Object-Oriented GUI Design of a Modeling Environment for Logical Discrete Event Systems

by

Haoming Wang

mémoire présenté au Département d'informatique en vue
de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, juillet 2004

Le 16 juillet 2004,
Date

le jury a accepté le mémoire de M. Haoming Wang dans sa version finale.

Membres du jury

M. Richard St-Denis
Directeur
Département d'informatique

Mme Reine Gagnon
Membre
Département d'informatique

M. Gabriel Girard
Président-rapporteur
Département d'informatique

Sommaire

Ce mémoire porte sur la conception et l'implémentation d'une partie de l'interface personne-machine orientée objet d'un environnement de modélisation de systèmes réactifs appelé MELODIES (*Modeling Environment for LOgical Discrete Event Systems*). Cet environnement permet la conception, l'analyse, la simulation et le contrôle de systèmes à événements discrets. L'architecture de l'interface est basée sur certaines idées empruntées au schéma de conception *Model-View-Controller* (MVC) et au paradigme *JSP Model 2 Architecture*. Il en résulte une nette séparation entre l'interface personne-machine, les structures de données ainsi que les fonctions sous-jacentes. L'adoption d'une approche orientée objet, comme celle supportée par VisualAge for Java 4.0, au lieu d'une approche orientée fichier supportée par plusieurs environnements de développement (par exemple JBuilder for Java 3.0) permet une plus grande convivialité et une meilleure organisation des artéfacts de modélisation. Afin de déployer cette interface sur différentes plateformes et d'assurer une rapidité d'exécution, la boîte à outils Qt 3.0 et le langage C++, avec sa librairie STL, ont été utilisés dans l'étape de codification. De plus, XML a été retenu comme langage de représentation de données afin de permettre un déploiement éventuel de MELODIES sur le Web.

Acknowledgments

I am very grateful to my supervisor Professor R. St-Denis. I have gotten a lot of advices and directions from him in my work. Under his guidance, I can finally accomplish my project and master thesis.

I am very thankful to Professor G. Girard and Professor R. Gagnon for their advices and guidances.

Thanks to Mr. I. Manimpire. His work and cooperation were very helpful to my work.

Table of contents

List of figures

List of abbreviations

ADT	Abstract Data Type
DES	Discrete Event System
DOM	Document Object Model
DTD	Document Type Definition
FSM	Finite State Machine
GUI	Graphic User Interface
JSP	JavaServer Page
MELODIES	Modeling Environment for LOgical Discrete Event Systems
MVC	Model-View-Controller
OO	Object-oriented
PLC	Programmable Logic Controller
SCT	Supervisory Control Theory
TTG	Timed Transition Graph
XML	Extensible Markup Language

Introduction

Much work and effort have been done in modeling discrete event systems (DESs) for control purpose in the past years since the Supervisory Control Theory (SCT) has been developed by Ramadge and Wonham [12]. MELODIES (Modeling Environment for LOgical Discrete Event Systems) is among one of them. MELODIES is a new paradigm in designing reactive systems that was originally proposed by Professor St-Denis in [13]. Now it has been developed into a large application project. The functionalities of MELODIES are shown in Fig. 1. MELODIES includes the following modules:

- OO Interface module includes an object-oriented graphic user interface (GUI) for the MELODIES software environment.
- I/O module is responsible for data format, files, and input/output functionalities.
- Editing module is responsible for editing timed transition graphs (TTGs), abstract data types (ADTs), formulas, mask functions, and feedback functions.
- Drawing module is responsible for drawing timed transition graphs (TTGs) and trees.
- Simulation module is responsible for simulating the behaviors of a closed-loop system that interact between the system and the controller.
- Synthesis module includes algorithms to synthesize controllers.
- Computation module includes basic algorithms for operations on automata.
- PLC (Programmable Logic Controller) module is responsible for code generation

to PLC.

– Help module includes on-line help about the MELODIES softwares.

– Customization module includes configuration for the GUI.

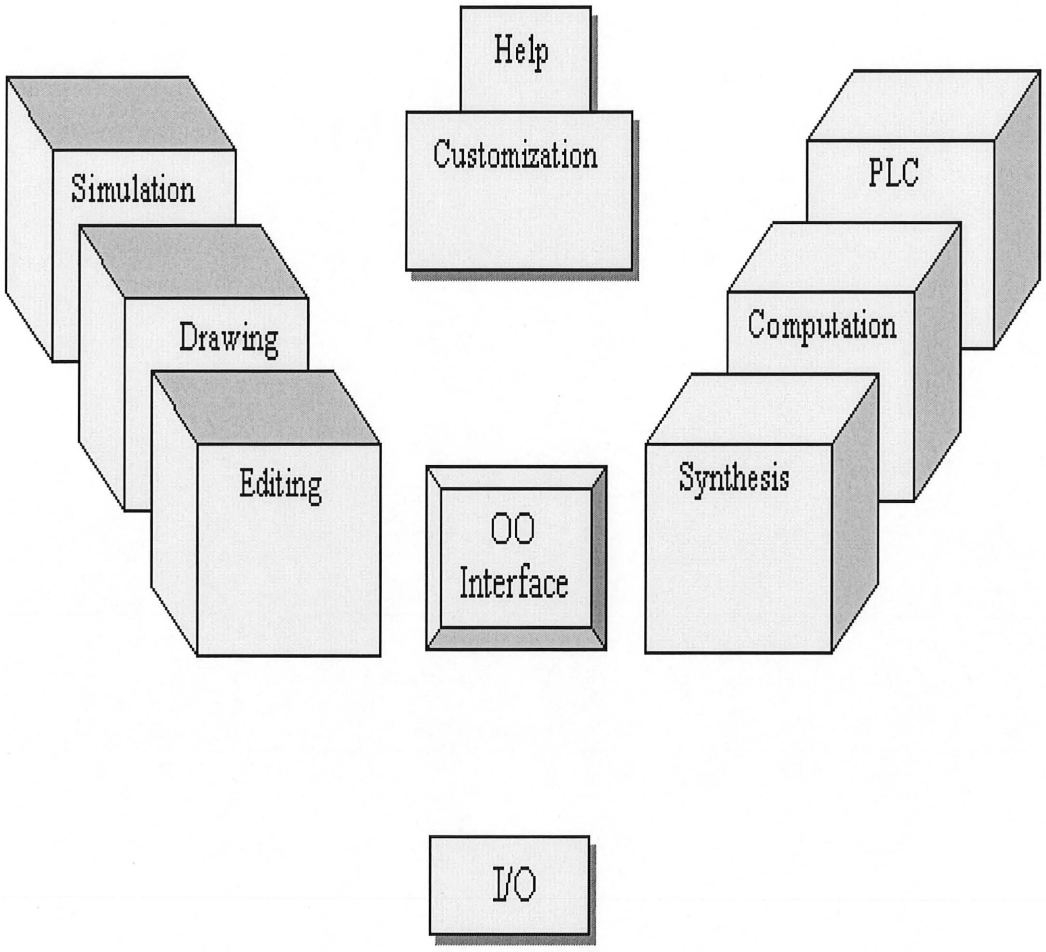


FIG. 1 – *Functionalities of MELODIES architecture*

In this thesis, we discuss the OO Interface and part of I/O modules as shown in Fig. 1.

In the GUI design, we need to represent information of DES project hierarchy and related processes. From the developer's point of view, the GUI design takes into consideration three possible abstract levels: top, middle, and low levels.

The top level includes classes that deal with presentation of all the items that need to be displayed on the GUI. Designers do not need to know much about DESs to develop these classes. They can base the interaction on a main menu and a pop-up menu to invoke their methods. In addition, these classes provide functions that are very common in GUI such as insertion, deletion, renaming, and modification. The implementation of these classes has less relationships with DESs. Hence the classes can be developed independently. From a functional point of view, this level includes the OO Interface and part of I/O modules in the MELODIES architecture as shown in Fig. 1.

The middle level includes classes whose implementation involves much knowledge about DESs. To implement these classes designers must master the theory that MELODIES is based on. Typical functions in this level are edition and creation of a new timed transition graph, and simulation. This level includes Editing, Simulation, Drawing, and part of I/O modules in the MELODIES architecture as shown in Fig. 1.

The low level includes underlying software that can be used by the two top levels such as synchronous composition and controller synthesis. This level includes Computation, Synthesis, and PLC modules in the MELODIES architecture as shown in Fig. 1.

In addition, there are accessory functions such as configuration and help. These functions belong to Help and Customization modules in the MELODIES architecture.

In this thesis, we mainly discuss the design and implementation of the top level. Simulation, Editing, Help, and Customization modules have been designed and implemented in a companion thesis [8].

In the GUI design, we take some ideas from the Model-View-Controller (MVC) design pattern, the JSP (JavaServer Pages) Model 2 Architecture, and the object-oriented approach used in VisualAge for Java 4.0. We use Qt 3.0 and C++ STL (Standard Template Library) in the implementation phase. Beside, XML is applied as data format for possible future use on the Web. The GUI can be used for design and analysis of discrete event systems by calling the underlying software.

The rest of the thesis is organized as follows. In Chapter 1, we discus the basic theory about modeling DESs and Supervisory Control Theory (SCT) that MELODIES is based on. In Chapter 2, we present some basic concepts of Qt and discuss practical issues about using Qt in GUI design. In Chapter 3, we discuss XML as data transmission format and give the corresponding DTD (*Document Type Definition*) design. In Chapter 4, we discuss the design issue used in the GUI design and the main classes involved. In conclusion, we give some points about future development.

Chapter 1

Discrete Event Systems and Supervisory Control Theory

The MELODIES software environment is based on the Supervisory Control Theory for discrete event systems [12] and a paradigm for designing reactive systems [13]. In this chapter we introduce some concepts of discrete event systems and the paradigm used in MELODIES.

1.1 Introduction to DES

Classical control theory uses differential equations to deal with time-driven systems. The input and output variables as well as the internal state variables of a time-driven system can be described by real, or possibly complex, numbers that change their values in accordance with time. Since the behaviors of such systems may be modeled as continuous functions of time, they can be appropriately described by sets of differential equations. However, behaviors of some systems such as traffic systems, for instance, cannot be appropriately described by differential equations at the logic level of abstraction. In this situation, classical control theory is not applicable. These systems may be modeled as discrete event systems instead.

A discrete event system (DES) is a dynamical system which evolves according to asynchronous occurrence of certain discrete changes, called events. The main characteristic of a DES is that the events (inputs and outputs) last a short period and that the system transitions between states happen on the occurrence of these events. Discrete event systems are useful to model the systems that are characterized with logical behavior. These types of systems may thus be called event-driven. Unlike the states of time-driven systems, the states of DESs are modeled by symbolic-valued variables that may indicate a certain kind of operation, for instance. In DESs, the relationships between state transitions and events are highly irregular and usually cannot be described using differential equations. So, a totally different approach is adopted to deal with DESs.

To conclude, the behavior of a DES is described by the sequences of events that occur and the sequences of states whose transitions depend on events. Typical application areas in which DES theory is applied are traffic systems, communication protocols, database systems, and manufacturing systems.

1.2 Modeling Formalisms

In this section, we introduce the main modeling formalisms for discrete event systems. Specifically, formal languages and finite state machines are discussed. As mentioned above, a discrete event system can be described by sequences of events and sequences of states. This approach leads to a state based model to be used to describe discrete event systems. There are, however, several formalisms that can be used to model discrete event systems. The basic ones are finite state machines and formal languages.

1.2.1 Events, Strings, and Languages

Before introducing formal language and finite state machine modeling formalisms, we first introduce the most important concepts that are needed for the following sections.

Given a finite set of events, called the alphabet, a system behavior is described by the sequences of events from the alphabet that may occur. At the logical level of abstraction, the behavior of a DES is described by the set of all possible sequences of state and event pairs:

$(x_0,e_0)(x_1,e_1)...$

We only consider deterministic DESs here. A deterministic DES is a DES such that the next state is uniquely determined by the current state and an event that occurs in that state. It is obvious that the behavior of a deterministic DES can be equivalently described by the initial state x_0 and the set of all the possible sequences of events $e_1e_2e_3...$, each such sequence is called a *trace* or *string* of the system, and a collection of strings is called a *language*. Let Σ^* denote the set of all finite length strings consisting of events from Σ, including the zero length string ϵ. Then a language is a subset of Σ^*.

The formal language modeling formalism is given next according to the concepts previously discussed. Definitions, concepts, texts, and results presented in the next sections are borrowed from Kumar and Garg [6].

1.2.2 Language Models

Let $K \subseteq \Sigma^*$ ($K \neq \emptyset$) be a language which consists of all the strings that can occur in a DES and $pr(K)$ be the prefix of K. It is easy to conclude that for a sequence of events to occur in a DES, all of its prefixes must occur first; so we get that $K = pr(K)$ in a DES. The language K is called the generated language of a DES. Let $K_m \subseteq K$ be the language consisting of those strings in the generated language of the DES whose execution represents completion of a certain task. This language is called the marked language of a DES. Thus, a DES can be modeled by a pair of languages (K,K_m). Given two language models (K,K_m) and (H,H_m), they are said to be equal if $K_m = H_m$ and $K = H$.

1.2.3 Finite State Machines

A formal language model is simple but it just gives a brief description of the system. From a formal language model, we only know what events may occur (the alphabet), and in which order they do occur (the language). There is no sufficient state information embedded in such a model. Because of this limitation, finite state machines may be used instead. Just like formal languages, finite state machines are mathematically well defined. A finite state machine introduces states, transitions that may happen between the states and events that are associated with transitions. Furthermore, marked states may be used to indicate the completion of desired tasks. A formal definition of a finite state machine is given in Definition 1.1.

Definition 1.1 Finite State Machine (FSM)

A finite state machine is defined by a 5-tuple $G := (X,\Sigma,\alpha,x_0,X_m)$, where X denotes a finite set of states, the state space, Σ is a finite set of event labels, the alphabet, $\alpha : X \times (\Sigma \cup \epsilon) \to 2^X$ is the transition function, $x_0 \in X$ is the initial state, and $X_m \subseteq X$ is a set of marked states. A state transition on ϵ represents a hidden transition, also called an ϵ-move. In general, the state transition function α does not uniquely determine the resulting state. Hence, G is referred as a non-deterministic state machine with ϵ-moves (ϵ-NSM). G is simply said to be a NSM if its state transition function can be written as a partial map $\alpha : X \times \Sigma \to 2^X$, i.e., if there are no hidden transitions in G. G is said to be a deterministic finite state machine (DFSM) if its state transition function can be written as a partial map $\alpha : X \times \Sigma \to X$, i.e., if there are no hidden transitions and the transition function uniquely determines the resulting state.

Example 1: Fig. 2 represents a non-deterministic finite state machine. Fig. 3 represents a deterministic finite state machine.

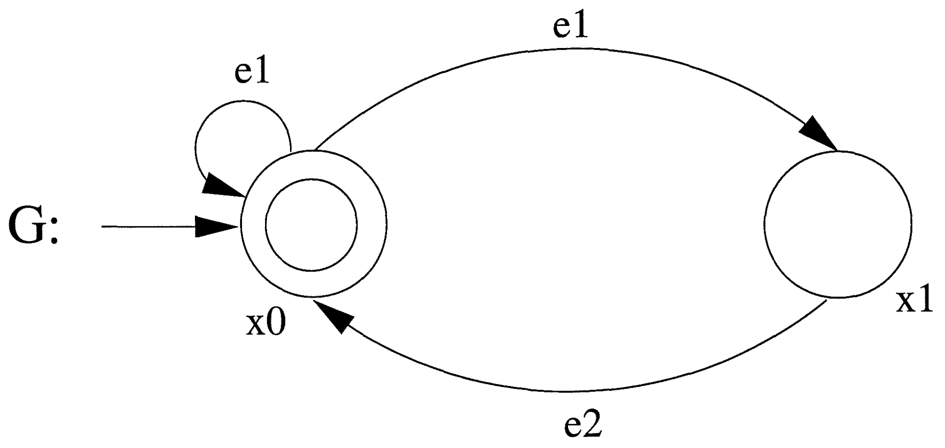


FIG. 2 – *A non-deterministic finite state machine*

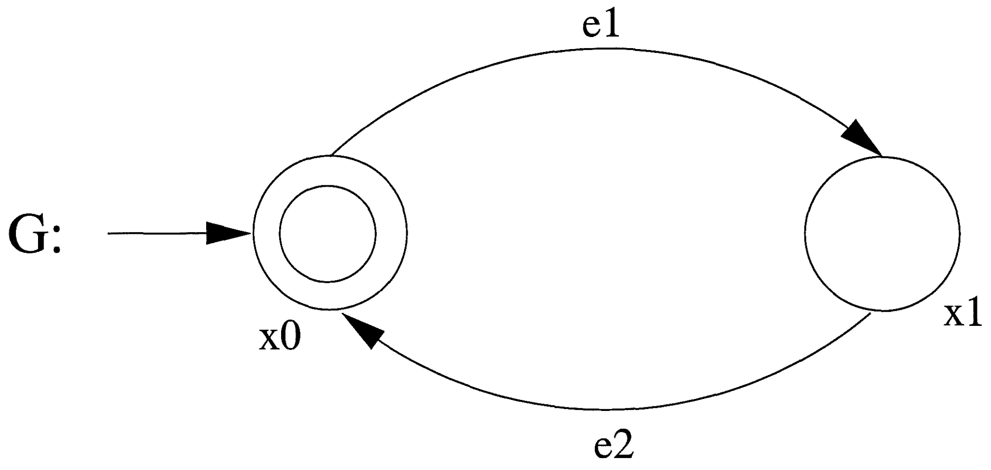


FIG. 3 – *A deterministic finite state machine*

We only consider DFSM here, since a FSM can be changed to an equivalent DFSM. As a finite state machine evolves on the occurrence of events, so it is possible to describe it by a formal language. Indeed, a finite state machine G is described by two languages, the prefix closed language $L(G)$ and the marked language $L_m(G)$. They are defined respectively as:

$$L(G) := \{s \in \Sigma^* \mid \alpha(x_0,s) \neq \emptyset\} \text{ and } L_m(G) := \{s \in L(G) \mid \alpha(x_0,s) \cap X_m \neq \emptyset\}$$

Normally, the prefix closed language $L(G)$ is straightforward to determine. Following we give some properties and operations of FSM which are important in MELODIES computation model.

Definition 1.2 Reachability

For a FSM G, a state $x \in X$ is reachable if there exists some string $s \in L(G)$ such that $\alpha(x_0,s) = x$.

Definition 1.3 Accessibility

A FSM is accessible if all states are reachable.

Definition 1.4 Co-reachability

For a FSM G, a state $x \in X$ is co-reachable if there exist some string $s \in L(G)$ such that $\alpha(x,s) = x_m$, where $x_m \in X_m$.

Definition 1.5 Co-Accessibility

A FSM is co-accessible if all states are co-reachable.

Definition 1.6 Trim

A FSM that is both accessible and co-accessible is trim. For a trim FSM, $pr(L_m(G)) = L(G)$.

Example 2: In Fig. 4, state x_3 is not co-reachable, so G is not trim.

Example 3: In Fig. 5, G' is trim FSM obtained from G given in Example 2.

Definition 1.7 Synchronous Composition of FSM

Given two FSMs $G_1 := (X_1,\Sigma_1,\alpha_1,x_{0,1},X_{m,1})$ and $G_2 := (X_2,\Sigma_2,\alpha_2,x_{0,2},X_{m,2})$, the synchronous composition of G_1 and G_2, denoted $G_1||G_2 := (X,\Sigma,\alpha,x_0,X_m)$, is defined as $X := X_1 \times X_2$; $\Sigma := \Sigma_1 \cup \Sigma_2$; $x_0 := (x_{0,1},x_{0,2})$; $X_m := X_{m,1} \times X_{m,2}$; and for each

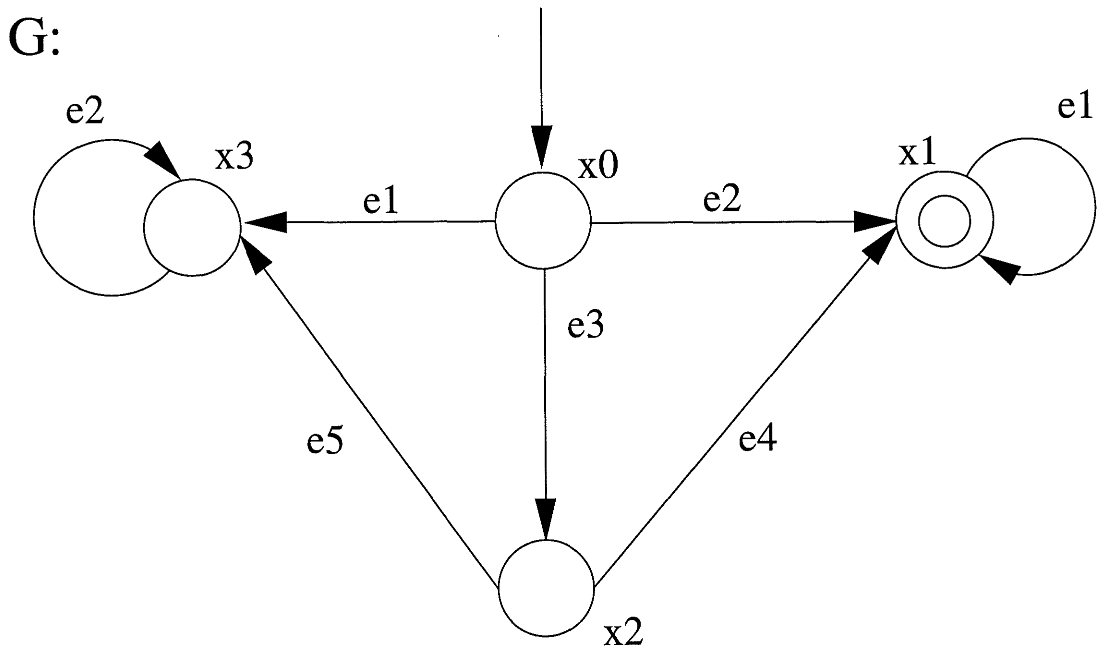


FIG. 4 – *G is not trim*

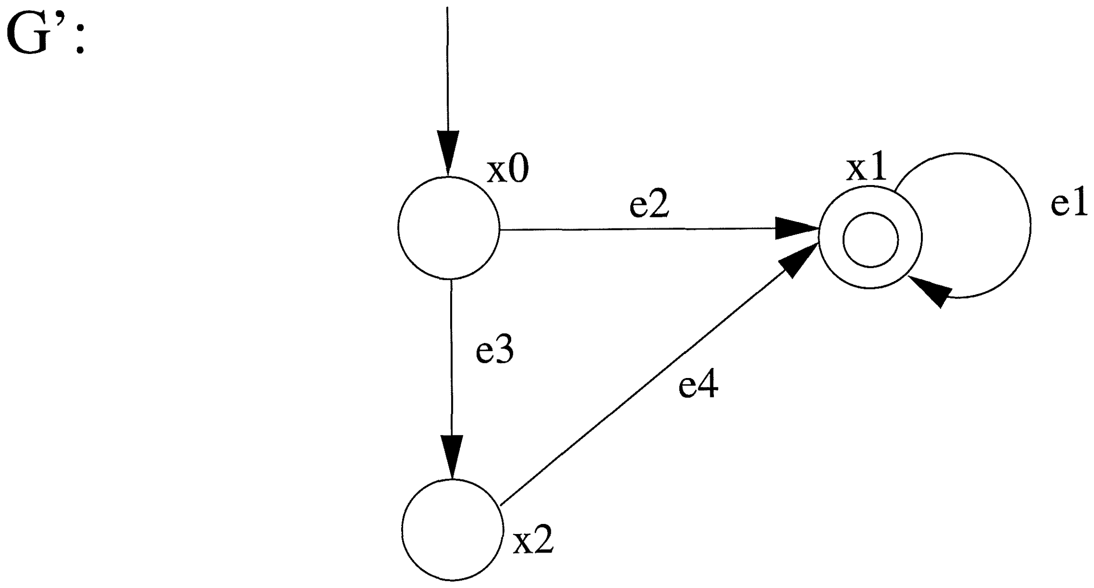


FIG. 5 – *Trim generator G′ obtained from G of Fig. 4*

$x = (x_1,x_2) \in X, \sigma \in \Sigma$:

$$\alpha(x,\sigma) := \begin{cases} (\alpha_1(x_1,\sigma),\alpha_2(x_2,\sigma)) & \text{if } \alpha_1(x_1,\sigma),\ \alpha_2(x_2,\sigma) \text{ defined}, \ \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\alpha_1(x_1,\sigma),x_2)) & \text{if } \alpha_1(x_1,\sigma) \text{ defined}, \ \sigma \in \Sigma_1 - \Sigma_2 \\ (x_1,\alpha_2(x_2,\sigma)) & \text{if } \alpha_2(x_2,\sigma) \text{ defined}, \ \sigma \in \Sigma_2 - \Sigma_1 \\ \text{undefined otherwise} & \end{cases}$$

For efficient computation in specific situations, the synchronous composition is classified as three specific operations in MELODIES computation model:

1. Shuffle product, if $\Sigma_1 \cap \Sigma_2 = \emptyset$. This is a parallel composition with no synchronization composition.
2. Cartesian product, if $\Sigma_1 = \Sigma_2$. This is a parallel composition with full synchronization.
3. Synchronous product, if $\Sigma_1 \cap \Sigma_2 \neq \emptyset$ and $\Sigma_1 \neq \Sigma_2$. This is a parallel composition with partial synchronization.

1.3 Supervisory Control Theory

The DES model described above is simply based on languages or automaton generators without considering external control. In practice, control is necessary in the case DESs need external inputs to evolve, or to fulfill the control requirements (called a specification). For this purpose, we introduce the Supervisory Control Theory (SCT) that was initially developed by Ramadge and Wonham [12]. It was a general approach to the synthesis of controllers for DESs. In its original formulation, the SCT is described in terms of formal languages and finite state automata that represent those languages.

1.3.1 Control under Complete Observation

The supervisory control problem of DES under complete observation of events can be stated as follows.

Given the controlled system, called the *process* or *plant*, the controlling system, called the *controller* or *supervisor*, is to be designed such that the process and the controller will work together to conform to a desired behavior, the *specification*. The interaction of a process and a controller is depicted in Fig. 6.

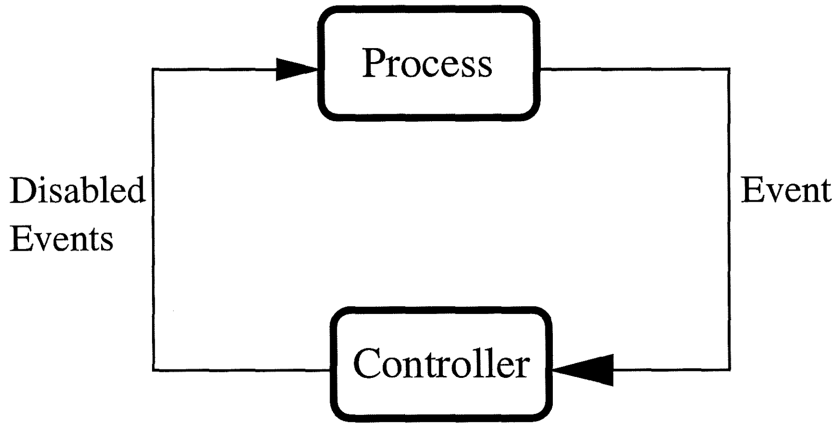


FIG. 6 – *A DES under complete observation*

The event set Σ is divided into two disjoint sets, the set of uncontrollable events, denoted Σ_u, and the set of controllable events, denoted Σ_c. That is $\Sigma = \Sigma_u \cup \Sigma_c$. On one hand, the generation of the controllable events may be dynamically disabled by the controller, on the other hand, the uncontrollable events cannot be disabled.

The process, controller, and specification can be modeled by FSMs. Let P denote a FSM representing a process, C denote a FSM representing a controller, and S denote a FSM representing a specification. It can be proved that the synchronous composition of P and C can restrict the behavior of the process. Moreover, we can get a synchronous composition based controller C such that $L(P||C) \subseteq L(S)$ or $L_m(P||C) \subseteq L_m(S)$.

1.3.2 Control under Partial Observation

It was assumed in the SCT under complete observation of events that a controller is able to observe all the events generated by the process. In many situations, it is difficult, if not impossible to observe all the events due to lack of sensors or failure of sensors. Thus, a controller is not able to observe a particular event, or distinguish some events. In such partial observation, we define the set of observable events Σ_o and an observation mask $M : \Sigma \rightarrow \Sigma_o \cup \epsilon$. If an event is mapped to ϵ, then it is unobservable to a controller. If an event is mapped to Σ_o, then it is observable to a controller. If two events are mapped to the same symbol, then they are indistinguishable to a controller. A controller is called observation-compatible if it takes identical control action following all sequences of events that have identical mask values. The interaction of a process and a controller is depicted in Fig. 7.

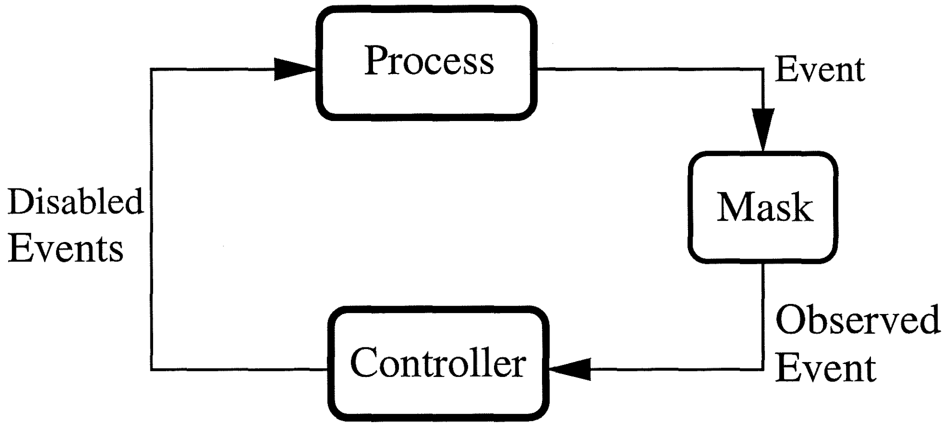


FIG. 7 – *A DES under partial observation*

Similarly, we can get a synchronous composition based observation-compatible controller C, such that $L(P||_{\Sigma_o}C) \subseteq L(S)$ or $L_m(P||_{\Sigma_o}C) \subseteq L_m(S)$.

1.3.3 Synthesis of a Controller

The SCT has given us a means to get a synchronous composition based controller. But the existence of a controller requires that one or more of the properties of prefix-closure, relative-closure, controllability, observability, or normality are satisfied by the desired behaviors in accordance with specifications [6]. If the conditions hold, then we can synthesize a controller by using the language K, that represents the desired behavior (specification), such that $L(P \parallel C) = K$. If the conditions fail to hold, then we can use a maximal sublanguage or a minimal superlanguage K' of K that satisfied the required properties to synthesize a controller called the minimally restrictive or maximally permissive controller such that $L(P \parallel C) = K'$.

In the SCT, Ramadge and Wonham provided an algorithm to compute a unique sublanguage of K, called the supermal sublanguage and denoted $sup(K)$, that satisfies one or more of the properties of prefix-closure, relative-closure, and controllability. The algorithm is also applied in MELODIES synthesis model.

1.3.4 Extension of SCT Used in MELODIES

The conventional framework of SCT is not directly applied to MELODIES. Instead a new framework developed within the framework of conventional SCT for DESs based on [2, 13] is applied. For simplicity, the former is called conventional SCT and the latter extension SCT, respectively. The main differences between the extension SCT and the conventional SCT are as follows.

1. Modeling formalism – In the extension SCT, the process to be controlled is described not only in terms of controllable active components but also in terms of uncontrollable passive components by using timed transition graphs and algebraic specifications, respectively [13]. In conventional SCT, only active components are used to describe the process to be controlled.

2. Synthesis of controllers – In the extension SCT, in addition to Ramadge and Wonham synthesis method, which is used in the conventional SCT, a synthesis method is used that automatically derives controllers for timed DESs with non-terminating behavior modeled by timed transition graphs and specifications of control requirements or constraints expressed by metric temporal logic formulas [2].

Since the representation of a DES project in our GUI design is based on the modeling formalism, more details of the modeling formalism are given in the rest of the section. Details of the new synthesis method, which is used in MELODIES synthesis model, can be found in [2].

As mentioned above, a process is described by using active components and passive components. Active components and passive components are expressed by timed transition graphs (TTGs) and abstract data types (ADTs), respectively. A factory example, which is given in [13], is used to illustrate how TTGs and ADTs are used to describe active components and passive components in a process. The factory is depicted in Fig. 8. It includes p' machines (producer machines) on the left, a table in the middle, and $p-p'$ machines (consumer machines) on the right. In a process cycle, a producer machine takes components from its own basket, produces a part, and puts the part on the table. A consumer machine removes a part from the table, performs some operations, and puts the part in its own basket. The machines may fail from time to time. In the example, both p' producer machines and $p-p'$ consumer machines are active components, and the table is a passive component. The behavior of active components is described by the TTG in Fig. 9, which includes three states: I_i (machine M_i is idle), W_i (machine M_i is working), and D_i (machine M_i is down). The arcs are labeled a_i (machine M_i starts a cycle), b_i (machine M_i completes a cycle), c_i (machine M_i breaks down), and d_i (machine M_i is repaired). Event a_i and d_i are controllable, while event b_i and c_i are not. Every event has a duration of one time unit.

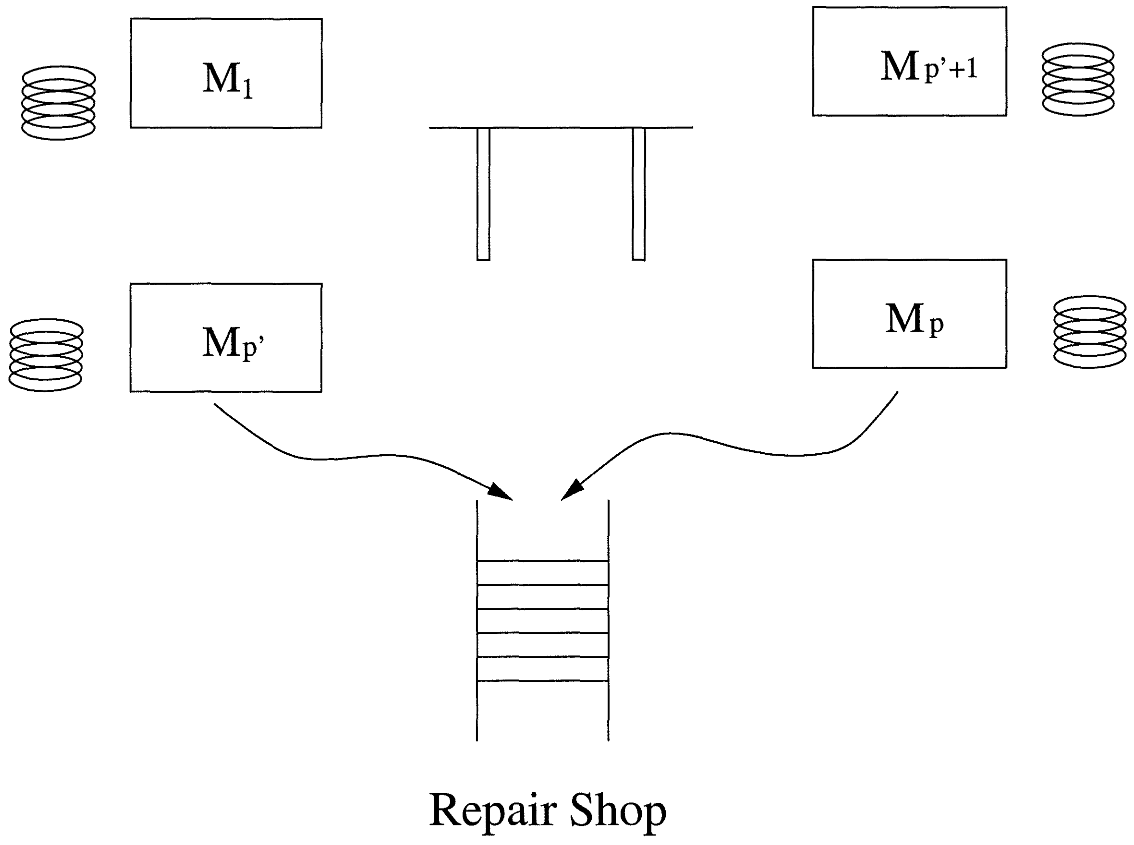


FIG. 8 – *The process to be controlled*

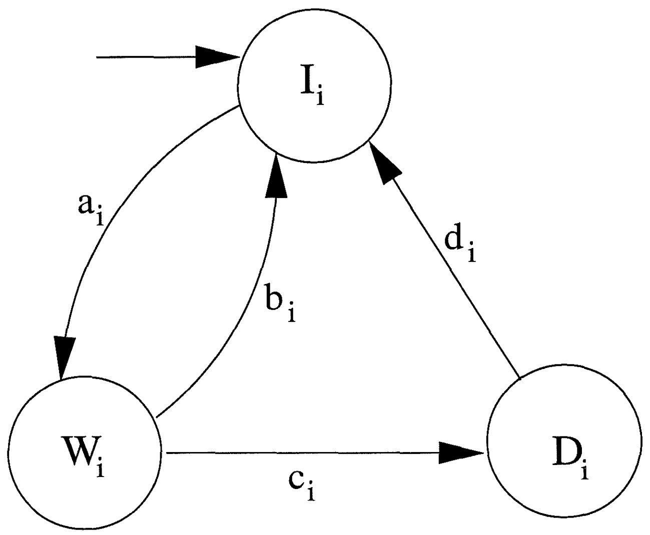


FIG. 9 – *The behavior of machine M_i*

The passive component table is described by an object b of type *Buffer*. The definition of ADT *Buffer* is given in Fig. 10.

$Buffer(capacity \in nat) :=$
  `import`: $bool$, nat
  `hidden sorts`: $buffer$
`operations`:
 New: $\rightarrow buffer$
 Put: $buffer \rightarrow buffer$
 Get: $buffer \rightarrow buffer$
 $IsEmpty$: $buffer \rightarrow bool$
 $IsFull$: $buffer \rightarrow bool$
 $Size$: $buffer \rightarrow nat$
`equations`: $b \in buffer$, $n \in nat$
 $Get(New) = ERROR$
 $Get(Put(b)) = b$
 $Put(Put^{capacity}(New)) = ERROR$
 $IsEmpty(New) = TRUE$
 $IsEmpty(Put(b)) = FALSE$
 $IsFull(New) = FALSE$
 $IsFull(Put^{n}(New)) = FALSE \ (n \neq capacity)$
 $IsFull(Put^{capacity}(New)) = TRUE$
 $Size(New) = 0$
 $Size(Put(b)) = Size(b) + 1$

FIG. 10 – *The abstract data type Buffer*

Now we give the composition of TTG and ADT. From [13], it is concluded that:

1. TTG is composed of states (name, initial, marked, duration, cost), events (name, controllable, observable, forceable, duration, sensor cost, actuator cost, operation), and transitions (source state, event, destination state).
2. ADT is composed of import sorts, hidden sorts, parameters (names and types), variables (names and types), operations, and equations.

Process, TTG, and ADT are discussed above. In MELODIES, the control of a DES is treated as a project and from the application's point of view a mask function is included into a process. So, a project includes processes, constraints, and controllers. The overall view of a project is listed below.

1. A process is composed of active components (TTGs), passive components (ADTs), and mask functions. The number of active components, passive components, and mask functions in a process can be zero to many. More specifically, when the number of passive components in a process equals zero, the process is only composed of active components. This situation falls into the conventional SCT domain. The number of processes in a project can be zero to many.
2. A constraint is composed of TTGs and/or metric temporal logic formulas. The number of both TTGs and formulas can be zero to many. Constraints are used to describe specifications of control requirements. The number of constraints in a project can be zero to many.
3. A controller is composed of TTGs and feedback functions. The number of both TTGs and feedback functions can be zero to many. The number of controllers in a project can be zero to many.

In this chapter, we have discussed basic theories in MELODIES. Next, we turn to programming aspects of the GUI design.

Chapter 2

GUI Programming in Qt

2.1 Introduction

Qt is a C++ toolkit for cross-platform GUI and application development. It is available for Windows, MacOS, Unix, including Linux, and many embedded systems. In addition to the Qt C++ class library, the toolkit includes tools to make writing applications fast and straightforward. With its mutiple platform support, Qt has always been an ideal C++ toolkit for building cross-platform applications. For the most part, you just need to recompile your Qt application to run on Unix, Windows, or embedded systems. Actually, the Qt application programming interface (API) is virtually identical for all desktop platforms, hence you can create a single source-code base and should be able to compile it without modification for all the supported operating systems. As compared with other two widely used GUI toolkits, Java JFC/Swing and Visual C++/MFC, Qt is faster than the former and supports more platforms than the latter. That is the main reasons why we use it to design the GUI. The two following features also make Qt more suitable for GUI programming [4]:

1. Object orientation and component support – Qt has a modular design and a strong focus on reusable software components. It allows objects to cooperate without any

knowledge of each other. A widget does not need to know its context and communicates with the outside world through signals and slots.

2. Full widget set – The widget is the basic built-in component of Qt. Qt contains all the ready-to-use widgets you need to create professional-looking user interfaces for your software.

Some concepts, definitions, results, and texts are based on or borrowed from [4] in the following sections.

2.2 Widgets

Widgets are visual elements that can be used to create user interfaces. Qt library contains a rich set of widgets. Buttons, menus, scroll bars, message boxes, and application windows are all examples of widgets. Such a rich set of widgets caters for most situations. You can just use these widgets to assemble graphic user interfaces. Qt's widgets are also flexible and easy to subclass for special requirements and some custom widgets are created in the GUI design. Widgets are instances of `QWidget` or one of its subclasses.

In contrast with other GUI toolkits, Qt's widgets are not arbitrarily divided between *controls* and *containers*; all widgets can be used both as *controls* and as *containers.* A widget may contain any number of child widgets. Child widgets are shown within the parent widget's area. A widget with no parent is a top-level widget (a *window*), which is similar to the frame or window in Java JFC/Swing or Visual C++/MFC. Qt imposes no arbitrary limitations on widgets. Any widget can be a top-level widget; any widget can be a child of any other widget. The position of child widgets within the parent's area can be organized automatically using layout managers, or manually if preferred. When a parent widget is disabled, hidden or deleted, the same action is applied to all its child widgets recursively.

In Qt, the predefined widgets can be classified as follows:

1. Abstract widgets – Abstract widget classes usable through subclassing.
2. Advanced widgets – Advanced GUI widgets such as list views and list view items.
3. Basic widgets – Basic GUI widgets such as buttons and combo boxes.
4. Main window and related classes – Everything you need for a typical modern main application window, including menus, toolbars, and workspaces.
5. Standard dialogs – Ready-made dialogs for file, font, or color selection and more.

2.3 Signals and Slots

The signal/slot mechanism, a unique feature of Qt, is used for communication between objects. In GUI programming a change in one widget often needs to be notified to another widget. For example, we want to add a new item in a `QListView` through a dialog. After inputing the item's name in the input field and clicking the accepted button (usually labeled «Ok») in the dialog, we need to notify the `QListView` that a new item is added. More generally, any kind of objects should be able to communicate with one another.

Older toolkits use callbacks to process such communication. A callback is a pointer to a function. If you want a processing function to notify you about some event, you pass a pointer to the callback to the processing function. The processing function then calls the callback when appropriate. But callbacks have two fundamental drawbacks:

1. They are not type-safe – It is not certain that the processing function will call the callback with the correct arguments.
2. The processing function and the callbacks are tightly coupled – The processing function must know which callback to call.

Qt uses an alternative to the callback technique instead. That is signal/slot mechanism.

Here is an example about how to define signals and slots in a class:

```
class myState: public QObject
 {
  Q_OBJECT
  public:
     myState();
     int getState() const { return state; }
  public slots:
     void setState( int );
  signals:
     void stateChanged( int );
  private:
     int state;
 };
```

It is necessary to add `Q_OBJECT` in the declaration in any class that contains signals and slots. Signals are automatically generated by `moc`, the meta object compiler, and must not be implemented in the `.cpp` file. They can never have return types (i.e. use `void`). Slots must be implemented in the `.cpp` file. All classes that inherit from `QObject` or one of its subclasses (e.g. `QWidget`) can contain signals and slots.

A signal is emitted when a particular event occurs. You can emit a signal from an object by using `emit signal(arguments)`. The `emit signal(arguments)` can be put into any methods or slots implementation. In the example above, we can do it like this:

```
void myState::setState( int s )
 {
   if(s!=state){
      state=s;
      emit stateChanged(s);
     }
 }
```

Signals use arguments to pass information to outside world. Only the class that defines a signal and its subclasses can emit the signal. When a signal is emitted, the slots connected to it are executed immediately, just like a normal function call. The signal/slot mechanism is totally independent of any GUI event loop. The `emit` command will return when all slots have returned.

A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them. Slot's arguments cannot have default values and, like signals, it is rarely wise to use your own custom types for slot arguments.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt. As a normal class method, a slot can be declared as public, protected, private, and virtual.

1. A public slot defined in a class can be connected to signals defined in any class. This is very useful for component programming. You create objects that know nothing about each other, connect their signals and slots so that information is passed correctly.
2. A protected slot defined in a class can only be connected to the signals defined in the class and its subclasses. This is intended for slots that are part of the class's

implementation rather than its interface to the rest of the world.

3. A private slot can only connect to the signals defined in the class itself.
4. Defining slots to be virtual is quite useful in practice.

A slot's access right determines who can connect to it.

Signals and slots are brought together at runtime by the `QObject::connect()` method, one of the central classes in Qt:

```
connect(object1, SIGNAL(), object2, SLOT());
```

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you want. We can treat the signal/slot relationship as a many-to-many relationship. It is even possible to connect a signal directly to another signal (This will emit the second signal immediately whenever the first one is emitted). If several slots are connected to one signal, the slots will be executed one after the other, in an arbitrary order, when the signal is emitted. Fig. 11 represents a general view of signal and slot connections.

Qt's widgets have many predefined signals and slots, but you can also add your own signals and slots in order to handle special situations.

The signal/slot mechanism is type-safe. The signature of a signal must match the signature of the receiving slot (in fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments). Since the signatures are compatible, the compiler can detect type mismatches. Signals and slots are loosely coupled. An object which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots technique ensures that a slot will be called with connected signal's parameters at the right time.

The signal/slot mechanism is efficient, but not as fast as «real» callbacks. Signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant.

The signal/slot mechanism allows for component-based programming. Objects can

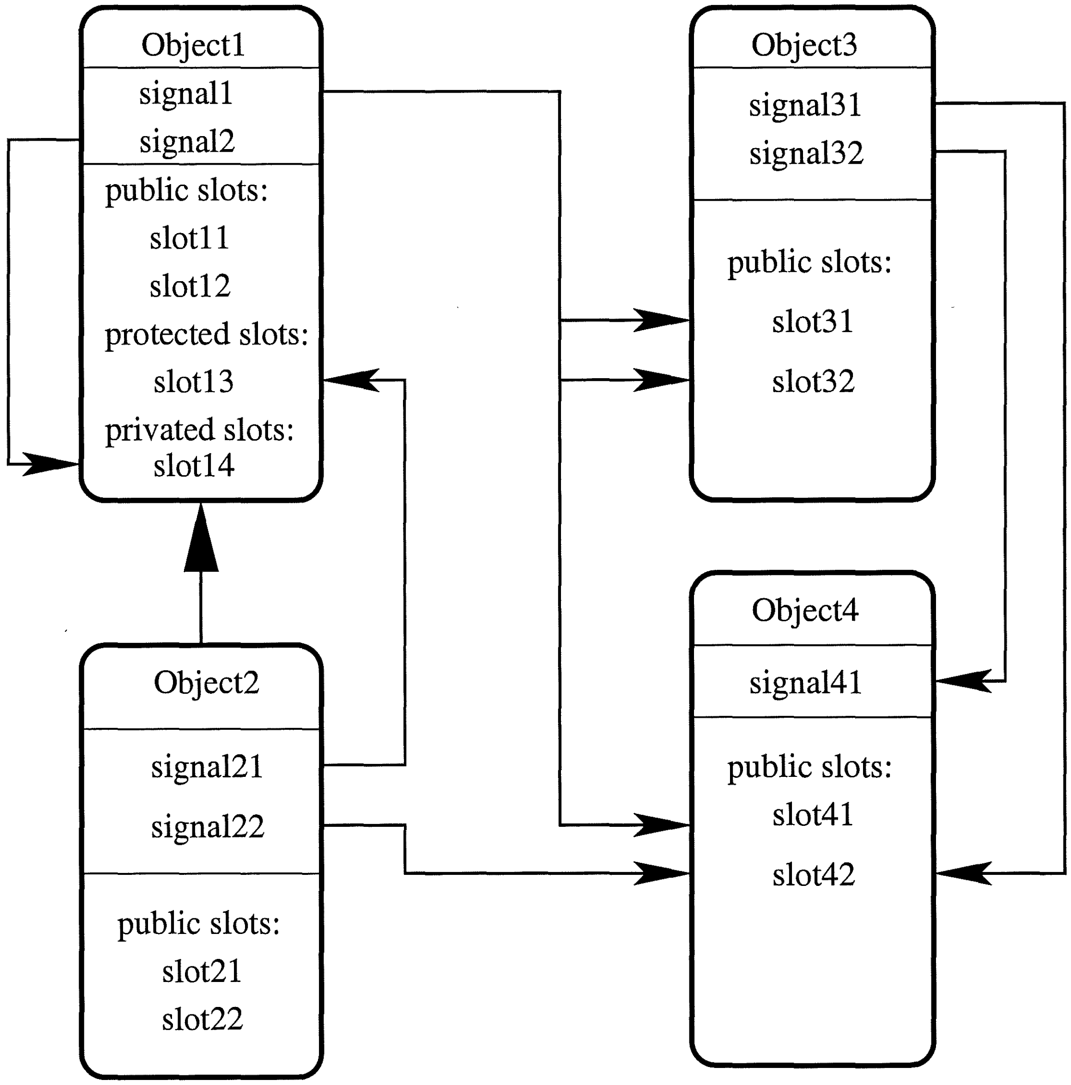


FIG. 11 – *A general view of signal and slot connections*

define signals that they emit when an event occurs. Objects can also define slots. The only way that an object communicates with other objects is through signals and slots. This makes information encapsulation and ensures that the object can be used as a software component. Together, signals and slots make up a powerful component programming technique.

2.4 Look of GUI Programming in Qt

GUI design incorporates artistic and technological concerns. The artistic aspect is mainly associated with look and feel, and layout management. The main technological concern in Qt is how the widgets communicate to fulfill some tasks.

2.4.1 Look and Feel of Widgets and Layout Management

Qt provides many classes used to customize a widget's style, fonts, and colors. Since Qt is cross-platform, it includes different looks and feels corresponding to different platforms it supports, such as Microsoft Window-like look and feel, Motif look and feel, and Mac/Platinum look and feel. On each platform, it has the default look and feel. But you can choose any existing look and feel of Qt for your application. On Solaris operating system, for example, the default look and feel of a widget is the Motif look and feel; you can change the look and feel of the widget to the Microsoft Window-like look and feel using `QWidget::QStyle(QWindowsStyle)`. You can also change fonts and colors using corresponding classes and methods. In addition, Qt provides several kinds of layout for geometry management. Among them, the following are most used.

1. `QBoxLayout` – Lines up child widgets horizontally or vertically.
2. `QGridLayout` – Lays out widgets in a grid.
3. `QHBoxLayout` – Lines up widgets horizontally.

4. `QVBoxLayout` – Lines up widgets vertically.

With Qt provided classes, we can customize an application's appearance and style.

2.4.2 Using Signals and Slots to Communicate between Widgets

We already know that signals and slots are used to communicate between widgets. More details need to be discussed through a simple example. The example is typical in GUI design: Users use a custom dialog box to update some data in an application. The dialog box is composed of some input fields and a push button labeled «Ok». When an user inputs data into the input fields and clicks on the button «Ok», the data in the application are updated accordingly and the dialog is closed. It is natural to use signals and slots to communicate between the dialog and the application. But things get complicated when there are many data that need to be updated simultaneously and especially when the updated data are not of simple data type, or the updated data are user-defined classes which include many instances that need to be updated. Even though signals and slots can take any number of arguments of any type, it is not efficient to define a signal and a slot with too many arguments or with user-defined data type, especially in a complicated structure. So, we must compromise which information need to be communicate with signals and slots in above situations. Some information may be passed in traditional ways. Taking it into account, we give three solutions for the example.

1. Update the data in the dialog class. This requires to expose internal structure of the application class to the dialog class or declare an instance of the application class in the dialog class. In the dialog class, a slot is defined to do the update and it is invoked by a signal:

```
public slot:
    void update(){
    //update by calling the application's corresponding methods
    }

  //Button OK connects its clicked() signal to dialog's update()
    slot, so that the update() is called when the button is clicked
  connect(Ok, SIGNAL(clicked()), this, SLOT(update()));
```

2. Update the data in the application. This requires to expose internal structure of the dialog class to the application or declare an instance of the dialog class in the application class. In the application class, a slot is defined to do the update and it is invoked by a dialog's signal:

```
  public slot:
    void update(){
     //do the update here by calling dialog's corresponding methods
    }

  //Button OK connects its clicked() signal to application's
    update() slot, and the update() is called when the button
    is clicked
  connect(Ok, SIGNAL(clicked()), this, SLOT(update()));
```

3. Using signals and slots to pass and receive the updated information, both the dialog class and the application class do not expose their internal structure to each other. In the dialog class, a protected slot updating() is defined to emit the do-update() signal:

```
  protected slots:
    void updating(){//set new information to the signal's args and
                        emits the signal
```

```
      ...
    emit do-update(...)
    }
  signals:
    do-update(arg1,arg2,...);//send information through args

  //Button OK connects its clicked() signal to dialog's
    updating() slot, so that the updating() is called
    when the button is clicked
  connect(Ok, SIGNAL(clicked()), this, SLOT(updating());
```

In the application class, a slot called update() is defined to receive the updated information that are passed from the dialog class:

```
  public slots:
    void update(arg1,arg2,...){//receive information through args
    //do the update here
    }
```

The dialog class connects its do-update() signal to application's update() slot and the update() is called when the dialog emits its do-update() signal. The connection happens outside both the dialog class and the application class.

```
connect(dialog, SIGNAL(do-update()), application, SLOT(update());
```

In the solution, the predefined signal `clicked()` of the button does not contain any arguments and hence cannot pass the needed information through arguments. So, we define a signal with arguments containing updated information and a protected (or private) slot that emits the signal. But we still use signal `clicked()` to trigger the whole communication. This way extends a solution while keeping the initial functionality in place.

Usually the third solution is recommended. It is based on the component-based programming concept of Qt. So, we should take this approach when it is possible. Unfortunately, if a class involved in communication contains many data needed to update and has a complicated data structure to organize these data, we must compromise in choosing those solutions. For example we use the `Repository` class in our GUI. The `Repository` class is used to contain all the GUI data which need to be updated. Too many slot calls will decrease the speed and efficiency of the application. So, we take the first solution instead of third one in the GUI design.

Chapter 3

Using XML as Data Transmission Format

3.1 Introduction

In this chapter, we introduce XML as a data transmission format in our GUI I/O module. Some concepts and texts are borrowed from the references as indicated.

The *Extensible Markup Language* (XML) specification was proposed by the *World Wide Web Consortium* (W3C), the organization that develops and recommends Web specifications and standards. XML is derived from the *Standard Generalized Markup Language* (SGML) and provides a simple, very flexible text format for representing structured information on the Web [9]. XML operates on two main levels: first, it provides syntax for document markup; and second, it provides syntax for declaring the structures of documents. Although XML is originally designed to meet needs of the Web, it has been used as a standard data transmission format for a wide variety of applications, not just Web applications.

XML allows developers to set standards for defining the information that should appear in a document, and in which sequence. XML, in combination with other standards, makes it possible to define the content of a document separately from its formatting, making it easy to reuse that content in other applications or in other presentation environments. Most important, XML provides a basic syntax that can be used to share information between different kinds of computers, different applications, and different organizations without the need to pass through many layers of conversion [10].

XML provides a simple format that is flexible enough to accommodate widely diverse needs. Even developers that perform tasks on different types of applications with different interfaces and different data structures can share XML formats and tools for parsing those formats into data structures that applications can use. XML offers many advantages, including [7]:

- simplicity,
- extensibility,
- interoperability, and
- openness.

In the following sections, we first discuss the XML design and processing issues, and then the XML design of the GUI data in details.

3.2 XML Document Design

The XML document design is mainly concerned with two aspects: one is the format of the XML document, and the other is its structure. The first point is related to well-formated document and the second to *Document Type Definition* (DTD) of the XML document.

3.2.1 Well-Formated Document

One of the basic requirements of XML document design is to produce well-formated documents. A well-formated document respects the following rules:

- XML elements must have a start and a corresponding end tag,
- tags are case sensitive,
- XML elements must be properly nested,
- XML elements must have a root tag,
- XML attribute values must be quoted (single or double quotes).

Obviously, these rules ensure that a well-formated document obeys the general XML syntax. In fact, you must guarantee that a XML document is well-formated if you want it to work.

3.2.2 Modeling Data with DTD

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. However, the DTD is only one way to build an XML design. There are other means of creating an XML design used in different contexts such as *XML Schema* or *XML Data-Reduced* (XDR). We focus on DTDs because the DTD is the most essential form of XML design and is used in our XML document design for the GUI data. A DTD describes how an XML document is constructed, which XML elements and their attributes are allowed, and how they put together to form a valid piece of XML. The DTD states the following rules [11]:

- the element types allowed within the document,
- the characteristics of each element type, including its attributes and the content,
- the notations that can be encountered within the document,
- the entities that can be encountered within the document.

A DTD of an XML document can be internal or external. The following example illustrates the two situations:

1. Internal DTD. The DTD is included in the document type declaration.

   ```
   <?XML version="1.0" standalone="yes"?>
   <!DOCTYPE greeting [
   <!ELEMENT greeting (#PCDATA)>]>
   <greeting>Hello World</greeting>
   ```

   The internal DTD is declared between the square brackets ([ ]).

   `#PCDATA` stands for parsed character data.

2. External DTD. Here, the DTD is given in the file referenced by the *Uniform Resource Identifier* (URL) *hello.dtd*.

   ```
   <?XML version="1.0" standalone="no"?>
   <!DOCTYPE greeting SYSTEM "hello.dtd">
   <greeting>Hello World</greeting>
   ```

   In the *hello.dtd*, there is an element declaration:

   ```
   <!ELEMENT greeting (#PCDATA)>
   ```

Here we just give a simple example to illustrate the DTD, the details of a DTD are given in the later part. You can also have both internal and external DTD in the document type declarations. This approach allows to define general declaration rules in an external DTD and the specific features of a document in an internal DTD. It is important to note that the declarations in the internal DTD are read first and take precedence over declarations in the external DTD [11].

With object-oriented terminology, a DTD can be considered as a class, and an XML document as an instance of that class [1].

3.2.3 Validation of XML Documents

A well-formated XML document provides a legal, convenient format for XML information. In contrast to well-formated XML documents, valid XML documents not only have a DTD, but must follow the rules of the DTD.

A well-formated XML can work without a DTD defined. However, not much information will actually be created and managed as well-formated XML, except in the simplest applications. When authoring XML documents, it is actually easier to work with a DTD than without one. More important, a DTD is necessary for a valid document. As mentioned above, a valid XML document has a DTD and each element must follow the rules of the DTD. XML's validity rules are an additional layer on top of the rules for being well-formated [10]. Every valid XML document must be well-formated as well.

Whether an application checks that an XML document is valid as well as being well-formated is determined by the XML parser it uses. If a non-validating XML parser is used, it just checks that the XML document is well-formated. If a validating XML parser is used, it reports any validity error in addition to checking that the XML document is well-formated.

3.3 XML Processing with Document Object Model

After discussing XML design, we turn to the XML processing. Parsing an XML document is the first step in processing it. To achieve this goal, an XML parser is needed. An XML parser is the most important part to any XML-based application. It takes an XML document as input and checks whether it is well-formated or valid. The result from an XML document being parsed is typically a data structure that can be manipulated and handled by other XML tools or APIs. There are two basic APIs to XML parsers: *Simple API for XML* (SAX), which supports the event-driven approach, and *Document Object Model* (DOM), which represents the result of parsing in a tree-like structure. As our choice, we will discuss DOM in details.

3.3.1 Document Object Model

The *Document Object Model* has its origins in the *World Wide Web Consortium.* The DOM is designed to represent the content and model of documents across all programming languages and tools. Hence, DOM is a cross-platform and cross-language specification. The DOM is organized as levels instead of versions [9, 11]:

- Level 0 is a restatement of the JavaScript syntax that was used to manipulate HTML documents.
- Level 1 allows access to all parts of the XML document. It does not allow access to the DTD associated with the document.
- Level 2 adds upon Level 1 by supplying modules and options aimed at specific content models, such as XML, HTML, and *Cascading Style Sheets* (CSS). It allows access to the DTD and actually provides APIs that can be shared by different applications.
- Level 3 adds a standard `Load and Save` package so that it will be possible to write completely implementation independent DOM programs. The saving part is based on the `DOMWriter` interface. `DOMWriter` is more powerful than `XMLSerializer`.

The following discussion is mainly based on DOM level 1 and level 2.

At the core of DOM is a tree model. It provides a complete in-memory representation of an XML document. A document is provided in a tree format, and all of this is built upon the DOM interfaces. Because of this model, all DOM structures can be treated either as their generic type, *Node*, or as their specific type (*Element*, *Attr*, etc.). A typical XML document might get a tree model like Fig. 12 [10]. Fig. 13 illustrates the diagram of XML processing with DOM.

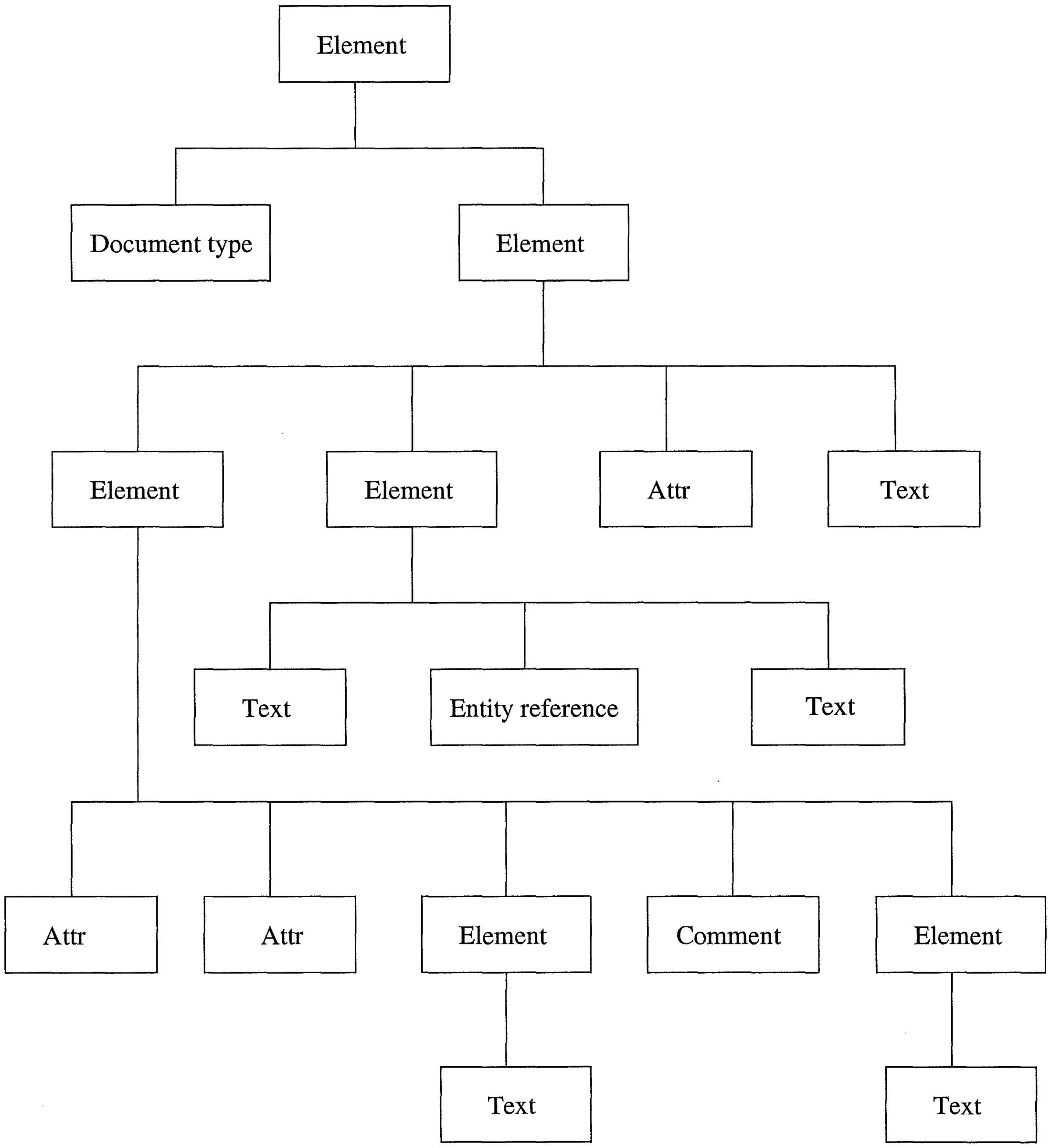


FIG. 12 – *DOM structure representing a typical XML document*

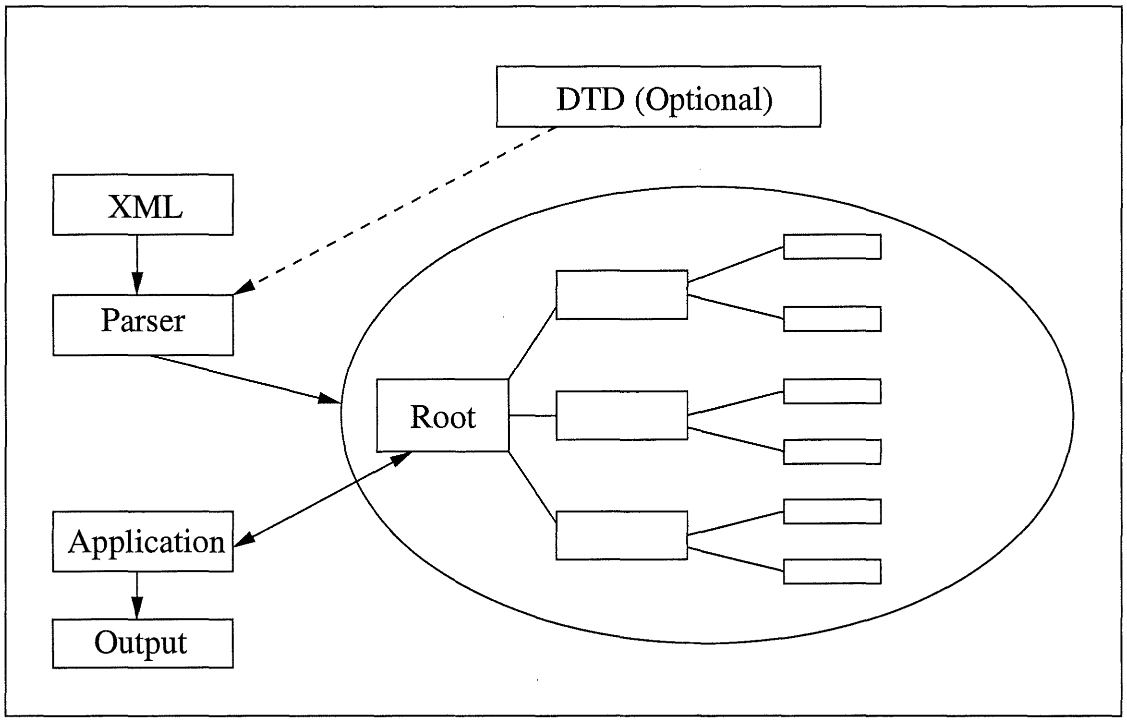


FIG. 13 – *XML processing with DOM*

3.3.2 DOM Interface

DOM defines several interfaces. The following interfaces are used in our application by binding to Qt.

Document This interface returns information about the main document itself that has been loaded into the object. The one that is most used is the `documentElement` attribute that creates a node of the root element of the document. This is a starting point to traverse the tree to get a node. The document interface has methods. The most important method is `getElementsByTagName()` which will return a `NodeList` of all the documents' elements of the name passed to it as a parameter. This is the method to use if you want to access an element by name.

Node It is the key interface of DOM. Everything in a document can be treated as a node such as elements, comments, the document itself, etc. This interface contains several general attributes and methods for manipulating any kind of node. Here are the read-only attributes of the node interface:

- `nodeName`,
- `nodeValue`,
- `nodeType`,
- `parentNode`,
- `childNodes`,
- `firstChild`,
- `lastChild`,
- `previousSibling`,
- `nextSibling`,
- `attributes`.

Here are the most used methods of the node interface:

- `parentNode()` – This method is used to access the parent node.

- `childNode()` – This method is used to access the first child of the node. If no such node exists, null is returned.
- `previousSibling()` – This method is used to access the previous sibling node. If no such node exists, null is returned.
- `nextSibling()` – This method is used to access the next sibling node. If no such node exists, null is returned.
- `insertBefore()` – This method is used to insert a node.
- `replaceChild()` – This method is used to replace a node.
- `removeChild()` – This method is used to remove a node.
- `appendChild()` – This method is used to append a node.

NodeList This is an ordered collection of nodes that can be accessed by using an index. You can use the item method of the `NodeList` interface to access the node object in the `NodeList`.

CharacterData This is an interface that deals with the character data or text of the document.

Attr This interface deals with the XML attributes of a node.

Element Most of the nodes of an XML document will be elements. This interface has properties and methods for dealing with elements and their XML attributes.

Text This deals with the text content of an element. Most of the required functionalities are handled by the `CharacterData` interface.

3.3.3 XML Module in Qt

Qt's XML module provides a SAX parser and a DOM Level 2 parser, both of which read well-formated XML. Many Qt applications use XML format to store their persistent data. The DOM parser reads the entire file into a tree structure in memory that can be traversed in some way [4]. Qt provides a series of classes that implement most interfaces of DOM. Qt uses a prefix `QDom` plus the name of a DOM interface as the corresponding class name that implements the DOM interface. For example, `QDomNode` class gives the implementation of `Node` interface. It is convenient to use these classes for processing XML documents in Qt.

3.4 XML Design for the GUI Data

We have discussed all the parts of building XML document and its processing. Now we will begin to design the XML and DTD for the GUI data. According to the requirements of the GUI project, we need to build the following XML documents and related DTDs:

- project list,
- timed transition graph (ttg) data,
- abstract data type (adt) data,
- mask function data,
- formula data,
- feedback function data.

Here, we only give the XML and DTD design of the project list in details. The project list contains a collection of projects, processes, constraints, controllers, components, ttgs, adts, mask functions, formulas, and feedback functions. Each project entry in the document is contained within a project element and has a variety of different pieces of information that need to be represented with either elements or attributes. Let us start with a project entry. From Chapter 1, the hierarchy of a project is shown in Fig. 14.

From the project hierarchy, it is easy to transfer the child nodes into corresponding elements that are included in a project entry.

A project element contains three child elements: process, constraint, and controller.

A process element contains two child elements: component and mask function, while a component element contains one child element either ttg or adt.

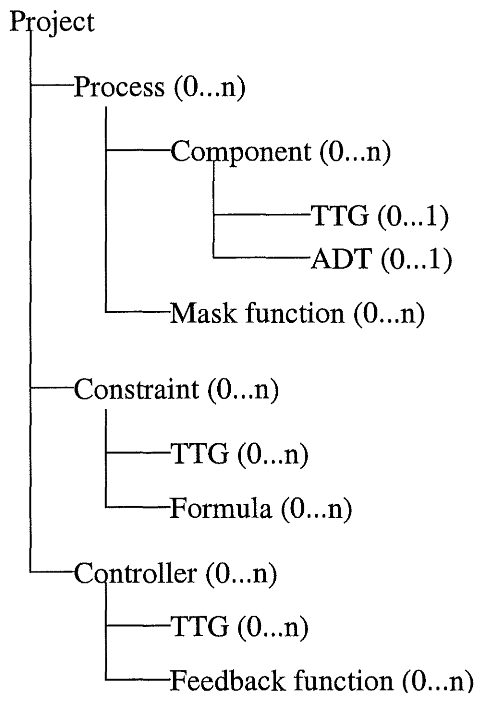


FIG. 14 – *Project hierarchy*

A constraint element contains two child elements: ttg and formula.

A controller element contains two child elements: ttg and feedback function.

According to the requirements, the GUI project should provide the following additional informations:

– project name;

– process name, creation date, update date, and comment;

– constraint name, creation date, update date, and comment;

– controller name, creation date, update date, and comment;

– component name, creation date, update date, and comment;

– ttg name, creation date, update date, and comment;

– adt name, creation date, update date, and comment;

– mask function name, creation date, update date, and comment;

– formula name, creation date, update date, and comment;

– feedback function name, creation date, update date, and comment.

After gathering all the information of a project entry, we need to determine which of the above additional informations is represented as an element or an attribute. There is no general rule for the determination. We use the following rule to organize the information: all the informations in a project that associate with other I/O data (such as ttg data, adt data, mask function data) are represented as elements, otherwise they are represented as attributes.

According to this rule, the names of ttg, adt, mask function, formula, and feedback function are represented as elements, the rest of the information is represented as attributes.

Next step in XML design is to build a DTD based on the information. A DTD for a project entry is shown as follows:

```
<!ELEMENT project (process,constraint,controller)*>
<!ATTLIST  project name ID  #REQUIRED>
<!ELEMENT  process (component,maskfun)*>
<!ATTLIST  process
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT constraint (ttg,formula)*>
<!ATTLIST  constraint
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT controller (ttg,feedbackfun)*>
<!ATTLIST  controller
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT component (ttg|adt)?>
<!ATTLIST  component
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  ttg  (#PCDATA)>
<!ATTLIST  ttg
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  adt  (#PCDATA)>
<!ATTLIST  adt
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
```

```
>
<!ELEMENT  maskfun  (#PCDATA)>
<!ATTLIST  maskfun
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  formula  (#PCDATA)>
<!ATTLIST  formula
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  feedbackfun (#PCDATA)>
<!ATTLIST  feedbackfun
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
```

Each of these elements must be defined within the DTD using the `ELEMENT` keyword. Each declaration of an element has two parts: the element name such as process, and its content model, which defines what the element can contain. In the declaration of elements:

- parentheses (()) enclose a sequence or choice group of child elements;
- comma (,) separates the items in a sequence and establishes the order in which they must appear;
- pipe (|) separates the items in a choice group of alternatives;

- question mark (?) indicates that a child element must appear exactly once or not at all;
- asterisk (*) indicates that a child element can appear any number of times;
- `#PCDATA` stands for parsed character data.

Each element must have its own attribute list declaration (using the `ATTLIST` keyword). Each attribute declaration consists of an identifier associating it with a specific element, the attribute's name (such as `name`, `creation`), and its type (such as `ID`, `CDATA`). In the attribute declaration:

- `ID` stands for a unique identifier;
- `CDATA` stands for character data;
- `#REQUIRED` indicates that the attribute is required.

Based on the DTD, a project entry of the factory example in Chapter 1 might look like the following:

```
<project name="Factory_Project">
  <process name="Factory">
   <component name="Producer" comment="active component">
     <ttg comment="behavior of machine M1">M1</ttg>
   </component>
   <component name="Consumer" comment="active component">
     <ttg comment="behavior of machine M2">M2</ttg>
   </component>
   <component name="Table" comment="passive component">
     <adt>Buffer</adt>
   </component>
  </process>
  <constraint name="Factory_Constraint">
```

```
    <formula comment="metric temporal logic formula">f</formula>
  </constraint>
</project>
```

After finishing the design of the project, we easily get the DTD for project list by adding the following declaration as first declaration in the DTD for the project:

```
<!ELEMENT project-list (project, process, constraint, controller,
  component, ttg, adt, maskfun, formula, feedbackfun)*>
```

In most situations, project-list only contains projects, while a project contains processes, constrains, and controllers. But there still exist some situations where an existing process does not belong to any project for the time being. We call the process «unused process». The unused process may be used by a project later. There exist unused constraints, controllers, and so on. Taking this into account, we use

```
<!ELEMENT project-list (project, process, constraint, controller,
  component, ttg, adt, maskfun, formula, feedbackfun)*>
```

instead of

```
<!ELEMENT project-list (project)*>
```

in the DTD for a project. In the design, we apply an external DTD for all the XML documents related to the GUI data. The external DTD becomes more valuable when creating multiple valid documents of the same class, while the internal DTD is inefficient in this situation because we would have to maintain copies of markup declaration in every document. Other DTDs for ttg data, adt data, mask function data, formula data, and feedback function data can be found in Appendix A.

At last, we briefly describe how to read and write an XML document in Qt. The following piece of code shows the way to read an XML file into a DOM tree:

```
//read the XML file and create DOM tree
QFile xmlFile (fileName);
```

```
if (!xmlFile.open (IO_ReadOnly)) {
    //error processing
}
if (!domTree.setContent (&xmlFile)) {
   //error processing
}
xmlFile.close();
```

After this, we get a DOM instance `domTree` for further processing. Because Qt's XML module does not support DOM level 3 and because we may not want to write DOM to XML for efficiency consideration, we use `QTextStream` to write to XML document directly. The following piece of code shows how to write a project element into an XML document named *project.xml*:

```
QFile f("project.xml");
if(f.open(IO_WriteOnly)) { // file opened successfully
QTextStream t(&f);         // use a text stream
t << "<?xml version="1.0" standalone="no"?>" << endl;
...
t << "<project>" << endl;
// write project items to xml
...
t << "</project>" << endl;
f.close();
}
```

So far we have finished all the preparation. Next we will touch the core part of the GUI design.

Chapter 4

Object-Oriented GUI Design

4.1 Design Concepts

In this chapter, we mainly concentrate on the design and implementation of the GUI. We begin with some design concepts used in the GUI design.

4.1.1 Model-View-Controller (MVC) Design Pattern

Model-View-Controller (MVC) design pattern is a widely used software design pattern that was created by Xerox PARC for Smalltalk-80 in the 1980s [5]. This type of interface has since been borrowed by the developers of the Apple Lisa and Macintosh. More recently, it has become the recommended model for web applications such as Sun's J2EE platform, Microsoft's ASP.Net and many other third party's products. MVC design pattern provides an object-oriented design and object-oriented programming environment for rapid development of applications that involve user interface, data transmission, and presentation. Under this design pattern, an application can be divided into three different functional parts:

1. Model – the object to be examined and/or modified. It manages the state and data

that the application represents. When changes occur in the model, it updates all of its views.

2. View – the user interface which displays information about the model to the user. Any object that needs information about the model needs to be a registered view with the model.
3. Controller – the object that encapsulates interactions with the model and view objects. Usually it is the user interface that manipulates the application.

The works of MVC can be briefly described as follows.

The model base class has associations with various view and controller classes. When changes occur in a base model, the base model sends message to itself to indicate that its state has changed. The message notifies each dependent model that the base model has changed. Each dependent model can then determine when to update itself based on the message and make corresponding update in the view through the controller. In MVC design pattern, different views can be attached to the same model. The views can be assembled into large or complex components as needed. The controllers may be interchanged, allowing different user interactions.

In general, MVC design pattern brings better organization and code reuse in software development. Nevertheless, MVC design pattern requires significant works and introduces more complexity on planning. So, in some situations, especially for small applications and even many medium-size ones, the extra work in taking the time to design a complex architecture may not be worth the payoff. In our design, we take a more practical approach as it is used in JSP Model 2 Architecture.

4.1.2 JSP Model 2 Architecture

JSP Model 2 Architecture is a server-side implementation of the MVC design pattern using Servlets and JavaServer Pages (JSP). Fig. 15 illustrates JSP Model 2 Architecture

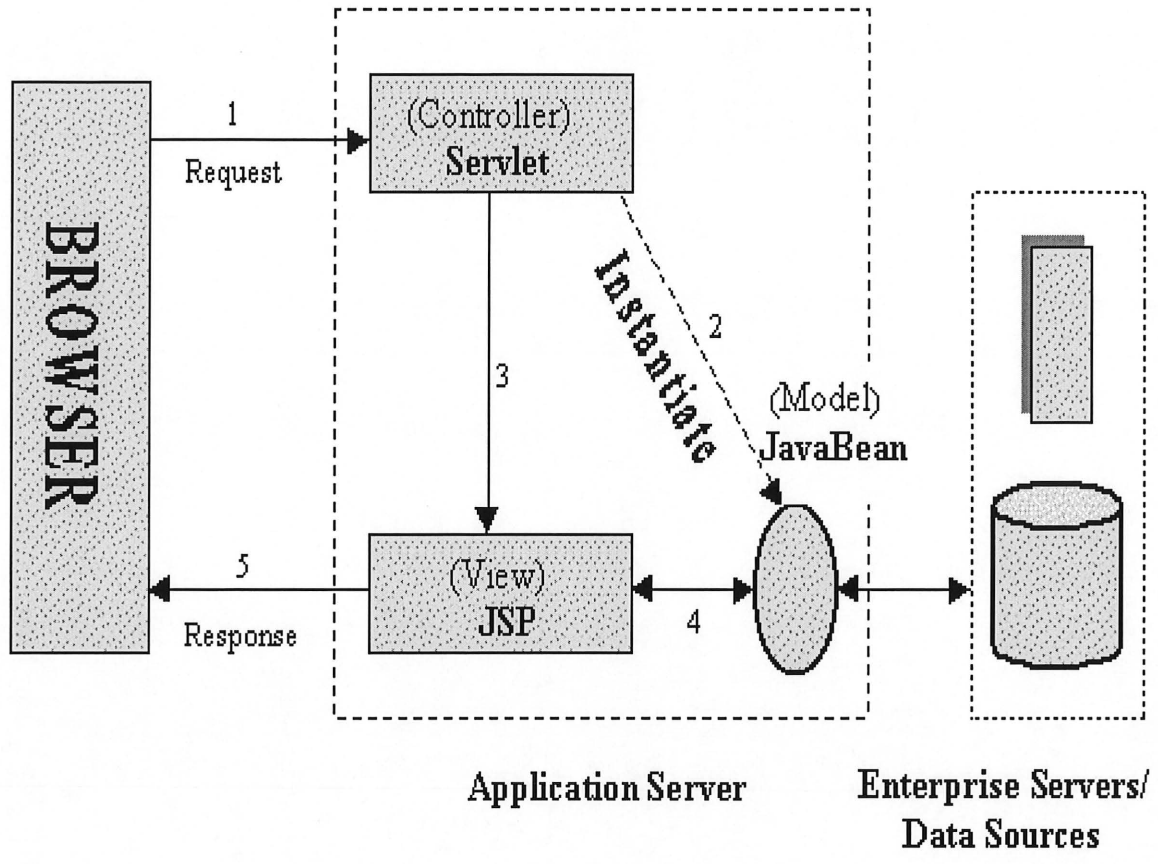


FIG. 15 – *JSP Model 2 Architecture*

[3]. In a server-side application, we usually classify the units of the application as business logic, presentation, and request processing. Business logic is the term used for the manipulation of an application's data, such as customer, product, and other information. Presentation refers to how the application data is displayed to the user, for example, position, font, and size. And finally, requesting processing is what ties the business logic and presentation parts together. In MVC terms, the model corresponds to business logic and data, the view to the presentation, and the controller to the requesting processing. In JSP Model 2 Architecture, servlets control the flow of the web application and delegate the business logic to either JavaBeans or EJBs, while JSP pages take care of producing

the HTML. So, servlets are commonly used as controllers, JSPs as view, and JavaBean and/or EJB components as models. Although many real-world web applications are developed in this way, JSPs may be used as both the controller and the view. An example is given to show how JavaBeans, JSPs, and servlets work together to implement the JSP Model 2 Architecture. In the example, a servlet checks user name and password, then dispatches the corresponding JSP page.

This is the servlet that acts as the controller:

```
...
public class LoginServlet extends HttpServlet {
  ...
  // the method is used to rceive user request, do processes,
     and send response to user
  public void doPost (HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
   {
    //get the session created by server to user
    HttpSession session = req.getSession(true);
    ...
    String username = req.getParameter("user");//get user name
    String password = req.getParameter("password");//get password
    if(username.equalsIgnoreCase("hmw")) //if it is registered user "hmw"
     { if(password.equalsIgnoreCase("www")) //if password is "www"
       {//do processing here
        IdBean idb=new IdBean();//initialize a JavaBean to store the data
        idb.setMatch(true);     //set flag for legal user
        session.setAttribute("id",idb);//set user id to the JavaBean
        //dispatch corresponding JSP page
```

```
         String url="../Data.jsp";//jsp to be dispatched
         ...
         rd.forward(req, res);//forwards user request
        }
        else //password is wrong
        {
           //error processing
        }
      }
     else //not registered user or wrong  user name
      {
          //error processing

      }
    }
}
```

This is the JSP (*Data.jsp*) that acts as the view:

```
<%@page session="true"%>
<%@page language="java"  import="../IdBean"%>
<html>
<body>
<%
//create a JavaBean from the JavaBean that is paased from user session,
  so that it can get user data
IdBean id=(IdBean)session.getAttribute("id");
if(id.getMatch()){//it is legal user
%>
<table BORDER=0 CELLSPACING=0 CELLPADDING=2 WIDTH="20%">
//send request to another servlet called RWServlet using post method
<form name="readwrite" action="../servlet/RWServlet" method="post">
 <input type="submit" name="action" value="read">
 <input type="submit" name="action" value="write">
</form>
</table>
//display corresponding jsp according to user's request
 <jsp:include page="Read.jsp" flush="true"/>
 <jsp:include page="Write.jsp" flush="true"/>
<%}%>
</body>
</html>
```

This is the JavaBean that acts as the model:

```
...
public class IdBean implements Serializable{
  private boolean match; //identify if legal login
   public IdBean(){//initialize the match flag to false
    match=false;
  }
  public boolean getMatch(){//get the match flag to identify a legal user
    return match;
  }
  public void setMatch(boolean isMatch){//set the match flag
    match=isMatch;
  }
}
```

This example can be easily understood by combining the pieces of code with Fig. 15. Any user request goes to the servlet first, then instantiates a `JavaBean` called `IdBean` and returns a JSP page to the user.

This model encourages a much cleaner separation of business and presentation logic, as well as of the responsibilities between developers and designers. As a result, it makes application development and maintenance easier. The most popular example of frameworks implementing JSP Model 2 Architecture is an `Apache Jakarta` project called `Struts`.

In my opinion, JSP Model 2 Architecture shows a simplified and straightforward way to use the MVC design pattern in real-world applications. It follows the MVC architecture but keeps relatively simple implementation because programming in JavaBeans, EJBs, and servlets is simpler than in Java or C++.

4.1.3 VisualAge for Java 4.0

VisualAge for Java 4.0 is an integrated visual environment that supports the complete cycle of Java program development. Here, we do not mention function or use details about VisualAge for Java 4.0. Our interest in VisualAge for Java 4.0 concentrates on its look and feel, and object-oriented approach. Some old IDE such as JBuilder and Visual C++ do not take an object-oriented approach when they present information to users. They rather used a file-oriented approach, because developers manipulate only files that are not organized hierarchically in user's point of view, users only interact with files by open, modify, and save buttons. VisualAge for Java 4.0 takes an object-oriented approach in the design. It organizes its files in a hierarchy of objects and manipulates these objects (classes, interfaces, packages, etc.) instead of files.

Fig. 16 illustrates the main interface window when VisualAge for Java 4.0 starts. Besides the toolbar and menu, we can see a tab widget in the main window as shown in Fig. 17. The tab widget can switch from one to another to display different contents. In our

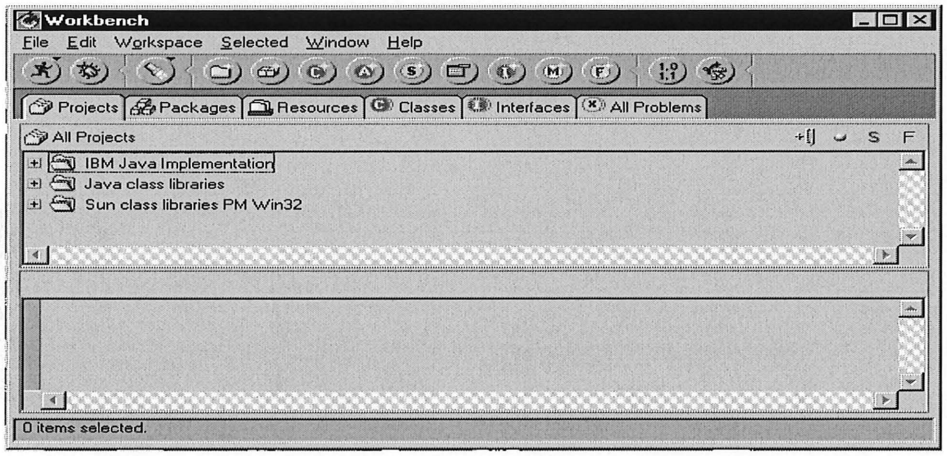


FIG. 16 – *Main window of VisualAge for Java 4.0*

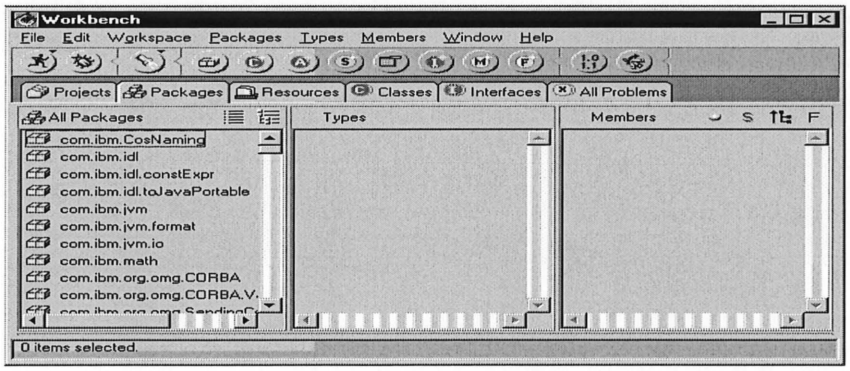


FIG. 17 – *Tab widget of VisualAge for Java 4.0*

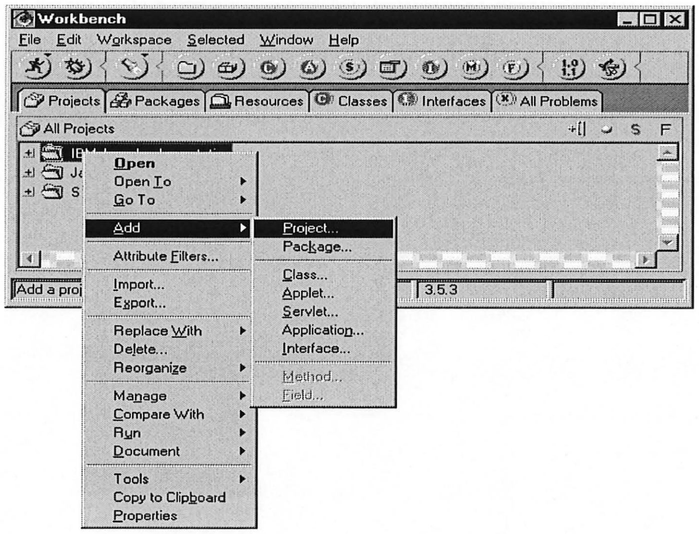


FIG. 18 – *A pop-up menu of VisualAge for Java 4.0*

GUI design, we need to display projects, processes, constraints, controllers, components, ttgs, adts, and feedback functions in the same way as VisualAge for Java 4.0 does. So, we use a tab central widget similarly. Fig. 18 illustrates the pop-up menu when a user adds a new item such as a project, a package, or a class. We also use a similar way to add a project or a process in our GUI design.

4.2 GUI Design and Implementation

We have discussed some basic ideas that are used in the GUI design. Now we discuss how to use these ideas to design the GUI. The GUI design takes some ideas from MVC design pattern, JSP Model 2 Architecture, and VisualAge for Java 4.0. We also describe how to use these ideas in the GUI design. We use these ideas in two aspects: architecture and implementation.

As we know from above, the MVC design pattern divides all the functional parts into three parts: model, view, and controller. Usually these three parts are totally separated. In a GUI, a typical situation is an interface that takes the user input from a keyboard or/and a mouse, then the GUI does the necessary update to the presentation and data. In this situation, a simple and typical solution is to use a GUI as both controller and view. Although this solution is not a true MVC architecture because it puts controller and view into one part, its architecture and implementation is clear and simple. We uses this solution as architecture design in the GUI. We also mention that JSP Model 2 Architecture follows MVC architecture but keeps relatively simple implementation because JavaBeans, EJBs, and servlets facilitate programming. In a JSP Model 2 Architecture application as shown in the above example, the servlet as a controller takes the user request in one of the two built-in methods (they are invoked by `GET` and `POST` methods in HTML form respectively.):

```
doPost (HttpServletRequest req, HttpServletResponse res);
doGet (HttpServletRequest req, HttpServletResponse res);
```

Then it updates a JavaBean or/and an EJB that acts as model and sends back the corresponding JSP as view. Message (request and response) passing is simple through above two methods. We do not need to know and consider the details of message passing but just how to process the message. This approach also avoids complicated design and implementation of message passing in using the MVC design pattern. Fortunately, Qt's powerful `signal/slot` mechanism provides an easy way for message passing between objects. So, we can use `signal/slot` mechanism to simplify the GUI implementation while keeping an architecture that approximately follows the MVC design pattern. According to the consideration above, the implementation of the GUI involves the following classes:

1. `Repository` class which is equivalent to the model;
2. `MelodiesCentralWidget` class which is mainly equivalent to the view and controller;
3. `GuiData` class which is responsible for reading and writing data from and to XML files;
4. `AddWindow` class which is used to add items on the GUI;

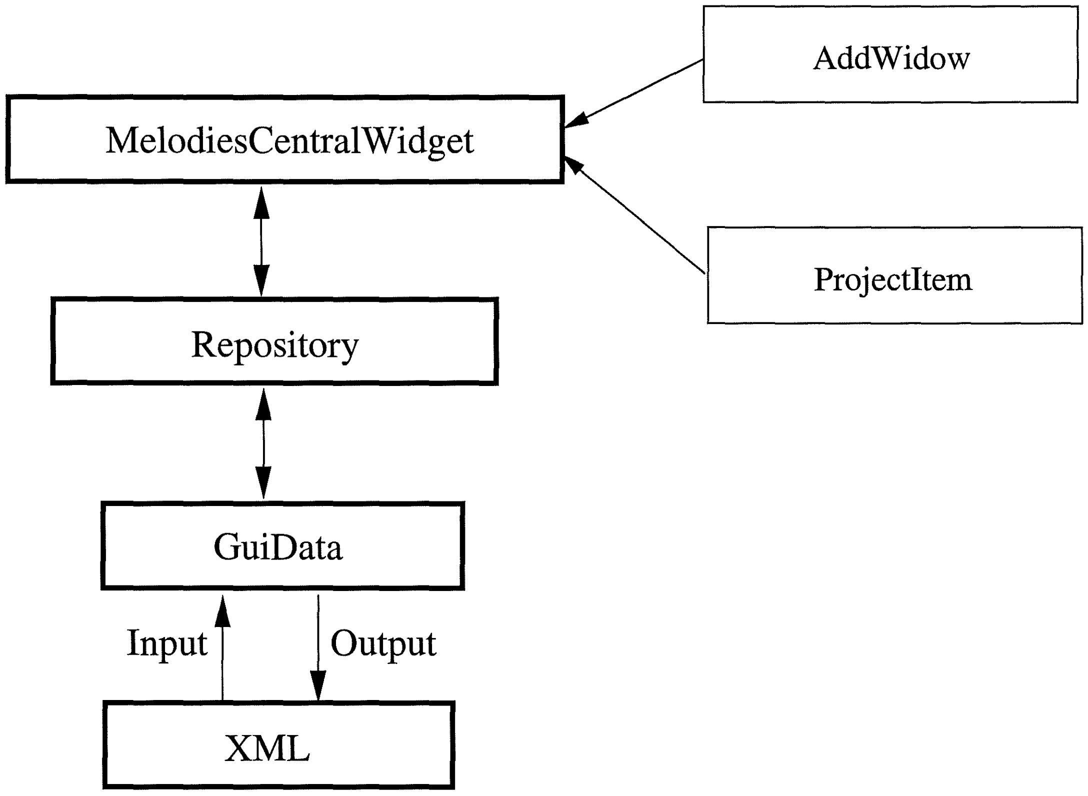


FIG. 19 – *Relations of GUI classes*

5. `ProjectItem` class which is used to identify item types on the GUI.

Fig. 19 shows how these classes work together. When the GUI program starts, the `GuiData` object reads data from an XML file and setups the `Repository` object, then the `MelodiesCentralWidget` object displays according to information from the `Repository` object. When changes occured, the `MelodiesCentralWidget` object updates itself as well as the `Repository` object. Last, the `GuiData` object can output data stored in the `Repository` object to an XML file. Following we discuss these classes.

4.2.1 *Repository* Class

The `Repository` class which has been written by Professor St-Denis builds a data structure that manages a project hierarchy. We can process a project hierarchy with it.

With Repository class, we can manipulate GUI data in an object-oriented approach as VisualAge for Java 4.0 does. Repository class declaration code fragment is as follows:

```
class Repository
 {
  public:
   typedef int Single_idx;                        // Single index
   typedef pair<int, int> Pair_idx;               // Pair of indexes
   typedef map<string, Info> Map_of_instances;    // Map of instances
  private:
   typedef list<int> List_of_associations;        // List of associations
   typedef Map_of_instances::iterator Instance_ptr; // Iterator on a map
   struct Entity                                  // Entity
    {
      string name;                        // name
      Map_of_instances instances;         // instances
      List_of_associations associations; // associations
    };
   struct Info                            // Information for an instance
    {
      List_of_list_of_instances *ass;     // association instances
      bool indicator;                     // application dependent
      int value;                          // application dependent
      string c_date;                      // creation date
      string u_date;                      // modification date
      string comment;                     // comment
    };
   unsigned int nb_entities;              // Number of entities
 vector<Entity> dictionary;            // Table of entities
public:
 Repository();                       // Default constructor
 bool Add_Entity(const string &);    // Add an entity in the dictionary
                                     // Get the number of an entity
 Single_idx Get_Entity_Number(const string &e) const;
 unsigned int Nb_Of_Entities()       // Return the number of entities
                               // Add an association in the dictionary
 bool Add_Association(const string & ,const string &);
                               // Return the number of associations
 int Nb_Of_Associations(Single_idx) const;
                               // Add an instance of an entity type
 bool Add_Instance(const string &, Single_idx);
                               // Return the number of instances
 int Nb_Of_Instances(Single_idx) const;
                               // Associate two instances
 bool Associate(const string &, Single_idx, const string &,
               Single_idx); // Return the number of instances
 int Nb_Of_Instances(const string &, Single_idx);
 int Nb_Of_Instances(const string &, Single_idx, int);
                               // Dissociate two instances
 bool Dissociate(const string &, Single_idx, const string &,
                Single_idx);
                              // Rename an instance
 bool Rename_Instance(const string &, const string &,
                     Single_idx);
                              // Delete an instance
```

```
    bool Delete_Instance(const string &, Single_idx);
                                 // Set information for an instance
    bool Set_Indicator(const string &, Single_idx, bool);
    bool Set_Value(const string &, Single_idx, int);
    bool Set_Creation_Date(const string &, Single_idx,
                           const string &);
    bool Set_Update_Date(const string &, Single_idx, const string &);
    bool Set_Comment(const string &, Single_idx, const string &);
                                 // Get information for an instance
    bool Get_Indicator(const string &, Single_idx);
    int Get_Value(const string &, Single_idx);
    string Get_Creation_Date(const string &, Single_idx);
    string Get_Update_Date(const string &, Single_idx);
    string Get_Comment(const string &, Single_idx);
  }
```

In `Repository` class, we use `Entity` and `Instance` to represent item's type and item itself in project hierarchy respectively. In MELODIES, there are nine entities: project, process, constraint, controller, component, ttg, adt, formula, and feedback function. Taking Fig. 9 and the factory described in Chapter 1 for example, we have the following instances: `Factory_Project`, `Factory`, `Producer`, `Consumer`, `M1`, `M2`, `Table`, `Buffer`, and `f`.

Obviously, the maximum number of entities is nine. But many instances can belong to one entity. We use two structures, `Entity` and `Info`, to represent entities and instances respectively. The `Repository` class provides some methods to perform operations on entities and instances. For example, we can use `Add_Entity(const string&)`, `Add_Instance(const string &, Single_idx)` to add an entity and an instance in the repository or `Delete_Instance(const string &, Single_idx)` to remove an instance from the repository. Because the number of entities is usually fixed, the `Repository` class does not define a method to remove entities. The `Repository` class also provides methods for updating information of instances, associating or disassociating entities and instances. The declarations of these methods are shown in the above code fragment. Besides, the `Repository` class declares and defines some private methods and data structures. The `Repository` class is written in C++ STL and it is completely independent of Qt. So, it is a total separation to GUI presentation.

4.2.2 *MelodiesCentralWidget* Class

The `MelodiesCentralWidget` class is responsible for building the central widget in the GUI main window. It is written in Qt 3.0. `MelodiesCentralWidget` class declaration code fragment is as follows:

```
class MelodiesCentralWidget : public QWidget
 {
   public:
    MelodiesCentralWidget( QWidget *parent, const char *name = 0 );
    ~MelodiesCentralWidget();
    void save();
    Repository & getRepository();
   protected:
    Repository  r;
    QTabWidget *tabWidget;
    QListView *project_view;
    QListView *process_view;
    QListView *pcom_view;
    QListView *msk_view;
    QListView *constraint_view;
    QListView *cst_ttg_view;
    QListView *for_view;
    QListView *controller_view;
    QListView *ctl_ttg_view;
    QListView *fbk_view;
    QListView *component_view;
    QListView *com_ttg_view;
    QListView *com_adt_view;
    QListView *ttg_view;
    QListView *adt_view;
    QListView *maskfun_view;
    QListView *formula_view;
    QListView *feedbackfun_view;
   protected:
    void setupTabWidget();
    void setView();
   public slots:
    void createView();
   protected slots:
    void addItem();
    void rename();
    void deleteListViewItem();
    void popup_menu(QListViewItem *, const QPoint &);
 };
```

The `MelodiesCentralWidget` class is a subclass of `QWidget` which is the base class of all user interface object. In the `MelodiesCentralWidget` class, some `QListViews`, which implement a list/tree view, are declared to represent the project hierarchy. A

QTabWidget is also declared as the container of all the QListViews. To arrange the QListViews in the QTabWidget, QGridLayout is used. According to the requirement, we use *Windows* look and feel. When a MelodiesCentralWidget object is initialized, it calls setupTabWidget() and setView() methods in its default constructor. The method setupTabWidget() initializes all the QListViews, QTabWidget, QGridLayout, and setups *Windows* look and feel. In the setView() method, a GuiData object is created and inputs GUI data from the corresponding XML file. Then the setView() method calls the createView() method to get data from Repository object and put it into the right QListView. The MelodiesCentralWidget class provides such methods as addItem(), rename(), deleteListViewItem() to update information. When updated information needs to be saved, a MelodiesCentralWidget object calls its save() method to accomplish the work. In the save() method, a GuiData object is created and outputs GUI data to the corresponding XML file. Another important method is popup_menu(QListViewItem *, const QPoint &). A pop-up menu is displayed when a user right clicks on a QListViewItem, then the right process is called. A Repository object is declared in the MelodiesCentralWidget class. To do so, we can update data in MelodiesCentralWidget object and the corresponding data in the Repository object concurrently. The getRepository() method is used when adding new items in QListViews.

4.2.3 *GuiData* Class

The GuiData class is used to input and output data from and to the XML file and initialize a Repository object used by MelodiesCentralWidget object. The methods of the GuiData class could be incorporated into the MelodiesCentralWidget class. There are, however, mainly two reasons why we encapsulate the methods into a separation class

GuiData:

1. These methods are usually used when the application starts and exits. Mostly they are not touched during execution.
2. The separation increases flexibility. If we want to choose another input/output format instead of XML file, we just need to modify the `GuiData` class.

The `GuiData` class declaration code fragment is as follows:

```
class GuiData
 {
   public:
    GuiData();
    void writeXml(Repository &, const QString &fname);
    void createDom(const QString &fname);
    void createRepository(Repository &);
   protected:
    Type node_type(QDomNode);
    QString node_stype(Type);
   protected:
    QDomDocument domTree;
 };
```

The `createDom(const QString &fname)` method is used to read data from the XML file and create a `QDomDocument` object called `domTree` to be processed further. After `domTree` is created, the `createRepository(Repository &r)` method is called to initialize the reference of the `Repositroy` object which is declared in the `MelodiesCentralWidget` class. The methods `node_type()` and `node_stype()` are used by the `createRepository(Repository &r)` method for the type cast from element of `QDomDocument` to `Entity` and `Instance` of the `Repository` class. The `writeXml(Repository &, const QString &fname)` method is called when updated data need to be saved. It consists of two parts:

extracting data from the Repository object and writing the data to the corresponding XML file. The writeXml(Repository &, const QString &fname) method can be treated as a converse process of the methods createDom(const QString &fname)and createRepository(Repository &r). The difference is that the former goes from Repository to XML, while the latter goes from XML file to QDomDocument then to Repository. In the methods, we use the similar code to read from or write to XML file as shown at the end of Chapter 3.

4.2.4 *AddWindow* and *ProjectItem* Classes

The AddWindow class is a subclass of QDialog which is the base class of dialog windows. It is used when the addItem() method of MelodiesCentralWidget is called. The AddWindow class can add an existing item or create a new item to corresponding list view and update the Repository class at the same time.

The ProjectItem class is a subclass of QListViewItem. The class QListViewItem is declared as:

```
QListViewItem( QListView * parent, QString label1,...)
```

Without type information included, it cannot distinguish a project item to process item in the project list view. That is why the ProjectItem class is created. In the ProjectItem class, item type information is added and declared as follows:

```
ProjectItem(QListView *, QString, Type);
ProjectItem(QListView *, QListViewItem *, QString, Type);
ProjectItem(QListViewItem *, QListViewItem *, QString, Type);
```

where Type is defined as:

```
enum  Type {Non, Project, Process, Constraint, Controller, Component,
           TTG, ADT, MaskFun, Formula, FeedbackFun};
```

With the ProjectItem class, we can identify proper item and call the right processing.

We have discussed the GUI design and implementation above. We also test properly and deploy it on Sun Solaris. At last, we show the running result of machine example given in Chapter 1 in Fig. 20 to Fig. 23

Fig. 20 shows the main window that displays Factory_Project hierarchy. The Factory_Project includes a process called Factory and a constraint called Factory_Constraint.

Fig. 21 shows the Process window that displays Factory process and it's three components (Consumer, Producer and Table).

Fig. 22 shows the Components window that displays three components (Consumer, Producer and Table) and their associated TTG or ADT.

Fig. 23 shows the TTG window that displays TTG M1 and M2 in the Factory process.

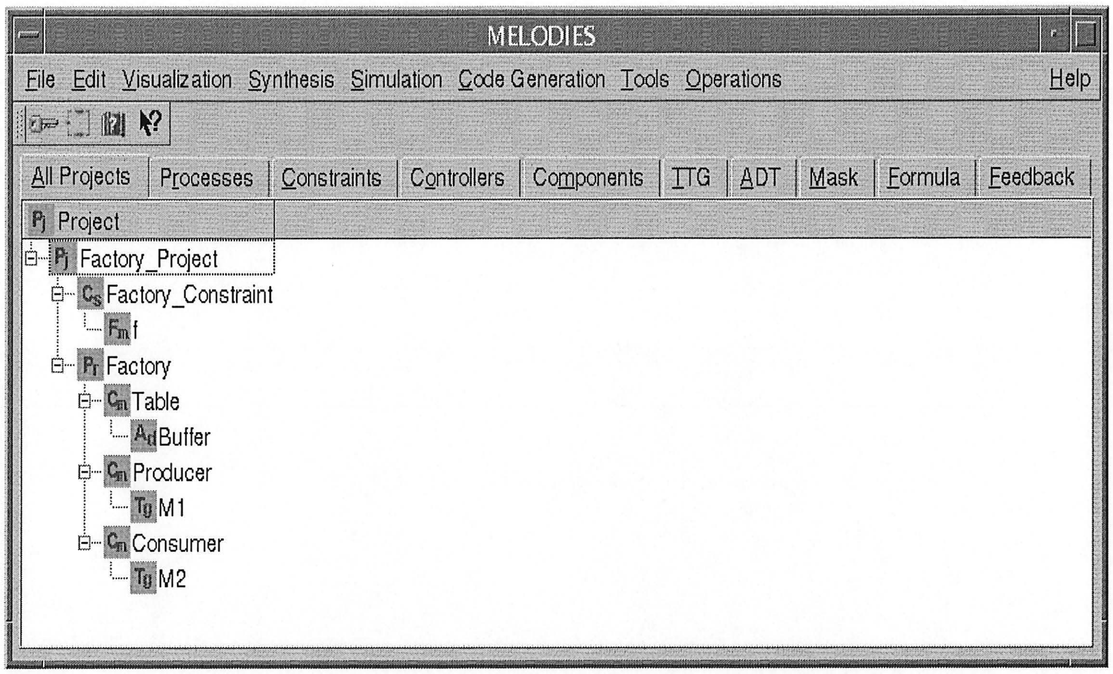
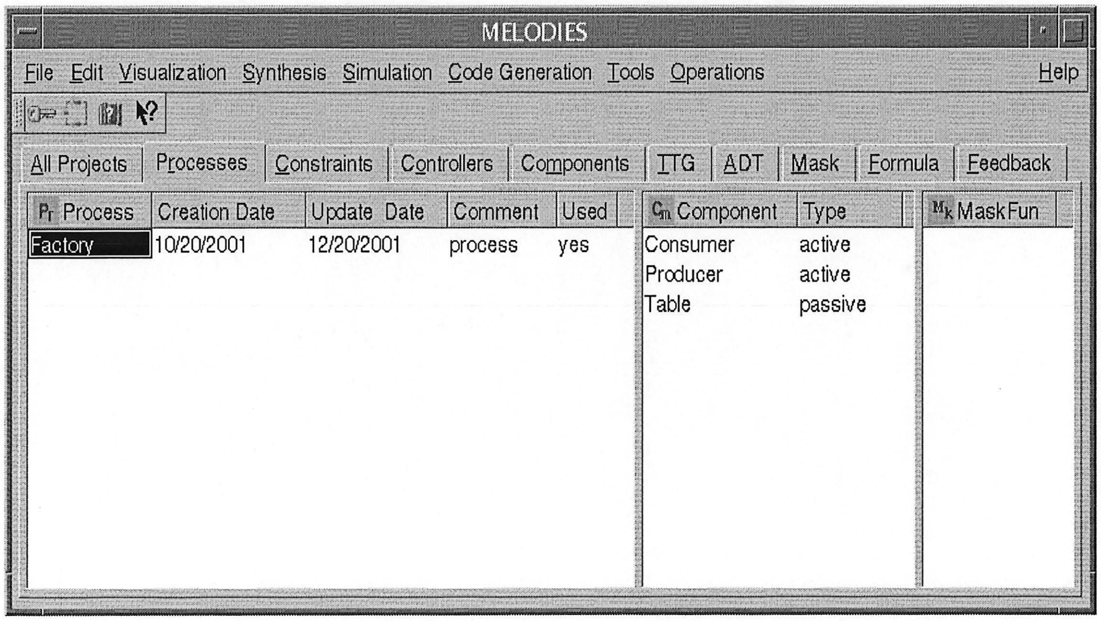

MELODIES
File Edit Visualization Synthesis Simulation Code Generation Tools Operations Help
All Projects Processes Constraints Controllers Components ITG ADT Mask Formula Feedback
Project
Factory_Project
Factory_Constraint
f
Factory
Table
Buffer
Producer
M1
Consumer
M2

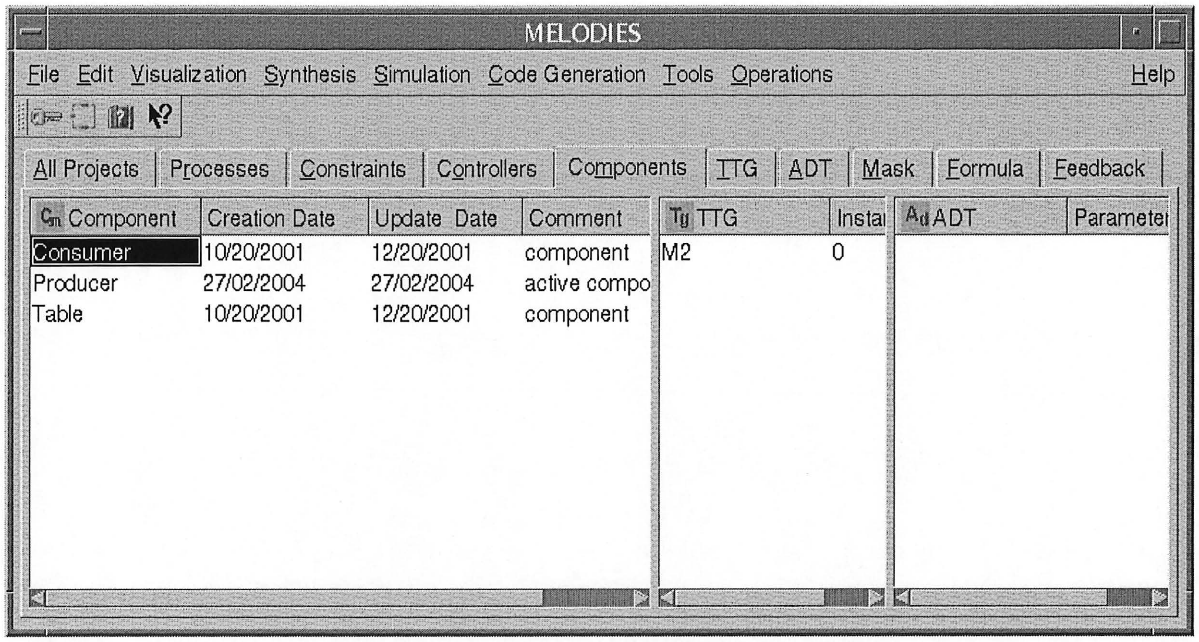
FIG. 20 – *MelodiesCentralWidget*


MELODIES
File Edit Visualization Synthesis Simulation Code Generation Tools Operations Help
All Projects Processes Constraints Controllers Components ITG ADT Mask Formula Feedback
MaskFun

Process	Creation Date	Update Date	Comment	Used
Factory	10/20/2001	12/20/2001	process	yes

Component	Type
Consumer	active
Producer	active
Table	passive

FIG. 21 – *Process tab widget*

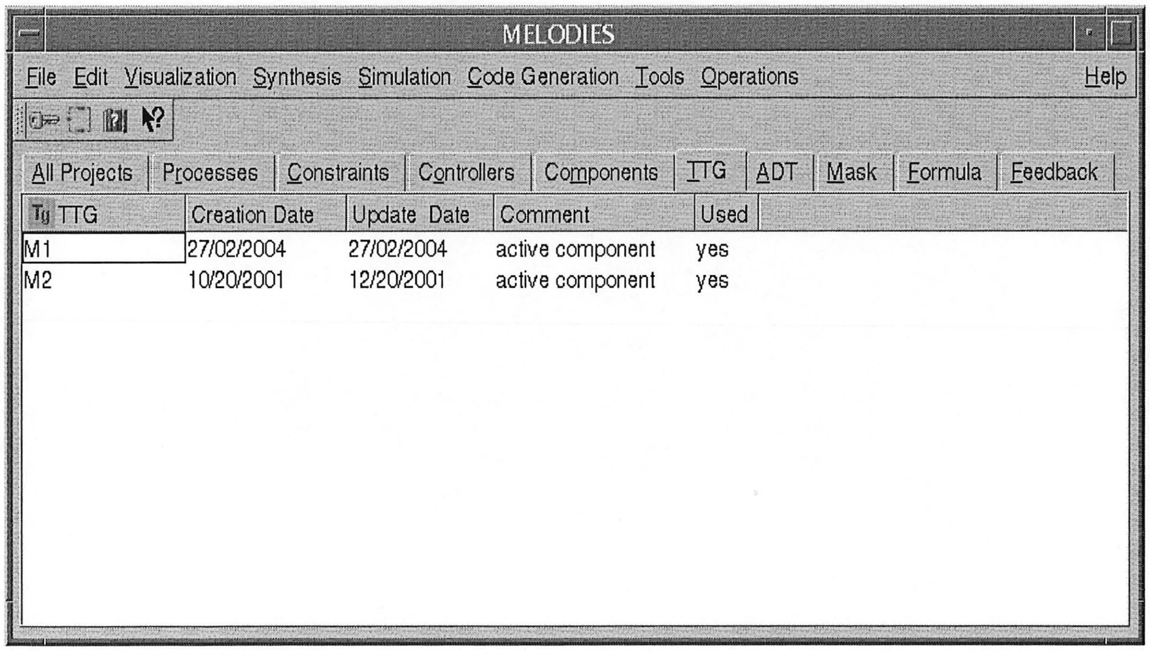

MELODIES
File Edit Visualization Synthesis Simulation Code Generation Tools Operations Help
All Projects Processes Constraints Controllers Components TTG ADT Mask Formula Feedback

Component	Creation Date	Update Date	Comment
Consumer	10/20/2001	12/20/2001	component
Producer	27/02/2004	27/02/2004	active compo
Table	10/20/2001	12/20/2001	component

TTG	Instai
M2	0

ADT	Parametei

FIG. 22 – *Component tab widget*


MELODIES
File Edit Visualization Synthesis Simulation Code Generation Tools Operations Help
All Projects Processes Constraints Controllers Components TTG ADT Mask Formula Feedback

TTG	Creation Date	Update Date	Comment	Used
M1	27/02/2004	27/02/2004	active component	yes
M2	10/20/2001	12/20/2001	active component	yes

FIG. 23 – *TTG tab widget*

Conclusion

This thesis has given an object-oriented GUI design for MELODIES. Because the conventional SCT situation can be treated as a special situation of the extension of conventional SCT that MELODIES is based on, we take the latter as general consideration. So, the GUI is available for the former as well as the latter. In the GUI, a DES is represented by a project and its processes, constraints, and controllers are organized into the project hierarchy. Unused processes, constraints, and controllers, which do not belong to any project, are represented in the corresponding tab widget instead of the project hierarchy. These components can be added to a project as needed. All the components are treated as objects, no matter whether it is a project, a process, or a controller. So, it can be processed independently in the GUI. With the GUI, users can easily create, add, delete, or modify a DES project, process, or controller. The GUI is partly integrated in the I/O and Editing modules and will be integrated with the other modules in the future. The GUI also provides a main menu and pop-up menus for calling the procedure to simulate a DES.

The GUI implementation is written in Qt and C++ STL. One advantage of developing the GUI in Qt and C++ STL is that it can be used on multiple platforms. We just need to recompile instead of modifying the code in different platforms to achieve this. Another advantage is its fast speed as compared with a Java implementation. XML is chosen as data transmission format in the GUI design because XML has become a standard data format that is widely used in Web and many other applications. Many software products include XML processors. It is easy to transfer and share XML data through Internet. One of the future goal of MELODIES is to use it on Internet. This choice is the first step to reach the goal.

Although the GUI reaches the requirement at the development stage, further improvements can be made in the future. One big issue that we have mentioned in the thesis is that the GUI design does not follow pure MVC design pattern because the view is not separated from the controller in the design. Total separation between model, view, and controller class would benefit us. First, it is easy for developers to maintain the application. Second, it permits new views or controllers to be added easily. This improvement can make the GUI design more scalable.

Appendix A

DTD Files

A.1 project.dtd

```
<!ELEMENT project-list (project,process,constraint,controller,
              component,ttg,adt,maskfun,formula,feedbackfun)*>
<!ELEMENT project (process,constraint,controller)*>
<!ATTLIST  project name  ID  #REQUIRED>
<!ELEMENT  process (component,maskfun)*>
<!ATTLIST  process
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT constraint (ttg,formula)*>
<!ATTLIST  constraint
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT controller (ttg,feedbackfun)*>
<!ATTLIST  controller
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT component (ttg|adt)?>
<!ATTLIST  component
  name      ID      #REQUIRED
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  ttg  (#PCDATA)>
<!ATTLIST  ttg
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  adt  (#PCDATA)>
<!ATTLIST  adt
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  maskfun  (#PCDATA)>
<!ATTLIST  maskfun
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  formula  (#PCDATA)>
<!ATTLIST  formula
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
<!ELEMENT  feedbackfun (#PCDATA)>
<!ATTLIST  feedbackfun
  creation  CDATA   #REQUIRED
  update    CDATA   #REQUIRED
  comment   CDATA   #REQUIRED
>
```

A.2 adt.dtd

```
<!ELEMENT adt (hname,imports,hsort,parameters,operations,
               signature,variables,equation)>
<!ELEMENT hname (#PCDATA)>
<!ELEMENT  imports (import)+>
```

```
  <!ELEMENT  import (#PCDATA)>
<!ELEMENT  hsort (#PCDATA)>
<!ELEMENT parameters (parameter)>
  <!ELEMENT parameter (name,type)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT type (#PCDATA)>
<!ELEMENT operations (operation)>
    <!ELEMENT operation (#PCDATA)>
<!ELEMENT signature (#PCDATA)>
<!ATTLIST  signature
  rank      CDATA   #REQUIRED
>
<!ELEMENT  variables (variable)+>
  <!ELEMENT  variable (name,type)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT type (#PCDATA)>
<!ELEMENT  equation  (#PCDATA)>
```

A.3 ttg.dtd

```
<!ELEMENT ttg (events,states,transitions)>
<!ELEMENT  events (event)+>
  <!ELEMENT  event (name,controllable,observable,forceable,
                   duration,cost,other_cost,operation)>
     <!ELEMENT name (#PCDATA)>
     <!ELEMENT controllable (#PCDATA)>
     <!ELEMENT observable (#PCDATA)>
```

```
    <!ELEMENT forceable (#PCDATA)>
    <!ELEMENT duration (#PCDATA)>
    <!ELEMENT cost (#PCDATA)>
    <!ELEMENT other_cost (#PCDATA)>
    <!ELEMENT operation (#PCDATA)>
<!ELEMENT  states (state)+>
  <!ELEMENT  state (name,initial,marked,duration,cost)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT initial (#PCDATA)>
    <!ELEMENT marked (#PCDATA)>
    <!ELEMENT duration (#PCDATA)>
    <!ELEMENT cost (#PCDATA)>
<!ELEMENT  transitions (transition)+>
  <!ELEMENT  transition (from,event,to)>
    <!ELEMENT from (#PCDATA)>
    <!ELEMENT event (#PCDATA)>
    <!ELEMENT to (#PCDATA)>
```

A.4 mask.dtd

```
<!ELEMENT mask_list (maskfun)+>
<!ELEMENT  maskfun (event,mask)>
  <!ELEMENT event (#PCDATA)>
  <!ELEMENT mask (#PCDATA)>
```

A.5 formula.dtd

```
<!ELEMENT formula(#PCDATA)>
```

A.6 feedback.dtd

```
<!ELEMENT feedbackfun_list (feedbackfun)>

<!ELEMENT  feedbackfun (state,event)>

  <!ELEMENT state (#PCDATA)>

  <!ELEMENT event (#PCDATA)>
```

Bibliography

[1] D. K. Appelquist. *XML and SQL.* Addison-Wesley, 2000.

[2] M. Barbeau, F. Kabanza, and R. St-Denis. A method for the synthesis of controllers to handle safety, liveness, and real-time constraints. *IEEE Trans. on Automatic Control*, 43(11):1543–1559, 1998.

[3] H. Bergsten. *JavaServer Pages.* O'REILLY, 2003.

[4] J. Blanchette and M. Summerfield. *C++ GUI Programming with Qt 3.* Prentice Hall, 2004.

[5] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[6] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems.* Kluwer Academic Publishers, 1995.

[7] R. Light. *Presenting XML.* Sams Publishing, 1997.

[8] I. Manimpire. *Conception d'une interface pour un environnement de modélisation et de simulation de systèmes réactifs.* Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, 2003.

[9] D. Martin and M. Birbeck. *Professional XML.* Wrox Press Ltd., 2000.

[10] B. Mclaughlin. *Java & XML.* O'Reilly Associates, Inc., 2001.

[11] M. Morriso. *XML Unleashed.* Sams Publishing, 2000.

[12] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[13] R. St-Denis. Designing reactive systems: integration of abstraction techniques into a synthesis procedure. *Journal of Systems and Software*, 60(2):103–112, 2002.