stiffness and estimating door closing force.



door seal with a round bulb

(round-seal.CATPart)

door seal with a convex bulb

(convex-seal.CATPart)

Figure 5-1: Two door seal designs modeled in CATIA

The following steps have been taken to create a catalog model satisfying the requirements described above:

1. Two DOME CATIA plug-in models wrapping each of the two CATIA files is created and deployed on a DOME server. They are named as *round door seal model* and *convex door seal model.*

2. A catalog model named door seal catalog model is created. An existing implementation called *default implementation* is renamed as *round seal implementation.* A new implementation called *convex seal implementation* is added to the catalog model.

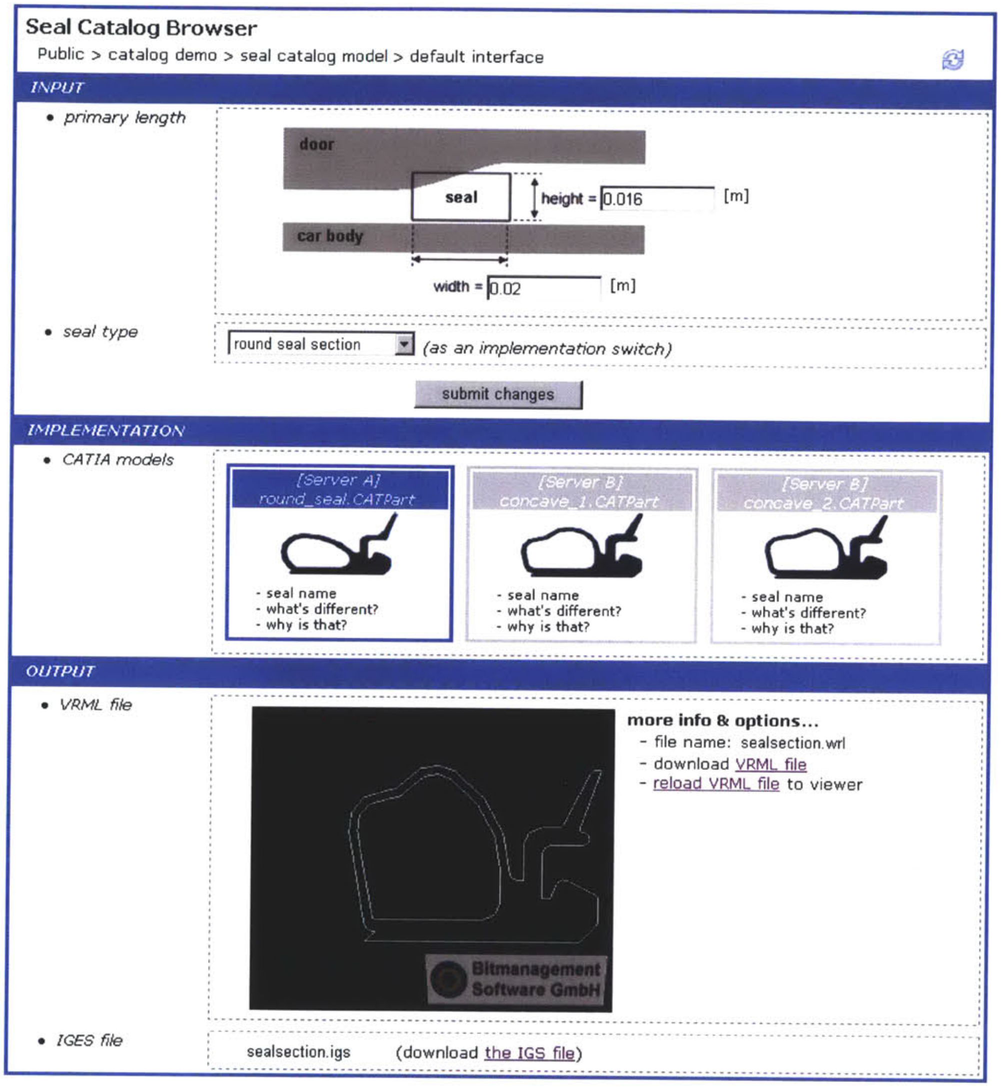
3. Modify interface parameters of the catalog model so that it has input parameters of *width* and *height* and output parameters of *IGES file* and *VRML file.*

4. Use the implementation navigation panel to open the implementation of the *round seal implementation.* Add a remote relation subscribing a model interface of the *round door seal model.* Map interface input parameters to input parameters of the remote relation. Also map output parameters of the remote relation to interface output parameters.

5. Use the implementation navigation panel to open the implementation of the *convex seal implementation.* Add a remote relation subscribing a model interface of the *convex door seal model.* Map interface input parameters to input parameters of the remote relation. Also map output parameters of the remote relation to interface output parameters.

6. Save the *door seal catalog model,* and deploy the generated files to a DOME server.

7. Now the *door seal catalog model* can be opened and executed by a DOME browser. To allow an access to the model with standard web browsers like Mozilla Firefox, a web page has been developed based on DOME run-mode API and Java Server Page (JSP) technology (Figure 5-2).



Seal Catalog Browser

Public > catalog demo > seal catalog model > default interface

INPUT

- primary length
  - height = 0.016 [m]
  - width = 0.02 [m]
- seal type: round seal section (as an implementation switch)

IMPLEMENTATION

- CATIA models

OUTPUT

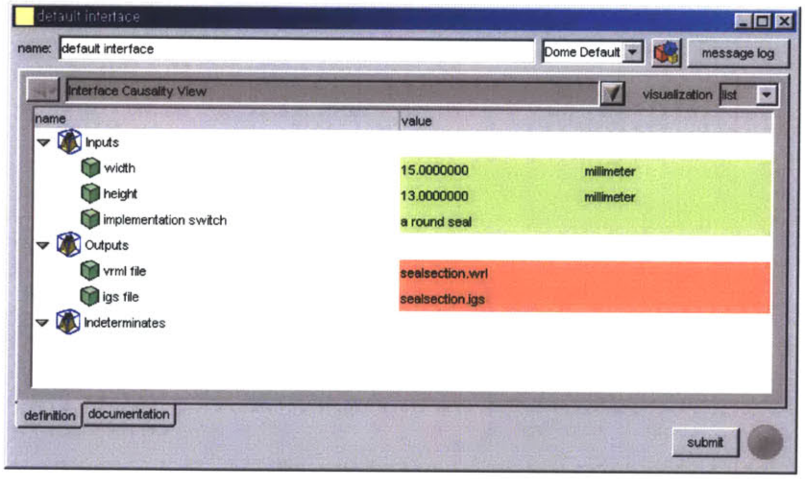
- VRML file
  - more info & options...
    - file name: sealsection.wrl
    - download VRML file
    - reload VRML file to viewer
- IGES file: sealsection.igs (download the IGS file)

door
seal
car body
submit changes
[Server A] round_seal.CATPart
[Server B] concave_1.CATPart
[Server B] concave_2.CATPart
- seal name
- what's different?
- why is that?
Bitmanagement Software GmbH

Figure 5-2: A web page developed for executing the *door seal catalog model*

5.1.3. Evaluation result

As we see in Step 4 and Step 5 in previous Section 5.1.2, the catalog allows us to add multiple implementations associated with one interface. This feature is achieved by the model definition component's capability to organize implementation and interface data with many-to-one relationship as well as the implementation dispatch mechanism that transform the model definition data into an executable form. Also the second sub-level requirement of switching implementation in the run-mode is satisfied, because the door seal catalog model provides a parameter named *implementation switch* which allows us to decide which implementation to use for each execution of the model interface as shown in Figure 5-3. Finally, the ***door seal catalog model***, utilizing two different implementations based on two CATIA models, has shown that it can produce IGES file having a large geometric variation: Not only width and height can be changed, but also the shape of the seal can be varied. From this representative application of the catalog model, we can conclude that the catalog model accomplishes the goal of decoupling the implementation from the interface.



name: default interface

Interface Causality View

visualization: list

name	value	
Inputs		
width	15.0000000	millimeter
height	13.0000000	millimeter
implementation switch	a round seal	
Outputs		
vrml file	sealsection.wrl	
igs file	sealsection.igs	
Indeterminates		

Figure 5-3: The *door seal catalog model* opened in a DOME browser

5.2. Power Window System Template

5.2.1. Evaluation goal

The second demo application called *power window system template* is to demonstrate how the catalog model can be used to realize the goal of reusing integration structure. As we have described in Section 2.1, we can reuse the integration structure of an integration model if the integration model is built with simulation models supporting the decoupling of the implementation from the interface; we call such an integration model as *system template*, indicating that the integration model can be served as a template for creating many different systems. The catalog model provides the decoupling mechanism, so it can be employed to build a system template. To be a practical solution for the integration structure reuse, the catalog model also needs to satisfy another implementational requirement: it has to be interoperable with existing DOME models. Not only the catalog model should be able to subscribe DOME models, which has already been implemented by the remote relation and demonstrated in Section 5.1, but also the DOME integration project should be able to subscribe the catalog model as its resource. As a test of the interoperability issue, the power window system template will use a DOME integration project as an integration model and import catalog models as resources for the DOME integration project.

5.2.2. Configuration of the power window system template

The power window system template, which itself is a DOME integration project, integrates three catalog models: *window catalog model*, *motor catalog model*, and *guide catalog model*. The information flow inside the power window system template is shown in Figure 5-4.

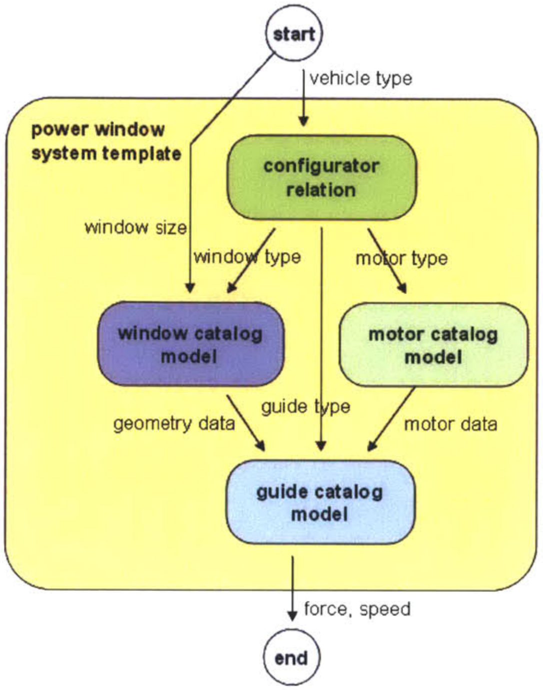


Figure 5-4: Data flow among simulation models

When the system template receives values for *vehicle type* and *window size* from its model interface, it passes *vehicle type* and *window size* to a local relation called *configurator relation* and a remote relation of the window catalog model, respectively. The configurator relation is a procedural relation to select appropriate *window type*, *motor type*, and *guide type* based on *vehicle type*; it has been implemented by a set of if-then rules. Once the configurator relation passes *window type* to the window catalog model, the model is executed, and as a result window properties, described in Table 5.1, are passed to the guide catalog model. Similarly, *motor type* is passed from the configurator relation; the motor catalog model is executed; and the motor properties, described in Table 5.1, are passed to the guide catalog model.

Catalog model name	*Input / output parameters*
Window catalog model	*Input:* implementation switch (window type: F150 or Fusion) *Output:* A pillar length, B pillar length

	A upper pillar length, B upper piller length A lower pillar length, B lower pillar length glass area, A to B edge length
Motor catalog model	*Input:* implementation switch (motor type: Danaher or Groschopp) *Output:* speed at 2nm torque, torque at 20rpm speed stall current, free current, voltage, resistance
Guide catalog model	*Input:* A pillar glass length, B pillar glass length A pillar upper runner length, B pillar upper runner length A pillar lower runner length, B pillar lower runner length glass mass, A to B edge length motor speed at 2NM, motor torque at 20 rpm current draw at stall, motor free current power supply voltage, inline resistance *Output:* max velocity, max stall force speed0, speed1, speed2, speed3, speed4, speed5 force0, force1, force2, force3, force4, force5

Table 5.1: Model interface of three catalog models

Finally, when the window catalog model and motor catalog model copy their results to the guide catalog model, the guide catalog model calculates *force* and *speed* and copies them to the model interface of the power window system template, whose definition is shown in Table 5.2.

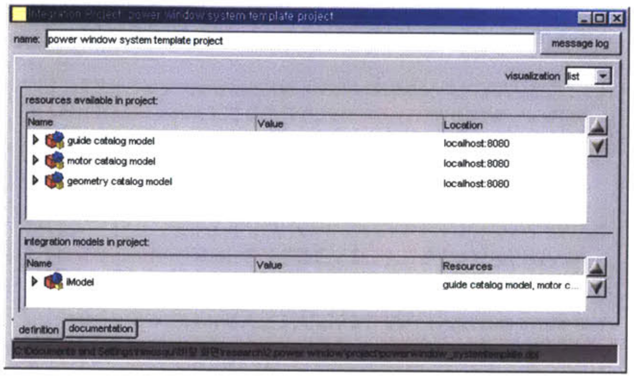
Integration project name	*Input / output parameters*
Power window system template	*Input:* vehicle type (F150 or Fusion) window width, window height *Output:* max force, max speed, guide type, geometry type, motor type speed0, speed1, speed2, speed3, speed4, speed5 force0, force1, force2, force3, force4, force5 torque at 20 rpm, rpm at 2 Nm, glass area, base width, stall current, free current, A lower pillar length

Table 5.2: Model interface of the power window system template, a DOME integration project

The following steps have been taken to set up the power window system template based on three catalog models, which are assumed to be already built through the same steps we have described in Section 5.1.2:

1. First create A DOME integration project called *power window system template.*

2. Create an integration model called *iModel* in the project, and add resources of three catalog models to the *power window system template* (Figure 5-5)



Integration Project: power window system template project

name: power window system template project

message log

visualization: list

resources available in project:

Name	Value	Location
guide catalog model		localhost:8080
motor catalog model		localhost:8080
geometry catalog model		localhost:8080

integration models in project:

Name	Value	Resources
iModel		guide catalog model, motor c...

definition | documentation

Figure 5-5: Three catalog models are added to a DOME integration project as available resources

3. Open *iModel,* and subscribe three model interfaces from the catalog models.

4. Add a procedural relation containing input and output parameters defined in Table 5.1 Write a script that transforms *vehicle type* into *window type, motor type,* and *guide type* based on a set if-then rules

5. Define all necessary mappings among subscribed parameters and relation parameters.

6. Create a project interface having input and output parameters defined in Table 5.2.

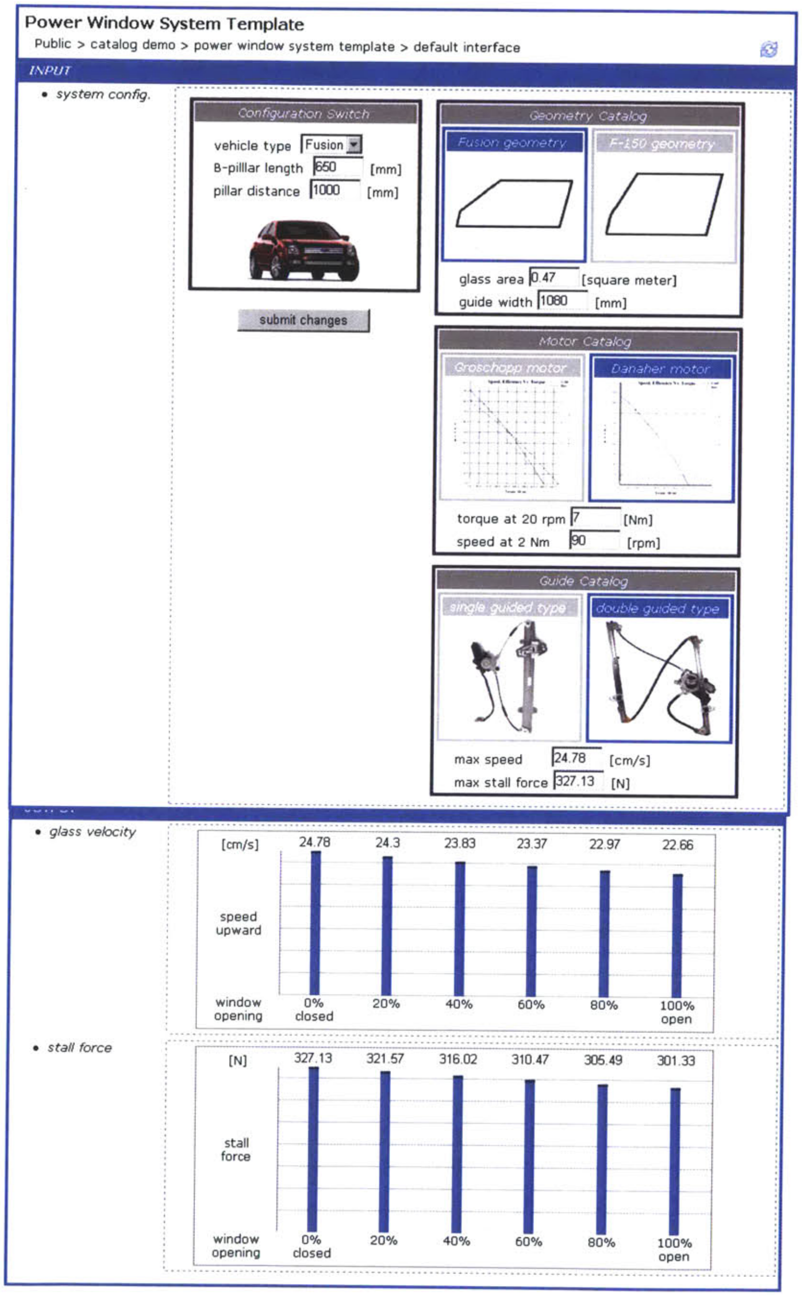
7. Deploy the *power window system template* on a DOME server.

8. Now the *power window system template* can be opened and executed by a DOME browser. A web page for accessing the system template has been developed based on DOME run-mode API and JSP technology.

5.2.3. Evaluation result

As depicted in the graphical user interface of the *power window system template* shown in

Figure 5-6, the resulting system template lets us explore various design opportunities without having to modify underlying integration structures. The successful execution of the demo application also signifies that the issue of interoperability between the catalog model and the DOME integration project has been resolved effectively. Because we have built the system template by subscribing catalog models, instead of by subscribing a certain simulation model directly, we can switch the implementation of its catalog models without having to affecting the overall integration structure; consequently our goal of reusing integration structure is achieved. The power window system template also addresses the issue of model configuration. Because implementation switches of catalog models can be mapped and transformed just like other parameters, we have developed a relation, called the configurator relation, which can coordinate implementation switches within the power window system template. The current version of the configurator is a simple procedural relation based on a few if-then rules, but it considerably reduces the burden of configuring catalog models one by one.



Power Window System Template

Public > catalog demo > power window system template > default interface

INPUT

- system config.

Configuration Switch

vehicle type Fusion

B-pilllar length 650 [mm]

pillar distance 1000 [mm]

submit changes

Geometry Catalog

Fusion geometry | F-150 geometry

glass area 0.47 [square meter]

guide width 1080 [mm]

Motor Catalog

Groschopp motor | Danaher motor

torque at 20 rpm 7 [Nm]

speed at 2 Nm 90 [rpm]

Guide Catalog

single guided type | double guided type

max speed 24.78 [cm/s]

max stall force 327.13 [N]

- glass velocity

- stall force

Figure 5-6: The power window system template demonstrating integration structure reuse: catalog models can switch their implementations without having to change the integration structure

5.3. Simulation models with Different Levels of Fidelity

5.3.1. Evaluation goal

The goal of the third demo application is to show the catalog model's capability to create a simulation model that can provide different levels of fidelity, which are also configurable in the run-mode. Such a capability is valuable when a simulation is used for multiple purposes which have different preferences on the trade-off relation between accuracy and time: some may prefer a very accurate, but rather slow simulation model, while others may prefer a fast model because of the tight time constraint. One solution to this challenge would be utilizing two simulation models serving different needs; however, this approach is not very efficient because it will require us to re-integrate simulation models for each level of fidelity. Also having to manage multiple simulation models with similar functionality can be another source of inefficiency because of the version control issues. The catalog model is a promising solution to meet the need because one catalog model can hold multiple implementations from simulation models with different levels of fidelity. A catalog model called *seal FEA catalog* will be built to verify this promising option.

5.3.2. Configuration of the seal FEA model

The seal FEA model subscribes two simulation models: ABAQUS-based seal FEA model called *full FEA model* and MATLAB-based neural-network model approximately emulating the behavior of the ABAQU-based model called *approximated FEA model*. The full FEA model is based on an ABAQUS model utilizing a time step analysis technique. To emulate the behavior of the *full FEA model*, a neural network is trained from fifty sample points and also validated against fifty points. The setup of neural network used in training is as follows:

- Number of inputs : 4 (seal_width, seal_height, seal_thickness, seal_gap)

- Number of outputs: 3 (contact_length, deflection_nom, load_per_length)
- Number of sampling points: 108

  seal_width: 20, 21, 22, 23 [mm]

  seal_height: 11, 12, 13 [mm]

  seal_thickness: 0.85, 1.0, 1.15 [mm]

  seal_gap: 7, 8, 9 [mm]
- Transfer function:

  input layer: logsig

  output layer: logsig
- Training performance: net error 0.0018~0.0041 (depending on randomly-generated seeds)

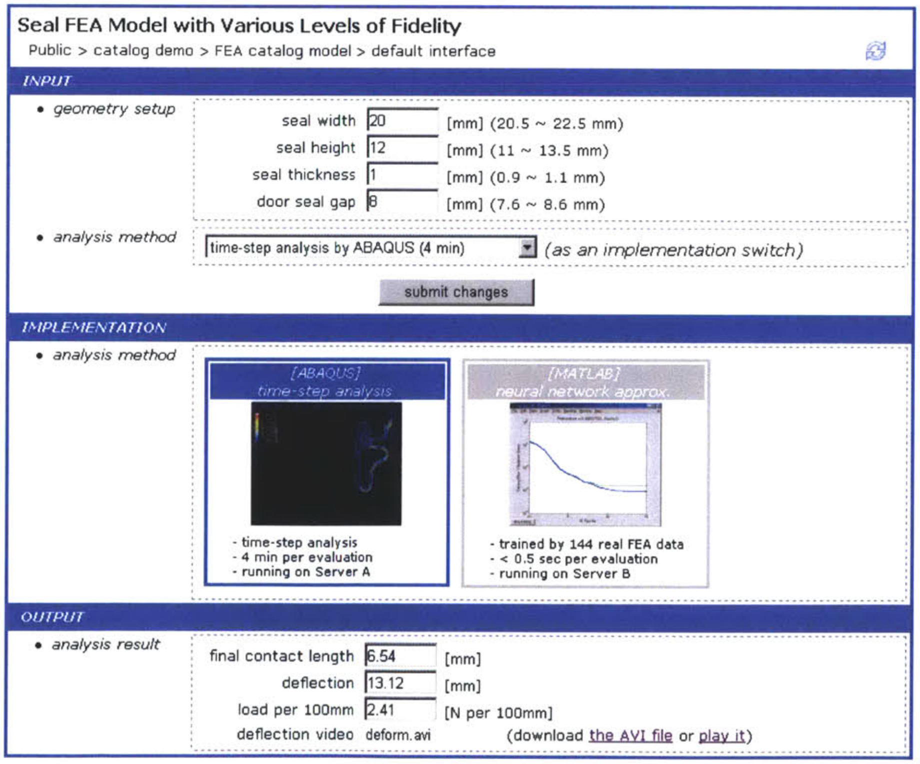
The model interface of the FEA emulation model has been created so that it matches the model interface of the *full FEA model* (Table 5.3). Note that both models have an output parameter called deform video, but the approximated FEA model will not generate a video file because the current neural-network setup lacks the capability to generate a video data from a given set of video data; instead, parameter *deform video* returns an empty file.

Simulation model name	*Input / output parameters*
Full FEA model (ABAQUS plug-in model) and Approximated FEA model (MATLAB plug-in model)	*Input:* seal width seal height seal thickness seal gap *Output:* final contact length deflection nom load per 100 mm deform video

Table 5.3: To emulate behaviors of the *full FEA model*, the model interface of the *approximated FEA model* has been copied from that of the *full FEA model*

Now we have two simulation models that have the same model interface, but provide different levels of fidelity. Following steps will create a catalog model and add the two simulation models as its implementations:

1. Create a catalog model called *seal FEA catalog model*.
2. Rename default implementation as *full FEA implementation*.
3. Add a new implementation called *approximated FEA implementation*.
4. Open *full FEA implementation*. Add a remote model subscribing *full FEA model* and define mapping between interface parameters and relation parameters.
5. Open *approximated FEA implementation*. Add a remote model subscribing *approximated FEA model* and define mapping between interface parameters and relation parameters.
6. Deploy the *seal FEA catalog model* on a DOME server.
7. Now the *power window* system *template* can be opened and executed by a DOME browser. Also a web page for accessing the catalog model has been developed.



Seal FEA Model with Various Levels of Fidelity

Public > catalog demo > FEA catalog model > default interface

INPUT

- *geometry setup*

seal width 20 [mm] (20.5 ~ 22.5 mm)
seal height 12 [mm] (11 ~ 13.5 mm)
seal thickness 1 [mm] (0.9 ~ 1.1 mm)
door seal gap 8 [mm] (7.6 ~ 8.6 mm)

- *analysis method*

time-step analysis by ABAQUS (4 min) *(as an implementation switch)*

IMPLEMENTATION

- *analysis method*

[ABAQUS]
time-step analysis

- time-step analysis
- 4 min per evaluation
- running on Server A

[MATLAB]
neural network approx.

- trained by 144 real FEA data
- < 0.5 sec per evaluation
- running on Server B

OUTPUT

- *analysis result*

final contact length 6.54 [mm]
deflection 13.12 [mm]
load per 100mm 2.41 [N per 100mm]
deflection video deform.avi (download the AVI file or play it)

Figure 5-7: The seal FEA catalog allows switching between simulation models having different

levels of fidelity. While the higher fidelity seal FEA model takes 4 minutes for one execution, the lower fidelity one takes less than 1 second.

5.3.3. Evaluation result

Through the steps described in the previous section, we have built a catalog model called the *seal FEA catalog*. The seal *FEA catalog* provides two levels of fidelity: a higher fidelity based on ABAQUS-based simulation and a lower fidelity based on neural network approximation. The implementation switch allows us to adjust the level of fidelity in the run-mode. As a result, the catalog model successfully accomplishes the goal of our demonstration: one simulation model that can be used for multiple purposes requiring multiple levels of fidelity. To estimate the level of fidelity of *approximated FEA model*, the net error of the model has been measured at fifty validation sample points. The error – the normalized vector distance between estimated and original output vectors – is quite small, ranging from 0.0018 to 0.0041, depending on randomly generated seeds used in the neural network training algorithm. The time required to execute each model has shown significant difference: the full FEA model takes average 4.8 min for one evaluation, while the approximated FEA model takes less than 1 second, excluding training time less than 3 seconds, when both models are deployed in the same computer.

A simulation model that can flexibly adjust the level of fidelity has interesting application areas. For example, it can be applied to enhance the optimization process. For a problem with large search space, optimization engines often employ an exploration technique at the beginning of its search steps to create a rough map of the search space. At this step, a faster simulation model with an acceptable level of fidelity is preferred over a slower model with a high level of fidelity because the faster model allows the exploration of many interested areas in a short time. Meanwhile, in the later step of optimization, when the engine is confident about in which region the optimum is, a simulation model with high fidelity is essential to find the exact location of the optimum. Therefore, the catalog model like the *seal FEA catalog* will be a valuable tool to realize such an adaptive optimization strategy.

Chapter 6 Conclusion

6.1. Summary

The goal of our new design tool to reuse two aspects of integration knowledge: the integration structure and the integration code pattern. The integration structure is defined as a graph structure consisting of parameters as nodes and mappings as arcs, which our design tool aims at reusing for other simulation models that are structurally compatible, but functionally different. The integration code pattern is a pattern based on regularity found in the script code of simulation models; the new design tool includes a pattern generalization algorithm and a code completion feature that enables designers to reuse the pattern.

The first use case scenario – in which a car company tries to replace a motor simulation model, which is part of a power window simulation model, with another without having to rebuild or modify the overall integration model – shows reusing integration structure can ease the process of design exploration by saving redundant integration efforts. In the second use case scenario, we have provided several exemplary code patterns, which can be easily generalized by human, but have not handled that well by computers. An algorithm that can perform similar generalization process human can do would saves integration effort because many of mapping scripts used in simulation models have regularity, within one model or within a group of models closely related to each other.

Since a simulation model is a kind of software, we have reviewed concepts for software reusability studied in software engineering to find which can be related to and utilized for our problem of reusing integration knowledge in simulate models. The missing and needed feature for integrated design frameworks is found to be a support of the polymorphism concept. The concept proposes decoupling between implementation and interface, and when it is implemented in a tool for designing integrated simulation models, the concept enables us to reuse the integration structure in simulate models. Another research discipline of code mining provides several techniques relevant to our implementation of integration code pattern reuse. Especially, previous research on code clone detection and example-based code completion has developed various code representation techniques including graph-based, vector-based and token-based.

We derived the list of software components based on the decomposition of functional requirements and design parameters using the axiomatic design method. Major software components implementing two goals of integration knowledge reuse include the catalog model definition, the dependency solving algorithm, the implementation dispatch mechanism, the pattern generalization algorithm, and the user interface components. Consequently, a design tool called the catalog model builder and an integration model called the catalog model have been developed.

We have evaluated how our new design tool satisfies its intended goal using three demo applications: the door seal catalog model, the power window system template, and the seal FEA catalog model. The applications not only show that the basic capability of the catalog model, decoupling implementation from interface and reusing the integration structure, is accomplished, but it also shows that the model can be used for creating simulation models having multiple levels of fidelity.

6.2. Future Work

The catalog model is a tool for integrating simulation models, and naturally the value of a catalog model as an engineering analysis tool relies on those of simulation models it subscribes. Because it allows subscription to DOME simulation models, the catalog model is considered to competent from this point of view: any computational tool for engineering analysis and modeling is represented as a standardized parametric model in the DOME integrated design framework, and thus the catalog model has a transparent access to various computation tools such as ABAQUS, CATIA, and MATLAB. However, the catalog model still has room for improvement because the current implementation subscribes DOME simulation models only, but cannot subscribe other potentially useful simulation models deployed outside the DOME integrated design framework. Since the inherent architecture of the remote relation allows more than one type of remote simulation services, it can be extended to support other simulation services. Considering the fact that the web service is widening its acceptance in the IT industry, supporting subscription to simulation services deployed on a web service container will be a valuable extension of the remote relation.

Supporting a new integration mechanism called *implementation delegate* also interests us. If a catalog model has N kinds of implementations for M interfaces, we need to define N x M implementation instances. For example, when we create a catalog model having five kinds of implementations with four interfaces, we need to twenty implementation instances, which is not a trivial task to do. However, we can reasonably reduce the effort when there are duplicated implementation instances. Such duplication is often introduced because an implementation may have different implementation instances for some of its interfaces, but have shared implementation instances for others. The implementation delegate will allow us to delegate the definition of an implementation instance, called *source implementation instance*, to other implementation instance, and therefore we can save the effort of copying the same implementation instance from one to another. Moreover, the catalog model becomes easier to maintain because a change made to the source implementation instance will be immediately reflected to other implementation instances which have delegated their definition to the source implementation instance.

The pattern generalization algorithm has an ability to find patterns in the source code. In our application to the code completion feature, the pattern is used to generate missing part of the code. Another interesting application of the algorithm is to find irregularity existing in the source code in order to detect possible errors. This task is challenging because it requires us to tell irregularity from randomness. The pattern generalization algorithm is a promising start point to develop an algorithm that can differentiate them because the irregularity is characterized by a consistent pattern in the source code having a few exceptions to the pattern. Introducing additional rules for generalization is also worth investigating. It is more interesting because a trade-off relation regarding introduction of a new rule should be considered: while a new rule may help capture regularity that could not be handled before, it may increase false positive code completion candidates because the added rule may lead the algorithm recognize unacceptable generalization.

Reference

[1] B. Meyer, *Object-Oriented Software Construction, Second Edition*: Prentice Hall, 1997.

[2] G. C. Cay S Horstmann, *Core Java 2*: Prentice Hall, 2001.

[3] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, pp. 471--522, 1985.

[4] C. Strachey, "Fundamental Concepts in Programming Languages," *Higher-Order and Symbolic Computation*, vol. 13, 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995.

[6] D. Box, *Essential COM*: Addison-Wesley, 1998.

[7] D. Slama, J. Garbis, and P. Russell, *Enterprise CORBA*: Prentice Hall, 1999.

[8] R. Englander, *Java and SOAP*: O'Reilly, 2002.

[9] D. Hand, H. Mannila, and P. Smyth, *Principles of Data Mining*. Cambridge, MA: MIT Press, 2001.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, pp. 654-670, 2002.

[11] B. S. Baker, "A Program for Identifying Duplicated Code," presented at Proc. Computing Science and Statistics: 24th Symposium on the Interface, 1992.

[12] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," Bethesda, MD, USA, 1998.

[13] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," Lisbon, Portugal, 2005.

[14] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code " presented at International Symposium on Static Analysis (SAS), 2001.

[15] B. S. Baker, "On finding duplication and near-duplication in large software systems," Toronto, Ont, Can, 1995.

[16] F. Van Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," Linz, Austria, 2004.

[17] S. Ducasse, M. Rieger, and S. Demeyer, "Language independent approach for detecting duplicated code," *Conference on Software Maintenance*, pp. 109-118, 1999.

[18] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting Similar Java Classes Using Tree Algorithms," presented at MSR, Shanghai, China, 2006.

[19] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," St. Louis, MO, USA, 2005.

[20] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," Chicago, IL, United States, 2005.

[21] R. Hill and J. Rideout, "Automatic method completion," Linz, Austria, 2004.

[22] C. Ware, *Information Visualization: Perception for Design*: Morgan Kaufmann Publishers, 2004.

[23] J. Nielsen, *Designing Web Usability : The Practice of Simplicity*: New Riders Press, 1999.

[24] P. Wildbur and M. Burke, *Information Graphics: Innovative Solutions in Contemporary Design*: Thames & Hudson, 1999.

[25] N. Senin, D. R. Wallace, and N. Borland, "Distributed object-based modeling in design simulation marketplace," *Journal of Mechanical Design, Transactions of the ASME*, vol. 125, pp. 2-13, 2003.

[26] D. R. Wallace, E. Yang, and N. Senin, "Integrated Simulation and Design Synthesis," MIT CADLAB 2002.

[27] E. software, "The Federated Intelligent Product EnviRonment (FIPER) project,," E. software, Ed. http://www.engineous.com/product_FIPER.htm, 2003.

[28] H. Elmqvist, S. E. Mattsson, and M. Otter, "Object-oriented and hybrid modeling in Modelica," *Journal Europeen des Systemes Automatises*, vol. 35, pp. 395-416, 2001.

[29] M. Otter, H. Elmqvist, and F. E. Cellier, "Modeling of multibody systems with the object-oriented modeling language Dymola," *Nonlinear Dynamics*, vol. 9, pp. 91-112, 1996.

[30] J. S. Glaser, F. E. Cellier, and A. F. Witulski, "Object-oriented power system modeling using the dymola modeling language," Atlanta, GA, USA, 1995.

[31] R. Mansfield, "Buzz Words - OOP vs. TOP," in *Inheritance (and it's problems)*. http://www.geocities.com/tablizer/buzzword.htm, 2002.

[32] R. Mansfield, "OOP Is Much Better in Theory Than in Practice," vol. 2006. http://www.devx.com/opinion/Article/26776, 2005.

[33] D. Rahmel, *Client/Server Applications with Visual Basic 4*: Sams Publishing, 1996.

[34] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance and Evolution*, vol. 18, pp. 37-58, 2006.

[35] S. K. Card, J. Mackinlay, and B. Shneiderman, *Readings in Information Visualization : Using Vision to Think*: Morgan Kaufmann, 1999.

[36] N. P. Suh, *Axiomatic Design : Advances and Applications* Oxford University Press, 2001.

[37] J. Strachan, "Groovy User Guide," in http://groovy.codehaus.org/, 2006.

[38] D. Systemes, "CATIA product overview," vol. 2006. http://www.3ds.com/products-solutions/plm-solutions/catia/overview/, 2006.