Improving Task Modelling to Support the Co-Evolution of Information Systems and Business Processes

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

David N. Paquette

Permission to Use

Abstract

In business environments, information systems are required to change in response to changes in business processes. We refer to this process as co-evolution: the process of reciprocal change in a software system and the activities and goals of the system's users. This research focuses on improving task modelling techniques to support the co-evolution of information systems and business processes.

We propose the Interaction Template approach to improve task modelling to support co-evolution. Interaction Templates make the task modelling process less tedious in both the design phase and the evolution phase of a system's lifecycle. Our approach adds data schemas and presentation components to task models, allowing us to build task models that adapt to data elements and parameters. Binding presentation components to task models allows us to generate user interface prototypes from task models. The generated user interface prototypes improve task model simulation and help make the effects of changes to business processes more clear.

This thesis describes a study of the seven year evolution of a real world information system. Through this study, we gain a better understanding of how information systems evolve in response to the evolution of an organization's business processes. This thesis presents the Interaction Template approach, as well as a notation for specifying Interaction Templates. A prototype system supporting the Interaction Template approach is provided, along with examples demonstrating the approach.

Acknowledgements

I would like to thank my supervisor, Dr. Kevin Schneider, for his helpful guidance and support throughout my graduate studies. Many thanks also to the members of the Software Research Lab: Jennifer Petrie, Nicole Stavness, Andrew Sutherland, and Mark Watson for their helpful feedback, advice, and friendship.

For my loving wife Jennifer.

Contents

Permission to Use **i**

Abstract **ii**

Acknowledgements **iii**

Contents **v**

List of Tables **viii**

List of Figures **ix**

List of Abbreviations **xi**

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Thesis Statement . . . 4
- 1.3 Approach . . . 4
- 1.4 Contributions . . . 5
- 1.5 Outline . . . 6

2 Related Work **8**
- 2.1 Model Based Approaches . . . 8
  - 2.1.1 Task Modelling . . . 9
- 2.2 Program Families . . . 20
- 2.3 Software Evolution . . . 21
- 2.4 Dynamic Systems . . . 23
- 2.5 Structured Documents . . . 24
  - 2.5.1 DTD and XML Schema . . . 24
  - 2.5.2 XML Path Language . . . 25
  - 2.5.3 Transforming Structured Documents . . . 25
- 2.6 Discussion . . . 26

3 Changing Business Processes **29**
- 3.1 Business Process Overview . . . 29
- 3.2 Software Overview . . . 30
- 3.3 Change Scenarios . . . 31
- 3.4 Change Scenario Effects . . . 34
- 3.5 Summary . . . 39

4 Interaction Template Model **42**
- 4.1 Interaction Templates . . . 42

4.2 Terminology . . . 43
4.2.1 Interaction Template Definition . . . 44
4.2.2 Task Template . . . 45
4.2.3 Component Bindings . . . 47
4.3 Task Model Simulation Semantics . . . 48
4.3.1 Hierarchical Semantics . . . 49
4.3.2 Temporal Operator Semantics . . . 50
4.3.3 Task Model Simulation . . . 52
4.3.4 Interaction Template Simulation . . . 52
4.4 Summary . . . 53

5 Interaction Template Notation 54
5.1 Specifying ConcurTaskTrees with XML . . . 55
5.2 Specifying Interaction Templates with XML . . . 56
5.2.1 Template Element . . . 57
5.2.2 Component Element . . . 58
5.2.3 Parameter Element . . . 59
5.2.4 Schema Element . . . 60
5.3 Specifying Task Model Adaptation Logic . . . 61
5.3.1 Referencing Parameter Values . . . 61
5.3.2 Referencing Schema Values . . . 63
5.3.3 Boolean Expressions . . . 66
5.3.4 Loops . . . 68
5.4 Using Existing Interaction Templates . . . 71
5.5 Binding to Presentation Components . . . 73
5.6 Summary . . . 76

6 Modelling with Interaction Templates 78
6.1 Model-IT Overview . . . 78
6.2 Interaction Templates . . . 80
6.3 Task Model Simulation . . . 82
6.3.1 Basic Simulation . . . 82
6.3.2 Enhanced Simulation . . . 82
6.4 Prototype Limitations . . . 90
6.5 Summary . . . 90

7 Interaction Template Examples 92
7.1 Data Entry . . . 93
7.1.1 Entering Attributes . . . 95
7.1.2 Entering Sub-Elements . . . 100
7.2 Data Access . . . 104
7.2.1 Modify View . . . 106
7.2.2 Select Data . . . 108
7.3 Dialog Box . . . 109
7.4 Using Interaction Templates . . . 111

7.4.1 CS1 Revisited 115
7.5 Summary 119

8 Conclusion **120**
8.1 Summary 120
8.2 Future Work 123
8.2.1 Prototype Layout and Sketching 123
8.2.2 Multiple User Task Models 124
8.2.3 Attaching Sample Data 124
8.2.4 In-Depth Field Study 125
8.3 Conclusion 125

A ConcurTaskTree XSD **131**

B Interaction Template XSD **132**

C Enter Data Element Interaction Template **135**

D Data Table Interaction Template **138**

E Print Labels **141**

List of Tables

2.1 Example UAN description 13

4.1 Temporal Operator Semantics 51

5.1 Interaction Template parameter types 60

List of Figures

2.1 A simple graph editing tool 9
2.2 Example HTA description 10
2.3 Example GOMS description 11
2.4 ConcurTaskTrees notation 14
2.5 Example CTT description 16
2.6 Example State Transition Network 17
2.7 Basic CTT Simulator 18
2.8 Dialogue Graph Editor 20
2.9 PetShop simulator 21

3.1 Lab Manager and Lab Assistant 31
3.2 Impact of change scenarios on the lab software 35
3.3 “Sample Priorities” change scenario effects 36
3.4 “Identifying Outliers” change scenario effects 39

4.1 Task Simulation State Transition Network 49
4.2 Hierarchical task structure 49

5.1 A simple ConcurTaskTree 56
5.2 A ConcurTaskTree adapted to an XML Schema 70

6.1 Model-IT prototype 79
6.2 Interaction Template Repository 80
6.3 Inserting an Interaction Template 81
6.4 Model-IT Basic Simulator 83
6.5 “Enter String Value” IT example 84
6.6 Enhanced Task Model Simulator prototype 89

7.1 Data Entry Examples 93
7.2 Data Table Examples 104
7.3 Example of a task tree that is added when AllowSort is true 107
7.4 Example of a task tree that is added when AllowColumnMove parameter is true 108
7.5 Example of a dialog box 110
7.6 Print Dialog Task Tree 111
7.7 ‘Enter Data Element’ Interaction Template Use 112
7.8 ‘View Data Table’ Interaction Template Use 113
7.9 ‘Print Labels’ Interaction Template Use 113
7.10 Lab Assistant Interaction Templates example 114
7.11 Simple Lab Assistant Prototype Before CS1 115
7.12 ‘Enter Data Element’ Interaction Template after CS1 117
7.13 ‘View Data Table’ Interaction Template after CS1 117

7.14 Simple Lab Assistant Prototype After CS1 118

List of Abbreviations

CTT	ConcurTaskTrees
CTTE	ConcurTaskTreesEnvironment
CUI	Concrete User Interface
ETC	Enabled Task Collection
ETS	Enabled Task Set
FSR	Field Service Representative
GOMS	Goals Operators Methods and Selectors
GPS	Global Positioning System
HTML	Hypertext Markup Language
IT	Interaction Template
ITDL	Interaction Template Definition Language
PRS	Plant Root Simulator
STN	State Transition Network
WIMP	Windows, Menus, Icons, and Pointing Devices
XML	eXtensible Markup Language
XPath	XML Path Language
XSL-FO	XSL Formating Objects
XSL	XML Style Sheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

A business process is a set of one or more linked activities which collectively realize a business goal, normally within the context of an organizational structure defining functional roles and relationships [44]. Businesses use information systems to support or automate many parts of the organizations' business processes. As business processes evolve, supporting information systems must also evolve in order to meet new requirements. We refer to this process as co-evolution: the process of reciprocal change in a software system and the activities and goals of the system's users. Co-evolution can occur in two directions: software systems changing in response to a change in user's activities and goals, or a user's activities and goals changing in response to a change in a software system.

Our research explores co-evolution in the direction of software systems changing in response to changes in activities and goals. We focus on a specific type of software system, information systems, where activities and goals are business processes. That is, we are interested in how information systems must respond to changes in business processes. Based on our focus, we define a co-evolution point as a specific change that has occurred to an information system in response to a change in a business process. We intend to support the co-evolution of information systems and business processes by improving the process of adapting information systems to changes in business processes.

We explore the use of model based approaches to support the co-evolution of information systems and business processes. Model based approaches to interactive system design are based on the specification and evaluation of interactive systems using high-level models [36]. We focus on improving a specific task modelling notation called ConcurTaskTrees. ConcurTaskTrees model systems in terms of goals and activities, which allows us to model information systems in terms of the business processes they support.

1.1 Motivation

Some small businesses, particularly knowledge-based and riskier innovative businesses, have business processes that are constantly evolving. These businesses find it difficult to obtain information systems that meet their needs. Off-the-shelf software promotes software reuse and helps businesses adopt best-practices. Unfortunately, the initial cost of purchasing software can be substantial, and expensive and lengthy customizations are often needed to adapt the software to accurately meet the business's needs. These costs, combined with constantly changing business processes, can result in customization expenses that greatly exceed a small business's means. Alternatively, businesses can choose to build custom information systems. A custom solution can lead to software that better meets the needs of the business. Unfortunately, evolving business processes can lead to maintenance costs that are, again, too large for small businesses.

Using task models, we are able to model information systems in terms of the business processes they support. Task models provide an effective abstraction of information systems. The models focus on users and their activities and goals, which can help developers design better systems. Task modelling also helps to ensure that systems meet user's requirements through early task model simulation, allowing users to simulate the activities involved in a system to ensure their goals can be reached. Unfortunately, task modelling can become tedious when modelling information systems of a significant size [29].

Task modelling can also be useful in the evolutionary phase of a system's lifecycle. When a business process changes, a system's task model can be modified in response to those changes, allowing developers to verify the changes with users. However, model changes must be tediously performed by hand, making the process inefficient. Many tasks must be added, deleted, moved, and/or renamed to correspond to the business process changes. Furthermore, when a task model changes, the effects on the system are unclear because there is no link between the model and the implementation.

This research addresses some of the issues that are present when modelling information systems using task models. We hope to provide techniques that will make task modelling less tedious in both the design phase and the evolution phase of a system's lifecycle by providing adaptable task templates for building task models. Furthermore, we hope to make the effects of task model changes more clear to both developers and users by binding presentation components to the tasks in a task model. We believe that these improvements to task modelling will help to support the co-evolution of information systems and business processes.

1.2 Thesis Statement

Building task models using adaptable task templates and binding presentation components to tasks will improve the task modelling process and support the co-evolution of information systems and business processes. Adaptable task templates allow task models to adapt their behaviour based on data schemas and parameter values. Providing adaptability in task models can help to alleviate the tedious aspects of building and evolving task models.

Binding presentation components to tasks will help to ensure a task model maintains its benefits throughout a system's lifecycle. Binding presentation components to tasks can be useful for developers by bridging the gap between a system's task model and a system's implementation. Furthermore, component bindings can be used to create user interface prototypes of a task model. These prototypes, involving concrete interface components, can help both users and developers to see how a system will change in response to changes in business processes.

1.3 Approach

We begin with a study identifying real world examples of co-evolution points. In this study, we explore how information systems must adapt as a result of changes in business processes.

Next, we propose techniques that aim to make task modelling less tedious in both the design phase and the evolution phase of a system's lifecycle by using adaptable task templates. These task templates represent reusable task model pieces that are self-adaptive to data schemas and parameter values. In particular, we hope to promote the reuse of task model pieces and their implementation using a concept we refer to as Interaction Templates [29]. Interaction Templates are task model pieces that model common interface interactions found in information systems.

We also propose an improvement to task model simulation. Our approach to task model simulation involves high-fidelity user interface prototypes that are generated from task models using concrete user interface components. The prototypes help users and developers to understand how a system will change in response to changes in business processes.

1.4 Contributions

Through the investigation of using task models to support the co-evolution of information systems and business processes, we hope to provide the following contributions:

- a rigorous definition of Interaction Templates,
- a technique for building task models using Interaction Templates,
- a technique for composing user interface prototypes for task model simulation,

- a description of the semantics of task model simulation,
- a prototype system for building task models using Interaction Templates, and
- a prototype system for task model simulation using concrete user interface components.

A case study of the seven year evolution of a real world information system, as discussed in Chapter 3, will help us to understand how changes in business processes can affect an organization's information systems. The study will provide real world examples of co-evolution points, and will be used to structure a new approach to task modelling information systems. The approach will involve a technique for building task models that are self-adaptive to data schemas and parameter values. An enhancement to task model simulation will also be proposed. The enhancement will include user interface prototypes that are generated from task models.

1.5 Outline

Chapter 2 discusses previous research in the areas of model based approaches, software evolution, program families, dynamic systems, and structured documents. A case study exploring how changes in business processes can affect a business's information systems is discussed in Chapter 3. Chapter 4 describes Interaction Templates, our approach to improving task models to support co-evolution. Chapter 5 outlines a notation for Interaction Templates, while Chapter 6 describes a prototype that was built to illustrate the approach. In Chapter 7, we demonstrate our approach using several examples. Finally, Chapter 8 concludes with an overview of the research that has been completed as well as possible directions for future work.

Chapter 2

Related Work

Model based approaches, and in particular task models, have been used to model interactive systems that support business processes. This Chapter explores the advantages and weaknesses of several task modelling approaches. Research in the areas of software evolution, program families, and autonomic systems have all explored issues related to changing software over time. This Chapter gives an overview of these areas and discusses their applicability to co-evolving information systems and business processes. Finally, an overview of research into structured documents and transforming structured documents is provided.

2.1 Model Based Approaches

Model based approaches to the development of interactive systems involve the specification and evaluation of interactive software systems at a more abstract level than source code. Using high-level models to specify interactive systems can help designers to focus on specifying the requirements and behaviour of the system rather than immediately worrying about implementation details. Model based approaches also have the benefit of early evaluation. High-level models can be evaluated before implementation has begun, allowing for a refinement of the system specification with less resources than if source code were involved in the change.

2.1.1 Task Modelling

Task models focus on describing interactive systems in terms of user goals and the tasks required to reach those goals. Many model based approaches, such as ADEPT [45], SUIDT [1], U-TEL/MOBI [42], and PetShop [27], acknowledge the importance of task models. This section will describe some of the more successful task modelling approaches that have been proposed. To illustrate each of the approaches, the simple graph editing tool shown in Figure 2.1 will be modelled.

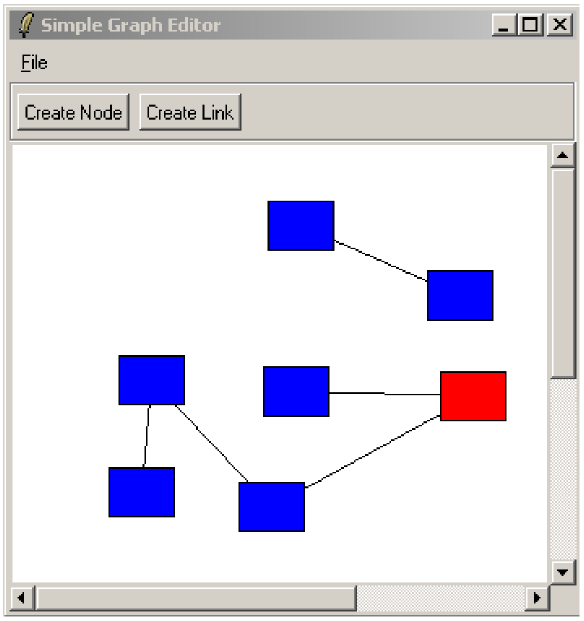


Figure 2.1: A simple graph editing tool

Hierarchical Task Analysis

One of the first approaches to task modelling, Hierarchical Task Analysis (HTA) [37], dates back to the late 1960s. HTA is used to break down activities into different levels. A HTA is represented with task names in numbered boxes. Although the order of the numbers has no relevance, the numbers can be used to describe a plan for reaching a specific user goal. Given the HTA shown in Figure 2.2 and the goal of editing a graph to add an edge between two existing nodes, a plan would be as follows:

0. Edit Graph
1. Load Graph
3. Add Edge
3.1 Select From Node
3.2 Select To Node
7. Save Graph

HTA is a very simplistic approach to task modelling. A hierarchical breakdown of tasks is given, but the temporal relationships between tasks is not known. For example, it is not possible to express concurrent tasks using HTA.

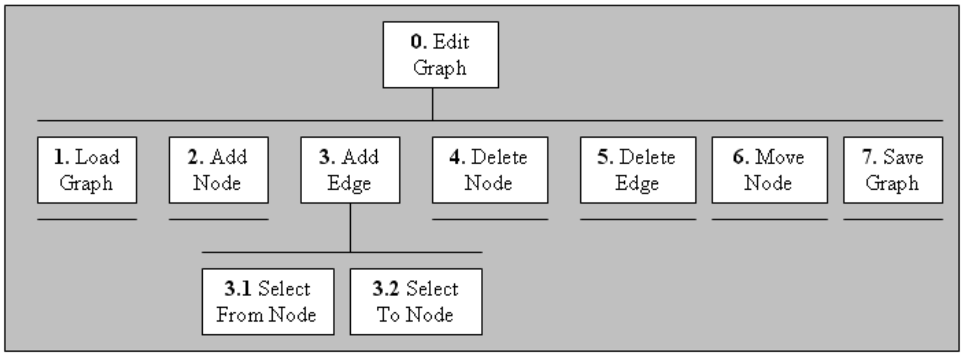
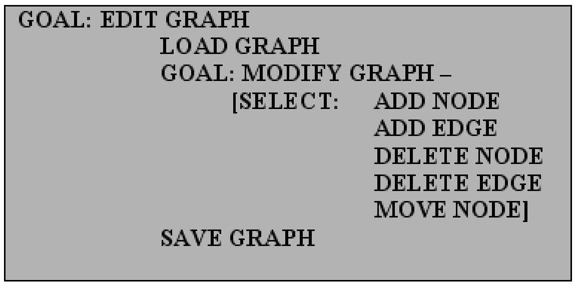


Figure 2.2: A HTA description of a simple graph editor

GOMS

Goals Operators Methods and Selectors (GOMS) [36, 37], another early approach to task modelling, has been used with a great deal of success. GOMS takes a cognitive approach to task modelling. It is a hierarchical description in which Goals, Operators, Methods and Selectors describe tasks. Goals are reached in terms of operators. Operators are elementary perceptual, motor and cognitive acts [37]. Methods are made up of sequences of subgoals and operators used to reach goals. Selection rules are used when there is a choice between methods or operators. An example of a GOMS specification is shown in Figure 2.3.

GOMS has been used successfully to predict task execution times. The main drawbacks of GOMS are that it only considers sequential tasks and error-free execution.



```
GOAL: EDIT GRAPH
          LOAD GRAPH
          GOAL: MODIFY GRAPH –
               [SELECT:   ADD NODE
                          ADD EDGE
                          DELETE NODE
                          DELETE EDGE
                          MOVE NODE]
          SAVE GRAPH
```

Figure 2.3: A GOMS description of a simple graph editor

Several different versions of GOMS have been proposed. The simplest of these, KLM-GOMS [6], uses only keystroke-level operators. KLM-GOMS does not use goals, methods or selection rules. A KLM-GOMS description only lists the actions a user must perform to reach a goal. The actions listed are keystrokes, mouse-movements, and mouse-button presses. Other versions of GOMS include NGOMSL

[18] and CPM-GOMS [15]. "NGOMSL includes a more rigorous set of rules for identifying the GOMS components and information such as the number of steps in a method, how goals are set and terminated, and what information needs to be remembered while performing the task" [36]. CPM-GOMS is an extension of GOMS that does consider non-sequential tasks.

The GOMS approach has seen more interest in North America than in Europe. In Europe, other techniques for task modelling have been developed and used more widely [36].

User Action Notation

The User Action Notation (UAN), developed in the late 1980's [37, 16], is a formal textual language used to describe the behaviour of graphical user interfaces. A UAN specification is made up of two parts.

The first part shows the task decomposition in terms of tasks and subtasks, as well as the temporal relationships among asynchronous tasks [36]. An example of a task decomposition is:

> *Task: Edit Graph*
>
> *Load Graph(Add Node | Add Edge | Delete Node | Delete Edge | Move Node | Save Graph) +*

This expression is a description of the main task *Edit Graph.* The steps involved in editing a graph are first *Load Graph* followed by one or more (specified by the + operator) of the tasks composing the expression *(Add Node | Add Edge | Delete*

Node | *Delete Edge* | *Move Node* | *Save Graph).* The | operator indicates a choice to be made by the user.

The second part of a UAN specification is a table that describes user actions, which are referred to as Primitive User Actions [16]. Each basic task is described in a table that indicates the user action, the interface feedback and interface state for each physical action performed by the user. An example is shown in Table 2.1.

Table 2.1: A UAN description of a simple graph editor

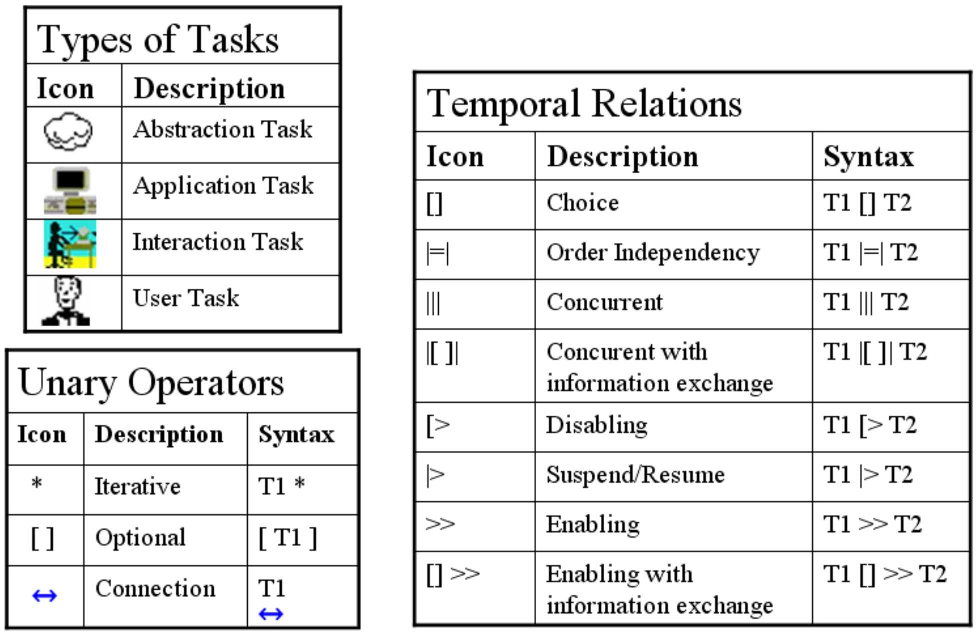
Task: Move Node		
User Action	Interface Feedback	Interface State
$\sim [node]Mv$	node!	selected = node
$\sim [x,y]^*$	node>$\sim$ node_edges(node)>$\sim$	
$M\wedge$		

In the first line, $\sim$*[node] Mv* indicates the mouse cursor ($\sim$) has entered the context of the node (*[node]*), a mouse button (M) was depressed (v), the node was highlighted (*node!*), and the interface's currently selected node was updated to indicate that the highlighted node was selected (*selected* = *node*). In the second line, $\sim$*[x,y]** indicates that the mouse was moved around on the screen, *node*> $\sim$ indicates that the node followed the mouse movement, and *node_edges(node)*>$\sim$ indicates that all edges attached to the node also followed the mouse movement. In the third line, $M\hat{}$ indicates that the mouse button was released.

The UAN provides a very detailed textual description of the behaviour of a graphical user interface. However, the level of description does not seem to be much more abstract than source code, making the UAN a notation that does not fit well with many other model based approaches.

ConcurTaskTrees

ConcurTaskTrees (CTT) is a graphical notation used to describe interactive systems [37]. With CTT, tasks are arranged hierarchically, with more complex tasks broken down into simpler sub-tasks. CTT includes a rich set of temporal operators that are used to describe the relationship between tasks, as well as unary operators that are used to identify optional and iterative tasks. A summary of the CTT Notation can be seen in Figure 2.4.



Types of Tasks

Icon	Description
☁	Abstraction Task
🖥	Application Task
🧑‍💻	Interaction Task
👤	User Task

Unary Operators

Icon	Description	Syntax
*	Iterative	T1 *
[ ]	Optional	[T1]
↔	Connection	T1 ↔

Temporal Relations

Icon	Description	Syntax
[]	Choice	T1 [] T2
|=|	Order Independency	T1 |=| T2
|||	Concurrent	T1 ||| T2
|[ ]|	Concurent with information exchange	T1 |[ ]| T2
[>	Disabling	T1 [> T2
|>	Suspend/Resume	T1 |> T2
>>	Enabling	T1 >> T2
[] >>	Enabling with information exchange	T1 [] >> T2

Figure 2.4: Summary of the ConcurTaskTrees notation

CTT includes four types of tasks: abstraction, application, interaction, and user tasks. An abstraction task is a high-level task that must be further broken down into application, interaction, or user tasks. An application task is a task that is performed by the system. An example of an application task is displaying data to the user. An interaction task is any task where the user is interacting with the system, for example, when a user enters data into a form. A user task refers to a cognitive task performed by the user, for example when a user decides between two options.

Three unary operators can be applied to tasks in CTT. A task can be repeated any number of times using the iterative operator, represented by an asterisk next to the task. Using the optional operator, represented by square braces surrounding a task, makes a task optional. Finally, for multi-user systems, a task in one user's task tree can be linked to a task in another user's task tree by using the connection operator. The connection operator is represented by adding a double-sided arrow below a task.

In CTT, sibling tasks are tasks in the tree that share the same parent. Sibling tasks can be related to each other using a set of eight available temporal operators, as shown in Figure 2.4. A temporal operator describes the temporal relationship between two neighbouring sibling tasks. The choice relationship identifies a choice between two tasks, where one of the tasks must be performed, but not both. An order independence relationship means that both the tasks must be performed, but the order in which they are performed does not matter. Two tasks that can be performed at the same time, without constraints, are related using the concurrency operator. Two tasks that can be performed at the same time, but with communication between the two, are related using the concurrent with information exchange operator. The enabling operator is used to show when the completion of one task enables another task. When one task enables another, as well as passes some information to the task it is enabling, the enabling with information exchange operator is used. The disabling operator is used to show that the first action of the second task disables the first task. When the first action of the second task disables the first task, but the last action of the second tasks re-enables the first task, the suspend/resume operator is used. An example of a CTT model specifying the example graph editing program is shown in Figure 2.5.

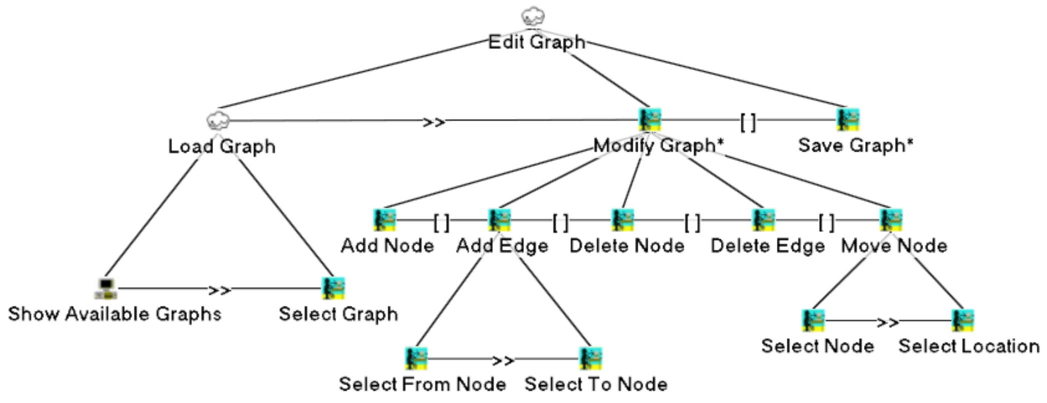


Figure 2.5: An example of a CTT model specifying a simple graph editing program

Task Model Simulation One of the powerful features of CTT is the ability to simulate task models at an early stage in the development process, allowing for a simulation of the system before implementation has started. Simulation can help to ensure the system that is built will match the user's conceptual model as well as help to evaluate the usability of a system at a very early stage. Several task model simulators have been built for ConcurTaskTrees. First, we discuss the process involved in simulating ConcurTaskTrees. Next, an overview of some of the task model simulators that are available is given.

The Simulation Process Simulating a ConcurTaskTree involves simulating, in some way, the performance of specific tasks in order to reach a pre-defined goal. In a ConcurTaskTree, tasks are related to each other according to their temporal relations and hierarchical breakdown. Depending on what tasks have been performed, some tasks are enabled and others are disabled. The first step in simulating ConcurTaskTrees is to identify the sets of tasks that are logically enabled at the same time. A set of tasks that are logically enabled at the same point in time is called an enabled task set (ETS) [36]. Enabled tasks sets are identified according to the rules laid out in [36]. The set of all enabled task sets for a specific task model is referred to as an enabled task collection (ETC).

Having identified the enabled task collection for a task model, the next step is to identify the effects of performing each task in each ETS. The result of this analysis is a state transition network (STN). In this state transition network, each ETS is a state, and transitions between enabled task sets occur when tasks are performed. The final preparation step for simulation is to calculate the initial state. A simple example illustrating a ConcurTaskTree and its STN is shown in Figure 2.6. A command-line tool called TaskLib [24] can be used to extract the ETC, STN, and initial state from a CTT. The details of TaskLib's implementation can be found in [25].

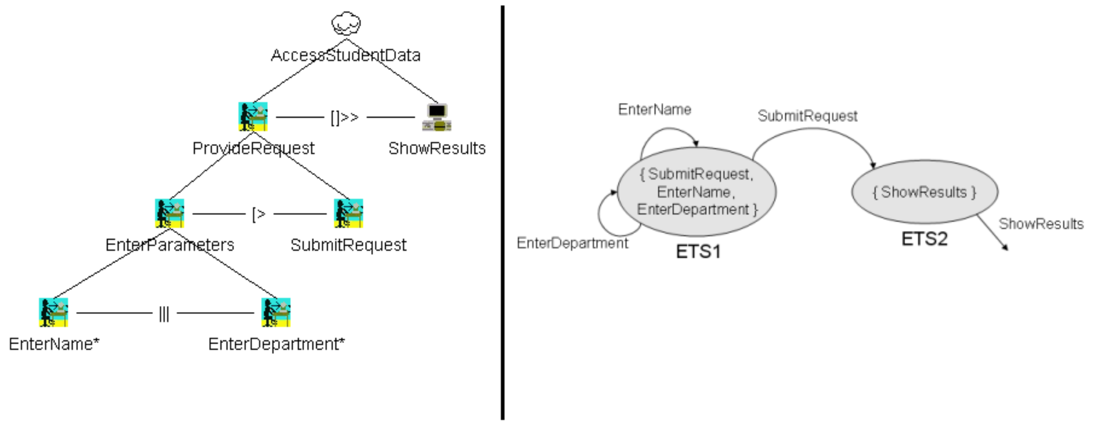


Figure 2.6: A CTT (left) and its State Transition Network (right)

Once the ETC, STN, and initial state have all be identified, task model simulation can be performed. This initial process is common to all ConcurTaskTree simulators. The actual simulation involves the user navigating through the STN by simulating the performance of tasks. As will be discussed shortly, the simulation of performing a task is done differently in the various simulation tools that exist.

Simulation Tools

Basic Simulators The most basic simulators, such as the one shown in Figure 2.7, simply display the currently enabled tasks in a list. In these simple simulators, double-clicking on a task simulates the performance of that task. When a task is performed, the enabled tasks are updated accordingly. A basic task model simulator can be found in ConcurTaskTreesEnvironment (CTTE) [35], a tool for both building and simulating task models.

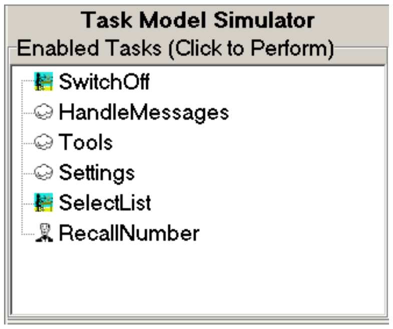


Figure 2.7: A simple ConcurTaskTrees task model simulator

Dialogue Graph Editor The Dialogue Graph Editor, a tool developed at the University of Rostock, provides a more complex simulation than the basic simulator found in CTTE. The Dialogue Graph Editor allows designers to create views and assign tasks from a task model to those views. The views can later be used to simulate the task model as shown in Figure 2.8. When simulating the task model, views are represented as windows, elements (as well as tasks) inside the windows are represented by buttons, and transitions between states are represented by navigation between windows [13]. Views become visible when they are enabled, and invisible when they are disabled. Likewise, buttons become enabled and disabled when their associated tasks are enabled or disabled. Users can simulate the task model by clicking buttons to perform tasks and navigate through windows to select between available tasks.

The windows and buttons generated by Dialogue Graph Editor for simulation purposes are considered to be abstract interface prototypes. However, clicking buttons to perform tasks does not seem to provide much of an advantage over the basic simulators, and at times might be more confusing. For example, clicking a button to simulate an interaction task that does not normally involve a button widget may seem strange to end users. The key advantage in Dialogue Graph Editor is the ability to organize tasks into a dialog. This requires an additional dialog model as well as a mapping between the dialog model and task model.

PetShop The PetShop [27] environment offers a different approach to interactive system simulation. In PetShop, ConcurTaskTrees are integrated with a Petrie net based notation called ICO. The two models are integrated using scenarios, and a mapping between a ConcurTaskTrees' user tasks and a system model's user services. ICO specifications are used to simulate interactive systems in the PetShop

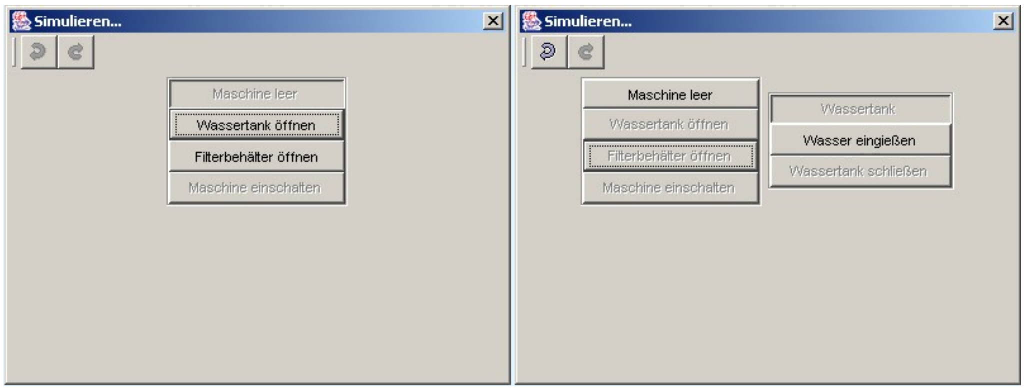


Figure 2.8: Simulator included in Dialogue Graph Editor [12]

environment using presentation objects that are manually implemented by system developers. The PetShop approach allows task models to be used to better understand and evaluate the system's behaviour, while a more detailed system model is used to model the dialogue and presentation parts of the system. The PetShop simulator, shown in Figure 2.9, provides a very detailed simulation of the system, but relies on a very detailed system model as well as manually implemented presentation objects.

2.2 Program Families

A program family is set of programs whose common properties are extensive enough that it is advantageous to study the common properties before studying the specific properties of the individual family members [32]. One method of designing and developing program families is module specification. In module specification a program is divided into information hiding modules [33], each of which hides a design decision that could change for the various family members. Developing new family members is simplified when using module specification because changes in one module do not

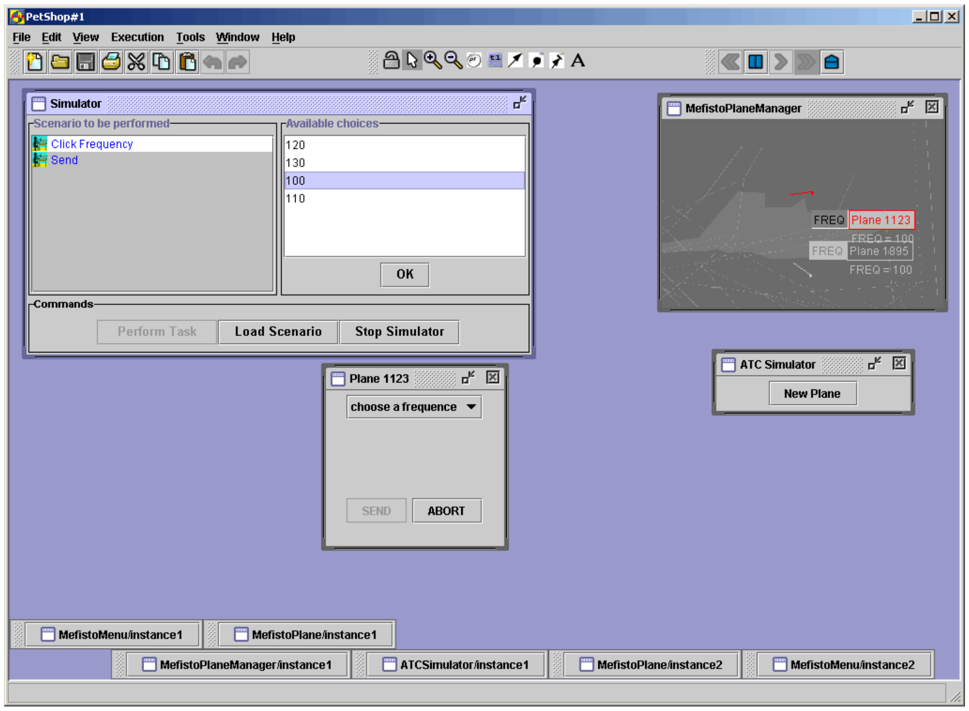


Figure 2.9: Screenshot of the PetShop simulator [27]

affect the system as a whole. The concept of program families has been studied extensively since the idea was originally introduced in 1976. The most recent version of the topic that has seen considerable success in industry is Software Product Lines [5] from the Software Engineering Institute at Carnegie Mellon.

When designing information systems to support business processes, selecting modules that can hide the effects of possible business process changes will help to support co-evolution.

2.3 Software Evolution

Software evolution includes any activities involved in ensuring that a software system continues to meet the organization's needs in an efficient, cost effective manner. Software evolution is a broad area and can include activities ranging from early design decisions to issues involved in managing unanticipated changes in an existing software system. In contrast co-evolution emphasizes the complex relationship between software systems and the activities and goals of users. In the direction of software adapting to changes in users activities and goals, co-evolution can be considered a subset of software evolution.

In general, software evolution research can be divided into two categories: anticipatory and reactive [3]. Anticipatory methods of software evolution are based on the belief that software change can be planned. Early planning for change can be seen in many requirements engineering methods [17, 46, 28]. Anticipatory methods focus on eliciting possible changes during requirements analysis as well as designing and planning for the possible changes that have been identified.

Reactive methods are based on the belief that despite early efforts in requirements engineering and design, there will always be some changes that cannot be anticipated. In reactive methods, the belief is that software change should only be dealt with once the changes are needed. Reactive methods focus on formal methods for managing changes as they occur, and are discussed in workshops such as Unanticipated Software Evolution [19].

We have taken an anticipatory view towards software evolution in trying to anticipate and design for the types of changes that can occur when a business process is changed. Our research focuses on a specific subset of software evolution, which we have defined as co-evolution.

2.4 Dynamic Systems

Dynamic systems are either systems that can be changed at runtime, or systems that can change their implementation without the need to recompile or reboot [26]. Dynamic systems can be classified as either open or closed. An open dynamic system allows the functionality of the system to evolve and allows for adaptations to be specified at runtime. Closed dynamic systems have all functionality and adaptation logic specified at build-time.

Another classification of dynamic systems is adaptable and self-adaptive. An adaptable system is one that provides a procedural or declarative interface by which an external actor can specify the changes that are needed. Adaptable systems are a type of open system. Self-adaptive systems are capable of adapting their behaviour based on elements in the system's deployment environment. Environment elements are anything that is observable by the system. Some examples of observable information are end-user input, external hardware, and internal data. A self-adaptive system can be classified as either an open or closed system, depending on its ability to change at runtime.

The approach we have taken to supporting the co-evolution of information systems and business processes involves both self-adaptive and adaptable components. In particular, we aim to provide task models that are self-adaptive to data and parameter values, while providing techniques that allow developers to adapt task models to changes in business processes that are not handled by the data and parameter adaptability.

2.5 Structured Documents

The Extensible Markup Language (XML), a subset of Standard General Markup Language (SGML), describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them [4]. XML is a grammatical system that is used to create custom markup languages. While every custom markup language contains custom tags that describe data objects, every language must adhere to the underlying rules of XML. XML documents are made up of entities, and every document begins with a 'root' entity. XML provides a powerful and extensible mechanism for storing and exchanging data, which may prove useful in co-evolving information systems and business processes.

2.5.1 DTD and XML Schema

A schema defines a custom language that is created with XML. A schema contains a set of rules that defines the elements and attributes that are allowed or required in a document. Currently, there are two schema definition languages available: Document Type Definition (DTD) and XML Schema. DTDs contain rules for each and every element and attribute that can appear in a document. XML Schema is a newer schema language that provides several advantages over DTDs. XML Schemas are written in XML, which means that the same parsers can be used to process both XML Schemas and XML documents. Also, XML Schemas contain a notion of scope, whereas all definitions in DTDs are global. In DTDs, no two elements can have the same name, even if they appear in different contexts. XML Schemas also provides more control over the type of information that can appear in an element or attribute.

2.5.2 XML Path Language

XML Path Language [9], XPath, is a powerful language used to navigate elements and attributes in an XML document. Using path expressions, XPath selects nodes or sets of nodes from an XML Document. Location paths are used to navigate through an XML document's hierarchical structure and select sets of nodes. Sets of nodes can be filtered using predicates. Axes can be used to further specify target nodes according to parent-child relationships. Finally, a set of functions is available to provide string manipulation and other useful functions.

2.5.3 Transforming Structured Documents

Structured documents are often transformed from one structure to another. Structured document transformations are needed when presenting documents to users, or when documents are shared across different systems. Several document transformation languages have been proposed, including XSL [8], XT3D [20], and VXT [39].

XSL

XSL, a style sheet language for XML documents, contains three parts: XSLT, XPath, and XSL-FO. XSL Transformations (XSLT) is an XML based language for transforming one XML document into another XML document for the purpose of presentation.

XML Path Language, XPath, is an expression language used by XSLT to refer to portions of an XML document. Finally, XSL Formatting Objects (XSL-FO) is a vocabulary for specifying formatting semantics in an XML document. While XSLT was initially designed for the purposes of transforming XML documents to documents containing formatting details expressed in XSL-FO, it has been used in more general applications such as generating HTML web pages from XML documents. However, XSLT is not a general purpose XML transformation language, as it was designed specifically for the types of transformations that are needed when XSLT is used as a part of XSL [8].

2.6 Discussion

Model based approaches to software development show promise in supporting the co-evolution of information systems and business processes. In particular, the activity and goal oriented task modelling approaches seem well suited to modelling information systems in terms of business processes. We have reviewed several different task modelling approaches. Hierarchical Task Analysis provides a useful hierarchical breakdown of tasks, but it does not provide any mechanism for modelling the temporal relationships between tasks. As such, HTA does not seem like a good choice for modelling complex information systems that support complex business processes. The GOMS approach only considers sequential task execution, making it a poor choice for our research. The User Action Notation (UAN) focuses on very detailed descriptions of tasks, going as far as describing mouse movements and mouse button presses. The UAN seems to be too detailed to model systems in terms of business processes. The ConcurTaskTrees (CTT) notation, with its hierarchical breakdown of tasks and temporal operators, seems to be the best choice for modelling information systems that support business processes. The notation allows us to include cognitive user tasks, tasks performed by the system, as well as tasks where the user is interacting with the system.

The simulation capabilities of the ConcurTaskTrees notation allows designers to validate a system design with users before the system is ever built. Simulation also shows promise for validating changes to a system that must occur as the result of a business process change. Most current task model simulation tools provide a very abstract simulation of the tasks in the system. While some simulation tools, such as PetShop, do provide simulation using concrete interface components, they require separate and very detailed models, as well as requiring developers to write code to build the simulated interfaces. Task model simulation involving concrete user interface components without additional and complex models would allow developers and users to more easily see how information systems would change in response to changes in business processes.

We have seen how co-evolution relates to the traditional definition of software evolution. We are focusing on co-evolution, a subset of software evolution, and have taken an anticipatory view of software evolution. We have also reviewed work in the area of dynamic systems, and seen how our research involves both adaptable and self-adaptive components. Structured documents and their related technologies have been studied, as these are used in our approach to task modelling.

The work of Parnas [32] has shown us the value of studying a family of programs and structuring software into modules that hide the effects of possible changes. We apply this approach in Chapter 3 by studying how changes in business processes can affect information systems, and in Chapter 4 by proposing an approach to task modelling that encapsulates the effects of some business process changes.

Chapter 3

Changing Business Processes

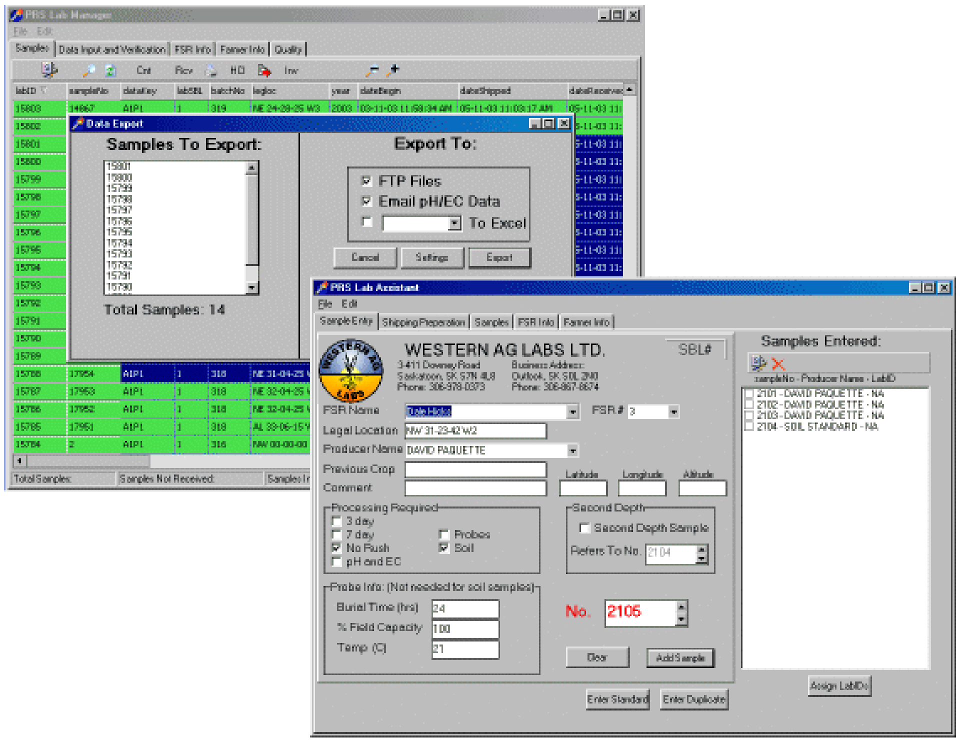
Supporting the co-evolution of information systems and business processes requires an understanding of how business processes can change within an organizational context. The business processes at Western Ag, a soil analysis lab, were studied to help better understand how business processes can change, and how those changes can affect the software that supports those business processes. Although the software is small, the evolution of the custom software used by Western Ag makes it an excellent example of an organization whose software has been affected by continually changing business processes. This chapter provides a brief overview of Western Ag's business processes and custom software, followed by a set of nine co-evolution points that have occurred at Western Ag.

3.1 Business Process Overview

The goal of the soil analysis lab at Western Ag is to provide soil nutrient values and pH/EC values to farmers. Initially, Field Service Representatives (FSRs) retrieve soil samples from farmers' fields. The soil is then shipped to the analysis lab, where it is analyzed using the PRS™-probes. Analyzing soil samples using the PRS™-probes begins by inserting the probes in the soil for a 24-hour burial. Over the burial time, the membrane on the probes absorbs soil nutrients that are available to a plant root. After the burial is completed, the probes are washed to remove all soil. The washed probes are then immersed in a mild acid solution, causing the nutrients to be transferred from the membrane to the solution. Finally, the solution is analyzed using traditional methods. The results of the final analysis, nutrient values and pH/EC values, are then sent to the FSRs via fax, email, and data files transferred over an FTP server.

3.2 Software Overview

Western Ag uses a custom software solution to track samples through the analysis process as well as to process the data generated from soil analysis. The software has undergone two major changes since the lab first opened in 1997. The first major change occurred in the fall of 2000, when the software was migrated from a Microsoft Excel based solution to a custom built Microsoft Windows application. The next major change came in the fall of 2002, which saw the addition of a central database server to support multiple users located in remote labs across Western Canada. The most recent version of the software, called the Lab Assistant and Lab Manager, was built using Borland Delphi 4 with a central MySQL database server. Screenshots of the Lab Manager and Lab Assistant can be seen in Figure 3.1. The Lab Manager and Lab Assistant are made up of approximately 24,000 lines of Object Pascal source code.



labID	sampleNo	dataKey	labSBL	batchNo	legloc	year	dateBegin	dateShipped	dateReceived
15803	14867	A1P1	1	319	NE 24-29-25 W3	2003	03-11-03 11:58:34 AM	05-11-03 11:03:17 AM	05-11-03 11:
15788	17954	A1P1	1	318	NE 31-04-25				
15787	17953	A1P1	1	318	NE 32-04-25				
15786	17952	A1P1	1	318	NE 32-04-25				
15785	17951	A1P1	1	318	AL 33-06-15				
15784	2	A1P1	1	316	NW 00-00-00				

Samples To Export:

15801
15800
15799
15798
15797
15796
15795
15794
15793
15792
15791
15790

Total Samples: 14

Export To:

[✓] FTP Files
[✓] Email pH/EC Data
[ ] To Excel

WESTERN AG LABS LTD.
3411 Downey Road
Saskatoon, SK S7N 4L8
Phone: 306-978-0373
Business Address:
Outlook, SK S0L 2N0
Phone: 306-867-8674

FSR Name: Dale Hicks
FSR #: 3
Legal Location: NW 31-23-42 W2
Producer Name: DAVID PAQUETTE

Processing Required: [ ] 3 day [ ] 7 day [✓] No Rush [ ] pH and EC [ ] Probes [✓] Soil
Second Depth: [ ] Second Depth Sample; Refers To No. 2104
Probe Info: (Not needed for soil samples)
Burial Time (hrs): 24
% Field Capacity: 100
Temp (C): 21
No. 2105

Samples Entered:
sampleNo - Producer Name - LabID
2101 - DAVID PAQUETTE - NA
2102 - DAVID PAQUETTE - NA
2103 - DAVID PAQUETTE - NA
2104 - SOIL STANDARD - NA

Figure 3.1: Lab Manager (left) and Lab Assistant (right) screenshots

3.3 Change Scenarios

A study of the lab software was conducted in order to identify how the changing business processes of the soil-testing lab affected the software over the last seven years. The goal of this study was to attempt to anticipate the types of changes that can result from business process changes. Anticipating the types of changes can help to specify modules that can minimize the effect of such changes. A key lab employee who was involved in both of the major evolution points of the software was interviewed to identify historical business process changes and how those changes affected the software.

This section will outline some example business process changes that occurred in the soil analysis lab. Example business process changes are described in terms of change scenarios. A change scenario outlines a concrete change that has occurred in the businesss process but does not discuss how the change will affect the software used in the business. Change scenarios provided an effective mechanism for communicating change between users and system designers.

CS1 - Sample Priorities:

On busy days, the number of samples that are waiting to be analyzed can exceed the number of samples that can be processed in a single day. The lab has implemented a method of prioritizing samples as “3 Day”, “7 Day”, and “No Rush”. Samples marked “3 Day” must be processed as soon as possible. “7 Day” samples must be completed in seven days, and therefore can wait up to four days after being received to begin processing. Processing of samples marked “No Rush” can wait until all samples with a higher priority rating have been processed.

CS2 - Method Blanks:

The lab has implemented a new quality control measure. A method blank must be processed with every batch of samples that is analyzed. A method blank is a clean PRS™-probe that is analyzed with the current batch of samples. Since the method blank has not been buried in a soil sample, the analysis should return values that are less than the detection limit for each nutrient that is tested. If the results of a method blank are too high, the batch of probes used to test the soil has likely been contaminated and the soil samples should be re-tested with a new set of probes.

CS3 - Remote Processing Labs:

As the number of soil samples begins to exceed the capacity of a single soil-testing lab, Western Ag has decided to open a remote-processing lab in Southern Saskatchewan. The remote-processing lab receives soil samples from nearby Field Service Representatives and performs the probe burial and probe cleaning steps of the soil analysis. The clean probes are then shipped to the central analysis lab for the remainder of the analysis process. The addition of a remote-processing lab increases the capacity of the central analysis lab, as it does not have to deal with soil from samples that have been processed by the remote lab.

CS4 - Sample Batching:

The lab no longer processes soil samples as individual samples. Many samples are grouped together and processed as a batch of samples. Batches are given a Batch ID, and samples within a batch are processed together as a group. Batching samples together helps lab technicians to keep track of the samples currently being analyzed.

CS5 - Second Depth Samples:

Western Ag has started to occasionally test samples for farmers who grow winter wheat. When a farmer wants to grow winter wheat, it is required that the field is sampled at two different depths. A shallow sample is tested for all nutrients, while the deeper (second-depth) sample is tested only for Nitrogen. When the results are reported to the farmer, the Nitrogen values of both samples are added together.

CS6 - pH/EC Email:

In an attempt to cut down on long-distance fees and increase the efficiency of the lab, Western Ag has decided to email pH/EC results rather than manually faxing the results. pH/EC results are now emailed as a Microsoft Excel file.

CS7 - Sample Forecasting:

It is important for lab technicians to have an indication of the number of soil samples to expect in a given week or month. Lab technicians have started to create summaries of historical data to help identify expected numbers for given time periods.

CS8 - GPS Format:

Field Service Representatives reference soil sample locations using GPS coordinates. While most reps use degree decimal format (eg. 42.45472°), some reps are now using degree-minute-second format (eg. 42°27'17"). The lab is required to record and report the locations in either of the two formats.

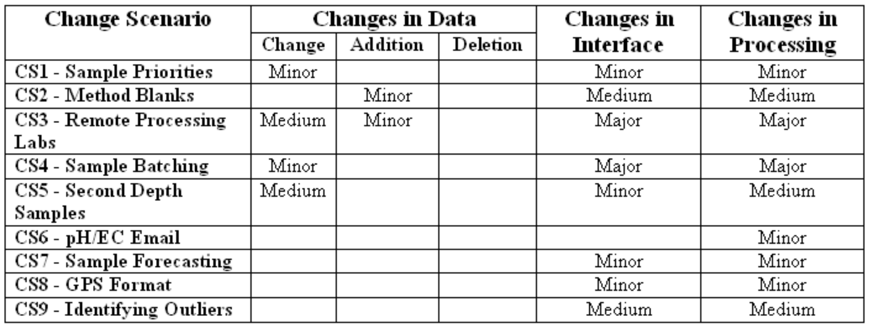
CS9 - Identifying Outliers:

Lab technicians have started to monitor the quality of soil sample data more closely by checking the quality control data for outliers. An outlier is defined as a number that is over two standard deviations from the mean value. If outliers are identified, technicians are able to check certain portions of the analysis process for possible errors.

3.4 Change Scenario Effects

The effects of the nine change scenarios on Western Ag's lab software are discussed in terms of three categories: changes in *data*, *interface*, and *processing*. Changes in data are considered when changes were made to the underlying database schema. Data schema changes are further divided into changes to existing database tables, the addition of new database tables, and the deletion of existing database tables. Changes in interface are considered whenever there has been a change to the user interface of the software. Changes in processing are considered when a change has occurred in the non-user interface specification portion of the source code, that is, whenever the source code that processes data in the information system has changed.

Figure 3.2 shows a summary of the magnitude of impact each of the change scenarios had on the lab software. The rating used to summarize the magnitude of impact of each change scenario is a relative rating based on the amount of development time that was needed to implement each change. A minor impact corresponds to less than a week of developer-hours to implement. A medium impact corresponds to between one week and one month, while a major impact corresponds to a month or more.

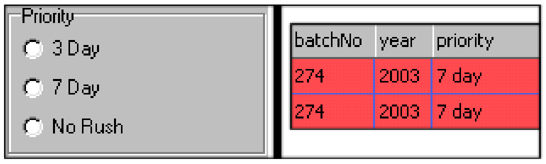


Change Scenario	Changes in Data			Changes in Interface	Changes in Processing
	Change	Addition	Deletion		
CS1 - Sample Priorities	Minor			Minor	Minor
CS2 - Method Blanks		Minor		Medium	Medium
CS3 - Remote Processing Labs	Medium	Minor		Major	Major
CS4 - Sample Batching	Minor			Major	Major
CS5 - Second Depth Samples	Medium			Minor	Medium
CS6 - pH/EC Email					Minor
CS7 - Sample Forecasting				Minor	Minor
CS8 - GPS Format				Minor	Minor
CS9 - Identifying Outliers				Medium	Medium

Figure 3.2: Impact of change scenarios on the lab software

CS1 - Sample Priorities:

Adding a priority rating to each soil sample caused a change to the existing 'sample' data element in the lab software. An enumerated 'priority' field, with possible values '3 Day', '7 Day', and 'No Rush', was added to the 'sample' data element. As shown in Figure 3.3, the change also caused two small interface changes. A radiogroup for selecting a sample's priority rating was added to the soil sample entry form, and a 'priority' column was added to the data table that displays the 'sample' data elements. Small changes in data processing were also needed in the functions that save 'sample' data elements to the database, as well as the functions that retrieve the 'sample' data elements for viewing in the data table.



batchNo	year	priority
274	2003	7 day
274	2003	7 day

Figure 3.3: Changes in interface caused by the "Sample Priorities" change scenario

CS2 - Method Blanks:

This change scenario resulted in changes in data, interface, and processing. A data element called 'mblk' was added to the data model of the lab software. In the interface, a method of adding a 'mblk' was added, as well as a data table to view all method blanks that have been analyzed. Several changes in processing also occurred. A function was added to ensure that a 'mblk' is analyzed with every batch of samples. Functions were also added to store and retrieve the 'mblk' data elements.

CS3 - Remote Processing Labs:

Of all the change scenarios, the addition of remote processing labs had the greatest impact on the lab software, as it was the main cause of the last major evolution point in the lab software. Major changes in data were seen, and a central database server was needed to handle concurrency due to multiple users. Two new data elements, the 'analysis lab' and the 'processing lab', were added, corresponding to the central analysis lab and remote processing lab respectively. The 'sample' data element was changed to include a reference to the 'processing lab' that processed the sample. The interface was changed significantly, as two separate interfaces were now needed. One interface, the Lab Assistant, was built for the remote processing labs that guides the remote processing labs through the pre-processing of soil samples. A separate interface, the Lab Manager, was built for the central analysis lab. Changes in processing were also seen in all functions that stored and retrieved data, because not only did the data change, but its location and storage type also changed from a local Microsoft Access database to a central MySQL database server.

CS4 - Sample Batching:

A 'batchID' field was added to the existing 'sample' data element. The interface was changed in the sample entry screen. Since samples are no longer entered one at a time, a listbox was added to display the current batch of samples. A button was also added to allow lab technicians to specify when the batch of samples has been completely entered. Changes in processing were needed when adding a batch of samples to the database. A function was needed to retrieve a valid 'batchID' and assign it to the current batch. Several other minor interface changes were made where a 'batchID' could be used to select a group of samples. Corresponding changes in processing were also made to retrieve and save samples based on 'batchID'.

CS5 - Second Depth Samples:

A boolean 'N-Only' field was added to the existing 'sample' data element. A 'second depth' field was also added to the existing 'sample' data element. The 'N-Only' field is set to true for the second depth sample, while the 'second depth' field of the first sample is set to the unique id of the second depth sample. A change to the interface was made to the sample entry screen. A checkbox was added to specify when a sample is a second depth sample. When the checkbox is checked, a text-entry box is activated. The unique id of the first sample is selected in the text-entry box. A change in processing was made in the functions that save the 'sample' data elements to the database. A change in processing was also made in the data export section of the lab software. When sample data is exported, a function was added that checks for second depth samples. When a second depth sample is found, the Nitrogen value from the second depth sample is added to the first sample before the data is exported.

CS6 - pH/EC Email:

When sending data to Field Service Representatives, the software was changed such that the pH/EC data was converted to Microsoft Excel format and emailed to the Field Service Representatives instead of printed and manually faxed.

CS7 - Sample Forecasting:

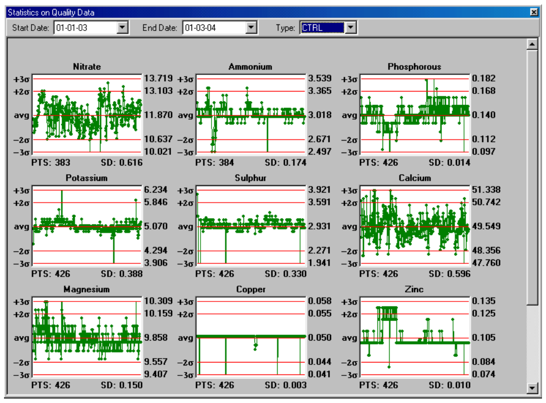
In this change scenario, there is a requirement for a new view of the data that already exists in the lab software, therefore no changes to the data have occurred. There has, however, been a minor change in both interface and processing. First, the interface was changed to include a status bar below the sample data table. Changes in processing were made with the addition of functions to populate the status bar with statistics such as the number of total samples currently shown in the data table. Existing soil sample filtering functionality of the lab software provided the remainder of the tools needed to generate the desired summaries.

CS8 - GPS Format:

GPS data is entered in the same way as before, therefore the interface does not change in the sample entry form. A small function that converts the GPS coordinates into decimal format if it was entered in degree-minute-second format was added to the software, therefore the processing has changed when saving the sample data to the database. A small interface change was made on the screen that displays sample data. A button to toggle between the two GPS formats was added. Here, a function was also added to convert from decimal degree format to degree-minute-second format when needed.

CS9 - Identifying Outliers:

A quality control view, shown in Figure 3.4 was added to the software, where quality control data is shown in a series of charts. The charts allow the lab technicians to quickly identify outliers. As well, a change in processing was required and functions were added to populate the charts and calculate standard deviation of quality control data.


Statistics on Quality Data

Start Date: 01-01-03 End Date: 01-03-04 Type: CTRL

Figure 3.4: New view needed due to the "Identifying Outliers" change scenario

3.5 Summary

This case study has shown us the impact of nine different business process changes on the information systems used by Western Ag. The study has provided us with real world examples of how business process changes can effect information systems. We can use these examples to structure our approach to supporting co-evolution of information systems and business processes. The examples also show that the ability of information systems to quickly adapt to changes in business processes is important to businesses.

We analyzed the impact of each of the nine change scenarios in terms of their impact on the software system in three categories: data, interface, and processing. Our analysis showed that almost all of the business process changes resulted in changes to the system's user interface. Furthermore, our analysis showed that changes in data always leads to corresponding changes to the system's user interface. Evidence of such changes can be seen in CS1 - Sample Priorities, CS2 - Method Blanks, CS3 - Remote Processing Lab, CS4 - Sample Batching, and CS5 - Second Depth Samples. Each of these change scenarios resulted in changes in data and corresponding changes to the system's user interface. Changes in processing occurred, with varying degrees of impact, in all of the nine changes scenarios.

We take a model based approach to supporting the co-evolution of information systems and business processes. We use a task modelling approach to take advantage of its focus on activities and goals. Using task modelling alone, we can model how the activities performed in the information system will change. However, our case study also showed how data elements and user interfaces also change. Data elements can be modelled using data modelling techniques such as XML Schema. User interfaces are best represented by user interface components as is the case in widely used graphical user interface builders such as Visual Basic, Delphi, and Visual Studio. Combining task models, data models, and user interface components in a way that can be quickly modelled to show how information systems change in response to business process changes would help developers co-evolve information systems and business processes.

In Chapter 4, we propose Interaction Templates, a template based approach to task modelling that binds data models and user interface components to task models. The Interaction Template approach will allow developers to quickly, and more accurately, co-evolve information systems and business processes.

Chapter 4

Interaction Template Model

Chapter 3 showed us real world examples of how continually evolving business processes can affect supporting information systems. In this Chapter, we introduce the Interaction Template approach to task modelling. Using Interaction Templates, we hope to improve task modelling to help support the co-evolution of information systems and business processes.

We begin with an introduction of Interaction Templates. Next, we define the terms used in the Interaction Template approach. We continue with a discussion of the semantics of task models as they relate to task model simulation. Finally, we conclude with an outline of the benefits of our approach. In Chapter 5, we will introduce a notation, called the Interaction Template Definition Language, that we use to specify Interaction Templates.

4.1 Interaction Templates

Task modelling, and the ConcurTaskTrees notation in particular, has been shown to be useful when designing interactive systems [37]. Unfortunately, when using ConcurTaskTrees, the task modelling process can be tedious because the models become very large when modelling non-trivial systems. Our previous research [29] has shown that when modelling information systems using ConcurTaskTrees, there are often subtrees that repeat with only slight variations throughout the task model. Furthermore, these repeating subtrees are often associated with common interface interactions found in information systems. We have proposed Interaction Templates [29] as a technique to ease task modelling of information systems by encapsulating these common interface interactions.

Interaction Templates include a detailed and adaptable task model, an execution path (i.e. dialog), and a presentation component. An Interaction Template is an adaptable subtree that can be inserted into a ConcurTaskTree and quickly customized by setting parameter values and assigning data schemas. Inserting and quickly customizing Interaction Templates reduces the need to repeatedly model similar interactions in a system, and thus, can greatly reduce the time spent modelling information systems.

An Interaction Template that has been inserted into a ConcurTaskTree can be adapted by changing its parameter values or updating its data schemas. Therefore, Interaction Templates can be used to adapt a system's task model to changes in business processes.

4.2 Terminology

In this section, we will introduce and define the terms used in the Interaction Template approach.

4.2.1 Interaction Template Definition

An *Interaction Template definition* (*IT definition*) consists of a set of zero or more *parameters*, a set of zero or more *schemas*, a set of zero or more *components*, and a *task template*. That is,

IT_definition ::= **IT** {*parameter*} {*schema*} {*component*} *task_template* **end IT**

parameter ::= **par** *name***:***type*=*value*

schema ::= **schema** *name*

component ::= **component** *name***:***type* {*event*} {*property*}

event ::= **event** *name*

property ::= **property** *name***:***type*

The *parameters* and *schemas* are used within the *task template* to define an adaptable task tree. A *parameter* corresponds to a variable of a specific *type*. A default *value* can be assigned to a *parameter* in an *IT definition*, and can be manually set when the Interaction Template is instantiated. A *parameter's value* is used in the adaptation logic of the Interaction Template's *task template*.

A *schema* represents the description of a *data element*. In an *IT definition*, a *schema* contains only a *name*. When an Interaction Template is instantiated, the location of the *schema's data element description* is specified. A *data element description* contains the description of the structure of a data element. An example of a *data element description* is a XML Schema. The *data element description* is used in the adaptation logic of the *task template*.

A *component* defines a concrete user interface component, also referred to as a presentation component, that implements some portion of the task tree defined by the Interaction Template. A *component* includes a *name*, a *type*, a set of *events*, and a set of behaviour modifying *properties*. The *type* specifies the name of the object that implements the *component*. In order to customize the behaviour of the *component*, the *properties* can be set within the *task template*. *Events* are bound to specific tasks in the *task template*, defining a relationship between the user interactions implemented in the *component*, and the *template's* tasks.

A *task template* is a structure that defines the adaptation logic for an Interaction Template's task tree. That is, a *task template* defines how the template's task tree will adapt to the *values* assigned to *parameters* and to the *data element descriptions* assigned to schemas.

4.2.2 Task Template

A task template is made up of either a *case*, a *loop*, an *Interaction Template Use*, or a *task*. That is,

task_template ::= *case* | *loop* | *IT_use* | *task*

case ::= **case** {*condition*} **end case**

condition ::= **condition** *expression* {*task_template*} **end condition**

loop ::= **loop** {*element*} {*task_template*} **end loop**

IT_use ::= **IT_use** *name* {*parameter_assignment*} {*schema_path*}

parameter_assignment ::= **par** *name*=*value*

schema_path ::= **schema_path** *name*=<*path*>

task ::= **task** *name category* {*unary_operator*} {*component_binding*} {*task_template*}

end task

unary_operator ::= **optional** | **iterative**

category ::= **abstraction** | **system** | **interaction** | **user**

A *case* is an adaptation logic structure that modifies a task tree's structure. A *case* defines a choice between a set of tasks based on the *value* of a *parameter*, or the *data element description* of a *schema*. A *case* contains one or more *conditions*.

A *condition* contains an *expression* and one or more *task templates*. A *condition* defines that the *task templates* will only be included in the task tree when the *expression* evaluates to true. An *expression* is a boolean expression that can contain references to *parameter values* and/or a *schema*'s *data element descriptions*.

A *loop* is an adaptation logic structure that repeats one or more *task templates* for a given set of *elements* selected from a *schema*'s *data element description*. A *loop* contains a set of one or more *elements* and a set of one or more *task templates*. The set of *task templates* is repeated for each of the *elements*. The *task templates* can contain references to the current *element*, allowing us to adapt tasks according to the *element*'s details. For example a *loop* could be used to repeat a task for each of the attributes of a data element.

An *Interaction Template use* (*IT use*) is a reference to another *IT definition*. An *IT use* consists of a *name*, a set of *parameter assignments*, and a set of *schema paths*. The *name* specifies the name of the Interaction Template being inserted in to the current template. The *parameter assignments* consist of *name, value* pairs to assign values to the template's *parameters*. The *schema paths* consist of *name, path* pairs assigning paths to data elements for the templates *schemas*. An *IT use* is used to instantiate the specified Interaction Template, using the *parameter assignments* and *schema paths*, when the current template is instantiated.

A *task* defines a task in the Interaction Template's task tree. A task contains a *name*, a *category*, and two unary operators: *optional* and *iterative*.

The task's *name* is a string specifying the name of the task. The *name* can contain a reference to a *parameter value*, or to a specific section of the *data element description* of a *schema*. That is, the name of a task can be adapted to parameters and schemas. The task's *category* specifies the category of the task (Interaction, Abstraction, System, or User). The two unary operators are boolean values. The unary operators can contain a boolean expression in the same way as a *condition*'s *expression*. Therefore, a task's unary operators can adapt to parameters and schemas.

A *task* may also contain other optional properties. A *task* may contain a *temporal operator* specifying a temporal operator for the task. A *task* can also optionally contain *component bindings* that define a relationship between a *component* and the current task. A *task* can also contain a set of one or more *task templates*, defining the *task*'s child *tasks*.

4.2.3 Component Bindings

Component bindings define a relationship between tasks and presentation components in a template. There are three types of component bindings: *property bindings*, *event bindings*, and *task bindings*. That is,

component_binding ::= *property_binding* | *event_binding* | *task_binding*

property_binding ::= **bind_prop** *component_name***.***property_name*=*parameter_name* *event_binding* ::= **bind_event** *component_name***.***event_name*

task_binding ::= **bind_task** *component_name*

Property bindings define a relationship between a *parameter* and a *component*'s *property*. A *property binding* is used to assign a *parameter value* to a *component*'s property. *Event bindings* define a relationship between a *component*'s *event* and a leaf task in the task tree. That is, when the *event* is fired by the *component*, a specific *task* is performed in the task model. *Task bindings* defines a relationship between a task and the component. That is, the interaction modelled by the *task* is implemented by the *component*.

4.3 Task Model Simulation Semantics

In order to simulate a task model, we need to keep track of two bits of state information for each task in the task tree. A task can be in one of four states: Disabled, Enabled, Performing, or Performed. Figure 4.1 shows the different states a task can be in during simulation, and the transitions that can occur between states. In this state transition network, the transitions occur when messages are sent to a task. The transition messages are sent by neighbouring siblings and parents as will be seen in the following sections.

Any task that is currently *enabled* can be *started*, at which point its state changes from *enabled* to *performing*. A non-iterative task that is in the *performing* state changes to the *performed* state only when that task is *completed*. For an iterative task, the task becomes *enabled* when the *complete* transition occurs. The *complete*

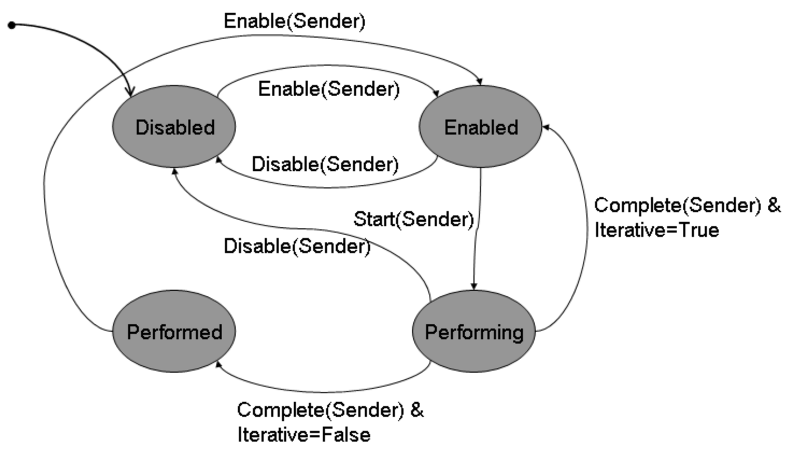


Figure 4.1: State transition network for tasks during task model simulation

transition occurs according to the hierarchical semantics of the task tree, as will be discussed shortly.

4.3.1 Hierarchical Semantics

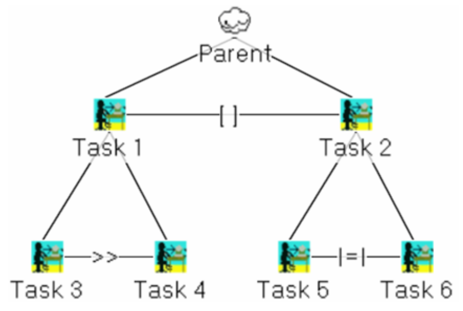


Figure 4.2: Hierarchical task structure of a ConcurTaskTree

The hierarchical structure of a task tree defines relationships between parent and child tasks. The following rules summarize our interpretation of the hierarchical semantics of the ConcurTaskTrees notation.

Enable When a task is *enabled*, its left-most child task is *enabled*.

Disable When a task is *disabled*, all of its children are *disabled*.

Complete A task is *completed* only once all of its non-optional sub-tasks have been *completed*. Therefore, leaf tasks are *completed* immediately after they are started.

4.3.2 Temporal Operator Semantics

Temporal operators define temporal relationships between sibling tasks in a task tree. In task model simulation, this means that a state transition in one task can result in a state transition in that task's neighbouring siblings. The resulting transition, if any, is defined by the task's temporal operator. When a task receives a transition message, it sends a transition message to its left and right siblings based on the value of the tasks left and right temporal operators. Guards are used to ensure that transition messages do not cause message loops. Table 4.1 summarizes our interpretation of the semantics of the temporal operators in the ConcurTaskTrees notation.

Rule 1, for example, describes the semantics of the $>>$ (*Enabling*) temporal operator. Rule 1 states that when the current task T receives a *Complete* message and the task's right temporal operator is $>>$, then an *Enable* message is sent to the right sibling, *RS*, of task T.

Rule 9 makes use of a guard to ensure that messages do not cause infinite message loops. Rule 9 states that when the current task T receives an *Enable* message, and the task's right temporal operator is $|||$ (*Concurrent*) and the *Sender* is not the task's right sibling (*RS*), then an *Enable* message is sent to *RS*.

Table 4.1: Rules defining the semantics of temporal operators in the ConcurTaskTree notation. When a transition message is received by a task, transition messages are sent to left and right sibling tasks based on the values of temporal operators. Guards are used to prevent transition messages from causing loops.

Rule	**Message Received**	**Temporal Operator**		**Message Sent**		
	Current Task (T)	Right	Left	Guard	Right Sibling (RS)	Left Sibling (LS)
Rule 1	T.Complete(Sender)	>>			RS.Enable(T)	
Rule 2	T.Complete(Sender)	[]>>			RS.Enable(T)	
Rule 3	T.Enable(Sender)	[]		Sender $\neq$ RS	RS.Enable(T)	
Rule 4	T.Disable(Sender)	[]		Sender $\neq$ RS	RS.Disable(T)	
Rule 5	T.Start(Sender)	[]			RS.Disable(T)	
Rule 6	T.Enable(Sender)		[]	Sender $\neq$ LS		LS.Enable(T)
Rule 7	T.Disable(Sender)		[]	Sender $\neq$ LS		LS.Disable(T)
Rule 8	T.Start(Sender)		[]			LS.Disable(T)
Rule 9	T.Enable(Sender)	|||		Sender $\neq$ RS	RS.Enable(T)	
Rule 10	T.Enable(Sender)		|||	Sender $\neq$ LS		LS.Enable(T)
Rule 11	T.Enable(Sender)	|[]|		Sender $\neq$ RS	RS.Enable(T)	
Rule 12	T.Enable(Sender)		|[]|	Sender $\neq$ LS		LS.Enable(T)
Rule 13	T.Enable(Sender)	|=|		Sender $\neq$ RS	RS.Enable(T)	
Rule 14	T.Enable(Sender)		|=|	Sender $\neq$ LS		LS.Enable(T)
Rule 15	T.Enable(Sender)	[>			RS.Enable(T)	
Rule 16	T.Start(Sender)		[>			LS.Disable(T)
Rule 17	T.Enable(Sender)	|>			RS.Enable(T)	
Rule 18	T.Start(Sender)		|>			LS.Disable(T)
Rule 19	T.Complete(Sender)		|>			LS.Enable(T)

4.3.3 Task Model Simulation

The first step involved in task model simulation is to calculate the set of tasks that are initially enabled. The initially enabled tasks are calculated by enabling the root task and following the rules we have outlined above. The task model simulation process consists of simulating the performance of currently enabled leaf tasks, and calculating the resulting enabled tasks according to the semantics of the task tree.

4.3.4 Interaction Template Simulation

Using current task modelling techniques, no link exists between a system's implementation and a system's task model. The effects of changes to a task model on a system's implementation are unclear. Binding presentation components to task models can be useful for developers by bridging the gap between a system's task model and a system's implementation. The structure of the task model defines which user interface components should be enabled at each point as the user progresses through activities to reach specific goals.

Furthermore, component bindings can be used to create user interface prototypes of a task model. When modelling with Interaction Templates, task model simulation can be enhanced through the use of concrete user interface components. User interface prototypes can be generated from the Interaction Templates found in a task model. When a task model is being simulated, users can interact with concrete user interface prototypes to perform specific tasks. These prototypes can help both users and developers to see how a system will change in response to changes in business processes. Since the prototypes involve concrete user interface components, some of the implementation details are explicitly shown to the developer. The prototypes resemblance to the system's user interface also helps users visualize how they will perform tasks in a new system. In Chapters 6 and 7, we will see how user interface prototypes are generated and used in task model simulation.

4.4 Summary

In Chapter 5, we will introduce the Interaction Template Definition Language [31] used to specify Interaction Templates. We will show how user interface prototypes can be generated from task models in Chapter 6. In Chapter 7, we will illustrate how Interaction Templates can be built to adapt to some of the changes that can occur as a result of changes in business processes.

Chapter 5

Interaction Template Notation

In the previous Chapter, we introduced the Interaction Template approach to task modelling and discussed its benefits in terms of supporting the co-evolution of information systems and business processes. In this chapter, we introduce an XML based notation for specifying Interaction Templates. In the next chapter, we will introduce a prototype of a task model editor that supports the Interaction Template approach.

We have developed a custom XML language for specifying Interaction Templates. There are multiple benefits to using XML to specify Interaction Templates. The tree structure of XML documents allows us to easily specify ConcurTaskTrees. Using XML also allows us to make use of existing XML processing technologies such as XPath, as we will see later in this Chapter. Our custom XML language is called the Interaction Template Definition Language (ITDL) [31]. The notation is used to specify Interaction Template definitions, that is, a set of parameters, a set of schemas, a set of components, and a task template.

We begin by describing the XML language used to specify ConcurTaskTrees. Then, we introduce the Interaction Template Definition Language.

5.1 Specifying ConcurTaskTrees with XML

The ITDL is based on an XML description of the Interaction Template's task tree. We use a modification of the XML language used by the ConcurTaskTreesEnvironment (CTTE) [35]. In this section, we will explain each of the elements used to specify ConcurTaskTrees. A complete XML Schema for the CTT elements can be found in Appendix A. The `ctt` namespace is used to denote elements that describe ConcurTaskTrees.

The root element for a ConcurTaskTree document is the `<ctt:TaskModel>` element. The `<ctt:TaskModel>` element contains a `name` attribute of type string that specifies the model's name, and a single `<ctt:Task>` element specifying the model's root task. A `<ctt:Task>` element contains four attributes: `ID`, `Category`, `Iterative`, and `Optional`. The `ID` attribute is a string specifying the task's name. The `Category` attribute is a restricted string type specifying the task's category. The possible values for `Category` are Abstraction, Interaction, System, and User. The `Iterative` and `Optional` attributes are boolean values specifying the task's iterative and optional operators respectively. Every `<ctt:Task>` element also contains a `<ctt:TemporalOperator>` element, specifying the task's temporal relation with its right sibling. The `<ctt:TemporalOperator>` is a restricted string type with possible values Choice, Order Independency, Concurrent, Concurrent with Information Exchange, Disabling, Suspend-Resume, Enabling, and Enabling with Information Exchange. The `<ctt:TemporalOperator>` element may also be empty when the task does not have a right sibling, because no temporal operator is required in this case. If a task has any children, it contains a `<ctt:SubTasks>` element. The `<ctt:SubTasks>` element contains one `<ctt:Task>` element for each of the task's children, and the children are listed in order of left to right as they appear in the task tree.

The following example shows a simple CTT where the root task has two abstract child tasks. The two child tasks are related using the Choice temporal operator. The graphical representation of this simple CTT is shown in Figure 5.1.

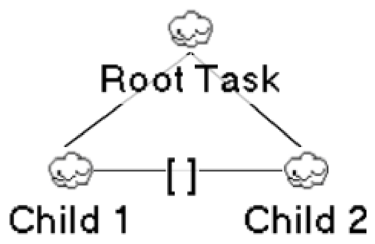


Figure 5.1: A simple ConcurTaskTree

```
<ctt:TaskModel name="Example Task Model">
  <ctt:Task ID="Root Task" Category="Abstraction" Iterative="false" Optional="false">
    <ctt:TemporalOperator></ctt:TemporalOperator>

    <ctt:SubTasks>
      <ctt:Task ID="Child 1" Category="Abstraction" Iterative="false" Optional="false">
        <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
      </ctt:Task>
      <ctt:Task ID="Child 2" Category="Abstraction" Iterative="false" Optional="false">
        <ctt:TemporalOperator></ctt:TemporalOperator>
      </ctt:Task>
    </ctt:SubTasks>

  </ctt:Task>

</ctt:TaskModel>
```

5.2 Specifying Interaction Templates with XML

In this section, we will describe each of the elements of the Interaction Template Definition Language. The `it` namespace is used to denote the elements that describe the parameters and behaviour of Interaction Templates. A complete XML Schema defining the Interaction Template elements can be found in Appendix B.

5.2.1 Template Element

The root element of an Interaction Template document is the `<it:template>` element.

```
<xs:element name="template">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="description" type="xs:string"
              minOccurs="1" maxOccurs="1"/>
      <xs:element name="component" type="it:component"
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="parameter" type="it:parameter"
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="schema" type="it:schema"
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="task" type="ctt:Task"
              minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="case" type="it:case"
              minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

The `<it:template>` element contains a `name` attribute that specifies a name that is meaningful to the template's user. The `<it:template>` element is a complex element containing the following elements:

- `<it:description>` is a simple string element that is required to occur once and only once. This element contains a string describing the Interaction Template and its intended use.

- `<it:component>` is an empty element specifying the name of a presentation component that implements the interactions defined by the Interaction Template.

- `<it:parameter>` is a complex element that defines one of the Interaction Template's parameters. This element occurs once for each of the template's parameters.

- `<it:schema>` is an empty element that specifies any XML Schemas the Interaction Template requires. This element occurs once for each schema required by the template.

- `<ctt:Task>` is a complex element describing the Interaction Template's task template. This element follows the ConcurTaskTree XML Schema Definition as discussed above, but with modifications that allow us to specify the task's adaptation logic.

- `<it:case>` is a complex element that allows the template to select from a set of tasks depending on the values of parameters and schemas.

The following example shows a skeleton Interaction Template document.

```
<it:template name="...">
  <it:description>...</it:description>
  <it:component name="..." />
  <it:parameter name="..." type="...">
    <it:value>...</it:value> ...
  </it:parameter>
  <it:schema name="..."/>

  <ctt:Task ID="..." Category="..." Iterative="..." Optional="...">
    <ctt:TemporalOperator>...</ctt:TemporalOperator>
    <ctt:SubTasks>...</ctt:SubTasks>
  </ctt:Task>

</it:template>
```

5.2.2 Component Element

An `<it:template>` element contains one `<it:component>` element for each presentation component that implements the interaction modelled by a specific task in the

Interaction Template.

```
<xs:complexType name="component">
  <xs:attribute name="name" type="xs:string" use="required" />
  <xs:attribute name="objectname" type="xs:string" use="required" />
</xs:complexType>
```

The `name` attribute indicates the name given to the current component. The name is referred to within the template when specifying component bindings. The `objectname` attribute indicates the name of the object that implements the presentation component.

Components can be defined at the beginning of an Interaction Template, as is often the case when a single component implements all of the interaction encapsulated within the template. Alternatively, components can be defined from within specific tasks in the template. Defining components within tasks allows us to design templates that make use of a number of different components, and gives us the ability to select different components depending on the structure of the resulting task tree. We will discuss presentation components further in Section 5.5.

5.2.3 Parameter Element

The `<it:parameter>` elements define the Interaction Template's parameters.

```
<xs:complexType name="parameter">
  <xs:sequence>
    <xs:element name="value" minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="it:parameterType" use="required"/>
</xs:complexType>
```

The `<it:parameter>` element has two attributes: `name` and `type`. The `name` attribute is a string that specifies the name of the parameter. The name will be used later in the Interaction Template specification to specify the template's adaptive behaviour. The `type` attribute is a restricted string specifying the parameter's type. Table 5.1 shows the possible parameter types for Interaction Templates.

The `<it:parameter>` element contains one `<it:value>` element. The `<it:value>` element's type is not restricted by the XML Schema shown above, but the value should be the same type as is specified in the current `<it:parameter>` element's `type` attribute. If a value is specified for the `<it:value>` element, that value will be the parameter's default value. If no default value is specified, the user of the Interaction Template will be required to specify a value when inserting the template into a task tree.

Table 5.1: Interaction Template parameter types

Parameter Type	**Description**
it:string	Alphanumeric String
it:decimal	Integer value
it:float	Floating point value
it:boolean	True or False

The following example specifies a boolean parameter called MyParam whose default value is false.

```
<it:parameter name="MyParam" type="it:boolean">
  <it:value>false</it:value>
</it:parameter>
```

5.2.4 Schema Element

Interaction Templates can be designed to adapt to the structure of data elements. Data elements are bound to Interaction Templates through XML Schemas that define the structure of data elements. We will see how Interaction Templates can adapt to data elements in section 5.3.

```
<xs:complexType name="schema">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>
```

Each `<it:schema>` element defines a data element that is bound to the template. This element contains a `name` attribute. The `name` attribute is a string specifying the name that will be used to refer to the data element from within the template. The `<it:schema>` element also contains a `path` attribute that remains empty until the Interaction Template is inserted into a ConcurTaskTree. The path attribute is used to specify the location of the data element that will be linked to the template.

5.3 Specifying Task Model Adaptation Logic

Within the context of an `<it:template>` element, the `<ctt:Task>` element is structured slightly differently than defined inside the ConcurTaskTree XML Schema discussed earlier. Certain commands are either embedded within the `<ctt:Task>` element, or surrounding the `<ctt:Task>` element. When interpreted, these commands transform the Interaction Template definition into a valid ConcurTaskTree `<ctt:Task>` element. The structure of the resulting `<ctt:Task>` element can vary depending on the parameter values and schemas that are specified, allowing for task model adaptability to be defined inside the Interaction Template definition.

5.3.1 Referencing Parameter Values

The simplest form of adaptation comes in the form of referencing parameter values from within a `<ctt:Task>` element's attributes and child elements. References to parameter values can be inserted into any of the attributes or elements of the `<ctt:Task>` element. These references, when interpreted, will result in attribute and element values that are valid as defined by the CTT XML Schema. Parameter values are referenced using the XML Path Language (XPath). XPath, as described in Section 2.5.2, provides a method of navigating the elements and attributes in an XML document. From anywhere within the `<it:template> element`, the XPath expression shown in Expression 5.1 will select the text within the `value` element of the `<it:parameter>` element whose `name` attribute equals `ParamName`:

$$\mathsf{ancestor::it:template/it:parameter[@name=\text{“ParamName”}]/it:value/text()} \tag{5.1}$$

First, `ancestor::it:template` selects the nearest ancestor element that is an `<it:template>` element. Next, `/it:parameter[@name="ParamName"]` selects the `<it:parameter>` element whose `name` attribute equals `ParamName`. Finally, `/it:value` selects the `<it:parameter>` element's `<it:value>` element, and `/text()` selects the text inside that element. To simplify parameter references within Interaction Templates, we use `$ParamName` as a short form of Expression 5.1.

The following example shows how a boolean parameter called `MyBool` is referenced to set the root task's *optional* operator. While `$MyBool` is not a valid value for the `Optional` attribute, the reference will be replaced with a valid boolean value when it is interpreted.

```
<it:template name="Boolean Example">
  ...
  <it:parameter name="MyBool" type="it:boolean">
    <it:value>False</it:value>
  </it:parameter>
  ...
  <ctt:Task ID="Root Task" Category="Abstraction" Iterative="False" Optional="$MyBool">
    <ctt:TemporalOperator></TemporalOperator>
    ...
  </ctt:Task>

</it:template>
```

Since parameter references can be inserted into any of the attributes or elements within the `<ctt:Task>` element, we must exercise caution when designing Interaction Templates. Inserting parameter references into incorrect attributes or elements could result in an invalid ConcurTaskTree. For example, inserting `$MyBool` in the `Category` attribute above would result in an invalid value for the `Category` attribute. We have chosen to assign the responsibility of ensuring the validity of the resulting tree to the Interaction Template designer, as we did not want to impose design limitations on our approach at the language level. Proper tool support could help to guide Interaction Template designers in this respect.

5.3.2 Referencing Schema Values

When a template is inserted into a ConcurTaskTree, a valid XML Schema data element must be specified for each of the `<it:schema>` elements in the template. A data element's path can be specified in one of two ways: as an XML Schema file path, or as an XPath expression leading to an XML Schema element in an XML Schema file linked elsewhere within the scope of the current template.

When linking to an XML Schema file, the `path` attribute is set to a string specifying the location of the file. From within an `<it:template>` element, Expression 5.2 will select the `filepath` attribute of the `<it:schema>` element whose `name` attribute equals `SchemaName`:

$$\text{ancestor::it:template/it:schema[@name=“SchemaName”]/@path} \tag{5.2}$$

If the XML Schema file specified above is loaded and parsed, we can use XPath expressions to select attributes and elements from the schema definition. For example, Expression 5.3 will select the data element that is specified in an XML Schema document.

$$\text{/xs:schema/xs:element} \tag{5.3}$$

Within the context of an `<it:template>` element, we will use `$SchemaName` as a shortcut to load and parse the file specified in the `<it:schema>` element whose `name` attribute equals `SchemaName`. Also, since we are always interested in the data element defined in the schema document, our shortcut will also execute Expression 5.3 to select the data element definition from the schema document.

When linking the `<it:schema>` to an element within an existing XML Schema file, we simply set the `path` attribute to an XPath expression leading to the desired element. In this case, the `$SchemaName` shortcut simply points to the specified XPath expression.

To select different elements and attributes from XML Schemas, we can continue XPath expressions from our shortcut reference. The expression `$SchemaName/@name`, for example, will select the name of the root xs:element. Much like parameter references, references to schema values can also be inserted into any of the `<ctt:Task>` element's attributes or elements.

The following example shows how the name of a task can be adapted to the name of the data element described in an XML Schema document.

```
<it:template name="Enter Data Element">
  ...
  <it:schema name="DataElement" path="" />

  <ctt:Task ID="Enter $DataElement/@name" Category="Interaction" ... >
     ...
  </ctt:Task>

</it:template>
```

If the `<it:schema>` element's `path` attribute is set to lead to the XML Schema document shown below, the `ID` of the template's root task will become "Enter Person" when the template is interpreted.

```
<xs:schema>
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FirstName" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
        <xs:element name="LastName" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Again, we must exercise caution to ensure the resulting task tree is valid when designing Interaction Templates. This requires some knowledge of XML Schema, and possibly some assumptions about the structure of the schema you are given.

We can also select a list of XML elements using XPath. For example, Expression 5.4 will select all of the `<xs:element>` elements that describe the data element's children. In the schema shown above, Expression 5.4 would select two `<xs:element>` elements: "FirstName", and "LastName". Element lists can be used in Interaction Template loops to adapt the structure of a task tree based on XML Schemas, as will be discussed shortly.

$$\text{\$DataElement/xs:complexType/xs:sequence/xs:element} \tag{5.4}$$

5.3.3 Boolean Expressions

Boolean expressions can be inserted into the `Optional` and `Iterative` attributes of the `<ctt:Task>` element, as well as the `expression` attribute of the `<it:condition>` element. The expressions can be built using the following atoms: (,), and, or, not, eq, ne, gt, lt, ge, le, true, and false. Parameter and schema references can be inserted into the expressions. References will be interpreted before the expression is evaluated. Expressions can be used to compare strings, integers, floats, and booleans. Boolean expressions are identified by surrounding `expr()`.

The following example shows how a boolean expression can be used inside the root task's `Optional` attribute. A parameter, called `MyBool` is negated inside the expression. While the expression is not a valid value for the `Optional` attribute, the expression will be evaluated and replaced with a valid boolean value when the Interaction Template is interpreted.

```
<it:template name="Boolean Example">
  ...
  <it:parameter name="MyBool" type="it:boolean">
    <it:value>False</it:value>
  </it:parameter>
  ...
  <ctt:Task ID="Root Task" Category="Abstraction" Iterative="False" Optional="expr( not $MyBool )">
    <ctt:TemporalOperator></TemporalOperator>
    ...
  </ctt:Task>

</it:template>
```

Cases

The `<it:case>` element is used to select a specific task or subtask based on the template's parameter values.

```
<xs:complexType name="condition">
  <xs:sequence>
    <xs:element name="Task" type="ctt:Task"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="SubTasks" type="ctt:SubTasks"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="case" type="it:case"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="foreach" type="it:foreach"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="var" type="it:var"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="bindevent" type="it:bindevent"
            minOcuurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="expression" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="case">
  <xs:sequence>
    <xs:element name="condition" type="it:condition"
            minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

The `<it:case>` element contains at least one `<it:condition>` element, and can appear anywhere inside the `<it:template>` element after the description, component, parameter, and schema elements. The `<it:condition>` defines a choice between different tasks or subtasks in the template. Any `<ctt:Task>`, `<ctt:SubTasks>` or `<it:foreach>` element in a template can be surrounded by an `<it:condition>` element. The `<it:condition>` element contains an `expression` attribute that defines when the condition should be chosen. When interpreted, the first `<it:condition>` element whose `expression` attribute evaluates to true is chosen. The expressions can contain references to parameter and schema values as described above.

The following `<it:case>` element describes a choice between three different tasks:

...

```
<it:case>
  <it:condition expression="expr( $Param1 eq true )">
    <ctt:Task ID="Task Option 1" ...> ... </ctt:Task>
  </it:condition>

  <it:condition expression="expr( $Param2 eq true )">
    <ctt:Task ID="Task Option 2" ...> ... </ctt:Task>
  </it:condition>

  <it:condition expression="expr( true )">
    <ctt:Task ID="Default Task Option" ...> ... </ctt:Task>
  </it:condition>
</it:case>
...
```

In this example, we assume two boolean parameters named `Param1` and `Param2` have been defined. The task named "Task Option 1" is chosen if the `Param1` parameter is equal to true. Otherwise, the task named "Task Option 2" is chosen if `Param2` is equal to true. If neither of the two parameters are true, then the task named "Default Task Option" is chosen.

Using the `<it:condition>` and `<it:case>` elements, we can design Interaction Templates that will result in different task tree structures depending on current parameter and schema values.

5.3.4 Loops

Interaction Template loops are used to repeat `<ctt:Task>` elements for each element in a specified list of XML elements. Loops are specified using the `<it:foreach>` element.

```
<xs:complexType name="foreach">
  <xs:sequence>
      <xs:element name="Task" type="ctt:Task"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="case" type="it:case"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="foreach" type="it:foreach"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="var" type="it:var"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="bindevent" type="it:bindevent"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>
```

The `<it:foreach>` element is a complex element containing a single `value` attribute and a single `<ctt:Task>` child element. The `value` attribute is a string containing a reference to a list of XML elements. As discussed earlier, element lists can be selected using references to schemas. When interpreted, the child `<ctt:Task>` element will be inserted into the resulting ConcurTaskTree once for each element in the list of elements specified by the `value` attribute. The current element can be referenced simply by inserting `$value` in the attributes and elements of the `<ctt:Task>` element. Like schema references, the reference to the current element can be navigated by adding an XPath expression to the end of the `$value` shortcut.

Let us extend the example that was used to show how to reference values inside a schema. We will now show how we can define the Interaction Template to build a task tree to enter the data element containing a sequence of elements as described in an XML Schema document.

```
<it:template name="Enter Data Element">
  ...
  <it:schema name="DataElement" filepath="" />

  <ctt:Task ID="Enter $DataElement/@name" Category="Interaction" ... >
     <ctt:TemporalOperator></ctt:TemporalOperator>
     <ctt:SubTasks>

       <it:foreach value="$DataElement/xs:complexType/xs:sequence/xs:element">
         <ctt:Task ID="Enter $value/@name" Category="Interaction" ... >
            <ctt:TemporalOperator>Order Independence<ctt:TemporalOperator>
         </ctt:Task>
       </it:foreach>

     </ctt:SubTasks>
  </ctt:Task>

</it:template>
```

If the `<it:schema>` element's `path` attribute is set to the XML Schema shown below, the `ID` of the template's root task will become "Enter Person" when the template is interpreted. Furthermore, the “Enter Person” task will have two child tasks: “Enter FirstName” and “Enter LastName”. The two child tasks will be related with the “Order Independence” temporal operator. The resulting ConcurTaskTree is shown in Figure 5.2.

```
<xs:schema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FirstName" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
        <xs:element name="LastName" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

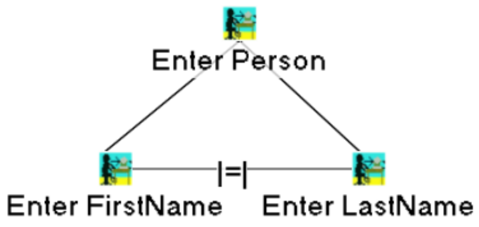


Figure 5.2: A ConcurTaskTree adapted to an XML Schema

In more complex `<it:foreach>` commands, it may be easier to express the tree traversals by creating variables. Interaction Template variables can be inserted anywhere within a template by using the `<it:var>` element. The `<it:var>` element contains two attributes: `name` and `path`. The `name` attribute is a string containing the variable's name. The `path` attribute is a string containing a parameter reference, schema reference, or boolean expression. A variable is accessible to any of its parent element's descendants. The variable can be referenced by using the `$VarName` shortcut in the same way as the current value is referenced within a `<it:foreach>` element.

The following example shows how an `<it:var>` element can be used within the "Enter Data Element" above. Inside the `<it:foreach>` element, a variable named "ElementName" is declared, and it is set to refer to the current element's name attribute. The variable is then referenced from within the `ID` attribute of the loop's root task.

```
<it:template name="Enter Data Element">
  ...
  <it:schema name="DataElement" filepath="" />

  <ctt:Task ID="Enter $DataElement/@name" Category="Interaction" ... >
     <ctt:TemporalOperator></ctt:TemporalOperator>
     <ctt:SubTasks>

       <it:foreach value="$DataElement/xs:complexType/xs:sequence/xs:element">

         <it:var name="ElementName" path="$value/@name" />

         <ctt:Task ID="Enter $ElementName" Category="Interaction" ... >
            <ctt:TemporalOperator>Order Independence<ctt:TemporalOperator>
         </ctt:Task>
       </it:foreach>

     </ctt:SubTasks>
  </ctt:Task>

</it:template>
```

5.4 Using Existing Interaction Templates

In some cases, it may be helpful to make use of existing Interaction Templates from within a template. This allows us to make use of simpler templates to compose more complex Interaction Templates. An existing template can be used by inserting the `<it:usetemplate>` element in place of a task in the template's task tree.

```
<xs:complexType name="usetemplate">
  <xs:sequence>
    <xs:element name="parameter" type="it:parameter"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="schema" type="it:schema"
            minOccurs="=" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>
```

The `<it:usetemplate>` element contains a `name` attribute that specifies the name of the Interaction Template that is being used. This element also contains `<it:parameter>` and `<it:schema>` child elements.

One `<it:parameter>` element exists for each `<it:parameter>` element in the template that is being used. Each parameter's `<it:value>` element will contain the values for the given use of the template. Parameter and schema references can be inserted into the `<it:value>` element, allowing the template to be flexible with respect to parameter and schema values.

Likewise, an `<it:schema>` element exists for every `<it:schema>` element in the template that is begin used. The schema's `path` attribute is set to the location of an XML Schema file. Alternatively, the `path` attribute can contain a reference to an element within a schema file that is linked elsewhere.

The following example shows how the "Enter Data Element" template can be made to use the very simple "Enter String" template.

```
<it:template name="Enter String">
  ...
  <it:parameter name="StringName" type="it:string">
    <it:value>it:var</it:var>
  </it:parameter>

  <ctt:Task ID="Enter $StringName" Category="Interaction" ... >
     <ctt:TemporalOperator></ctt:TemporalOperator>
  </ctt:Task>

</it:template>


<it:template name="Enter Data Element">
  ...
  <it:schema name="DataElement" filepath="" />

  <ctt:Task ID="Enter $DataElement/@name" Category="Interaction" ... >
     <ctt:TemporalOperator></ctt:TemporalOperator>
     <ctt:SubTasks>

       <it:foreach value="$DataElement/xs:complexType/xs:sequence/xs:element">

         <it:usetemplate name="Enter String">
           <it:parameter name="StringName">
             <it:value>$value/@name</it:value>
           </it:parameter>
```

```
        </it:usetemplate>

      </it:foreach>

    </ctt:SubTasks>
  </ctt:Task>

</it:template>
```

5.5 Binding to Presentation Components

We will now discuss how presentation components are bound to the tasks defined in an Interaction Template. First, we will show how components are declared and bound to specific subtrees in the template's task tree. Next, we will show how presentation component properties can be set from within an Interaction Template definition. Finally, we will show how specific component events are bound to specific tasks in the task tree.

Components are defined by the `<it:component>` elements as was discussed earlier. Components are defined either at the beginning of the template, or inside the tasks found in the Interaction Template definition. The location of the `<it:component>` element affects the scope of the component inside the template. Components that are defined at the beginning of the template are available to all tasks inside the template. Components that are defined inside of a task are available only to descendants of the task in which the `<it:component>` element appears.

Once a component is defined it can be bound to any task in which the component is in scope. Components must be bound to the task that models the user interface interaction provided by the component. The task might be a leaf task, or a task that has many children. Components are bound to tasks using the `<it:bindcomponent>`

element. The `<it:bindcomponent>` element contains a single `component` attribute specifying the name of one the components being bound to the current task. The value of the `component` attribute must match the value of the `name` attribute of one of the `<it:component>` elements that is currently in scope.

Presentation components have properties that affect their behaviour. These properties can be set from within an Interaction Template definition using the `<it:setproperty>` element. The `<it:setproperty>` element contains three attributes: component, property, and value. The `component` attribute specifies the name of the component whose property you wish to set. The value of the `component` attribute must match the value of the `name` attribute of one of the `<it:component>` elements that is currently in scope. The `property` specifies the name of the property you wish to set. Finally, the `value` attribute specifies the value you wish to assign to the property. The `value` attribute can contain a parameter reference, a schema reference, or a boolean expression.

Finally, specific tasks must be bound to the event structure of the components being modelled by the template. When defining an Interaction Template, we add an `<it:bindevent>` to some of the basic task elements. Basic tasks are tasks that are leaf nodes in the task tree. Leaf nodes are the only tasks that can be performed in simulation. The `<it:bindevent>` element allows us to define how a presentation component's behaviour is linked to specific tasks in the template's task tree. The `<it:bindevent>` element contains two string attributes: `component` and `event`. The `component` attribute specifies the name of the component. The value of the `component` attribute must match with the `component` attribute of one of the

`<it:bindcomponent>` elements of one of the ancestor tasks of the current task. The `event` attribute specifies the name of the event you wish to bind to the current task.

In the simple example below, a TButton object is bound to an Interaction task called “Cancel”. The `<it:bindevent>` element specifies that when the OnClick event of the TButton object is fired, the “Cancel” task is performed. In this example, both the component and the event are bound to the same task. It is possible, however, for the component to be bound to a higher level task, and for several events to be bound to different basic tasks.

```
...
  <it:parameter name="AllowCancel" type="it:boolean>
    <it:value>True</it:value>
  </it:parameter>
...
  <it:component name="CancelButton" objectname="TButton" />
  <it:setproperty component="CancelButton" property="Enabled" value="$AllowCancel"/>
...

  <ctt:Task ID="Cancel" Category="Interaction" Optional="True" Iterative="False">
    <it:bindcomponent component="CancelButton" />
    <it:bindevent component="CancelButton" event="OnClick">...</it:bindevent>
  </ctt:Task>
...
```

The event binding may not always be a simple one to one mapping between events and tasks. In some cases, the task that is performed when an event is fired may depend on the current state of the presentation component, or on the values of an event’s parameters. In this case, a small amount of code may be required to decide on the name of the task that is being performed. Custom event code is defined inside the `<it:bindevent>` element’s `<it:taskname>` child element. The code that is defined in the `<it:taskname>` element will be executed when the event is fired. The code, therefore, has access to the same variables, methods, and parameters as any other code that would be written inside that specific event handler. The code must follow the usual Object Pascal syntax, and must set the value of the TaskName variable equal to the task that is being performed. This type of binding is commonly found inside of Interaction Template loops.

The following example shows how the "OnSelect" event of the "TComboBox" object is linked to all of the tasks generated by a `<it:foreach>` element. A single line of code is executed when the "OnSelect" event is fired, that is, when an item in the ComboBox is selected. The code simply sets the TaskName variable according to the currently selected item.

```
...
  <it:foreach value="$DataElement/xs:restriction/xs:enumeration/@value">
    <ctt:Task ID="Select $value" Category="Interaction" Optional="True" Iterative="False">
      <it:bindevent component="TComboBox" event="OnSelect">
        <it:taskname>
          TaskName := ‘Select ’ + Self.Text;
        </it:taskname>
      </it:bindevent>
    </ctt:Task>
  </it:foreach>
...
```

Using the `<it:bindevent>` element allows us to make use of any pre-built presentation component without manually coding any modifications to the component. As defined inside the Interaction Template definition, an extension to the component is generated by the Model-IT environment. The details of how prototypes are generated from Interaction Template definitions will be discussed in Chapter 6.

5.6 Summary

We have introduced the Interaction Template Definition Language, our notation for specifying Interaction Templates. We have shown how we use the notation to bind data elements and interface components to task templates. We have also shown how the notation is used to define a template's adaptation logic. In the next Chapter, we will introduce a prototype of a task modelling tool that supports the Interaction Template approach.

Chapter 6

Modelling with Interaction Templates

In Chapter 4, we introduced the Interaction Template approach. In Chapter 5, we outlined the Interaction Template Definition Language. In this Chapter, we introduce a prototype of a task modelling tool that supports the Interaction Template approach. The prototype system is named Model-IT, which stands for Modelling with Interaction Templates. We begin with an overview of the Model-IT prototype. Next, we show how Interaction Templates can be inserted into task models using Model-IT. Finally, we discuss how task models are simulated in the Model-IT prototype.

In the next Chapter, we will illustrate the Interaction Template approach with three Interaction Template examples, followed by an example of how the templates can be used together to support the co-evolution of information systems and business processes.

6.1 Model-IT Overview

The Model-IT prototype is a ConcurTaskTree editor that includes support for building task models using Interaction Templates. The prototype includes a direct manipulation interface for editing ConcurTaskTrees. Model-IT was built using Borland's Delphi. The graphical ConcurTaskTrees representation was implemented in OpenGL using GLScene, an OpenGL library for Delphi.

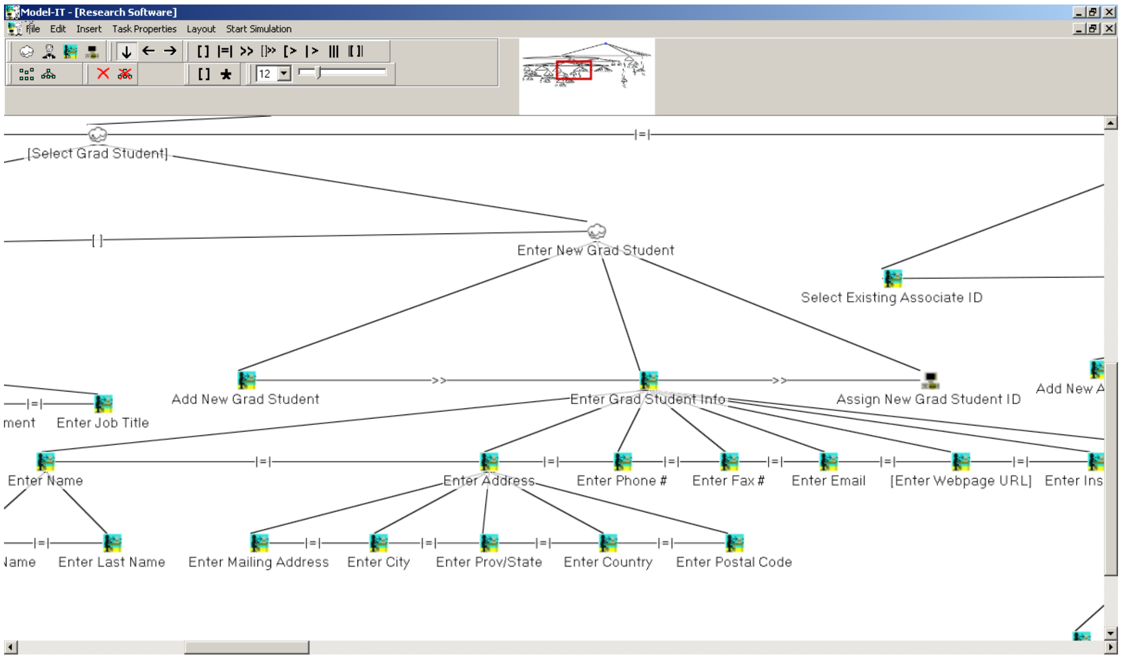


Figure 6.1: Model-IT, the prototype task model editor with Interaction Template support

The Model-IT prototype allows users to add and delete tasks to a ConcurTaskTree. Tasks can be added either as children or as siblings of the currently selected task. A task's properties can be modified using the "Task Properties" menu, or by clicking the controls on the toolbar. Task properties include a task's name, category, temporal operator, and unary operators. A slider allows the user to zoom in and out of the task model, while a small overview shows the current context within the overall model. The "Layout" menu can be used to layout the ConcurTaskTree nicely. The Model-IT prototype is shown in Figure 6.1.

The Model-IT prototype provides users with the ability to insert Interaction

Templates into a task model. Also, the prototype provides two task model simulation modes: basic and enhanced. We discuss these features in the following sections.

6.2 Interaction Templates

The Interaction Template Repository is a collection of Interaction Templates that is made available to users of the Model-IT prototype. The repository is simply a folder containing Interaction Template definition files. Interaction Templates can be added or removed from the repository by adding and removing files in the respository folder. When Model-IT is started, the files are loaded and the names of the templates are listed in the "Insert Template" menu as shown in Figure 6.2.

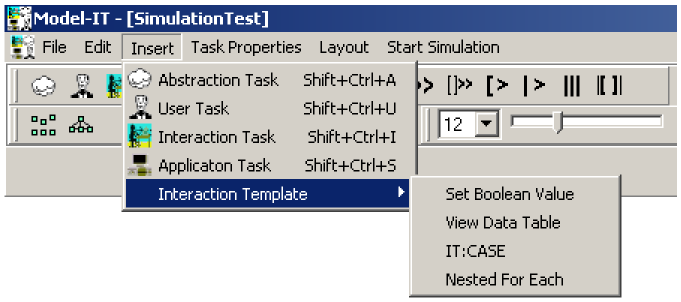
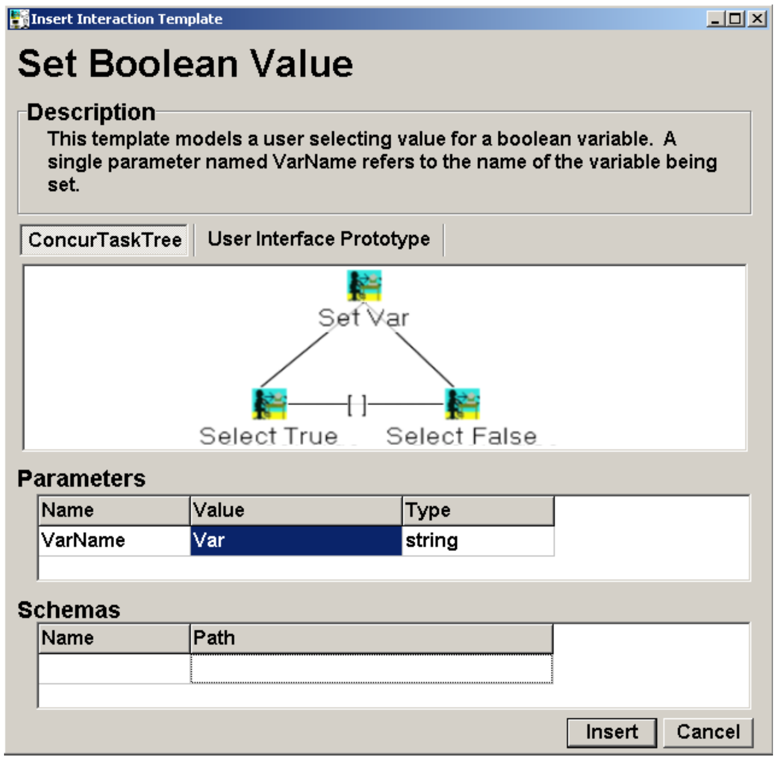


Figure 6.2: Interaction Template Repository in Model-IT

To insert an Interaction Template into the task tree, the user first selects a task, then selects an Interaction Template from the repository. When a template is selected, the "Insert Interaction Template" window, as shown in Figure 6.3, is displayed. The "Insert Interaction Template" window displays information about the selected template. The template's name and description are shown at the top of the window. In the middle of the window, an example of the template's resulting task tree is shown. A sample of the generated user interface prototype for the template can be seen by clicking the "User Interface Prototype" tab.

At the bottom of the "Insert Interaction Template" window, the user can set values for the template's parameters and schemas. Once the required parameters and schemas have been set, the user can click "Insert" to add the template to the ConcurTaskTree. At this point, the template is interpreted by Model-IT, and the resulting task tree is added to the task that was initially selected. Inserting an Interaction Template into a ConcurTaskTree is not much different than inserting a task. The only difference is that some parameter values and schema paths must be set. Subtrees that were added using an Interaction Template are identified by a small "IT" icon to the left of the task's icon.



This template models a user selecting value for a boolean variable. A single parameter named VarName refers to the name of the variable being set.

Name	Value	Type
VarName	Var	string

Name	Path

Figure 6.3: Inserting an Interaction Template using Model-IT

6.3 Task Model Simulation

The Model-IT prototype includes two task model simulation modes: basic and enhanced. Basic simulation provides similar functionality as is provided in CTTE [35]. With enhanced simulation, users interact with concrete user interface components to simulate tasks that are part of an Interaction Template.

6.3.1 Basic Simulation

The Model-IT prototype includes a basic task model simulator, shown in Figure 6.4. The basic simulator implements the functionality discussed in Section 2.1.1. Enabled tasks are calculated using the task model semantics defined in Section 4.3.

The currently enabled tasks are shown in two ways. First, enabled tasks are listed in the simulation window. Secondly, enabled tasks are highlighted with a green checkmark in the model view. Performing a task is simulated by either double-clicking a task in the simulation window, or by selecting a task and clicking the "Perform Task" button. When a task is performed, the enabled tasks are updated, and the user can continue to the next task in the simulation.

6.3.2 Enhanced Simulation

We originally introduced the Enhanced Task Model Simulator in [31]. The simulator has since been extended and incorporated into the Model-IT prototype. In the Enhanced Task Model Simulator, user interface prototypes are generated from the

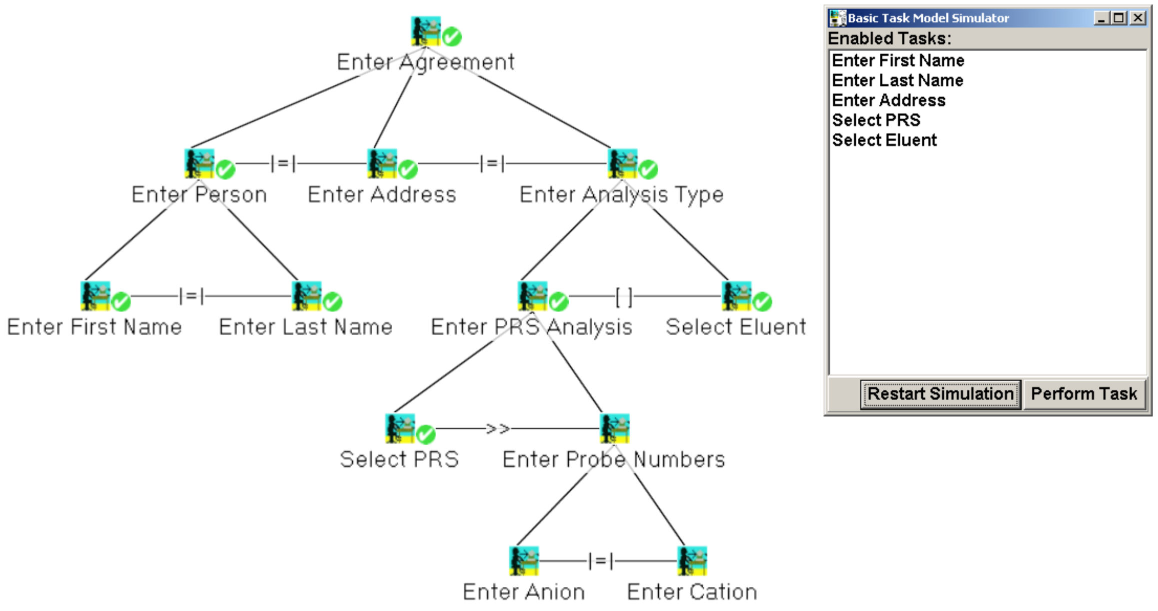


Figure 6.4: Basic task model simulator implemented in the Model-IT Prototype

Interaction Templates that are used in a task model. Users are then able to interact with the generated prototypes to simulate portions of the task model.

Generating User Interface Prototypes

In Section 5.5, we discussed how presentation components are bound to tasks using the Interaction Template Definition Language. This section shows how high-level user interface prototypes can be generated from task models that are built using Interaction Templates.

In this example, we will consider a simple task model built using the "Enter String Value" Interaction Template shown below. The "Enter String Value" template allows users to enter a string value. The template contains a single parameter, "VarName", that refers to the name of the variable being entered. The template results in a single task named "Enter VarName".

```
<it:template name="Enter String Value"
   xmlns:it="http://www.cs.usask.ca/ns/it"
   xmlns:ctt="http://www.cs.usask.ca/ns/ctt">

  <parameter name="VarName" type="string" value="var" />

  <Task ID="Enter $VarName" Category="Interaction" Iterative="False" Optional="$Optional">
    <it:component name="Enter$VarName" objectname="TLabeledEdit"/>
    <it:setproperty component="Enter$VarName" property="EditLabel.Caption" value="$VarName"/>
    <it:bindcomponent component="Enter$VarName"/>
    <it:bindevent component="Enter$VarName" event="OnExit"> </it:bindevent>
  </Task>

</it:template>
```

In this example, we will insert the “Enter String Value” Interaction Template twice as a child of the task model’s root task. The first time, we will set the “VarName” parameter to “Name”. The second time, we will set the “VarName” parameter to “Email”. Finally, we will set the temporal operator of the resulting “Enter Name” task to “Enabling”. The resulting task tree, shown in Figure 6.5, specifies that we first must perform the “Enter Name” task, then the “Enter Email” task.

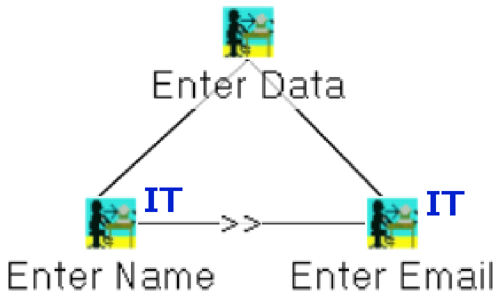


Figure 6.5: ConcurTaskTree resulting from two instantiations of the “Enter String Value” Interaction Template

Before we begin describing how user interface prototypes are generated, we will give a brief introduction to the Delphi naming conventions that we used. The Delphi environment uses the Object Pascal language. In Object Pascal, objects are grouped in files called units. By convention, unit names begin with a “u”, and object names begin with a “T”. In the Delphi environment, forms and presentation components are all objects. To distinguish between form objects and component objects, form object names begin with “Tfrm”.

To generate a user interface prototype from a task model, we begin by selecting all of the subtrees that were inserted using Interaction Templates. In this example, we select the “Enter Name” and “Enter Email” tasks.

Let us start with the “Enter Name” task. First, we retrieve all of the component declarations in the subtree. For the “Enter Name” task, a single component declaration is selected. The component’s name is “EnterName” and the component’s objectname is “TLabeledEdit”. For each of the component declarations, a new unit is generate. Each new unit contains a single object that inherits from the ITPrototype object. Prototype behaviour such as communication with the Enhanced Task Model Simulator is inherited from the ITPrototype object. We name the new unit “uEnterName”, corresponding to the name of the component, and we save the unit to a file called “uEnterName.pas”. Spaces must be removed from component names, since unit and variable names cannot contain spaces. We name the object we are creating by appending ‘Tfrm’ to the component name, in this case “TfrmEnterName”. A private variable is declared. The variable’s type is defined by the component’s objectname attribute, and the variable is named according to the component’s name attribute. In this case, the variable is named “EnterName” of type “TLabeledEdit”.

Inside the object’s constructor, the “EnterName” variable is initialized. First, the object is created, and its parent is set to self. Next, any `<it:setproperty>` elements for the current component are selected using Expression 6.1. In this example, the following `<it:setproperty>` element is selected.

```
<it:setproperty component="EnterName" property="EditLabel.Caption" value="Name"/>
```

Each `<it:setproperty>` element results in a single line of code that is added to the constructor. The line of code sets the specified property to the specified value.

Finally, the component is bound to tasks through its event structure by generating event handlers. All the `<it:bindevent>` elements for the current component are selected using Expression 6.2. In this example, we select the `<it:bindevent>` element shown below.

```
<it:bindevent component="EnterName" event="OnExit"> </it:bindevent>
```

A private procedure called "ComponentOnExit" is declared. In the procedure's implementation, a variable named "sTaskName" of type string is declared. The value of this variable is then set to the current task. If custom code is specified in the `<it:bindevent>` element's `<it:taskname>` element, that code is inserted in place of the code setting the "sTaskName" variable to the current task's name. Finally, "PerformTask" is called to send a message to the Enhanced Task Model Simulator. The "PerformTask" procedure is inherited from the ITPrototype object.

$$\text{//it:setproperty[@component=@name]} \tag{6.1}$$

$$\text{//it:bindevent[@component=@name]} \tag{6.2}$$

The code resulting from the "Enter Name" Interaction Template instantiation is shown below. Following the same prototype generation method, a similar unit would also be generated for the "Enter Email" Interaction Template instantiation.

```
unit uEnterName;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, InteractionTemplatePrototype;

type
  TfrmEnterName = class(ITPrototype)
  private
    EnterName : TLabeledEdit;
    procedure ComponentOnExit(Sender : TObject);
  public
    constructor Create(AOwner : TComponent);override;
  end;

var
  frmEnterName: TfrmEnterName;

implementation

{$R *.dfm}

procedure TfrmEnterName.ComponentOnExit(Sender: TObject);
var
  sTaskName : String;
begin
  sTaskName := ‘Enter Name’;

  PerformTask(sTaskName);
end;

constructor TfrmEnterName.Create(AOwner: TComponent);
begin
  inherited;
  EnterName := TLabeledEdit.Create(Self);
  EnterName.Parent := Self;

  EnterName.EditLabel.Caption := ’Name’;

  EnterName.OnExit := ComponentOnExit;

  EnterName.AutoSize := True;

end;

end.
```

Once all of the required units have been generated, the following Delphi project file is generated to link all the prototype elements together.

```
program UIPrototype;

uses
  Forms,
  Prototype in ‘Prototype.pas’,
  InteractionTemplatePrototype in ‘InteractionTemplatePrototype.pas’,
  EnterName in ‘EnterName.pas’,
  EnterEmail in ‘EnterEmail.pas’,
  SocketConnection in ‘SocketConnection.pas’;

{$R *.res}
```

```
begin
  Application.Initialize;
  Application.CreateForm(TfrmPrototype, frmPrototype);
  Application.CreateForm(TfrmEnterName, frmEnterName);
  Application.CreateForm(TfrmEnterEmail, frmEnterName);
  Application.Run;
end.
```

The project is then compiled, resulting in an executable application. The resulting application is a user interface prototype that can communicate with the Enhanced Task Model Simulator. The prototype can send messages to the simulator, telling the simulator when to perform specific tasks. Additionally, the simulator can send messages to the prototype, telling the prototype when to show and hide specific components of the prototype.

Enhanced Task Model Simulator

Once a prototype has been generated, as discussed in the previous section, the Enhanced Task Model Simulator begins simulation in the same way as the basic simulator. However, whenever a task is enabled, disabled, or completed, the simulator checks to see if a component is bound to that task. Components are bound using the `<it:bindcomponent>` element of the ITDL. If the task is being enabled and a component is bound to that task, the simulator sends a message to the prototype telling that component to show itself. If the task is being disabled or completed, the simulator sends a message to the prototype telling that component to hide itself. The current state of the prototype is dictated by the current state of the task model simulation. That is, the components that are currently available to the user depend on the currently enabled tasks in the task model.

In the ETMS, users can simulate performing tasks in the same way as in the basic simulator. Alternatively, the user can interact with any of the components that are currently visible in the prototype. When the user interacts with the prototype, the generated event code in the prototype will send "PerformTask" messages back to the simulator. When the simulator receives a "PerformTask" message, the specified task is completed, and the enabled tasks will be updated.

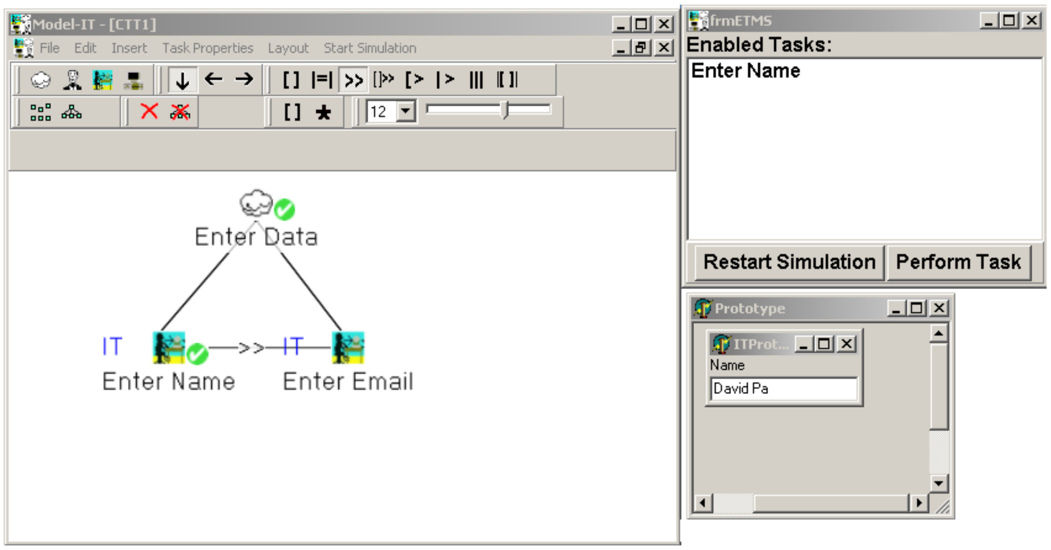


Figure 6.6: A prototype of the Enhanced Task Model Simulator

Figure 6.6 shows Model-IT with the early example loaded. The ETMS is running, along with the generated prototype. In the prototype window, we can see how the user has started to enter a name into a concrete user interface component. When the user is finished entering the name, the "Enter Name" task will be performed in the simulator, and the "Enter Email" task will be enabled. When the "Enter Email" task is enabled, a message will be sent to the prototype, and the "EnterEmail" component will be shown.

While other task model simulators use abstract interface objects to simulate tasks, concrete user interface components can be used to simulate Interaction Templates that have been inserted into ConcurTaskTrees. Using the Enhanced Task Model Simulator, users can interact with concrete user interface components to simulate portions of a larger task model.

6.4 Prototype Limitations

The Model-IT prototype provides support for the Interaction Template approach, but does have some limitations. The current version of the Model-IT prototype supports a subset of the Interaction Template Definition Language. In particular, Interaction Template variables and nested Interaction Template loops are not supported. More time and research would allow for a more sophisticated ITDL interpreter that supports the complete language. We believe the level of Interaction Template support currently implemented is sufficient to demonstrate the advantages of the approach.

6.5 Summary

In this Chapter, we introduced Model-IT, a prototype of a task model editor and simulator that supports the Interaction Template approach. The prototype shows how task models can be built using different Interaction Templates. We showed how user interface prototypes can be generated from the Interaction Templates that were used in a task model. We also showed how a task model can be simulated using the genreated user interface prototypes. We showed, by example, how a user interface prototype can be generated, and we described how communication can occur between the task model simulator and the prototype.

In Chapter 7, we will illustrate our approach with three examples of complex Interaction Templates. The examples will be followed by an evaluation of the approach involving a change scenario from Chapter 3.

Chapter 7

Interaction Template Examples

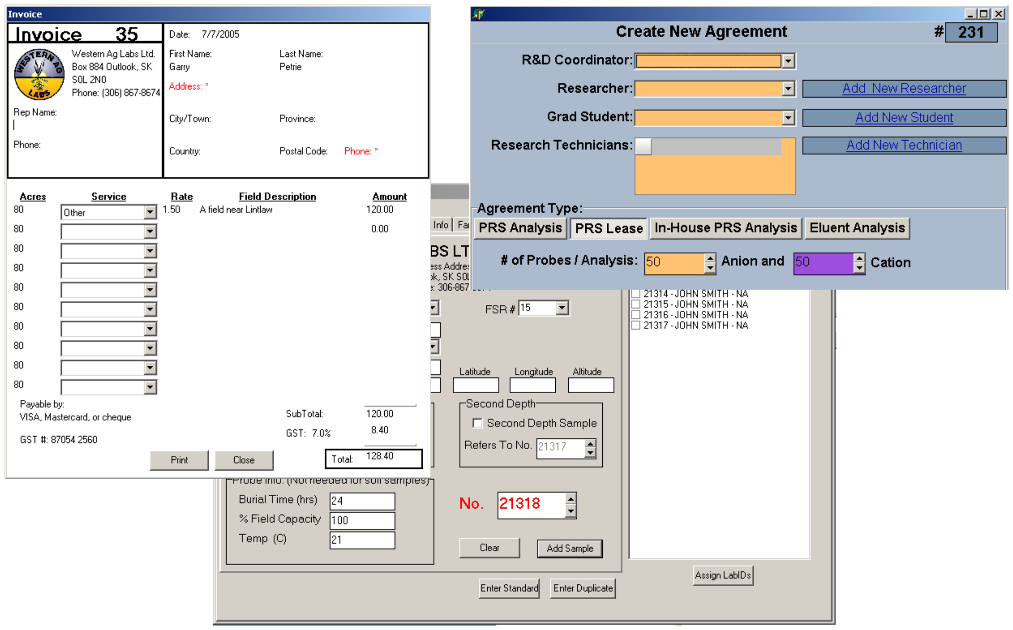
In Chapters 4 and 5, we introduced the Interaction Template approach to task modelling, and a notation for specifying Interaction Templates. In Chapter 6, we presented a prototype of a task model editor that includes support for the Interaction Template approach.

In this Chapter, we illustrate our approach by introducing three Interaction Template examples. First, the examples discuss the user interface interaction that each template is designed to encapsulate. Next, each Interaction Template is specified using the Interaction Template Definition Language. We also show how presentation components are bound to the Interaction Templates using the ITDL.

We have taken these examples from three common types of interface interactions found in the information systems studied at Western Ag: a data entry form, a data access form, and a dialog box. Throughout the examples, we discuss how each Interaction Template is able to support co-evolution by adapting a task model to specific changes. We conclude this Chapter with a look back at a change scenario from Chapter 3. We use a specific change scenario to see how the Interaction Templates we have introduced can help us to co-evolve information systems and business processes.

7.1 Data Entry

Data entry is a very common task in information systems. Figure 7.1 shows several data entry forms found in the custom built information systems used at Western Ag.


Invoice

Invoice 35

Western Ag Labs Ltd.
Box 884 Outlook, SK
S0L 2N0
Phone: (306) 867-8674

Rep Name:

Phone:

Date: 7/7/2005

First Name: Garry

Last Name: Petrie

Address: *

City/Town:

Province:

Country:

Postal Code:

Phone: *

Acres	Service	Rate	Field Description	Amount
80	Other	1.50	A field near Lintlaw	120.00
80				0.00
80				
80				
80				
80				
80				
80				
80				
80				

Payable by:
VISA, Mastercard, or cheque

GST #: 87054 2560

SubTotal: 120.00
GST: 7.0% 8.40
Total: 128.40

Create New Agreement # 231

R&D Coordinator:

Researcher:

Grad Student:

Research Technicians:

Agreement Type:

# of Probes / Analysis: 50 Anion and 50 Cation

FSR # 15

Latitude

Longitude

Altitude

Second Depth

Second Depth Sample

Refers To No. 21317

Probe Info: (Not needed for soil samples)

Burial Time (hrs) 24

% Field Capacity 100

Temp (C) 21

No. 21318

21314 - JOHN SMITH - NA
21315 - JOHN SMITH - NA
21316 - JOHN SMITH - NA
21317 - JOHN SMITH - NA

Figure 7.1: Examples of data entry forms found in custom built software at Western Ag

The following example shows how an Interaction Template can be designed to create a ConcurTaskTree from an XML Schema defining the data that must be entered. The tedious work of repeatedly modelling data entry tasks can be greatly reduced by using the Data Entry Interaction Template. The template can also help to ensure consistency among different data entry tasks within a system or even across several systems within an organization.

This example also shows how an Interaction Template can help to more quickly adapt a system in response to some business process changes. In Chapter 3, we saw several different change scenarios that resulted in changes to the structure of data elements. In particular, CS1 - Sample Priorities, CS3 - Remote Processing Labs, CS4 - Sample Batching and CS5 - Second Depth Samples all resulted in changes to existing data elements. This template's adaptation logic allows the task model to adapt to changes in the structure of a data element. In the above mentioned change scenarios, the changes to existing data elements also had corresponding changes in the system's user interface. The presentation components that are bound to this template's tasks allow us to automatically generate a user interface prototype for the data entry task. The generated prototypes help show how the system's user interface will change according to the change in the structure of a data element.

The Data Entry Interaction Template is named "Enter Data Element". The template requires a single schema named "DataElement", and contains no parameters. The schema is an XML Schema description of the data element that must be entered. The template's root task contains a reference to the data element's name, allowing the task to be identifiably associated with the data element assigned to the template. For example, if the data element is named "Person", then the template's root task would be named "Enter Person".

There is no presentation component that implements the "Enter Data Element" template's root task. Instead, several different presentation components implement each of the root task's children. Therefore, in this Interaction Template, components are declared at the same time as they are bound. This allows the template to dynamically select the set of presentation components that implement the current data entry task. The presentation component declarations and bindings will be discussed as they appear in the template definition.

The root task's children are generated from the schema using two Interaction Template loops. The first loop adds a child task for each of the data element's attributes, denoted by `<xs:attribute>` in the schema. The second loop adds a child for each of the data element's child elements. As was discussed in Section 5.1, loops are implemented using the `<it:foreach>` element of the Interaction Template Definition Language. The details of the template's two loops are discussed in the following sections. A complete implementation of the "Enter Data Element" Interaction Template can be found in Appendix C.

7.1.1 Entering Attributes

Data elements described by XML Schema can contain a number of attributes. In XML, attributes are the nodes that appear within an element's opening tag. For example, the element "`<person name="Donald" age="57"></person>`" contains two attribute nodes: name and age. In this example, the name attribute is a string, and the age attribute is an integer. The XML Schema defining the person element is as follows:

```
<xs:element name="person">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="age" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

The first loop in the "Enter Data Element" template adds a task for each of the data element's attributes. We will refer to this loop as the attribute loop. The list of attribute definitions is selected by referencing the XML Schema with the XPath expression shown in Expression 7.1.

$$\mathsf{\$DataElement/xs{:}complexType/xs{:}attribute} \tag{7.1}$$

If the data element contains no attributes, then Expression 7.1 does not return any XML nodes, and no tasks are added to the tree. The structure of the task that is added for each of the attributes is specified within the attribute loop. The structure of the task depends on the attribute's definition in the XML Schema. An Interaction Template case statement is used to select the appropriate task structure. The case statement, containing two conditions, is used to define the task tree's adaptation logic according to the attribute's type.

The first condition tests whether or not the attribute's type is one of the many built-in data types of XML Schema. This test is accomplished using Expression 7.2, which evaluates to true when the current attribute type is a built-in type.

$$\mathsf{expr(starts\text{-}with(\$value/@type,`xs{:}'))} \tag{7.2}$$

When the attribute's type is a built-in data type, a single leaf task is added to the tree. The task's name contains a reference to the attribute's name using Expression 7.3. The task's optional operator is set according to the value of the use attribute in the current attribute's definition using Expression 7.4. The use attribute can be set to either "optional" or "required". When the use attribute is set to "optional", the task's optional operator is set to true. Otherwise, the optional operator is set to false.

$$\mathsf{\$value/@name} \tag{7.3}$$

$$\mathsf{expr(\$value/@use = ``optional")} \tag{7.4}$$

The task's temporal operator is set to Order Independence. The Order Independence operator is used because the XML Schema contains no indication of the order in which the attributes and the element should be entered.

Inside the task, a presentation component is declared and bound to the current task. A "TLabeledEdit" presentation component is declared and named with a reference to the attribute's name using Expression 7.3. A "TLabeledEdit" presentation component is used to enter the attribute. When the "TLabeledEdit" component is exited by pressing the Tab or Enter key, the "OnExit" event is fired. Therefore, the component's "OnExit" event is bound to the current task. The "EditLabel.Caption" property of the "TLabeledEdit" component is set to the attribute's name using Expression 7.3.

The next condition in the attribute loop's case statement handles the case when the attribute is not a built-in data type. The condition's expression is set to true, meaning that it is automatically selected if and only if the previous condition is not selected. In this situation, the attribute is a custom type that is defined somewhere in the current XML Schema file. The XML node that defines the type is selected using Expression 7.5 and the result is assigned to an Interaction Template variable named CurAttribute.

$$/\mathsf{xs{:}schema}/\mathsf{xs{:}simpleType}[@\mathsf{name} = \$\mathsf{value}/@\mathsf{type}] \tag{7.5}$$

The following is an example XML node that might be selected by Expression 7.5.

```
<xs:simpleType name="agreementType">
  <xs:restriction base="xs:string">
      <xs:enumeration value="PRS Analysis"/>
      <xs:enumeration value="PRS Lease"/>
      <xs:enumeration value="In-House PRS Analysis"/>
      <xs:enumeration value="Eluent Analysis"/>
  </xs:restriction>
</xs:simpleType>
```

The current condition contains another Interaction Template case statement that is used to adapt the task model's structure and behaviour according to the node selected by Expression 7.5. The first condition succeeds when the selected type is an enumerated type, such as the one shown above. With enumerated types, the attribute's value is restricted to those defined in the `<xs:enumeration>` elements. In this case, a task is added to the tree with a reference to the name of the attribute, and the optional operator is set in the same manner as it was for built-in data types. Additionally, an Interaction Template loop is used to add a child task to the current attribute's task for each of the enumeration options. The optional and iterative operators of each of the tasks added by the loop are both set to false, and the temporal operator is set to Choice. The task resulting from enumerated type attributes specifies an interaction such that when entering the attribute, the user must select one, and only one, of the possible values defined in the schema. The possible values are selected using Expression 7.6, and each child task's name contains a reference to the current value using Expression 7.7.

$$\text{\$CurAttribute/xs:restriction/xs:enumeration/@value} \tag{7.6}$$

$$\text{\$value} \tag{7.7}$$

When entering an enumerated type attribute, a “TComboBox” component is used. The “TComboBox” component allows the user to select from a set of options. The “TComboBox” implements the interactions modelled by the entire subtree generated above. Therefore, before the Interaction Template loop, a “TComboBox” component is declared and named with a reference to the attribute’s name. The component is then bound to the current task. Finally, the component’s “OnSelect” event is bound to the tasks that specify the options for the current attribute’s value. To retrieve the currently selected value from the component’s state, a single line of custom event code is required. The entire “TComboBox” declaration and binding is as follows:

```
...
<it:component name="Enter_$value/@name" objectname="TComboBox"/>
<it:setproperty component="Enter_$value/@name" property="Items"
              value="$CurAttribute/xs:restriction/xs:enumeration/@value"/>
<it:bindcomponent component="Enter_$value/@name" />
<it:bindevent component="Enter_$value/@name" event="OnSelect">
  <it:taskname>
    TaskName := ‘Select ’ + Self.Text;
  </it:taskname>
</it:bindevent>
...
```

If the selected attribute type is not an enumerated type, then a single leaf task is added in the same way as for built-in data types. Again, a “TEdit” component is declared and bound to the leaf task in the same way as it was for built-in data types.

More adaptation logic could be defined according to the attribute's type definition by adding more conditions to the case statement. For the purposes of this example, we will limit the attribute type adaptation logic to that which has been discussed so far.

7.1.2 Entering Sub-Elements

Data elements described by XML Schema optionally contain a sequence of child elements. In XML, a child element is an element that appears between an element's opening and closing tags. For example, the person element shown below has two child elements: name and age. In this example, the name element is a string, and the age element is an integer.

```
<person>
  <name>Donald</name>
  <age>57</age>
</person>
```

The XML Schema defining the person element above is as follows:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"
              minOccurs="1" maxOccurs="1"/>
      <xs:element name="age" type="xs:integer"
              minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

An Interaction Template loop is used to add a child task to the template's root task for each of the data element's child elements. We refer to this loop as the child element loop. A list of the XML nodes describing the child elements is selected using the XPath expression shown in Expression 7.8. If the data element does not contain a sequence of child elements, Expression 7.8 returns an empty list, and no tasks are added to the tree.

$$\mathsf{\$DataElement/xs{:}complexType/xs{:}sequence/xs{:}element} \tag{7.8}$$

The structure of the task that is added for each child element is specified within the child element loop. The loop contains an Interaction Template case with three conditions. The first condition tests if the element is a built-in data type. The expression used in this condition is the same as the expression used to test if attributes are built-in data types, as shown in Expression 7.2. When the element's type is a built-in data type, Expression 5.2 evaluates to true, and a single leaf task is added to the tree. The task's name contains a reference to the element's name using Expression 7.3. The task's optional operator is set according to the value of the element's minOccurs attribute. If minOccurs is equal to 0, then the task of entering the current child element is optional. The task's optional operator is set using Expression 7.9. Additionally, the task's iterative operator is set according to the value of the element's maxOccurs attribute. If maxOccurs is greater than 1, then the task of entering the current child element is iterative. The task's iterative operator is set using Expression 7.10. The task's temporal operator is set to Order Independence, because the XML Schema contains no indication of the order in which attributes and element should be entered. A "TEdit" component is declared and bound to the task. The component's name is set using a reference to the element's name using Expression 7.3. Finally, the component's "OnExit" event is bound to the task.

$$\mathsf{expr(\ \$value/@minOccurs = 0\ )} \tag{7.9}$$

$$\mathsf{expr(\ \$value/@maxOccurs\ gt\ 1\ )} \tag{7.10}$$

The next two conditions are needed to handle the two cases that can occur when the current child element is a custom data type. A child element can be either a complex element or a simple element. A complex element is an element that contains attributes and/or child elements, while a simple element is an element that contains only text. To simplify the adaptation logic inside this Interaction Template, a variable named “CurElement” is declared. The value of “CurElement” is assigned using Expression 7.11, which selects the current element’s type definition from the XML Schema file.

$$\begin{gathered}\mathsf{/xs{:}schema/xs{:}complexType[@name = \$value/@type]\ |} \\ \mathsf{/xs{:}schema/xs{:}simpleType[@name = \$value/@type]}\end{gathered} \tag{7.11}$$

Expression 7.12 is used to test if the current child element is a complex element. When Expression 7.12 evaluates to true, the template simply reuses itself. The “Enter Data Element” template is used, and the “DataElement” schema is set to the current element. Reusing the “Enter Data Element” template allows detailed task trees to be built for complex data types with little effort. The template is referenced using the following ITDL statement.

```
<it:usetemplate name="Enter Data Element" >
  <it:schema name="DataElement" path="$CurElement" />
</it:usetemplate>
```

$$\mathsf{\$CurElement/name()} = \text{“}\mathsf{xs:complexType}\text{”} \tag{7.12}$$

$$\mathsf{\$CurElement/name()} = \text{“}\mathsf{xs:simpleType}\text{”} \tag{7.13}$$

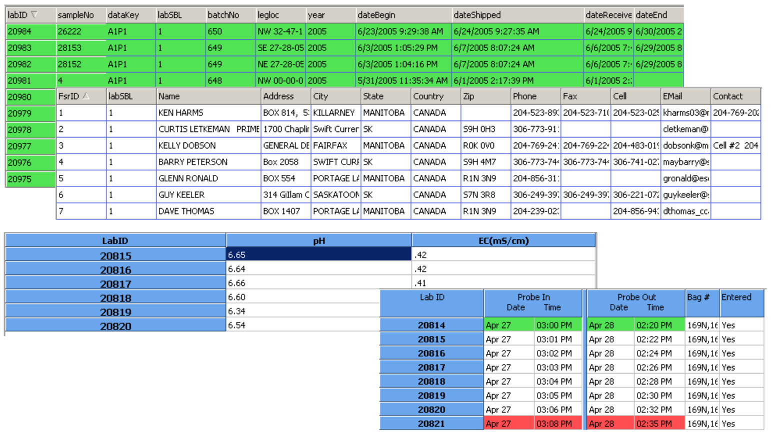
The next condition uses Expression 7.13 to test if the current child element is a simple element. When Expression 7.13 evaluates to true, another Interaction Template case statement is needed to adapt the tree's structure and behaviour according to the type's definition in the XML Schema. This case statement is similar to the case statement used when an attribute's type is a simple type. The first condition succeeds when the current child element is an enumerated type. In this case, a task is added to the tree with a reference to the name of the element. The optional and iterative operators are set in the same manner as they were for built-in data types. Additionally, an Interaction Template loop is used to add a child task to the current element's task for each of the enumeration options. The optional and iterative operators for each of the tasks added by the loop are both set to false, and the temporal operator is set to Choice.

As was the case with enumerated type attributes, a “TComboBox” component is declared and bound to the current element's task. The declaration, task binding, and event binding are all the same as for enumerated type attributes.

If the current element's type is not an enumerated type, then a single leaf task is added in the same way as it was for built-in data types. Again, a “TEdit” component is declared and bound to the task as it was for built-in data types.

7.2 Data Access

In information systems, users often access large amounts of data using data tables. An informal analysis of one particular information system at Western Ag, the Lab Assistant, shows that a large percentage of the tasks in the system involve viewing data in or selecting data from data table components in some manner. Figure 7.2 shows some of the data tables found in the Lab Assistant software used at Western Ag.



labID ▽	sampleNo	dataKey	labSBL	batchNo	legloc	year	dateBegin	dateShipped	dateReceive	dateEnd
20984	26222	A1P1	1	650	NW 32-47-1	2005	6/23/2005 9:29:38 AM	6/24/2005 9:27:35 AM	6/24/2005 9	6/30/2005 2
20983	28153	A1P1	1	649	SE 27-28-05	2005	6/3/2005 1:05:29 PM	6/7/2005 8:07:24 AM	6/6/2005 7:	6/29/2005 8
20982	28152	A1P1	1	649	NE 27-28-05	2005	6/3/2005 1:04:16 PM	6/7/2005 8:07:24 AM	6/6/2005 7:	6/29/2005 8
20981	4	A1P1	1	648	NW 00-00-0	2005	5/31/2005 11:35:34 AM	6/1/2005 2:17:39 PM	6/1/2005 2:	
20980										
20979										
20978										
20977										
20976										
20975										

FsrID △	labSBL	Name	Address	City	State	Country	Zip	Phone	Fax	Cell	EMail	Contact
1	1	KEN HARMS	BOX 814, 5	KILLARNEY	MANITOBA	CANADA		204-523-89	204-523-71	204-523-02	kharms03@	204-769-20
2	1	CURTIS LETKEMAN PRIME	1700 Chapli	Swift Currer	SK	CANADA	S9H 0H3	306-773-91			cletkeman@	
3	1	KELLY DOBSON	GENERAL DE	FAIRFAX	MANITOBA	CANADA	R0K 0V0	204-769-24	204-769-22	204-483-01	dobsonk@m	Cell #2 204
4	1	BARRY PETERSON	Box 2058	SWIFT CURI	SK	CANADA	S9H 4M7	306-773-74	306-773-74	306-741-02	maybarry@	
5	1	GLENN RONALD	BOX 554	PORTAGE L	MANITOBA	CANADA	R1N 3N9	204-856-31			gronald@es	
6	1	GUY KEELER	314 Gillam C	SASKATOON	SK	CANADA	S7N 3R8	306-249-39	306-249-39	306-221-07	guykeeler@	
7	1	DAVE THOMAS	BOX 1407	PORTAGE L	MANITOBA	CANADA	R1N 3N9	204-239-02		204-856-94	dthomas_cc	

LabID	pH	EC(mS/cm)
20815	6.65	.42
20816	6.64	.42
20817	6.66	.41
20818	6.60	
20819	6.34	
20820	6.54	

Lab ID	Probe In		Probe Out		Bag #	Entered
	Date	Time	Date	Time		
20814	Apr 27	03:00 PM	Apr 28	02:20 PM	169N,16	Yes
20815	Apr 27	03:01 PM	Apr 28	02:22 PM	169N,16	Yes
20816	Apr 27	03:02 PM	Apr 28	02:24 PM	169N,16	Yes
20817	Apr 27	03:03 PM	Apr 28	02:26 PM	169N,16	Yes
20818	Apr 27	03:04 PM	Apr 28	02:28 PM	169N,16	Yes
20819	Apr 27	03:05 PM	Apr 28	02:30 PM	169N,16	Yes
20820	Apr 27	03:06 PM	Apr 28	02:32 PM	169N,16	Yes
20821	Apr 27	03:08 PM	Apr 28	02:35 PM	169N,16	Yes

Figure 7.2: Examples of data tables from the Lab Assistant software used at Western Ag

The following example shows how an Interaction Template can be designed to encapsulate the common task of viewing and interacting with data in a data table component. The template includes parameters that modify the structure and behaviour of the resulting task tree. The parameters also modify the behaviour of the Interaction Template's presentation component. The resulting task tree includes points specifically intended for manually adding tasks associated with the data in the table, allowing the template's user to customize the template for specific uses. The template also allows the task tree to be easily adapted to changes in the structure of data elements.

The Data Table Interaction Template is named “View Data Table”. The template requires a single schema called “DataElement”. The “DataElement” schema must be set to a data element that contains attributes and/or a sequence of child elements. Since the template is using the “DataElement” to setup the data table’s columns, it is important that the input data element only contains simple elements as children. Any complex elements are treated as simple elements, which may result in the loss of expected columns. Several behaviour modifying parameters are defined in this template. Each parameter is discussed as it is referenced in the template’s definition.

The “View Data Table” template root task contains a reference to the data element’s name, allowing the task to be identifiably associated with the data element assigned to the template. For example, if the data element is named “Person”, then the template’s root task is named “View Person Table”.

The template’s root task contains a number of child tasks that refer to the different interactions offered by the template. The three child tasks are called “Modify View”, “Modify Data”, and “Select Data”. Each of these child tasks will be discussed in detail in the following subsections.

The “View Data Table” template contains several Interaction Template loops that are used to repeat a task for each of the columns in the data table. Expression 7.14 is used to select all the attributes and child elements of the specified data element. The resulting list of nodes is assigned to an Interaction Template variable named “Columns”. The “Columns” variable is used by the Interaction Template loops found in the templates definition.

$$\begin{gathered}\mathsf{\$DataElement/xs{:}complexType/xs{:}attribute} \mid \\ \mathsf{\$DataElement/xs{:}complexType/xs{:}sequence/xs{:}element}\end{gathered} \tag{7.14}$$

7.2.1 Modify View

The “Modify View” task includes child tasks that model the interactions in which the user modifies how the data is displayed in the data table. Such interactions include sorting data and moving columns.

Interaction Template parameters are used to adapt the behaviour that is permitted under the “Modify View” task. The “AllowSort” parameter, of type it:boolean, specifies whether or not the user can sort the rows in the table according to the values of a specified column. The “AllowSort” parameter’s default value is true. A case statement is used to add the appropriate tasks to the tree when “AllowSort” equals true. Expression 7.15 is used to accomplish this test. When “AllowSort” is true, a task called “Sort By Column” is added to the tree. An Interaction Template loop is then used to add a child task to “Sort By Column” for each of the data element’s attributes and child elements. The added task’s temporal operator’s are set to Choice, and each task contains two child tasks. The first task is an interaction task where the user clicks on the current column to sort the rows, and the second is an application task where the system sorts the rows by the specified column. These two child tasks are related using the Enabling with Information Exchange operator.

$$\text{expr(\$AllowSort = True)} \tag{7.15}$$

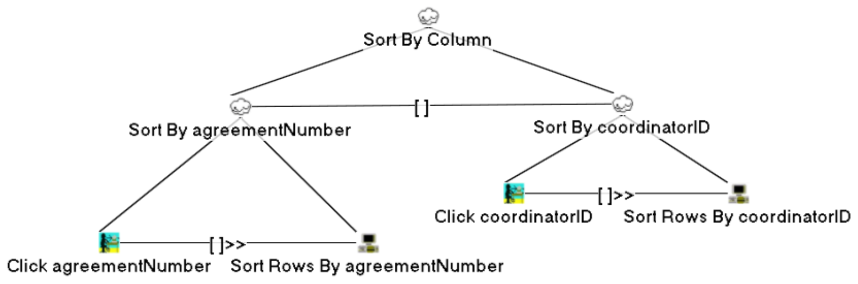


Figure 7.3: Example of a task tree that is added when AllowSort is true

Another view modification task is the "Move Column" task. The "AllowColumnMove" parameter, of type it:boolean, specifies whether or not users are allowed to move columns in the data table. The "AllowColumnMove" parameter's default value is true. A case statement is used to add the appropriate tasks when "AllowColumnMove" is true. Expression 7.16 is used to accomplish this test. When "AllowColumnMove" is true, a task called "Move Column" is added as a child of the "ModifyView" task. Two tasks, "Select Column", and "Select New Location", are added as children of the "Move Column" task. The "Select Column" task is related to the "Select New Location" task using the Enabling with Information Exchange temporal operator. Finally, an Interaction Template loop is used to add a child task to the "Select Column" task for each of the columns in the data table.

$$\text{expr(\$AllowColumnMove = True)} \tag{7.16}$$

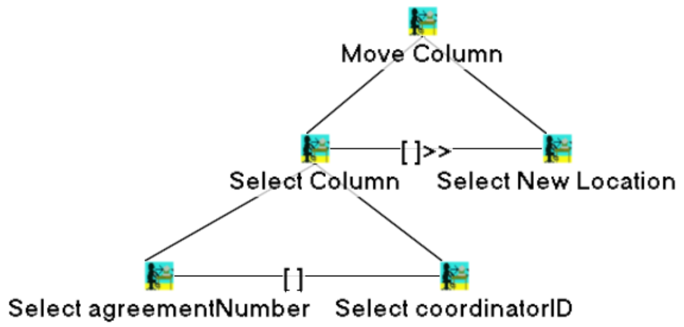


Figure 7.4: Example of a task tree that is added when AllowColumnMove parameter is true

7.2.2 Select Data

The "Select Data" subtask of the "View Data Table" Interaction Template is a task that models the selection of either a single row or a set of rows, and performing an operation on the selected row(s). The resulting tree is structured in a way that allows the developer to easily add different operations that the user can perform on the rows.

First, an Interaction Template case statement is used to modify the structure of the tree depending on whether or not the user is allowed to select multiple rows. The row selection option is specified using the "MultiSelect" Interaction Template parameter. This it:boolean type parameter, whose default value is true, is tested using Expression 7.17.

When "MultiSelect" equals true, two children are added to the "Select Data" task: "Select Rows" and "Perform Operation on Rows". The "Select Rows" task is an iterative interaction task, while "Perform Operation on Rows" is an abstraction task. The "Suspend-Resume" temporal operator is used to define the relationship between these two tasks. The resulting "Select Data" task defines an interaction such that the user can select any number of rows from the table, then perform an operation on those rows. While the operation is being performed, the user can not select any new rows. When the operation is completed, the user is permitted to select rows from the table.

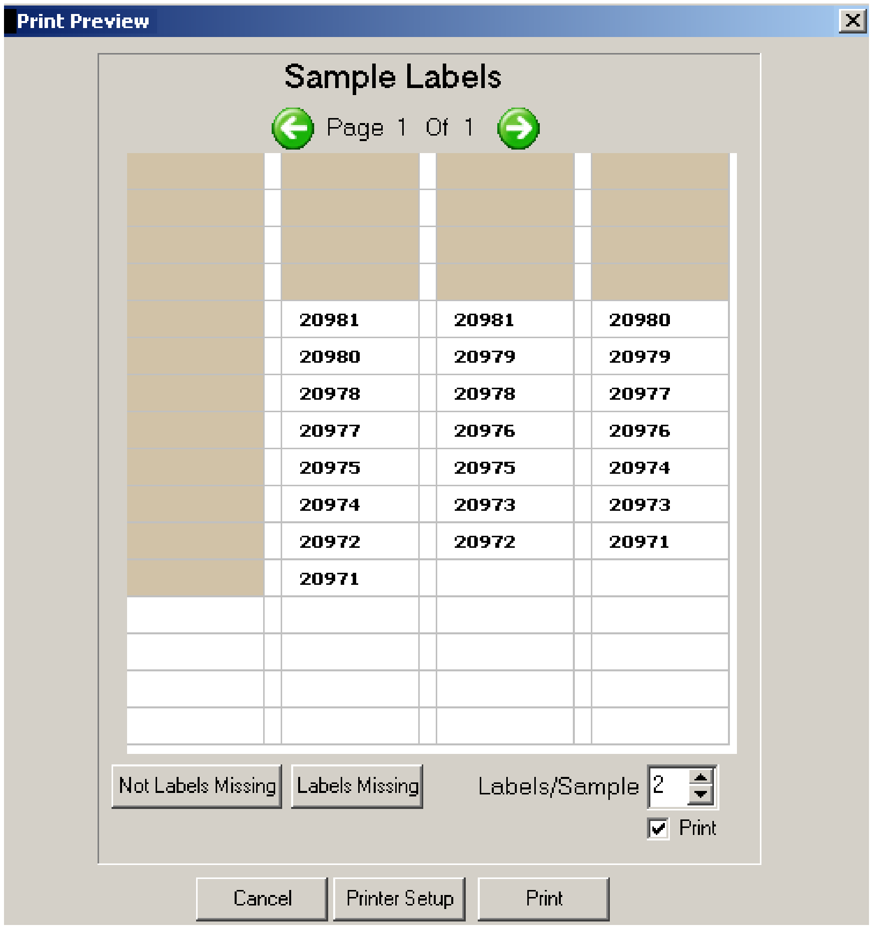
$$\mathsf{expr(\ \$MultiSelect = True\ )} \tag{7.17}$$

When “MultiSelect” equals false, two children are added to the “Select Data” task: “Select Row” and “Perform Operation on Row”. The “Select Row” task is an interaction task, while “Perform Operation on Row” is an abstraction task. In this case, the “Enabling with Information Exchange” temporal operator is used to define the relationship between the two tasks. The resulting “Select Data” task defines an interaction such that the user can select a single row from the table, then perform an operation on that row.

To add tasks to the template, tasks can be added as children of either the “Perform Operation On Row” or the “Perform Operation on Rows” task. The example in Section 7.4 will demonstrate the additions of custom tasks to the “View Data Table” Interaction Template..

7.3 Dialog Box

Dialog boxes are user interface windows that display information and/or request information from the user. Typically, they remain in focus until their function is complete. Dialog boxes are often used to encapsulate common tasks, allowing for reuse of existing code, and promoting consistency across systems. Some common dialog boxes include those used to open and save files, as well as to set printing options when printing documents.



Print Preview

Sample Labels

Page 1 Of 1

▨	▨	▨	▨
▨	▨	▨	▨
▨	▨	▨	▨
▨	▨	▨	▨
▨	20981	20981	20980
▨	20980	20979	20979
▨	20978	20978	20977
▨	20977	20976	20976
▨	20975	20975	20974
▨	20974	20973	20973
▨	20972	20972	20971
▨	20971		

Labels/Sample 2

[✓ Print]

Figure 7.5: A dialog box used in the Lab Manager and Lab Assistant software

Figure 7.5 shows a dialog box, called Print Labels, used in the Lab Manager and Lab Assistant software at Western Ag. The Print Labels dialog box receives a set of data elements from the system, and displays a print preview of the labels. The user is then able to specify several options for printing the labels. The user can set the number of labels per data element. The pages can also be customized to maximize the use of label sheets that are partially used. Specific labels can be set as used or unused directly on the preview, and the dialog box will adjust the print preview and print job accordingly.

The task tree resulting from the “Print Labels” Interaction Template is shown in Figure 7.6. The complete ITDL implementation of the “Print Labels” Interaction Template can be seen in Appendix E.

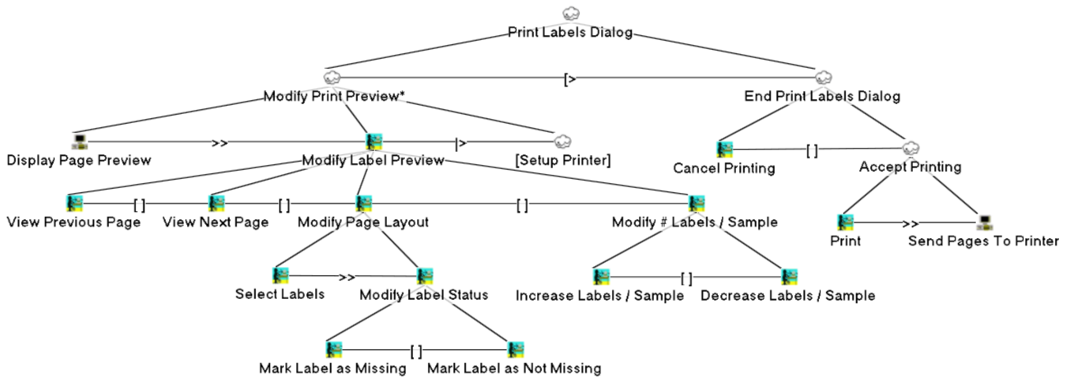


Figure 7.6: ConcurTaskTree resulting from the “Print Labels” Interaction Template

7.4 Using Interaction Templates

Task models can be built to describe information systems using Interaction Templates. In this example, we compose a task model describing a simplification of the Lab Assistant software using the three Interaction Templates we introduced earlier in this Chapter. We consider a simplification of the Lab Assistant that allows users to enter soil samples, view soil samples in a data table, delete soil samples from a data table, post soil sample data to farmers, and print soil sample labels.

The following is a simplified XML Schema of the soil sample data element that is used throughout the Lab Assistant and Lab Manager software at Western Ag.

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="sample">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="sampleNo" type="xs:integer" minOccurs="1" maxOccurs="1"/>
      <xs:element name="year" type="xs:year" minOccurs="1" maxOccurs="1"/>
      <xs:element name="farmerID" type="xs:integer" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The root task of our Lab Assistant task model is named ‘Use Lab Assistant’. The first child of the root task, shown in Figure 7.7, is inserted using the ‘Enter Data Element’ Interaction Template. The root task’s second child, shown in Figure 7.8, is inserted using the ‘View Data Table’ Interaction Template. With both of the templates, the ‘DataElement’ schema is set to the ‘sample’ data element shown above. For the ‘View Data Table’ template, we set the ‘AllowSort’, ‘AllowColumnMove’, and ‘MultiSelect’ parameters to true.

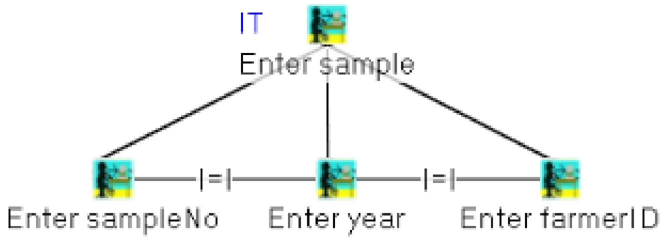


Figure 7.7: ‘Enter Data Element’ Interaction Template used for the soil sample data element

Finally, we add three child tasks to the ‘View Data Table’ template’s ‘Perform Operation On Row’ task: Delete Rows, Post Data to Farmer, and ‘Print sample Labels’. For the purposes of keeping this example simple, the ‘Delete Rows’ and ‘Post Data to Farmer’ tasks are both modelled as abstract tasks and are not modelled

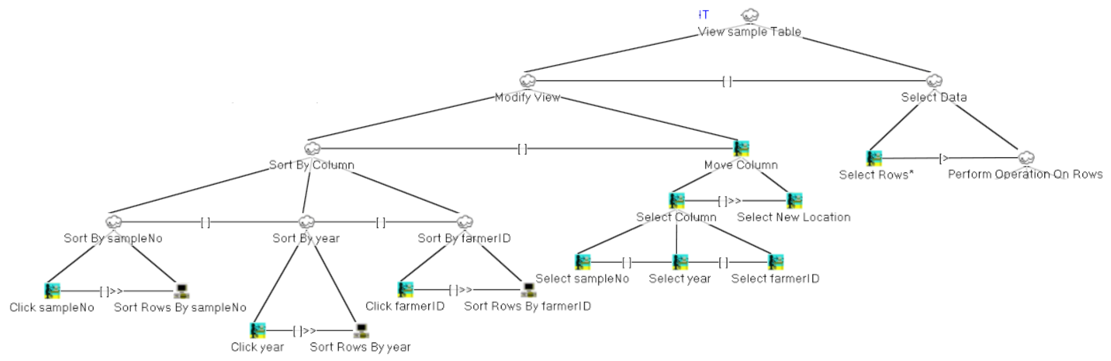


Figure 7.8: 'View Data Table' Interaction Template used for the soil sample data element

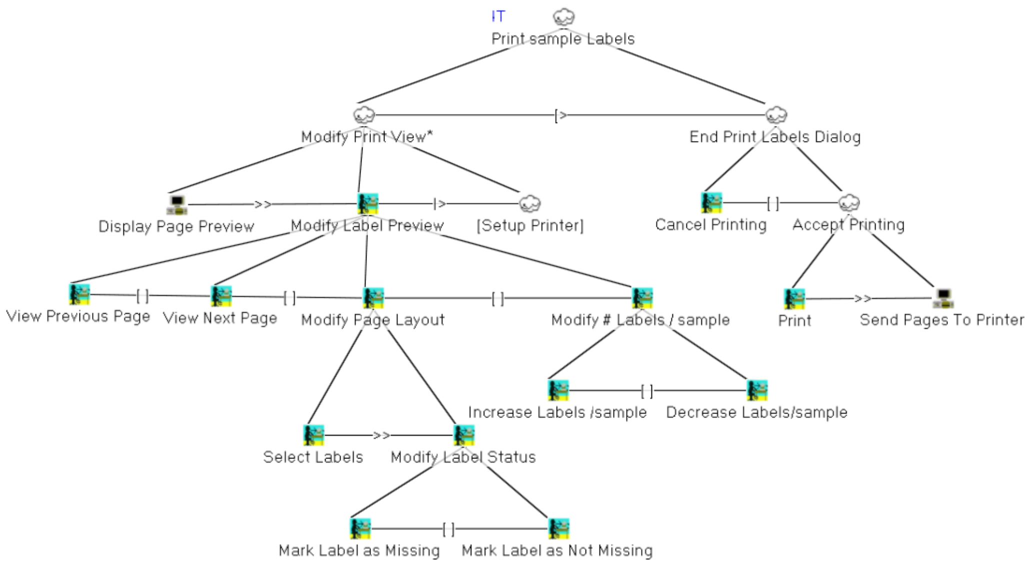


Figure 7.9: 'Print Labels' Interaction Template used for the soil sample data element

in any more detail. The 'Print sample Labels' task, shown in Figure 7.9, is inserted using the 'Print Labels' Interaction Template.

The task model of the Lab Assistant is shown in Figure 7.10. The model contains 48 tasks and was composed using the three Interaction Templates introduced earlier in this Chapter. Of the 48 tasks in the model, only three of the tasks were inserted manually. The task model contains detailed information about the soil sample data element that was bound to the three template instantiations. This example shows us that we are able to model a system at a significant level of detail in relatively few steps as compared to building a task model without Interaction Templates. Using the Interaction Template approach, we have made the task modelling process less tedious during the design phase.

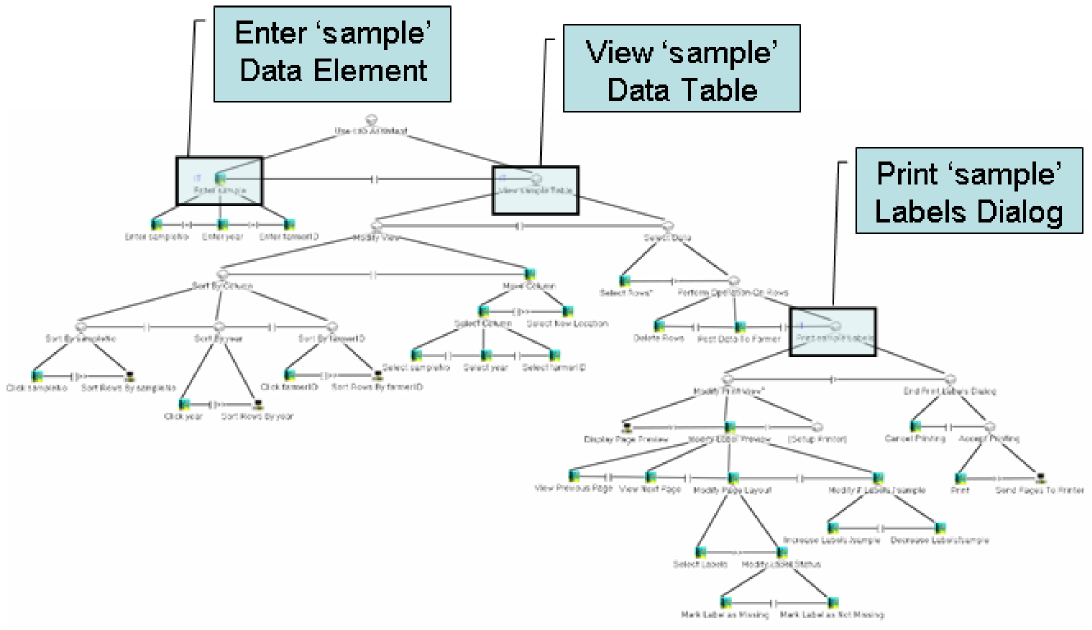
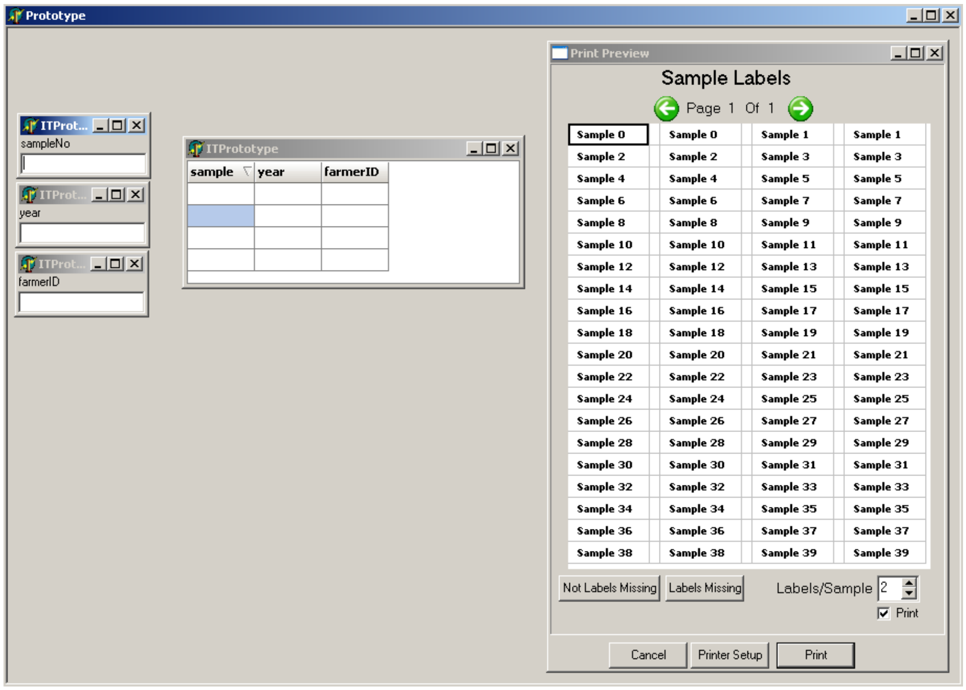


Figure 7.10: The Lab Assistant software modelled using the Interaction Template approach

Using the Enhanced Task Model Simulator from Model-IT, we are able to interact with a user interface prototype of the simple Lab Assistant. The prototype can be seen in Figure 7.11. In the prototype, users are able to enter the details for a new soil sample: the 'sampleNo', 'farmerID', and 'year'. Users are also able to view and interact with a data table that would display the soil samples in the system. The prototype also allows users to interact with the print labels dialog box. As the user interacts with the generated prototype, 'PerformTask' messages are sent to the Enhanced Task Model Simulator in Model-IT. As tasks are enabled and disabled in the task model simulator, messages controlling which interfaces components are currently enabled are sent to the prototype.



Sample Labels

Sample 0	Sample 0	Sample 1	Sample 1
Sample 2	Sample 2	Sample 3	Sample 3
Sample 4	Sample 4	Sample 5	Sample 5
Sample 6	Sample 6	Sample 7	Sample 7
Sample 8	Sample 8	Sample 9	Sample 9
Sample 10	Sample 10	Sample 11	Sample 11
Sample 12	Sample 12	Sample 13	Sample 13
Sample 14	Sample 14	Sample 15	Sample 15
Sample 16	Sample 16	Sample 17	Sample 17
Sample 18	Sample 18	Sample 19	Sample 19
Sample 20	Sample 20	Sample 21	Sample 21
Sample 22	Sample 22	Sample 23	Sample 23
Sample 24	Sample 24	Sample 25	Sample 25
Sample 26	Sample 26	Sample 27	Sample 27
Sample 28	Sample 28	Sample 29	Sample 29
Sample 30	Sample 30	Sample 31	Sample 31
Sample 32	Sample 32	Sample 33	Sample 33
Sample 34	Sample 34	Sample 35	Sample 35
Sample 36	Sample 36	Sample 37	Sample 37
Sample 38	Sample 38	Sample 39	Sample 39

Labels/Sample 2
[✓ Print]

Figure 7.11: A generated user interface prototype for the simplified Lab Assistant software

Using the Interaction Template approach to modelling the Lab Assistant software, we are able to build a task model in fewer steps than if we were building the task model without using Interaction Templates. We are also able to generate user interface prototypes that contain concrete user interface components. Furthermore, users are able to perform tasks during task model simulation by interacting with the user interface prototype.

7.4.1 CS1 Revisited

Let us reconsider the first change scenario from Chapter 3. Recall that in CS1 - Sample Priorities, the lab implemented a method of prioritizing samples as "3

Day", "7 Day", and "No Rush". This change scenario resulted in minor changes in data, interface, and processing. In the data, a "priority" element was added to the "sample" data element. In this section, we show how we quickly adapt the system's task model and user interface prototype to this change in data.

First, we must model the change in data that has occurred as a result of this change scenario. The following schema shows the addition of sample priorities in the soil sample data schema.

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:simpleType name="priority">
   <xs:restriction base="xs:string">
    <xs:enumeration value="3 Day"/>
    <xs:enumeration value="7 Day"/>
    <xs:enumeration value="No Rush"/>
 </xs:restriction>
</xs:simpleType>


<xs:element name="sample">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="sampleNo" type="xs:integer" minOccurs="1" maxOccurs="1"/>
      <xs:element name="year" type="xs:year" minOccurs="1" maxOccurs="1"/>
      <xs:element name="farmerID" type="xs:integer" minOccurs="1" maxOccurs="1"/>
    <xs:element name="priority" type="priority" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

From the simple change to the XML Schema shown above, we can model the changes to the Lab Assistant by simply applying the schema changes to the Interaction Templates in the task model. Figures 7.12 and 7.13 show the resulting changes in the Lab Assistant task model. The Interaction Templates are able to adapt to the changes in the data that resulted from CS1 - Sample Priorities.

Furthermore, the generated user interface prototype is also able to adapt to the changes that resulted from CS1 - Sample Priorities. Figure 7.14, shows the

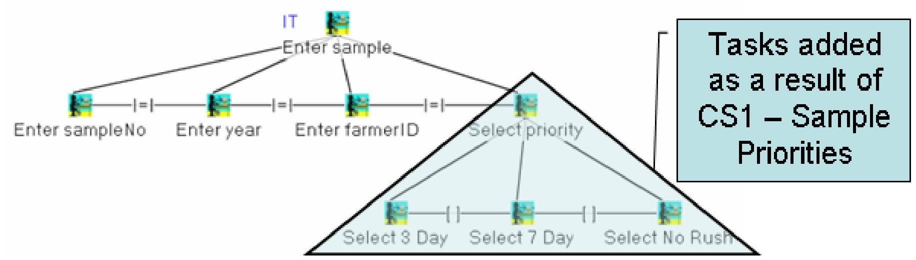


Figure 7.12: 'Enter Data Element' Interaction Template after CS1

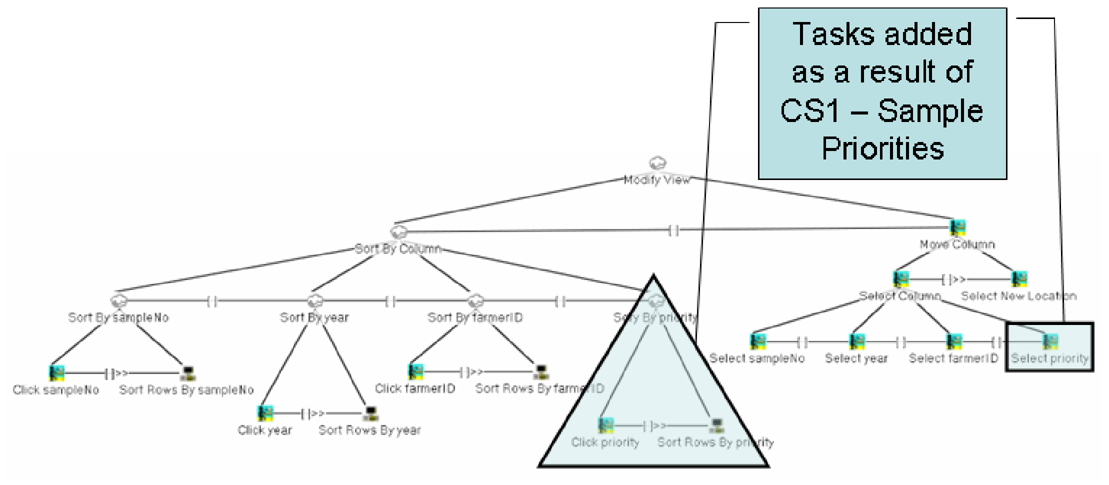


Figure 7.13: 'View Data Table' Interaction Template after CS1

Lab Assistant prototype that was generated after the changes are made to the task model. In the prototype, a combo-box for selecting the sample's priority is added. A 'priority' column is also added to the data table component in the prototype. The additions to the generated prototype allow both the system designers and the users to see how the Lab Assistant's user interface should change as a result of CS1 - Sample Priorities.

Using the Interaction Template approach to task modelling, we are able to adapt the Lab Assistant's task model by modifying the data model that is bound to the task model. We are also able to generate a prototype of the Lab Assistant's user

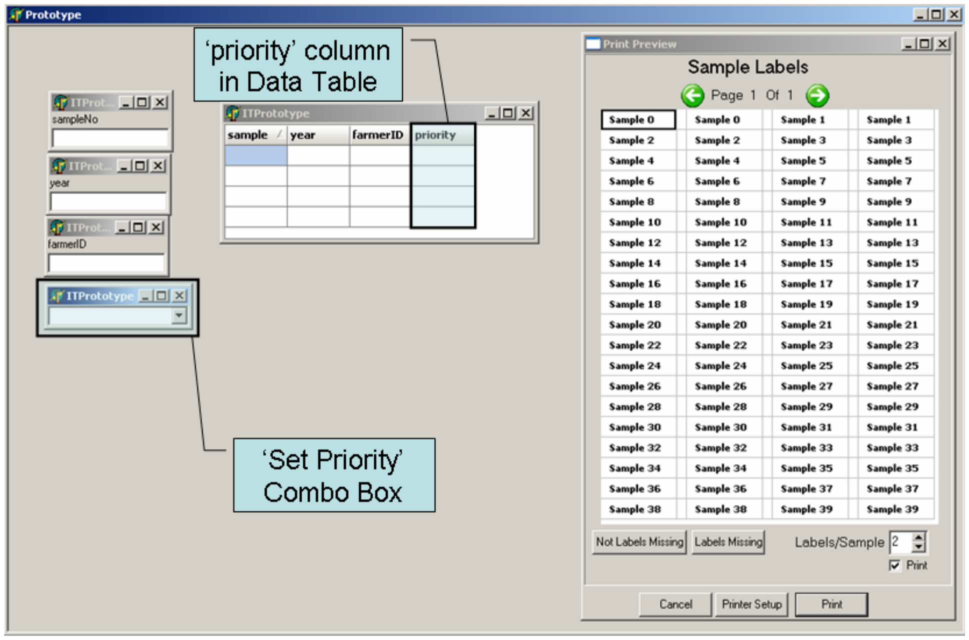


Figure 7.14: A generated user interface prototype for the simplified Lab Assistant software showing the change in the user interface that resulted from CS1 - Sample Priorities

interface using the concrete user interface components that are bound to the task model through Interaction Templates. Using our approach, designers and users can make changes to an existing system's task model, and interact with user interface prototypes to validate changes. The binding of a data model to the task model helps us to make change to a system's task model, while the binding of concrete user interface components to the task model allows us to generate user interface prototypes. Also, the encapsulation of possible changes into the adaptation logic of Interaction Templates improves the task modelling process by making it less tedious in both the design phase and evolution phase of the software development lifecycle.

7.5 Summary

In this Chapter, we illustrated our approach with three examples of complex Interaction Templates. We showed how each template is specified using the Interaction Template Definition Language, and discussed how each template is able to adapt to specific types of changes in the structure of data elements and input parameters.

We modelled a simplification of the Lab Assistant software using the three Interaction Templates we defined earlier in the Chapter. Finally, we revisited CS1 - Sample Priorities from Chapter 3 to show how the Interaction Template approach supports the co-evolution of information systems and business processes.

In Chapter 8, we will conclude our research with a summary of our motivation and contributions as well as an overview of directions for future work.

Chapter 8

Conclusion

This research focused on improving task modelling to support the co-evolution of information systems and business processes. In this Chapter, we summarize our motivation and contributions. We also outline areas for future work, including improving our user interface prototypes by incorporating sketching and layout techniques, supporting multiple user task models, and performing an in-depth field study using the Interaction Template approach.

8.1 Summary

Businesses use information systems to support or automate many parts of their organization's business processes. Some small businesses, particularly knowledge-based and riskier innovative businesses, have business processes that are constantly evolving. As business processes evolve, supporting information systems must also evolve in order to meet new requirements. As well, the introduction of a new information system may result in changes to an organization's business processes. We refer to this process as co-evolution: the process of reciprocal change in a software system and the activities and goals of the system's users.

Our research explored how we can improve the task modelling process to support the co-evolution of information systems and business processes. In particular, we investigated how building task models using adaptable task templates and binding presentation components to tasks would improve the task modelling process to support co-evolution. This research has resulted in the following contributions:

- a rigorous definition of Interaction Templates,
- a technique for building task models using Interaction Templates,
- a technique for composing user interface prototypes for task model simulation,
- a description of the semantics of task model simulation,
- a prototype system for building task models using Interaction Templates, and
- a prototype system for task model simulation using concrete user interface components.

In Chapter 2 - Related Work, we reviewed research in task modelling, software evolution, program families, dynamic systems, and data modelling. In Chapter 3 - Changing Business Processes, a case study of the seven year evolution of a real world information system provided insight into how changing business processes can affect supporting information systems. The case study resulted in concrete change scenarios, illustrating several real world co-evolution points.

In Chapter 4 - Interaction Template Model, we proposed the Interaction Template approach that incorporates data modelling and user interface components into task modelling. Incorporating data modelling allows us to provide task model adaptability with respect to the structure of data elements. Incorporating user interface components supports the automated generation of user interface prototypes. The combination of the three allows us to build task models that can adapt to changes in business processes and generate user interface prototypes for task model simulation. Using user interface prototypes for simulation allows users and developers to see how a system must change in response to a business process change.

In Chapter 4, we also outlined the operational semantics of ConcurTaskTrees. These semantics clearly outline the meaning of temporal and unary operators, as well as the semantics of the hierarchical structure of a task tree. Our outline of the semantics of ConcurTaskTrees helped us to overcome ambiguities and inconsistencies that are present in current literature describing ConcurTaskTree simulation.

In Chapter 5 - Interaction Template Definition Language, we defined an XML based notation for specifying Interaction Templates. In Chapter 6 - Modelling With Interaction Templates, we introduced our prototype system, Model-IT, a task model editor with support for building task models using Interaction Templates. Our prototype also generates simple user interface prototypes from task models, which allows users to perform task model simulation using the generated prototypes.

We demonstrated our approach in Chapter 7 - Interaction Template Examples, where we outlined the use of three Interaction Templates: Enter Data Element, View Data Table, and Print Labels. The three examples illustrated how data elements and presentation components are bound to task models through Interaction Templates. Finally, we demonstrated how Interaction Templates can be used to build task models that adapt to changes in business processes by modelling a simplified version of the Lab Assistant software from Western Ag. We also re-visited one of the change scenarios from Chapter 3, which we used to show how Interaction Templates can be used to support the co-evolution of information systems and business processes.

8.2 Future Work

This research could benefit from some extensions and improvements that we would like to pursue in the future. These include incorporating layout techniques and sketching in the generated user interface prototypes, considering multiple user task models, and performing a more in-depth case study using the Interaction Template approach.

8.2.1 Prototype Layout and Sketching

We would like to incorporate layout techniques to produce more aesthetically pleasing user interface prototypes. Currently, the generated user interface prototypes consist of a set of presentation components that can be placed in specific locations on the prototype screen. Each individual component is either shown or hidden, depending on the state of the task model simulation. We would like to investigate how automatically generated presentation components could be grouped into container components when they are logically enabled at the same time.

Furthermore, we would like to allow designers to layer presentation components with user interface sketches. We believe this would allow designers to create more complete screen designs, which would be more meaningful to both designers and users than our current prototypes.

8.2.2 Multiple User Task Models

Organizations often have complex business processes, some of which may involve multiple users cooperating to reach particular goals. ConcurTaskTrees can be used to build cooperative task models to describe systems involving multiple users [36]. Our prototype could be extended to include support for cooperative ConcurTaskTrees. We would like to extend our prototype to generate user interface prototypes for each different type of user or role. Since communication between the simulator and the generated prototypes is already implemented using TCP/IP sockets, each user's prototype could run on a different machine. Each prototype would then be able to communicate with the central task model simulator, and the task model simulator would communicate with each of the different prototypes. Multiple-user simulation sessions using generated prototypes could be useful in validation and verification of systems involving multiple users. Simulation using the prototypes would help us to see how different user's systems work together to accomplish business goals.

8.2.3 Attaching Sample Data

We have shown how we can build task models and user interface prototypes that are adaptable to the structure of data elements. We believe that attaching sample data to some of our user interface prototypes would help to give users a better understanding of what the end system might look like. A prototype of a data table, for example, might be more meaningful to end-users if the table component was populated with some sample data.

8.2.4 In-Depth Field Study

We would like to evaluate the Interaction Template approach more thoroughly by performing an in-depth and comprehensive field study. Such a field study would involve designing and developing a real world information system using the Interaction Template approach from the very beginning. Once the system has been implemented and deployed, we would like to continue to follow the evolution of the system for a significant period of time. Business process changes would be documented using our change scenario approach we introduced in Chapter 3. Each change scenario would be studied and we would use the Interaction Template approach to adapt the system in response to the change scenarios.

An in-depth field study would be very beneficial in fully evaluating our approach to improving task modelling to support the co-evolution of information systems and business processes. Unfortunately, such a study would be lengthy and require significant resources.

8.3 Conclusion

Businesses require information systems that are able to quickly adapt to continually changing business processes. This research has focused on improving existing task modelling techniques to help developers to better respond to changes in system requirements that result from changes in business processes. Through a study of the seven year evolution of a real world information system, we gained a better understanding of how changing business processes affect supporting information systems. We proposed the Interaction Template approach to improve task modelling to support the evolution of information systems and business processes. Our approach, by adding data modelling and presentation components to task models, has allowed us to provide adaptable building blocks for task modelling. Interaction Templates provide task model pieces that are adaptable to data elements and parameters. Binding presentation components to task models allows us to generate user interface prototypes from the Interaction Templates that are used in a task model. By re-visiting one of the change scenarios from our case study, we demonstrated how the Interaction Template approach is able to help developers to co-evolve information systems and business processes.

References

[1] Mickaël Baron and Patrick Girard. SUIDT: A task model based GUI-Builder. In *TAMODIA*, pages 64–71. INFOREC Printing House, 2002.

[2] Keith Bennett, Steven Golver, Xiang Li, and Stephen Rank. Designing software for change: evolvable architectures. In *Proceedings of the Principles of Software Change and Evolution; SCE'99'*, pages 65–69. ICSE'99, 1999.

[3] Keith H. Bennett and Vclav T. Rajlich. Software evolution: A roadmap. In *Proceedings IEEE International Conference on Software Maintenance*, page 4. IEEE, 2001.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (XML) 1.0 (third edition). W3 Recomendation available at: http://www.w3.org/TR/2004/REC-xml-20040204 - Accessed on November 3, 2005.

[5] Lisa Brownsword and Paul Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Carnegie Mellon Software Engineering Institute, 1996.

[6] Stuart K. Card, Thomas P Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1985.

[7] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, and Donghui Zhang. Storing and querying multiversion XML documents using durable node numbers. In *The 2nd International Conference on Web Information Systems Engineering (WISE)*, pages 232–, 2001.

[8] James Clark. XSL transformations (XSLT) 1.0. W3 Recomendation available at: http://www.w3.org/TR/xslt - Accessed on November 3, 2005.

[9] James Clark and Steve DeRose. XML path language (XPATH). W3 Recomendations available at: http://www.w3.org/TR/xpath - Accessed on November 3, 2005.

[10] Isabelle Comyn-Wattiau, Jacky Akoka, and Nadira Lammari. A framework for database evolution management. In *Second International Workshop on Unanticipated Software Evolution*, 2003.

[11] J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[12] Anke Dittmar and Peter Forbrig. Dialogue graph editor. http://wwwswt.-informatik.uni-rostock.de/deutsch/Projekte/DialogGraphEditor/dialoggraph-editor.html - Accessed on May 11, 2005.

[13] Anke Dittmar and Peter Forbrig. The influence of improved task models on dialogues. In *Fourth International Conference on Computer-Aided Design of User Interfaces*, pages 1–14, 2004.

[14] Pierre Dragicevic, David Navarre, Philippe Palanque, Amlie Schyn, and Rmi Bastide. Very-High-Fidelity prototyping for both presentation and dialogue parts of multimodal interactive systems. In *Design, Specification and Verification of Interactive Systems 2004 (EHCI-DSVIS'04)*, pages 61–88, 2004.

[15] W. Gray, B. John, and M. Atwood. Project ernestine: A validation of GOMS for prediction and explanation of real-world task performance. *Human-Computer Interaction*, 8(3):237–309, 1993.

[16] H. Rex Hartson, Jeffrey L. Brandenburg, and Deborah Hix. Different languages for different development activities: behavioral representation techniques for user interface design. pages 303–326, 1992.

[17] Kathryn L. Heninger. Specifying software requirements for complex systems: new techniques and their application. pages 111–135, 2001.

[18] David Kieras. A guide to GOMS model usability evaluation using ngomsl. In *The Handbook of Human-Computer Interaction*. North Holland, 2nd edition edition, 1996.

[19] Gnter Kniesel, Joost Noppen, Tom Mens, and Jim Buckley. The first workshop on unanticipated software evolution. In *First International Workshop on Unanticipated Software Evolution*, pages 1–15. ECOOP 2002, 2002.

[20] S. Krishnamurthi, K. E. Gray, and P. T. Graunke. Transformation-by-example for XML. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, pages 249–262. Springer-Verlag, 2000.

[21] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124. Springer-Verlag, 1996.

[22] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 70–74. ACM Press, 2002.

[23] Meir M. Lehman and Juan F. Ramil. Software evolution: background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.

[24] Kris Luyten and Tim Clerckx. TaskLib: a command line processor and library for ConcurTaskTrees specifications. http://www.edm.luc.ac.be/software/TaskLib/ - Accessed on May 11, 2005.

[25] Kris Luyten, Tim Clerckx, Karin Choninx, and Jean Vanderdockt. Derivation of a dialog model from a task model by activity chain extraciton. In *Design, Specification and Verification of Interactive Systems 2003 (DSV-IS 2003)*, pages 191–205. Springer-Verlag, 2003.

[26] Finbar McGurren and Damien Conroy. X-Adapt: An architecture for dynamic systems. In *First International Workshop on Unanticipated Software Evolution*, pages 10–18, 2002.

[27] David Navarre, Philippe A. Palanque, Fabio Paternó, Carmen Santoro, and Rmi Bastide. A Tool Suite for Integrating Task and System Models through Scenarios. In *DSV-IS '01: Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers*, pages 88–113, London, UK, 2001. Springer-Verlag.

[28] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference On the Future of Software Engineering*, pages 35–46. ACM Press, 2000.

[29] David Paquette and Kevin A. Schneider. Interaction templates for constructing user interfaces from task models. In *Fourth International Conference on Computer-Aided Design of User Interfaces*, pages 223–235, 2004.

[30] David Paquette and Kevin A. Schneider. Interaction templates for constructing user interfaces from task models. In Robert Jacob, Quentin Limbourg, and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces IV*, pages 223–234. Kluwer Acedemic Publishers, 2005.

[31] David Paquette and Kevin A. Schneider. Task model simulation using interaction templates. In *Design, Specification, and Verification of Interactive Systems 2005 (DSVIS'05)*, pages 49–60, 2005.

[32] David L. Parnas. Designing software for ease of extension and contraction. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals*, chapter 14, pages 267–290. Addison-Wesley, 2001.

[33] David L. Parnas. On the criteria to be used in decomposing systems into modules. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals*, chapter 7, pages 137–155. Addison-Wesley, 2001.

[34] David L. Parnas. On the design and development of program families. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals*, chapter 10, pages 191–213. Addison-Wesley, 2001.

[35] Fabio Paternó. ConcurTaskTreesEnvironment (CTTE). http://giove.cnuce.-cnr.it/ctte.html - Accessed on November 3, 2005.

[36] Fabio Paternó. *Model-Based Design and Evaluation of Interactive Applications.* Springer, 2000.

[37] Fabio Paternó. Task models in interactive software systems. In S. K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Co., 2001.

[38] Fabio Paternó. Tools for task modelling: Where we are, where we are headed. In Costin Pribeanu and Jean Vanderdonckt, editors, *First International Workshop on Task Models and Diagrams for User Interface Design - TAMODIA 2002*, pages 10–17, 2002.

[39] E. Pietriga, J.Y. Vion-Dury, and V. Quint. VXT: a visual approach to XML transformations. In *Proceedings of the 2001 ACM Symposium on Document engineering*, pages 1–10. ACM Press, 2001.

[40] J. Roddick. A survey of schema versioning issues for database systems, 1995.

[41] Kevin A. Schneider and James R. Cordy. Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems. In Chris Johnson, editor, *DSV-IS*, volume 2220 of *Lecture Notes In Computer Science*, pages 28–48. Springer, 2001.

[42] R. Chung-Man Tam, David Maulsby, and Angel R. Puerta. U-TEL: A tool for eliciting user task models from domain experts. In *Intelligent User Interfaces*, pages 77–80, 1998.

[43] Holger Uhr. TOMBOLA: Simulation and user-specific presentation of executable task models. In *Proceedings of HCI International*, volume 1, pages 263–267, 2003.

[44] WfMC. Workflow management coalition terminology and glossary, wfmc-tc-1011, document status- issue 2.0. In *Specifying Task Models. (Proceedings Interact97)*, pages 362–369. Chapman and Hall, June 1997.

[45] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, and P. Markopoulos. Beyond Hacking: A Model Based Approach to User Interface Design. In J.L. Alty, D. Diaper, and S. Guest, editors, *People and Computers VIII, Proceedings of HCI'93*, pages 217–231. Cambridge University Press, September 93.

[46] E Yu. Towards modelling and reasoning support for early- phase requirements engineering. In *3rd International Symposium on Requirements Engineering*, pages 226–235, 1997.

Appendix A

ConcurTaskTree XSD

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.cs.usask.ca/ns/ctt.xsd"
       xmlns:ctt="http://www.cs.usask.ca/ns/ctt.xsd">

    <xs:simpleType name="TaskCategory">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Abstraction"/>
        <xs:enumeration value="Interaction"/>
        <xs:enumeration value="System"/>
        <xs:enumeration value="User"/>
      </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="TemporalOperator">
      <xs:restriction base="xs:string">
        <xs:enumeration value="Choice"/>
        <xs:enumeration value="Order Independency"/>
        <xs:enumeration value="Concurrent"/>
        <xs:enumeration value="Concurrent with Information Exchange"/>
        <xs:enumeration value="Disabling"/>
        <xs:enumeration value="Suspend-Resume"/>
        <xs:enumeration value="Enabling"/>
        <xs:enumeration value="Enabling with Information Exchange"/>
      </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="Task">
      <xs:sequence>
        <xs:element name="TemporalOperator" type="ctt:TemporalOperator"/>
        <xs:element name="SubTasks" type="ctt:SubTasks"/>
      </xs:sequence>
      <xs:attribute name="ID" type="xs:string" use="required"/>
      <xs:attribute name="Category" type="ctt:TaskCategory" use="required"/>
      <xs:attribute name="Iterative" type="xs:boolean" use="required"/>
      <xs:attribute name="Optional" type="xs:boolean" use="required"/>
    </xs:complexType>

    <xs:complexType name="SubTasks">
      <xs:sequence>
        <xs:element name="Task" type="ctt:Task"
                    minOccurs="2" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="TaskModel">
     <xs:complexType>
      <xs:sequence>
        <xs:element name="Task" type="ctt:Task"/>
      </xs:sequence>
      <xs:attribute name="ModelName" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>

</xs:schema>
```

Appendix B

Interaction Template XSD

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       xmlns:ctt="http://www.cs.usask.ca/ns/ctt.xsd"
       targetNamespace="http://www.cs.usask.ca/ns/it" xmlns:it="http://www.cs.usask.ca/ns/it">

<xs:import namespace="http://www.cs.usask.ca/ns/ctt.xsd"
           schemaLocation="http://www.cs.usask.ca/ns/ctt.xsd"/>

<xs:simpleType name="parameterType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="string"/>
    <xs:enumeration value="decimal"/>
    <xs:enumeration value="float"/>
    <xs:enumeration value="boolean"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="component">
  <xs:attribute name="name" type="xs:string" use="required" />
  <xs:attribute name="objectname" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="setproperty">
  <xs:attribute name="component" type="xs:string" use="required" />
  <xs:attribute name="property" type="xs:string" use="required" />
  <xs:attribute name="value" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="bindcomponent">
  <xs:attribute name="component" type="xs:string" use="required" />
</xs:complexType>

<xs:complexType name="bindevent">
  <xs:sequence>
    <xs:element name="taskname" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="component" type="xs:string" use="required" />
  <xs:attribute name="event" type="xs:string" use="required" />
</xs:complexType>


<xs:complexType name="parameter">
  <xs:sequence>
    <xs:element name="value"
            minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="it:parameterType" use="required"/>
</xs:complexType>


<xs:complexType name="schema">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="foreach">
  <xs:sequence>
      <xs:element name="Task" type="ctt:Task"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="case" type="it:case"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="foreach" type="it:foreach"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="var" type="it:var"
            minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="bindevent" type="it:bindevent"
                minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="var">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="condition">
  <xs:sequence>
    <xs:element name="Task" type="ctt:Task"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="SubTasks" type="ctt:SubTasks"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="case" type="it:case"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="foreach" type="it:foreach"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="var" type="it:var"
            minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="bindevent" type="it:bindevent"
            minOcuurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="expression" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="case">
  <xs:sequence>
    <xs:element name="condition" type="it:condition"
            minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="usetemplate">
  <xs:sequence>
    <xs:element name="parameter" type="it:parameter"
                minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="schema" type="it:schema"
            minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required" />
</xs:complexType>

<xs:element name="template">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="description" type="xs:string"
              minOccurs="1" maxOccurs="1"/>
      <xs:element name="component" type="it:component"
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="parameter" type="it:parameter"
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="schema" type="it:schema"
```

```
              minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="task" type="ctt:Task"
              minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="case" type="it:case"
              minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Appendix C

Enter Data Element Interaction Template

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<it:template name="Enter Data Element"
             xmlns:it="http://www.cs.usask.ca/ns/it">

<it:description>
  A template defining the interaction task of entering a data element.  The
  "DataElement" schema describes the task's data element.
</it:description>

<it:schema name="DataElement" path="" />

  <ctt:Task ID="Enter $DataElement/@name" Category="Interaction"
              Iterative="False" Optional="False">
    <ctt:SubTasks>
      <it:foreach value="$DataElement/xs:complexType/xs:attribute">
        <it:case>
          <it:condition expression="expr(starts-with($value/@type,'xs:'))">
            <ctt:Task ID="Enter $value/@name" Category="Interaction"
                    Iterative="False" Optional="expr( $value/@use eq "optional" )">
              <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
              <it:component name="Enter_$value/@name" objectname="TLabeledEdit"/>
              <it:setproperty component="Enter_$value/@name" property="EditLabel.Caption"
                        value="$value/@name"/>
              <it:bindcomponent component="Enter_$value/@name" />
              <it:bindevent component="Enter_$value/@name" event="OnExit"></it:bindevent>
            </ctt:Task>
          </it:condition>
          <it:condition expression="true()">
            <it:var name="CurAttribute"
                  path="/xs:schema/xs:simpleType[@name = $value/@type]" />
            <it:case>
              <it:condition expression="$CurAttribute/xs:restriction/xs:enumeration/last() gt 0">
                 <ctt:Task ID="Select $value/@name" Category="Interaction"
                        Iterative="False" Optional="expr( $value/@use eq "optional" )">
                  <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
                  <it:component name="Enter_$value/@name" objectname="TComboBox"/>
                  <it:setproperty component="Enter_$value/@name" property="Items"
                            value="$CurAttribute/xs:restriction/xs:enumeration/@value"/>
                  <it:bindcomponent component="Enter_$value/@name" />
                  <it:bindevent component="Enter_$value/@name" event="OnSelect">
                    <it:taskname>
                      TaskName := 'Select ' + Self.Text;
                    </it:taskname>
                  </it:bindevent>
                  <ctt:SubTasks>
                    <it:foreach value="$CurAttribute/xs:restriction/xs:enumeration">
                      <ctt:Task ID="Select $value/@value" Category="Interaction"
                              Iterative="False" Optional="False">
                        <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                      </ctt:Task>
                    </it:foreach>
                  </ctt:SubTasks>
                </ctt:Task>

              </it:condition>

              <it:condition expression="true()">
                <ctt:Task ID="Enter $value/@name" Category="Interaction"
                          Iterative="False" Optional="expr( $value/@use eq "optional" )">
                  <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
                  <it:component name="Enter_$value/@name" objectname="TLabeledEdit"/>
                  <it:setproperty component="Enter_$value/@name" property="EditLabel.Caption"
                                  value="$value/@name"/>
                  <it:bindcomponent component="Enter_$value/@name" />
                  <it:bindevent component="Enter_$value/@name" event="OnExit"></it:bindevent>
                </ctt:Task>
              </it:condition>
            </it:case>
          </it:condition>
        </it:case>
      </it:foreach>

      <it:foreach value="$DataElement/xs:complexType/xs:sequence/xs:element">
        <it:var name="CurElement"
                path="/xs:schema/xs:complexType[@name = $value/@type] |
                      /xs:schema/xs:simpleType[@name = $value/@type]" />

        <it:case>
          <it:condition expression="expr(starts-with($value/@type,'xs:'))">
            <ctt:Task ID="Enter $value/@name" Category="Interaction"
                      Iterative="expr( $value/@maxOccurs gt 1 )"
                      Optional="expr( $value/@minOccurs eq 0 )">
              <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
              <it:component name="Enter_$value/@name" objectname="TLabeledEdit"/>
              <it:setproperty component="Enter_$value/@name" property="EditLabel.Caption"
                              value="$value/@name"/>
              <it:bindcomponent component="Enter_$value/@name" />
              <it:bindevent component="Enter_$value/@name" event="OnExit"></it:bindevent>
            </ctt:Task>
          </it:condition>

          <it:condition expression="$CurElement/name() eq "xs:complexType"">
            <it:usetemplate name="Enter Data Element" >
              <it:schema name="DataElement" path="$CurElement" />
            </it:usetemplate>

          </it:condition>

          <it:condition expression="$CurElement/name() eq "xs:simpleType"">

            <it:case>
              <it:condition expression="$CurElement/xs:restriction/xs:enumeration/last() gt 0">

                <ctt:Task ID="Select $value/@name" Category="Interaction"
                          Iterative="False" Optional="expr( $value/@minOccurs eq 0 )">

                  <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
                  <it:component name="Enter_$value/@name" objectname="TComboBox"/>
                  <it:setproperty component="Enter_$value/@name" property="Items"
                                  value="$CurElement/xs:restriction/xs:enumeration/@value"/>
                  <it:bindcomponent component="Enter_$value/@name" />
                  <it:bindevent component="Enter_$value/@name" event="OnSelect">
                    <it:taskname>
                      TaskName := 'Select ' + Self.Text;
                    </it:taskname>
                  </it:bindevent>

                  <ctt:SubTasks>
```

```
                    <it:foreach value="$CurElement/xs:restriction/xs:enumeration">
                      <ctt:Task ID="Select $value/@value" Category="Interaction"
                                Iterative="False" Optional="False">
                        <ctt:TemporalOperator>Choice</ctt:TemporalOperator>

                      </ctt:Task>
                    </it:foreach>

                  </ctt:SubTasks>
                </ctt:Task>

              </it:condition>

              <it:condition expression="true()">
                <ctt:Task ID="Enter $value/@name" Category="Interaction"
                          Iterative="expr( $value/@maxOccurs gt 1 )"
                         Optional="expr( $value/@minOccurs eq 0 )">
                  <ctt:TemporalOperator>Order Independence</ctt:TemporalOperator>
                  <it:component name="Enter_$value/@name" objectname="TLabeledEdit"/>
                  <it:setproperty component="Enter_$value/@name" property="EditLabel.Caption"
                                  value="$value/@name"/>
                  <it:bindcomponent component="Enter_$value/@name" />
                  <it:bindevent component="Enter_$value/@name" event="OnExit"></it:bindevent>
                </ctt:Task>
              </it:condition>
            </it:case>
          </it:condition>

        </it:case>
      </it:foreach>
    </ctt:SubTasks>
  </ctt:Task>

</it:template>
```

Appendix D

Data Table Interaction Template

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<it:template name="Data Table"
             xmlns:it="http://www.cs.usask.ca/ns/it">

  <it:description>
    A template defining the interaction task of viewing and manipulating data in table format.  The
    "DataElement" schema describes the data element that will be displayed in each row of the table.
  </it:description>

  <it:parameter name="AllowSort" type="it:boolean">
    <it:value>True</it:value>
  </it:parameter>

  <it:parameter name="AllowColumnMove" type="it:boolean">
    <it:value>True</it:value>
  </it:parameter>

  <it:parameter name="MultiSelect" type="it:boolean">
    <it:value>True</it:value>
  </it:parameter>

  <it:schema name="DataElement" path="" />

  <it:component name="View$DataElement/@name Table" type="TAdvStringGrid"/>

  <it:var name="Columns" path="/$DataElement/xs:complexType/xs:attribute |
              /$DataElement/xs:complexType/xs:sequence/xs:element" />

  <it:setproperty component="View$DataElement/@name Table" property="ColumnHeaders"
                  value="$Columns/@name"/>
  <it:setproperty component="View$DataElement/@name Table" property="AllowSort"
                  value="$AllowSort"/>
  <it:setproperty component="View$DataElement/@name Table" property="MultiSelect"
                  value="MultiSelect"/>
  <it:setproperty component="View$DataElement/@name Table" property="AllowColumnMove"
                  value="AllowColumnMove"/>

  <ctt:Task ID="View $DataElement/@name" Category="Abstraction"
          Iterative="False" Optional="False">

    <it:bindcomponent component="View$DataElement/@name Table"/>

    <ctt:SubTasks>
      <ctt:Task ID="Modify Table View" Category="Abstraction"
              Iterative="False" Optional="False">
        <ctt:TemporalOperator>
          Concurrent With Info Exchange
        </ctt:TemporalOperator>
        <ctt:SubTasks>

          <it:case>
            <it:condition expression="$AllowSort">

            <it:bindevent component="View$DataElement/@name Table" event="OnClickSort">
              <it:taskname>
                 sTaskName := ‘Click ’ + Self.Cells[ACol,0] + ‘ Header’;
              </it:taskname>
            </it:bindevent>

            <ctt:Task ID="Sort By Column" Category="Abstraction"
                      Iterative="False" Optional="False">
              <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
              <ctt:SubTasks>

                <it:foreach value="$Columns">
                  <ctt:Task ID="Sort By $value/@name" Category="Abstraction"
                            Iterative="False" Optional="False">
                    <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                    <ctt:SubTasks>
                      <ctt:Task ID="Click $value/@name Header" Category="Interaction"
                                Iterative="False" Optional="False">
                        <ctt:TemporalOperator>
                          Enabling with Information Exchange
                        </ctt:TemporalOperator>
                      </ctt:Task>
                      <ctt:Task ID="Sort Rows By $value/@name" Category="System"
                                Iterative="False" Optional="False">
                      </ctt:Task>
                    </ctt:SubTasks>
                  </ctt:Task>
                </it:foreach>

              </ctt:SubTasks>
            </ctt:Task>
          </it:condition>
        </it:case>

        <it:case>
          <it:condition expression="$AllowColumnMove">
            <it:bindevent component="View$DataElement/@name Table" event="OnColumnMove">
              <it:taskname>
                sTaskName := ‘Select ’ + Self.Cells[ACol,0] ;
              </it:taskname>
            </it:bindevent>


            <ctt:Task ID="Move Column" Category="Interaction"
                      Iterative="False" Optional="False">
              <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
              <ctt:SubTasks>

                <ctt:Task ID="Select Column" Category="Interaction"
                          Iterative="False" Optional="False">
                  <ctt:TemporalOperator>
                    Enabling With Information Exchange
                  </ctt:TemporalOperator>
                  <ctt:SubTasks>
                    <it:foreach value="$Columns">
                      <ctt:Task ID="Select $value/@name" Category="Abstraction"
                                Iterative="False" Optional="False">
                        <ctt:TemporalOperator>Choice</ctt:TemporalOperator>

                        </ctt:SubTasks>
                      </ctt:Task>
                    </it:foreach>
                  </ctt:SubTasks>
                </ctt:Task>
                <ctt:Task ID="Select New Location" Category="Interaction"
                          Iterative="False" Optional="False">
                  <it:bindevent component="View$DataElement/@name Table" event="OnColumnMoved">
                  </it:bindevent>
                </ctt:Task>
              </ctt:SubTasks>
```

```
          </ctt:Task>
        </it:condition>
      </it:case>

    </ctt:SubTasks>

    <it:case>
      <it:condtion expression="expr( $MultiSelect = True )">
        <ctt:Task ID="Select Rows" Category="Abstraction"
                  Iterative="True" Optional="False">
          <it:bindevent component="View$DataElement/@name Table" event="OnSelectionChanged">
          </it:bindevent>
          <ctt:TemporalOperator>
            Suspend-Resume
          </ctt:TemporalOperator>
        </ctt:Task>
        <ctt:Task ID="Perform Operation On Row" Category="Abstraction"
          Iterative="False" Optional="True">
        </ctt:Task>
      </it:condition>
      <it:condtion expression="expr( $MultiSelect = False )">

        <ctt:Task ID="Select Row" Category="Abstraction"
                  Iterative="False" Optional="False">
          <ctt:SubTasks>
            <ctt:Task ID="Click Row" Category="Interaction"
                      Iterative="True" Optional="False">
              <it:bindevent component="View$DataElement/@name Table" event="OnSelectionChanged">
              </it:bindevent>
              <ctt:TemporalOperator>
                Enabling With Information Exchange
              </ctt:TemporalOperator>
            </ctt:Task>
            <ctt:Task ID="Perform Operation On Row" Category="Abstraction"
                      Iterative="False" Optional="True">
            </ctt:Task>
          </ctt:SubTasks>
        </ctt:Task>
      </it:condition>
    </it:case>

  </ctt:SubTasks>
</ctt:Task>

</it:template>
```

Appendix E

Print Labels

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<it:template name="Print Labels"
             xmlns:it="http://www.cs.usask.ca/ns/it">

  <it:description>
    A template defining the interaction task of printing labels for a series of data elements.  The
    "DataElement" schema describes the data element that will printed.
  </it:description>

  <it:schema name="DataElement" path="" />

  <it:component name="PrintLabels" type="TPrintLabelsDialog"/>

  <it:setproperty component="PrintLabels" property="LabelName" value="$DateElement/@name"/>

  <ctt:Task ID="Print $DateElement/@name Labels" Category="Abstraction"
            Iterative="False" Optional="False">

    <it:bindcomponent component="PrintLabels"/>

    <ctt:SubTasks>

      <ctt:Task ID="Modify Print View" Category="Abstraction"
                Iterative="True" Optional="False">
        <ctt:TemporalOperator> Disabling </ctt:TemporalOperator>
        <ctt:SubTasks>

          <ctt:Task ID="Display Page Preview" Category="System"
                    Iterative="False" Optional="False">
            <ctt:TemporalOperator>Enabling</ctt:TemporalOperator>
            <it:bindevent component="PrintLabels" event="OnShowLabels">
            </it:bindevent>
          </ctt:Task>

          <ctt:Task ID="Modify Label Preview" Category="Interaction"
                    Iterative="False" Optional="False">
            <ctt:TemporalOperator>Suspend-Resume</ctt:TemporalOperator>
            <ctt:SubTasks>

              <ctt:Task ID="View Previous Page" Category="Interaction"
                        Iterative="False" Optional="False">
                <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                <it:bindevent component="PrintLabels" event="OnPrevPage">
                </it:bindevent>
              </ctt:Task>

              <ctt:Task ID="View Next Page" Category="Interaction"
                        Iterative="False" Optional="False">
                <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                <it:bindevent component="PrintLabels" event="OnNextPage">
                </it:bindevent>
              </ctt:Task>

              <ctt:Task ID="Modify Page Layout" Category="Interaction"
                        Iterative="False" Optional="False">
                <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                <ctt:SubTasks>

                  <ctt:Task ID="Select Labels" Category="Interaction"
                            Iterative="False" Optional="False">
                    <ctt:TemporalOperator>Enabling</ctt:TemporalOperator>
                    <it:bindevent component="PrintLabels" event="OnLabelSelectionChanged">
                    </it:bindevent>
                  </ctt:Task>

                  <ctt:Task ID="Modify Label Status" Category="Interaction"
                            Iterative="False" Optional="False">
                    <ctt:SubTasks>

                      <ctt:Task ID="Mark Label As Missing" Category="Interaction"
                                Iterative="False" Optional="False">
                        <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                        <it:bindevent component="PrintLabels" event="OnLabelMissingClick">
                        </it:bindevent>
                      </ctt:Task>
                      <ctt:Task ID="Mark Label As Not Missing" Category="Interaction"
                                Iterative="False" Optional="False">
                        <it:bindevent component="PrintLabels" event="OnLabelNoMisssingClicked">
                        </it:bindevent>
                      </ctt:Task>

                    </ctt:SubTasks>
                  </ctt:Task>

                </ctt:SubTasks>
              </ctt:Task>

              <ctt:Task ID="Modify # Labels / $DataElement/@name" Category="Interaction"
                        Iterative="False" Optional="False">
                <ctt:SubTasks>

                  <ctt:Task ID="Increase Labels / $DataElement/@name" Category="Interaction"
                            Iterative="False" Optional="False">
                    <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                    <it:bindevent component="PrintLabels" event="OnIncreaseLabelsClick">
                    </it:bindevent>
                  </ctt:Task>

                  <ctt:Task ID="Decrease Labels / $DataElement/@name" Category="Interaction"
                            Iterative="False" Optional="False">
                    <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
                    <it:bindevent component="PrintLabels" event="OnDecreaseLabelsClick">
                    </it:bindevent>
                  </ctt:Task>

                </ctt:SubTasks>
              </ctt:Task>

            </ctt:SubTasks>

          </ctt:Task>

          <ctt:Task ID="Setup Printer" Category="Abstraction"
                    Iterative="False" Optional="True">
          </ctt:Task>

        </ctt:SubTasks>

      </ctt:Task>

      <ctt:Task ID="End Print Labels Dialog" Category="Abstraction"
                Iterative="False" Optional="False">
        <ctt:SubTasks>
```

```
     <ctt:Task ID="Cancel Printing" Category="Interaction"
          Iterative="False" Optional="False">
      <ctt:TemporalOperator>Choice</ctt:TemporalOperator>
    <it:bindevent component="PrintLabels" event="OnCancel">
    </it:bindevent>
     </ctt:Task>
     <ctt:Task ID="Accept Printing" Category="Abstraction"
          Iterative="False" Optional="False">
      <ctt:SubTasks>

       <ctt:Task ID="Print" Category="Interaction"
               Iterative="False" Optional="False">
      <ctt:TemporalOperator>Enabling</ctt:TemporalOperator>
        <it:bindevent component="PrintLabels" event="OnPrint">
        </it:bindevent>
     </ctt:Task>
       <ctt:Task ID="Send Pages To Printer" Category="System"
               Iterative="False" Optional="False">
     </ctt:Task>

      </ctt:SubTasks>
     </ctt:Task>

    </ctt:SubTasks>
   </ctt:Task>
  </ctt:SubTasks>
 </ctt:Task>

</it:template>
```