Automatic Generation of Interactive Systems from Declarative Models

A THESIS

Presented to

The Academic Faculty

By

R. E. Kurt Stirewalt

In Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy in Computer Science

Georgia Institute of Technology

December, 1997

Automatic Generation of Interactive Systems from Declarative Models

Approved:

Spencer Rugaber, Chairman

Gregory Abowd, Co-Chairman

Edmund Clarke

James Foley

Scott Hudson

H. Venkateswaran

Date approved by Chairman _____

Contents

List of Tables

List of Figures

Summary

This dissertation applies formal methods to the automatic generation of interactive systems from multiple declarative models. We are interested in two kinds of models: user-task and presentation. The Mastermind Dialogue Language (MDL) is presented. MDL is a notation for describing interactive system behavior in terms of user tasks. MDL task models are expressed independently of other models, like presentation, but are later composed with the behavior of these other models. Such separation is important for preserving the integrity of models over the lifetime of a system. The technical challenge in this approach is to generate code that combines the functionality of task and presentation models without violating that integrity. To meet this challenge we implement MDL task models as run-time dialogue constraints that synchronize with presentation components. The constraint engine is implemented as a virtual machine that simulates the execution of tasks and resolves the dependencies that arise as a result of task and presentation model composition. To simplify the generation process, a toolkit of reusable run-time components is provided. Each component in this toolkit implements an MDL operator, and components aggregate into trees whose structure corresponds one-to-one with the abstract syntax tree of a corresponding MDL model. Thus implementations can be generated through a simple syntactic transformation of the MDL source code. The design correctness of these components is validated by a novel application of symbolic model checking. The run-time attributes of systems generated using this approach are measured, and we conclude that this strategy of model composition is feasible for use with real interactive systems.

Chapter 1

Introduction

Interactive systems are difficult to analyze, design, and implement. The model-based approach to interactive system design alleviates this problem by basing analysis, design, and implementation on a common repository of models. Unlike conventional software engineering, in which designers construct artifacts whose meaning and relevance easily drift from that of the delivered code, in the model-based approach, designers build models of critical system attributes and then analyze, refine, and synthesize these models into running systems. This research focuses on those models which most directly influence the user interface of an interactive system: user task models and presentation models. Specifically, we investigate the synthesis of these models into running systems and the techniques for automating this synthesis. Szekely[98] laments that while several model-based interface development tools have been built, none has achieved a level of maturity to generate industrial strength applications. It is the thesis of this work that these weaknesses are symptomatic of an unclear delegation of responsibility in model-based code generation technology.

Unlike traditional programming language compilers, interactive system code generators seems to require input from multiple models. The MASTERMIND project[18, 74] is concerned with the design, integration, and automatic generation of interactive systems from declarative models. In the MASTERMIND environment[74, 97, 18], interactive systems are subject to three kinds of models:

- User task models specifying the protocols of end user interaction in the context of performing a task. This includes a description of end user actions, how they are ordered, and how they affect the presentation and the application.
- Presentation models specifying the appearance of user interfaces in terms of their widgets and how they behave.
- Application models specifying which parts (functions and data) of applications are accessible

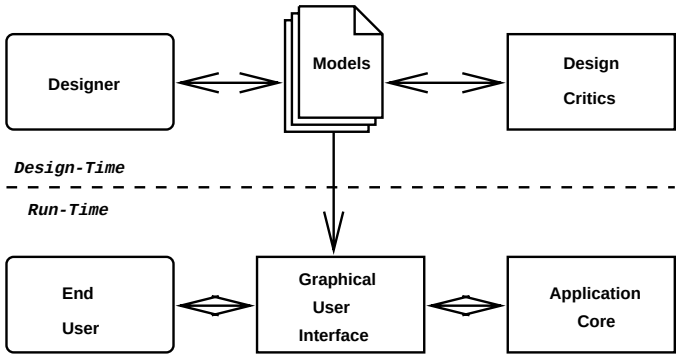

Figure 1: The Conceptual Architecture of the MASTERMIND environment.

from the interface.

These models often make overlapping and inter-dependent prescriptions about the behavior of a system. Moreover, resolving these dependencies is complex. It is the thesis of this work that a rigorous definition of model composition and a run-time virtual machine which implements such composition is required in order for multi-model based code generation to overcome the weaknesses noted by Szekely.

This research presents a solution. The MASTERMIND Dialogue Language (MDL) is a formally grounded notation for expressing task models and the binding of task and presentation behavior (multi-model composition). The MASTERMIND Toolkit (MMTK) is a collection of components which implement MDL operators and execute on top of a virtual machine. MDL models are easily encoded as trees of MMTK components which execute over the virtual machine. We claim that this approach solves the multi-model code generation problem thereby enabling the generation of more powerful user interfaces from declarative models. We present the details of the MDL language and MMTK and then validate our claims.

1.1 Design and Generation in MASTERMIND

MASTERMIND is an interactive system development environment that supports task based design through a repository of shared models and a suite of tools for defining, analyzing, and prototyping these models. Figure 1 shows the high level architecture of the MASTERMIND environment. At the core of this approach is a collection of models. Designers create models and can use automated

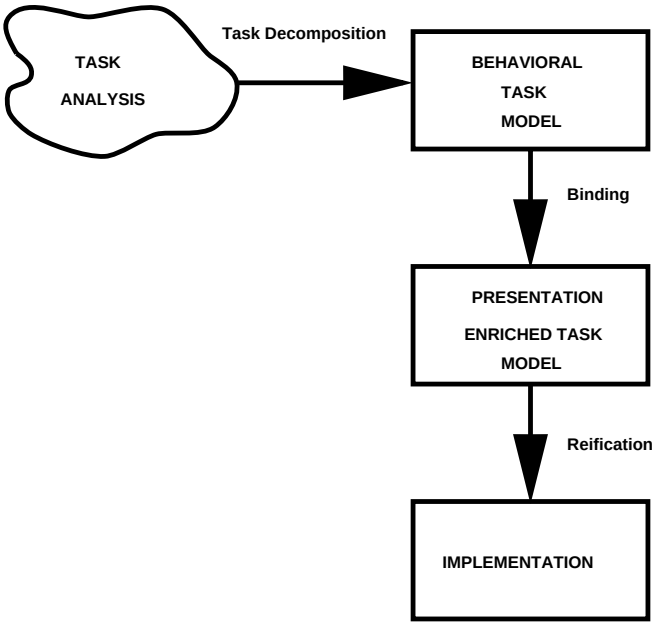

Figure 2: MASTERMIND design process model.

design critics to validate and critique these models. At some point, the designer invokes the code generator to synthesize the models into an executable user interface. This interface provides the end-user with access to core application functionality.

A view of the process of model design and generation is given in Figure 2. Designers perform user task analysis[30] to begin to gain an understanding of the requirements of the system. This analysis results in the initial design model (an hierarchical decomposition of user tasks). Tasks are specified in the MASTERMIND Dialogue Language (MDL). Once specified, task models are augmented with functionality from a presentation model (which is defined separately). This augmentation is called **binding**. Binding enriches task models with behavior defined in a separate presentation model. Finally, a code generation process implements these models on top of a virtual machine which simulates the execution of MDL. Components of this virtual machine comprise the MASTERMIND toolkit (MMTK).

One of the chief technical contributions of this research is the MASTERMIND Dialogue Language (MDL). MDL is a formal notation for representing user task decompositions. The design of MDL was influenced primarily by the LOTOS notation[17] with influences from CSP[51], and the User Action Notation (UAN)[46, 49]. Designers can use MDL to specify the behavior of tasks in terms

of objects, actions, and the sequencing of actions.

Binding enriches task models with presentation functionality. Binding is necessary because task descriptions are abstract control directives; whereas presentations describe physical entities which the user may observe and influence. Presentation functionality refines task models into a more concrete specification of an interactive system.

By design, task models are neutral with respect to the presentation aspects of an interactive system. Yet clearly tasks impact presentation. Presentation events like a mouse click symbolize user-events in task models, and task state must be communicated to the end user by reflection in presentation entities. In this sense, presentation entities implement tasks which lends credence to the refinement analogy of binding. Moreover, binding is not fully automatable, but rather must be supported by design tools. In the MASTERMIND [74] vision, binding is postulated as a design phase in which tasks are treated as obligations which must be resolved by attaching presentation functionality.

Figure 3 illustrates the runtime reification of task and presentation models and how they interoperate. Solid line shapes represent actual run-time components; whereas dashed line shapes represent the models from which these components are generated. The **Control Scheduler** is a run-time entity which simulates the task virtual machine mentioned earlier. It dispatches control to the **Task Components** which implement the MDL operators. Note that the task components are connected into a tree. This tree connection reflects the abstract syntax of MDL expressions. The idea is that, with a task component for each MDL operator, run-time systems can be rapidly constructed from MDL models by a syntactic transformation of the MDL expression. On the presentation side, there are **Presentation Components** which get generated from a presentation model. These components are typically organized into part-hierarchies and are controlled by a run-time system. MASTERMIND uses the Amulet[72] toolkit for presentation support, and so the run-time controller for presentation components is the **Amulet Run-time** system.

Systems with this structure must be generated from the bound task models. Implementation adds detail in the following dimensions:

Manifestation What tasks become in a running interface (i.e. objects, modules, or perhaps specifications of objects or modules),

Cooperation How, precisely, run-time tasks cooperate with other user interface objects, and

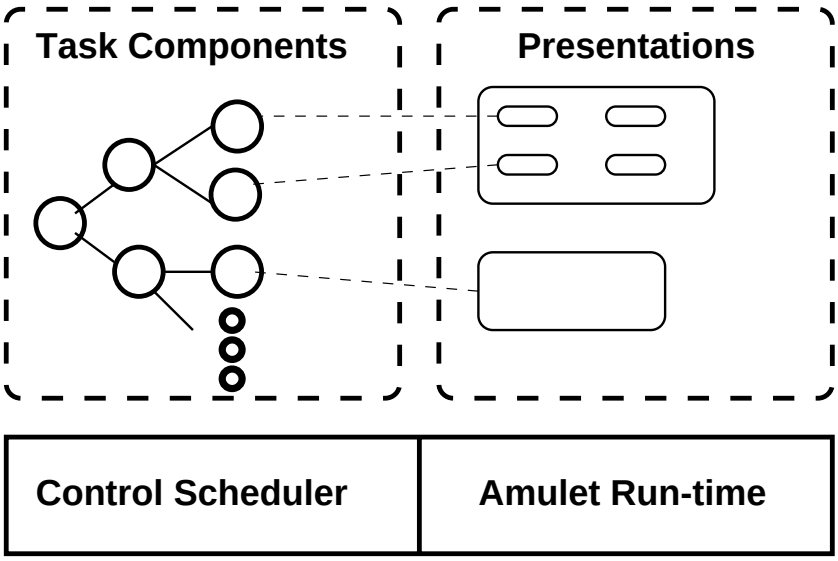

Figure 3: Run-time components of systems generated by MASTERMIND .

Behavior How, precisely, task ordering invariants are implemented.

Each MDL task manifests itself as a tree of task components whose leaves are connected to presentation components. Cooperation is supported through a binding scheme which composes arbitrarily with task tree leaves. Behavior is supported through a signaling protocol between the components in the task tree. This protocol expresses status information like enabling and disabling which eventually propagates over to presentation components (the dashed lines in the diagram). We went to great lengths to argue the correctness of this protocol, for without it, this seamless composition of run-time components would be moot.

1.2 Remainder of Dissertation

The existence of multiple models makes model-based code generation challenging. This thesis identifies two conditions we think are necessary to meet this challenge:

1. The composition mechanisms in modeling notations must be formalized (especially the inter-model composition mechanisms), and

2. There must be a virtual machine which correctly implements all of these composition operators.

Under these conditions, model-based code generation is feasible because code generators can concentrate on mapping models into software without worrying about dependencies introduced by other models. These dependencies will be resolved correctly by the virtual machine. In support of this thesis, we present a task modeling language (MDL) and toolkit which embodies a virtual task machine (MMTK). Components in MMTK compose in accordance with the MDL operators. We claim that MDL into MMTK satisfies the conditions mentioned above.

The remainder of this dissertation supports these claims. We begin by exploring the background of this problem (Chapter 3). We then introduce MDL and the generation of a running user interface from MDL through a motivating example (Chapter 2). This introduces a user interface for an air-traffic controller in a modern airport. It serves as a running example and helps to motivate the complexity of a toolkit to support the composition of MDL operators. We then formally introduce MDL (Chapter 4) and MMTK (Chapter 5). In defining MMTK, we identify a body of theory which must be fleshed out in order to validate that components cooperate to implement MDL operators. This theory culminates in a formal model of MMTK components and a pair of correctness properties that these components must respect (Chapter 6). In order to demonstrate that the correctness properties hold, we use an automated state space analysis tool called a model checker. This serves as a testing scaffolding. It allowed us to debug MMTK component logic without having to build large applications. We hypothesized that a finite number of such tests sufficiently tests the components in arbitrary compositions. We then proved this hypothesis to be true through a testing theorem (Chapter 7). With the testing theorem, we can claim the MMTK components are guaranteed to exhibit certain correctness properties. We then validate MDL and MMTK on examples and observe the system in operation (Chapter 8).

Chapter 2

A Motivating Example

In this chapter, we introduce MDL through an example which we refer to and extend throughout the document. The example is a system to support the tasks of an air-traffic controller in a modern airport. We use MDL to define the task model and then show how to *bind* this task model to a model of presentation (the graphical part of the system). We then demonstrate the C++ code that implements this bound task model and comment on the technical obstacles implied by this mode of code generation. The commentary frames the technical challenges which are the subject of this research. We conclude the chapter with a high level description of our solution to these technical challenges and a road-map of the remainder of the dissertation.

2.1 A Motivating Example

We introduce MDL by using it to model the tasks of an air-traffic controller in a modern airport. This section gives some background on the nature of these tasks. An air traffic controller must coordinate the airspace of an airport having a small number of runways. Flights enter the airspace at random and request permission to land. At the same time, flights on the ground request permission to takeoff. The controller must make sure that flights are serviced fairly, that throughput is maximized, and that safety is never threatened. An hierarchical task analysis of this domain describes the objects, actions, and action sequences. The analyst identified four high level tasks:

1. Monitor Air Space,
2. Assign Aircraft to Runway,
3. Land Aircraft, and

4. Handle Emergency Landing.

Further refinement identified the following subtasks:

1. Record flight information for new aircraft.
2. Assign flight to a runway.
3. Grant flight permission to land.
4. Put flight in a holding pattern.
5. Modify course of a flight.

Runway objects are passive in that they are not modified or acted upon, but rather are resources which influence the performance of tasks. In case of emergency landings, controllers dispatch emergency vehicles to the runway upon which the accident occurred.

2.2 MDL and User Task Modeling

MDL describes tasks in terms of their behavior patterns. The behavior pattern of a task is expressed as a *process*. A process can be thought of as a machine for performing actions in some prescribed manner. Processes perform and observe actions and interact with other processes. processes. Process interactions are built up out of atomic units of synchronization called *events*. Complex processes may be built by either combining sub-processes through an ordering operator (i.e.- process C is the sequential composition of sub-processes A and B) or by conjoining sub-processes making them synchronize on common events. Figure 4 lists the MDL description of a task called *ManagePlaneInFlight*. As its name suggests this task describes the sequence of actions (events) a controller must perform to service planes in the air. To understand the task, we first describe the events which make up the behavior and then the task description which specifies how the events are sequenced.

2.2.1 Events

Events represent basic actions which a user will perform. Figure 4 declares two events: *commitToLand*, and *newPosition* (lines 1 and 2). These correspond to the air traffic controller choosing a plane

```
event commitToLand;                                                    (1)
event newPosition : int;                                               (2)

task ManagePlaneInFlight                                               (3)
parameters                                                             (4)
    flight : string ; pos : integer;                                   (5)
is                                                                     (6)
    ModifyPosition* >> InstructToLand                                  (7)
where                                                                  (8)
    task ModifyPosition                                                (9)
    is                                                                 (10)
        newPosition?pos ; stop                                         (11)
    endtask                                                            (12)

    task InstructToLand                                                (13)
    is                                                                 (14)
        commitToLand? ; stop                                           (15)
    endtask                                                            (16)
endtask                                                                (17)
```

Figure 4: MDL description for the task of managing a plane in flight.

from the many within the airspace and entering a new position for a plane respectively. All events in MDL must be declared before they are used. This is done using the **event** declarator. Once an event has been declared, it may be used to specify the behavior of a task.

When declaring an event, the data type of the event must be specified. Events are used to synchronize interacting tasks, and, in the course of synchronization, data may be transferred between the tasks participating in the synchronization. For this reason, the type of data must be declared and associated with the event. In the figure, *newPosition* is declared to be an event with data type **string**. By associating a data type with the event *newPosition*, we may form input and output *communications* using the _?_ and _!_ operators respectively. Communications are event synchronizations which are accompanied by the flow of data. An input communication is expressed: $e?x$ where e names an event and x names a data storage location. The type of x must match the data type declared for e in an **event** statement. An output communication is expressed $e!x$ where e names a parameter and x names a value. There are two input communications specified in Figure 4 at lines (11) and (15). Note that the types of data expressed in the communication are consistent with the data type of the event. In *newPosition*?*pos*, the data parameter *pos* is of type **int**

which matches the type of the event *newPosition*. Some events, like *commitToLand* are declared to carry no data (line 1). Communications involving this event are expressed *commitToLand*?. The distinction between an event and a communication (which names an event) is important, and we will assume the distinction in subsequent discussion.

2.2.2 Tasks

Like any MDL task, *ManagePlaneInFlight* contains three sections:

1. a list of parameters,
2. a behavior expression, and
3. a list of subtasks.

We now describe the meaning of each of these sections.

The **parameter** section of an MDL task description specifies zero or more names that denote either local events or data locations. The name parameter was chosen because MDL supports task *instantiation* by providing values for these parameters. Parameterization allows designers to specify an infinite number of task instances, and it provides local storage for values and events used in computations internal to the task. A parameter names either a data values or an event. The task in Figure 4 has two parameters: *flight* and *pos* (line 5). The first parameter is a string containing the unique flight number of a flight and the second is an indicator of position within the airspace. Examples of flight numbers include `Delta 111` and `USAir 763`. An example of a position includes `10000` (feet). Flight numbers are used by controllers to uniquely distinguish flights in the air (as well as by passengers to make reservations and check gate location). Positions are used to record the location of flights in the airspace. Since this task has two parameters, it must be *instantiated* with two values. Instances are specified in MDL by affixing the task with a bracketed comma-separated list of parameter values. An instance of *ManagePlaneInFlight* for the flight mentioned above would be expressed in MDL as:

ManagePlaneInFlight[`Delta111`, `10000`]

Of course, the parameter need not be specified by a literal during instantiation. In fact, the most common use instantiates a task from a value which is input from some external source as in:

$$newFlight?x \,;\, position?y \,;\, ManagePlaneInFlight[x, y]$$

which specifies an instance of the task to be instantiated with values input by the events *newFlight* and *position*.

In addition to these two data values, *ManagePlaneInFlight* contains information designating how its instances will behave. This information follows the **is** keyword and is called a *behavior expression*. Behavior expressions specify how the task decomposes into subtasks and events. The behavior expression for *ManagePlaneInFlight* (line 7 in Figure 4) indicates two subtasks, *ModifyPosition* and *InstructToLand*, and an ordering which is to be imposed over these subtasks. The *ModifyPosition* task is designated to execute zero or more times by the unary operator $*$, while *InstructToLand* is *enabled* by the successful completion of *ModifyPosition*. The enabling operator $>>$ specifies sequential composition, and we read it to mean that *InstructToLand* may be performed after zero or more complete performances of *ModifyPosition*, but that it may not be performed during the performance of *ModifyPosition*.

Unlike *newFlight* and *position*, which are primitive events, *ModifyPosition* and *InstructToLand* are themselves tasks which decompose further. This further decomposition is specified in the **where** section of a task, and it contains complete task descriptions. The *ModifyPosition* task decomposes into a simple input event followed by the empty task **stop** (line 11). The ; operator prefixes a task **stop** by a communication $newPosition?pos$. The result of prefixing a task by a communication is a new task. Note that the input event *newPosition* deposits its input value into the parameter *pos* of the containing task *ManagePlaneInFlight*. That is, subtasks may reference the parameter scopes of containing tasks in block structure.

2.3 MDL and Presentation Binding

While the description of the task in Figure 4 documents the way an air-traffic controller performs one aspect of his job, it does not lend any insight into how this task manifests itself in a graphical user interface. For this we need to express the presentation aspect of the system. Figure 5 shows how the interface will look to the controller. The light gray bars demarcate legal air-traffic lanes;

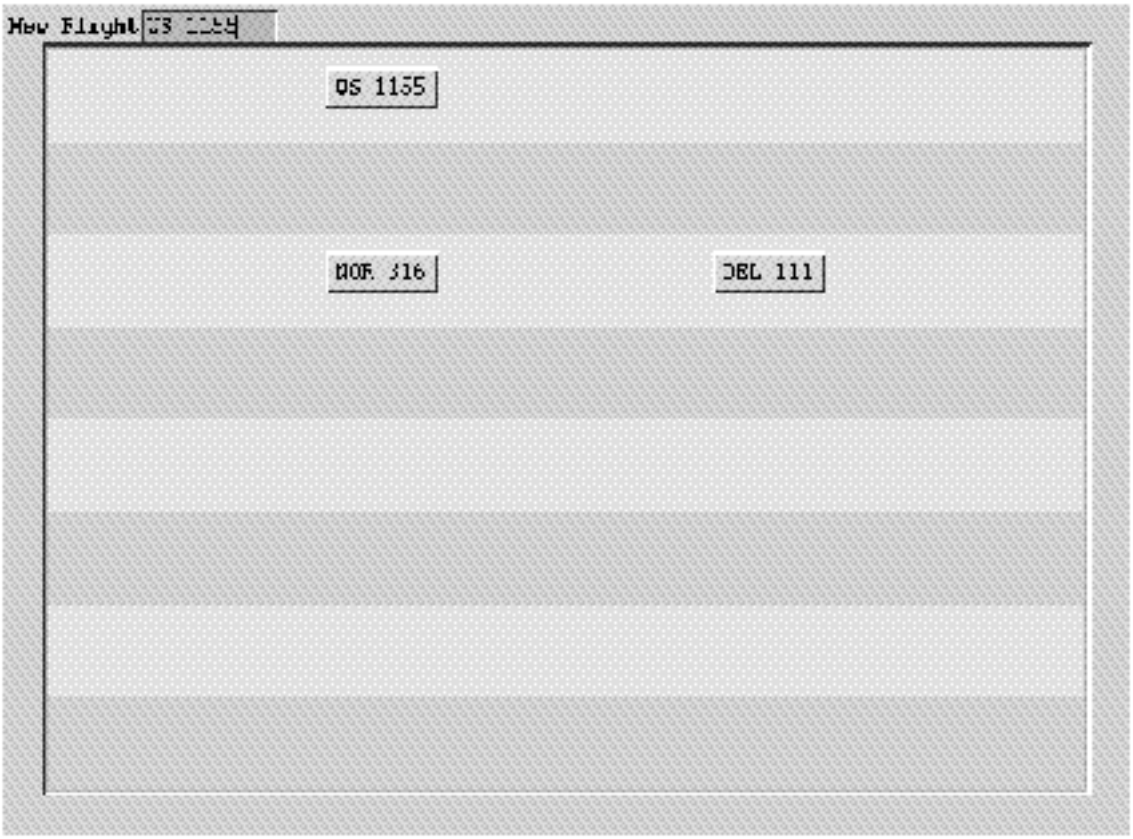

Figure 5: Presentation of the Air Traffic Control (ATC) system.

whereas the dark gray bars demarcate buffer lanes which are needed to ensure a safe distance between planes in flight. In this interface, planes are represented as buttons positioned throughout the traffic lanes to reflect the real position of planes in the airspace. New planes are incorporated into the airspace when the controller enters a new flight number in the text entry box at the top left of the interface.

In the MASTERMIND environment, the code for presentation objects, like the airspace background and the plane widgets, is generated from a declarative presentation model [25] which is specified separately from the task model. Of course, task and presentation models are not entirely orthogonal. When an air-traffic controller performs the *ManagePlaneInFlight* task, for example, he does so by interacting with the buttons generated by the presentation model. It is, therefore, necessary that some aspect of the presentation model be represented in the task model. In this example, the controller indicates a change of position by dragging a particular button to another spot in a legal air-traffic lane, and he indicates the desire to land a plane by double-clicking on a particular button. These gestures, pressing a button and dragging and dropping a button, represent actions which have meaning in both the presentation and the task world. We, therefore, model these gestures in MDL so that we may reason about the binding of task and presentation behavior. Specifically, gestures are modeled as output communications ($e!x$ for some event e and data value x), and the cumulative behavior of a presentation is modeled as a process. Since the behavioral aspect of presentations is modeled using processes, tasks and presentations are bound by synchronizing these events.

Figure 6 describes the behavior of a single air-plane widget (a button). The description resembles that of a task, but note two important differences. The process is bracketed with the keywords **pres** and **endpres** as opposed to **task** and **endtask**, and the communications reference a wild-card symbol (#) as opposed to literals or parameters. This notation describes that aspect of presentations which can be reasoned about in the task model. In this example, the *select* event represents the result of double clicking on the button; whereas the *drag* event represents the result of dragging and dropping the button onto a new location. Even though some detail is missing, we can understand what type of data the *select* and *drag* events communicate by looking at their declared data types. The *select* event communicates only control, which means that a corresponding input communication of the *select* event must be of the form *select*? as opposed to *select*?x. The

```
pres Airplane
parameters
      flight : string; pos : int;
      select : event; drag : event(int);
is
      ((select!# ; stop) | (drag!# ; stop))*
endpres
```

Figure 6: MDL description of the behavior of an airplane presentations

drag event communicates an integer, which corresponds to the newly selected position of the plane. When these specifications are realized in code, the communications will actually be initiated by *interactors*. Interactors[70] are user-interface toolkit devices which manage stereotypical gesture behavior and perform actions at various points during a gesture.

In MASTERMIND , a task T is bound to a presentation P using the notation: $P \parallel T$. The operator $\parallel$ (read "parallel") declares that T and P execute independently and concurrently, but that they synchronize on common events. We can see how this works by looking at the binding of task *ManagePlaneInFlight* to the presentation *Airplane* in Figure 7. Bindings are declared with the **bind** ... **endbind** keywords. Figure 7 defines a binding called *ManageFlight* with two parameters *flight* and *pos* (line 3). These parameters are local to the binding. The binding behavior is expressed in lines 7, 8, and 9. An instance of the *ManagePlaneInFlight* task is composed with an instance of the *Airplane* presentation through the $\parallel$ operator. Line 7 instantiates *ManagePlaneInFlight* with the values of *flight* and *pos* local to the binding. Line 9 instantiates *Airplane* with these same two values and also the two events, *newPosition* and *commitToLand*. These events guard input communications in task *ManagePlaneInFlight*. Since the events also occur in the instance of the *Airplane* presentation, and since these two instances are composed by the $\parallel$ operator, the instances will synchronize on these two events.

The binding behavior is wrapped up in a larger declaration that begins with the **hide** keyword. Hiding allows one to make events *unobservable* outside the scope of the **hide** declaration. Unobservable events are local to the binding in the sense that tasks within the binding (like *ManagePlaneInFlight* and *Airplane*) can synchronize on the events, but tasks external to the binding cannot. Hiding tends to be used in bindings like *ManageFlight* because there will be many run-time instances of this binding, and we do not want these separate instances to synchronize on

```
bind ManageFlight                                                    (1)
parameters                                                           (2)
    flight : string; pos : int;                                      (3)
is                                                                   (4)
    hide commitToLand, newPosition                                   (5)
    in                                                               (6)
        ManagePlaneInFlight[flight, pos]                             (7)
        ||                                                           (8)
        Airplane[flight, pos, commitToLand, newPosition]             (9)
endbind                                                             (10)
```

Figure 7: MDL description of the manage plane task binding to the airplane presentation.

each others' events!

The operation and utility of MDL should now be more clear. Designers can define tasks and the behavior of presentations and then compose them to arrive at the actual behavior of an interactive system. That is the essence of MDL. As we will now see, these bound specifications can then be implemented by a model-based code generator.

2.4 Generating Code from MDL

Once task and presentation models have been defined and bound together, these bound specifications can be translated into a program in a conventional programming language. It is the job of a model-based code generator to perform the translation, and it must do so without input from the designer. Our goal was to make the translation as straight-forward as possible. To support this goal, we created a toolkit (called the MASTERMIND Toolkit or MMTK) of reusable components that directly implement MDL specifications on a run-time virtual machine. We now demonstrate the use of this toolkit in implementing the behavior specification of *ManageFlight* and comment on the complexity of the toolkit components. This commentary motivates the technical contributions which follow.

2.4.1 Example of Generated Code

MASTERMIND implements bound task models in C++ using the Amulet[72] user interface toolkit for presentation support. The general strategy is to create a C++ class for each declared binding. The class declarations aggregate:

1. local data corresponding to binding parameters,
2. MMTK components corresponding to MDL communications and ordering operators, and
3. locally managed events (those which result from applying the MDL **hide** operator).

The constructor associated with such a class must then:

1. establish the location of data parameters are used by input and output communications and initialize these communications so that they know the locations,
2. establish the location of hidden events, and initialize communications that synchronize on these events so that they know the locations,
3. connect all of the aggregate MMTK components for communication.

To see this in action, consider again the binding *ManageFlight* of Figure 7. The implementation of this binding appears in Figure 8. The binding has become the C++ class `Binding_ManageFlight`. As the comments suggest, constituents within this class correspond to the task (*ManagePlaneInFlight*), the presentation (*Airplane*), the data parameters (*flight* and *pos*), and the events (*newPosition* and *commitToLand*) in the MDL binding.

To understand the design of the MMTK components, we now look at the details of these constituents and how they interoperate. The data parameters *flight* and *pos* of *ManageFlight* appear as private data members in class `Binding_ManageFlight`. The values of these parameters can be queried using the `getFlight()` and `getPos()` methods and can be set using the `setFlight()` and `setPos()` methods. Immediately we must consider the visibility of this data. MDL specifications, require these values to be visible to input and output communications both within and external to the binding which houses the data. To support this, the class exports (public) methods for querying and setting these values. Thus, when the implementation of the input communication *newPosition*?*pos* needs to set the *pos* parameter, it can do so by invoking the method `setPos(x)` where `x` is the value the input receives during synchronization. This leads to the following observation.

```
class Binding_ManageFlight : public MdlParOrdering,  // The binding operator.
                                 // The events.
                             public MMBlockingEvent_NewPosition,
                             public MMBlockingEvent_CommitToLand {
public:
  Binding_ManageFlight();
  virtual ~Binding_ManageFlight();
     // Parameter access methods.
  string& getFlight();
  void    setFlight( const string& );
  int     getPos();
  void    setPos( const int& );
protected:
  //
  // Implementation of the task ManagePlaneInFlight.
  //
  MdlSeqOrdering                                    inputNewPositionThenLand;
  MdlLoopOrdering                                   loopInputNewPosition;
  InputConcealedEvent<string,
                      Binding_ManageFlight,
                      MMBlockingEvent_NewPosition,
                      Binding_ManageFlight>         inputNewPosition;
  InputConcealedEvent<string,
                      Binding_ManageFlight,
                      MMBlockingEvent_CommitToLand,
                      Binding_ManageFlight>         inputCommitToLand;
  //
  // Implementation of the presentation Airplane.
  //
  MdlLoopOrdering                                  loopoutputNewPositionOrLand;
  MdlAltOrdering                                   outputNewPositionOrLand;
  PresOutputConcealedString<MMBlockingEvent_NewPosition,
                            Binding_ManageFlight>        outputNewPosition;
  PresOutputConcealedString<MMBlockingEvent_CommitToLand,
                            Binding_ManageFlight>        outputCommitToLand;
protected:
  virtual void MMenable();
private:
    // The parameters.
  string  flight;
  int     pos;
};
```

Figure 8: C++ header file implementation of ManageFlight binding.

Observation 2.4.1. *MMTK components need a uniform facility for connecting input (respectively output) communications to target (respectively source) data that are affected when the communications synchronize on an event.*

The next thing to note is that both task *ManagePlaneInFlight* and presentation *Airplane* are implemented as data members in this class. Consider the implementation of *ManagePlaneInFlight*. Recall that this task has the following behavior:

$$(newposition?pos \,;\, \mathbf{stop})* >> (commitToLand? \,;\, \mathbf{stop})$$

This protocol involves two MDL operators: $>>$ and $*$, and two input communications: *newposition*?*pos* and *commitToLand*?. These coincide with the four declarations under the comment "implementation of task ..." in Figure 8. Specifically, $>>$ is implemented by the member of type `MdlSeqOrdering`, $*$ is implemented by the member of type `MdlLoopOrdering`, and the input communications are implemented by members of the parameterized class `InputConcealedEvent<>`. The parameters to the `InputConcealedEvent<>` template specialize it with data type, data location, and event location information. This is discussed in detail in Chapter 5. The reader should note the strong structural similarity between the MDL expression and its MMTK/C++ implementation.

Observation 2.4.2. *Data members in the implementation of an MDL behavior expression occur in one-to-one correspondence with the operators and communications in a corresponding MDL behavior expression.*

Figure 9 shows the constructor for class `Binding_ManageFlight`. The code in this constructor relates the objects that implement task *ManagePlaneInFlight*. Specifically, it organizes them into a tree. The same is done for the presentation. And of course, the binding itself links the task and the presentation as siblings in the tree. One thing to note is that any class created from an MDL behavior expression may itself be aggregated into a containing class. It is therefore necessary that such classes aggregate the top-most MMTK tree component by inheritance rather than creating a data member of that type. This is why the class `Binding_ManageFlight` subclasses `MdlParOrdering` (which implements the MDL ordering $\|$). In fact, because bindings tend to be of the form $P \parallel T$ for some presentation P and task T, they typically subclass `MdlParOrdering`. There is a one-to-one syntactic correspondence between MDL behavior expressions and the MMTK components which are aggregated by the C++ class generated from these expressions.

```
Binding_ManageFlight::Binding_ManageFlight()
  : inputNewPosition("ppp" ,"ppp"), inputCommitToLand("pp" ,"pp"),
    outputNewPosition("ppp"), outputCommitToLand("ppp")
{
  addChild(&inputNewPositionThenLand);
  addSecondChild(&loopoutputNewPositionOrLand);
  // Task part.
  inputNewPositionThenLand.addChild(&loopInputNewPosition);
  inputNewPositionThenLand.addSecondChild(&inputCommitToLand);
  loopInputNewPosition.addChild(&inputNewPosition);
  // Presentation part.
  loopoutputNewPositionOrLand.addChild(&outputNewPositionOrLand);
  outputNewPositionOrLand.addChild(&outputNewPosition);
  outputNewPositionOrLand.addSecondChild(&outputCommitToLand);
}
```

Figure 9: C++ constructor for ManageFlight binding implementation.

Observation 2.4.3. *The syntactic structure of a behavior expression is preserved in the implementation of that expression.*

2.4.2 Shifting the Complexity of Code Generation

The correspondences noted in Observations 2.4.2 and 2.4.3 greatly simplify MDL code generation. The code generator merely aggregates MMTK components into a class and generates a constructor which connects these aggregates into a tree (isomorphic with the abstract syntax tree (AST) of the MDL behavior expression). This simplicity satisfies our design goals because code generation does not require input from the designer and does not arbitrarily make choices for the designer. We have effectively shifted the complexity of code generation into the design of a reusable toolkit. The idea is that we will carefully design the MMTK toolkit and then deploy a simple code generator that leverages it.

Of course, shifting the complexity of code generation into toolkit design means that the toolkit is difficult to design! That is, it is difficult to design correctly. The power of MDL operators makes correct design non-trivial. One such problem occurs when one task can effectively terminate another task. The MDL expression $A\,[>\,B$ (read "A is disabled by B") expresses that the performance of task B terminates any further performance of task A. A familiar application of this operator comes up in Internet browsers like the NetScape Navigator. The task model for this is shown in Figure 10. The ([>) operator is implemented by the MMTK component `MdlDisOrdering`. In the class `Binding_ChaseLink`, which is generated from the MDL in Figure 10, there is an instance class

```
task ChaseLink
is
    choose?url ; LoadURL[url]
where
    task LoadURL
    parameters
        url : string
    is
        (httpRequest!url ; httpComplete? ; stop) [> (quit? ; stop)
    endtask endtask
```

Figure 10: Example of disabling operator in MDL description.

`MdlDisOrdering`. This instance is connected to an instance of class `MdlSeqOrdering`, which implements the sequencing of the http requests, and to a member of class `InputEvent<>` which implements the quit communication. The semantics of [> ensure that if the *quit*? communication synchronizes before the *httpComplete*? communication synchronizes, that the whole process controlled by the `MdlSeqOrdering` will be terminated. This means objects of the class `MdlDisOrdering` need to be able to sense activity in their right component and terminate their left components. It also means that any object which could be the left component of an object of class `MdlDisOrdering` must know how to be terminated! This leads to the following observation:

Observation 2.4.4. *To implement MDL ordering operators as tree controllers, the internal logic of these controllers must take into account the contexts in which they may occur.*

This observation implies an inflation in the size of MMTK component specification and, consequently, the complexity of MMTK component design. Here is the problem. MMTK ordering components must internalize some aspect of the internal states of machines with which they might connect in a run-time tree. This makes the number of states in even the most seemingly simple of components (like `MdlSeqOrdering`) grow into the hundreds. It is difficult to design components with this much state, and it is even more difficult to do it correctly. After introducing MDL in more detail (Chapter 4) we return to the problem of MMTK component design and address these correctness issues (Chapter 5).

Chapter 3

Background and Related Work

The last chapter provides context for the application of automated tools to interactive system design and generation. This problem has been studied extensively throughout the history of HCI research, but to date the most widely used tools are rapid prototyping tools and static GUI layout generators. The model-based approach aims to increase the degree of design support and code generation using multiple explicit models of user interfaces. We believe that in order to achieve these lofty goals, model-based code generators requires a new mechanism of software composition. The purpose of this chapter is to frame this assumption in its historical context and introduce ideas we used when crafting the MDL/MMTK solution.

3.1 Interactive System Design

Interactive systems are a subclass of reactive systems, the main distinguishing characteristic being that humans are an active component of an interactive system. Interactive system design, therefore, must take human factors into account. Systems which accommodate a human actor must provide user interfaces through which the actor receives input from and issues output to other system components. Experience has shown that user interfaces are difficult and costly to design and maintain. In fact, a survey done by Myers[73] points out that up to half of the development cost and delivered application are associated with designing and implementing the user interface. HCI researchers and software engineers have concentrated on this domain because of the huge potential in development savings. Interactive system design is characterized by two novel aspects: the *ontology* of interactive system software and the software engineering process which supports interactive system development.

Ontologies define the concepts and relationships that make up a problem domain. They form

the objective grounding to which designers appeal when talking about the structure of a problem. There have so far been two ontologies proposed for the structure of the user interface. The first was proposed by Foley [38, 39] and defines the user interface as an input language for the user, an output language for the machine, and a protocol for interaction. The other ontology, due to Moran[69], considers the user interface to be "those aspects of the system that the user comes in contact with–physically, perceptually, or conceptually." Foley's linguistic model has proven particularly influential for automatic generation technology. The idea is that interface design is similar to programming language design. There is an input language, through which the user communicates with the computer and an output language, through which the computer communicates to the user. The act of engineering the interface is then thought of as the simultaneous design of these two languages. The Foley ontology suggests the existence of three *levels* of user-interface software. The lexical level describes atomic entities in the language like mouse motion, key clicks, and visible output. The syntactic level describes the sequencing of these tokens. The semantic level groups syntactic structures into collections of functionality which take on meaning in the specific problem space an interactive system is built to support.

The software engineering process side of interactive system design admits the human designer as an evaluator of the fitness of a user interface. The paradigm is one of rapid prototyping and continual evaluation[48]. Bass and Coutaz[11] refine this description into an iterative process model similar to the spiral model of Boehm[16]. Within this paradigm, analysis and evaluation occur throughout the life-cycle of a product. The focus of analysis in this domain are user tasks. According to Diaper[30, preface], task analysis is potentially the most powerful method for producing interactive system requirements specifications. There are many different approaches to task analysis. Some, like Task Analysis for Knowledge Description (TAKD)[31], produce a generalization hierarchy of tasks. Others, like Task Knowledge Structures (TKS)[56, 65] provide a theory of the structure of task knowledge and a methodology for identifying and modeling this knowledge. Others, like Goals, Operators, Methods, and Selection (GOMS)[23] and Cognitive Complexity Theory (CCT)[60], incorporate a model of human psychology in order to evaluate the performance of users on tasks.

The Mastermind environment, which will be discussed below, uses a task modeling notation derived from Lotos[17] in support of an incremental design, evaluate, iterate development model.

3.2 Automated Interactive System Development

Automated approaches to interactive system development reduce the complexity of development by raising the level of abstraction in system specification. There have been many approaches to this in the history of HCI. It turns out that the linguistic ontology is useful for classifying approaches. The lexical level, for example, represents the issues addressed by palette-based interface builders; whereas the syntactic level represents issues of dialogue and global sequencing in a User Interface Management System (UIMS), and the semantic level represents issues of domain-based user interface component selection. Within the confines of a specific level, application-generation tools perform well, but each level embodies system information which is influenced by, but cannot be derived from, the system information embodied by other levels. This motivates the conclusion of this section which states that a solution to the automatic-generation problem must deal with the simultaneous expression of information at multiple levels.

3.2.1 Lexical Tools and Technology

Tools at the lexical level provide designers with the ability to choose and specialize the presentation of a system with little support for behavior. Commercial tools like the User Interface Manager for X (UIMX)[101] allow designers to rapidly construct and specialize presentations from a palette of reusable widgets. Once the designer is satisfied with the display, the tool generates the source code which creates and initializes the display. It is then up to the designer to add the behavior to the widgets. In the case of UIMX-generated code, this activity amounts to providing call-back functions which are invoked when the end user interacts with the widgets at run-time.

With the emergence of the Java language[7] and user interface library AWT[43], palette-based presentation generators have become popular. These allow designers to rapidly construct presentations and then generate code which creates and initializes these presentations. Like UIMX, designers must follow this generation step with a programming step in which they add behavior to the call-backs of these widgets.

Incidentally, not all lexical tools are graphically based. The User Interface Language (UIL) compiler produces presentation code from a textual description of the display[11].

Tools at this level generate code which programmers then specialize. This specialization usually

takes the form of extension, i.e.-adding call-back functions. Often however, programmers modify the parameters or application of the generated code itself. It is typical for example, to find code that was originally generated by UIMX and then extended without going through the generator again. This behavior is certainly a drawback to these kinds of approaches because the delivered code easily drifts from the model which generated it. Still, tools which support the lexical level of design are very popular and have proven successful commercially.

3.2.2 Syntactic Tools and Technology

Tools at the syntactic level allow designers to express the sequencing of behavior in a user interface. Most of the tools at this level provide designers with a *dialogue* language through which sequencing constraints can be specified. Dialogue notations are so-called because they specify the syntax of interaction. Tools in this genre can be classified and differentiated by the power and style of their dialogue languages. Dialogue notations assume an underlying machine model which is either sequential or concurrent. Sequential dialogue notations tend to be based on formal language and automata theory (formal grammars, state transition diagrams, etc); whereas concurrent dialogue notations tend to be based on production rules which react to asynchronous events by performing some action.

The production rule model of dialogue fits well with the architecture of user-interface toolkits. The model does not require machine concurrency, but it has an expressive feel which is event based rather than thread based. The Propositional Production System (PPS)[76] is an example of such a model, and many UIMS's adopt it. The Serpent[11] UIMS, on the other hand, takes a more domain-independent approach. Using Slang (the Serpent dialogue language), functionality is assigned to graphical objects through guarded actions. These actions fire whenever the guards are satisfied (the standard production rule model).

3.2.3 The Multiple Level Problem

The approaches mentioned thus far have two recurring similarities. First, each approach adopts the same general framework for automating interactive system generation:

1. A run-time system that implements some form of virtual machine (be it an event loop, control scheduler, constraint engine, etc),

2. A language (visual or textual) for expressing the features of a system, and

3. A translator that maps expressions written in the language onto the virtual machine.

Second, these languages do not express the entire functionality of an application. This means that code generators must either:

1. generate a partial application which must then be augmented with programmer-written code, or

2. infer system attributes which are not expressible in the language (usually taking advantage of a knowledge-base).

This second feature is a source of difficulty and inadequacy of the current approaches to automated generation. In a truly automated development environment, programmers should not need to complete the user interface with hand-written code. Conversely the designer should expresses salient detail of an interface as opposed to a code generator.

We shall henceforth refer to this problem as the *multiple-level problem* in interactive system specification. Systems are not a product of structure at any one level, but rather are the confluence of structure at all three levels simultaneously. We believe this problem is inherent in automated approaches to interactive system development. As evidence of the problem, consider the following. In their work on the George Washington UIMS, Sibert et al. [90] commented that designers experienced difficulty when having to understand behavior at the *boundary* of the semantic, syntactic, and lexical levels. The success of tools and techniques that focus on a specific level, however, demonstrates that levels contain salient information which cannot be inferred from decisions made at higher levels. A semantic level decision, for example, cannot be derived from a lexical decision. Likewise, a lexical decision cannot *necessarily* be derived from a semantic decision. Designers need to be able to express interface detail at the appropriate level using tools and concepts appropriate to the level. Levels contain salient information that does not exist at other levels. At the same time, the information contained in a level is not independent with the information contained in other levels. Successful approaches to automated generation of interactive systems support the confluence of detail from all three levels. The model-based approach adopts this paradigm.

3.3 Model Based System Development

The model-based approach to interactive system design bases system analysis, design, and implementation on a common repository of models. Unlike conventional software engineering, in which designers construct artifacts whose meaning and relevance can diverge from that of the delivered code, in the model-based approach, designers build models of critical system attributes and then analyze, refine, and synthesize these models into running systems. Model-based user interface development environments (MB-UIDE's) work on the premise that development and support environments may be built around declarative models of a system. Developers using this paradigm build interfaces by building models that describe the desired interface, rather than writing a program that exhibits the behavior[97].

Pre-cursors to the model based approach were UIMS architectures and application generators that began to support the semantic level of UI design. Foley et al.[39] describe two types of models at this level: user-task models and application models. All of the model-based approaches contain at least one of these models, though, at present, different approachs use different models[87]. This section investigates current approaches to model-based code generation and identifies a software composition problem inherent in the approach. This problem, which we call the binding problem, motivates one of the technical contribution of this thesis.

3.3.1 Model-Based Analysis and Generation

The model-based approach is relatively new, but many precursors have been developed. These tools vary in the models they support and the notations used to express the models[87]. All of the approaches contain either a user task model, an application model, or both, with other models (like presentation) augmenting their functionality. We use these models as a basis for comparison.

3.3.1.1 Application Modeling

The earlier model-based approaches chose the application model as the cornerstone. Many of these approaches operate directly on software engineering models. The earliest applications of this approach began as semantic level extensions to UIMS architectures and technology. In Genius[55], designers specify data models and augment these with a behavior abstraction called a *dialogue net*

(similar to a Petri-net[86]). From these specifications, GENIUS uses a knowledge base to select interaction objects, arrange layout, and incorporate the dynamics into an executable implementation. In the context of our earlier distinction, GENIUS provides languages for the semantic level (data models) and the syntactic level (dialogue nets) but no language for the lexical level. As a result, interfaces generated by GENIUS have stereo-typical forms-based presentations.

A similar data/behavior approach is used in the TRIDENT environment[100]. Data models in TRIDENT are expressed using entity-relationship diagrams, and the behavior is expressed using a notation called an activity chaining graph (ACG). In addition, TRIDENT represents lexical attributes in the form of Abstract Interaction Objects (AIOs) and Concrete Interaction Objects (CIOs). Though TRIDENT does not contain an explicit task model, its representations are derived from a TKS[57] specified hierarchical task model. The melding of methodology and model representation in TRIDENT is impressive. In [15], Bodart et al. demonstrate how TKS models suggest entities and relationships, from which the data model can be derived, and temporal ordering cues from which the Activity Chaining Graph can be derived. Furthermore, these models suggest candidate abstract interaction objects that constitute the presentation of an interactive system[14].

The pinnacle of application modeling is the User Interface Design Environment (UIDE)[37, 95]. Systems are expressed in terms of an application model, and the environment provides run-time support for model inferencing. Salient features include explicit pre- and post-conditions which can be attached to any actions in the model. Rather than inferring dialogue once, UIDE actually infers dialogue at run-time by computing actions whose pre-conditions are satisfied. This requires an active dialogue manager component that observes the state of the system and reacts by disabling actions whose pre-conditions are not satisfied and enabling actions whose pre-conditions are satisfied.

3.3.1.2 Task Modeling

According to Diaper[30, preface], task analysis is potentially the most powerful method for producing interactive system requirements specifications. If this is truly the case, it makes sense to include task models in a model-based development environment. The idea is that user interfaces will be more usable if they comfortably implement the user's mental model of the task he or she is attempting to perform. Just as dialogue notations became a unifying feature of UIMS generators, user task models have become the unifying model of different model-based approaches. On the

analysis side, tools like Glean[61] allow designers to analyze task models in terms of human mental cognition and observable human action (GOMS[22]).

Other approaches view task models as evolving entities within the life cycle of an interactive system. The ADEPT environment[58, 102] supports the evolution of task models throughout a system's lifetime. In addition to iterative task model support, ADEPT models can be refined into an implementation model for which there is a code generator. The MASTERMIND environment[74, 97, 18], uses a task model as its design requirements model.

3.3.1.3 Presentation Modeling

Systems generated from task and application models often have interfaces that are not highly graphical. Szekely et al.[98] demonstrate the magnitude of this point in a side-by-side comparison of a screen-shot from a tree visualization tool (TreeViz) and the small portion of it that is describable using a UIMS. This is an important point because many of the model-based approaches generate UIMS scripts as their output. A natural conclusion is to extend these approaches with new models. The most common addition is an explicit presentation model. Humanoid[98], and now MASTERMIND[74], provide a more elaborate presentation model.

3.3.2 The MASTERMIND Approach

The MASTERMIND (**M**odels **A**llowing **S**hared **T**ools and **E**xplicit **R**epresentations **M**aking **I**nterfaces **N**atural to **D**evelop) project uses explicit task, presentation, and application models for the design and implementation of interactive systems. The research in this thesis began with an attempt to generate applications from the integration of MASTERMIND models.

The MASTERMIND vision is to provide support for designers of interactive systems. There may be more than one designer, and they may be broken into teams. Support for model analysis and multiple design teams requires infra-structure for modularization and integration of models. To this end, the MASTERMIND environment has a model server for storing and retrieving models at run-time. With dynamically queryable models, the MASTERMIND environment can support design critics and the management of context sensitive run-time help[94]. A similar vision was undertaken by the Mecano (now Mobi-D) project[85, 84]. Mobi-D uses an object oriented modeling language called MIMIC to describe and relate a multitude of models.

MASTERMIND is intimately concerned with generating applications from the models. The code generation meta-model is one in which each MASTERMIND model has an associated code generator and an elaborate model-based linking process connects data and behavior in the code generated from each model. The presentation model generates code using features from the Amulet[72] toolkit, the task model generates code using features from the MASTERMIND toolkit (MMTK), and the application model code generator is currently under development.

3.3.3 The Multi-Model Binding Problem

Models are not programming languages. They represent some aspects of a system and are neutral with respect to others. Inevitably, in an executable application, model-generated code depends upon (and is depended upon by) non-model generated code. Consider, for example, compilers built in part using parser generator technology like yacc[59]. Part of these systems is generated from a model (the grammar), and the other part is written by a programmer. Often the generated code must include snippets of programmer written code in the form of embeddable actions. The code in these actions is affected by the parse, and conversely, action code can affect the parse. The point is that there are (often complex) dependencies between model-generated code and human-written code. When multiple models generate code that must cooperate, the complexity of these dependencies can be greatly exacerbated. We call this the **multiple model binding problem**.

The binding problem is inherent in interactive system design. A task modeling notation, for example, is useful for describing and reasoning about the sequencing of activity necessary to perform a user's task. In actual implementations, however, users do not directly engage in tasks. Rather, they perform tasks by observing and manipulating graphical entities (like menus, draggable icons, and buttons). As run-time components, tasks must inter-operate with these graphical entities in order to share control, communicate status, and exchange data. What's more, in the MASTERMIND vision, these graphical entities are generated from a presentation model.

Whereas the binding problem refers to the dependencies that arise when different components need to co-operate, **binding** is the act of resolving these dependencies. At the level of models, binding is the unification of behavior in one model with behavior in another. At run-time, binding is a policy for sharing data and control among components generated from separate models.

Unfortunately, at the implementation level, binding is non-trivial because the presentation components with which task components must cooperate are designed to encapsulate the very functionality that must be woven together through binding. As a result, implementation binding is difficult to express declaratively. Our research overcomes this obstacle by:

1. formally defining binding within the confines of our task language (MDL), and
2. designing composable software toolkit abstractions which implement bindings directly.

3.4 Formal Approaches to Composition

Composition and de-composition in user interface software has been the subject of many different projects. On one end of the spectrum are conceptual "architectures" like Seeheim and its derivatives[82, 9], Model-View-Controller (MVC)[62], and Presentation-Abstraction-Control (PAC) [29]. These give general guidelines for separating functionality into components and then linking these components into a final system. On the other end are rigorous semantic models of interactive software structure like Paterno's theory of interactors[80]. The thesis of our work is that multi-model composition must be rigorously specified. We, therefore, follow in the spirit of Paterno, but we end up with radically different systems because of a difference in modeling philosophy.

3.4.1 Decomposition Heuristics

In the early days of UIMS systems the need for user interface structuring guidelines became clear. The first so-called UIMS architecture was presented at the Workshop on User Interface Management Systems in Seeheim Germany[82]. This architecture equated each of Foley's linguistic levels with a software component. In making the levels components, detail is encapsulated with access provided only through an application programming interface (API).

Problems with the rigid approach in the Seeheim architecture led researchers to consider other manifestations of levels. The so-called *multi-agent* architectures, which include SmallTalk's Model-View-Controller (MVC)[62] and Coutaz's Presentation-Abstraction-Control (PAC)[29], fall into this category. Multi-agent architectures structure a system as a collection of cooperating agents. An agent is a complete information processing system with attributes from each level. Agents communicate with other agents by observing, acting on, and issuing events rather than making sequential

procedure calls and waiting for their return. Multi-agent architectures support a higher degree of interface concurrency, feedback, and locality than layered architectures.

At present, there is no consensus on the proper interactive system decomposition[11]. The current population of generic decompositions serve rather as design heuristics which may or may not be appropriate to a specific application. Of course, being heuristics, these decomposition mechanisms are not rigorous enough to model software composition to the extent we require.

3.4.2 A Connection Oriented View of Composition

The connection-oriented approach views composition as a mechanical process of connecting outputs of one component to inputs of another to synthesize a new component. This new component itself has input and output connections and may be connected with other components to synthesize still larger components. Connection-oriented composition is a natural formalization of decomposition heuristics (like PAC) that decompose a problem recursively into smaller instances of the same problem. The view is conceptually appealing because it evokes images of constructing an interactive system by *wiring* together independent components in much the same way one would wire up a hardware device. The most formal treatment of the approach is Paterno's theory of interactors[80]. The approach begins historically with the PAC architecture.

The multi-agent frameworks like PAC decompose a user interface implementation by hierarchically distributing functionality. A PAC triad (presentation, abstraction, control), for example, decomposes into PAC sub-triads whose constituents are wired together to implement the higher level functionality. The resulting structure distributes semantic, syntactic, and lexical properties across a hierarchy of entities. Figure 11 demonstrates how this works. The example is a presentation which has a pie chart and a numeric display field. The abstraction maintains the values to be displayed, and the controller C, which controls the panel containing the pie chart and the numeric field, decomposes into two smaller controllers C_1, which controls the pie chart presentation, and C_2, which controls the numeric field presentation. The controller C aggregates these sub-controllers and resolves their presentation/abstraction connection needs. It is claimed by Coutaz[29] that this distribution allows the structure of the software to better match the cognitive organization of human knowledge because it does not organize functionality into hermetic layers[6].

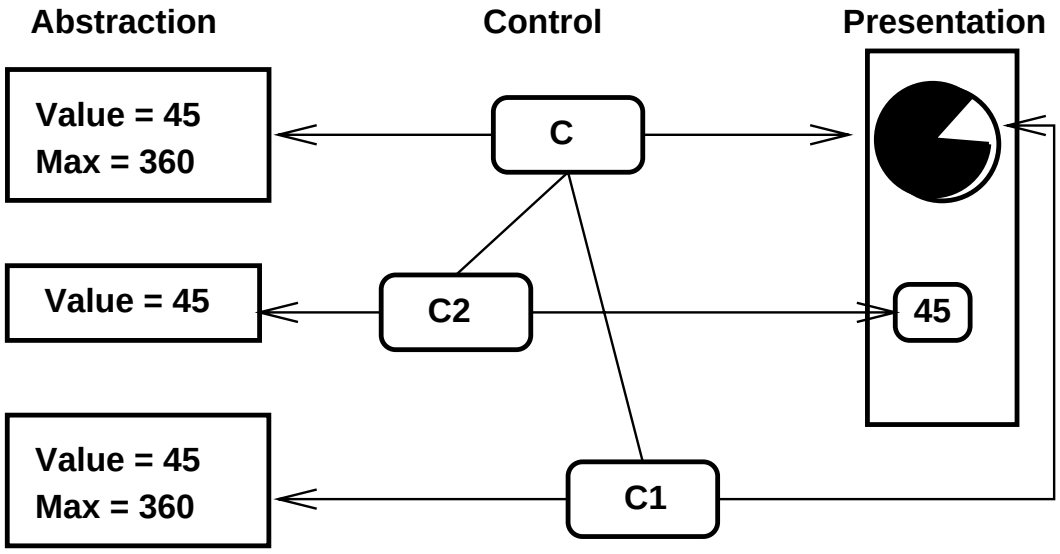

Figure 11: Composition by Connection in the PAC architecture.

Paterno's theory of interactors[80] adds precision and mathematical rigor to this connection-oriented view of interface composition. An *interactor* (not to be confused with the Amulet notion of an interactor) is a software component with a fixed set of incoming and outgoing data-flow paths. These paths support the transfer of control, data, and presentation and may be connected to paths of the opposite polarity in other interactors. In fact, Paterno identifies five different classes of composition which may be attained by connecting data paths in various configurations. Interactors are defined formally as LOTOS processes[81], and can be related to task models specified in LOTOS. This theory culminates in a methodology called TLIM for analyzing task models and system models (interactor-based). This is discussed in [78]. The idea is that by expressing a task as a LOTOS process P_T, an interactor-based solution can be derived from this specification. The interactor solution is the synchronous composition $P_{I1} \parallel P_{I2} \parallel \ldots \parallel P_{In}$ where each P_{Ii} is the LOTOS process describing an interactor. Related to Paterno's work is that of Markopoulos et. al.[63, 64] which use a slightly different formalization of interactor called the ADC interactor. ADC interactors are more PAC-like than the interactors used by Paterno.

One consequence of connection-oriented composition techniques is the distribution of functionality among independent components. The components view themselves (and their neighbors) as concurrent processes which occasionally receive input data, compute some function of this input

and then issue the output down an output data path. Concurrency and data-flow are implicit in this model of control; yet modern PC's and workstations typically only have one processor. Somehow the concurrency assumed by these frameworks is removed and implemented on a sequential architecture. This aspect of control will, in general, be difficult to detect in a LOTOS specification, although many task and interactor specifications use only a *regular* subset of LOTOS.

A concurrent modeling notation which captures this distributed locus of control is the Petri-net[86]. Petri-nets represent complex concurrent control by passing tokens along control paths in a graph. Palanque et al.[77] use a special variant of nets called hierarchical Petri-nets to represent the relationship of task models and system models in user interface software. In this work, they demonstrate a degree of completeness by modeling the operators of Hartson's User Action Notation (UAN). This allows them to define complex dialogues in terms of an hierarchical net and then analyze this net for properties like deadlock freedom. Petri-nets can not only express concurrency but can also be simulated on sequential machinery by an interpreter. For this reason, they are often used as a target of translation from more abstract user interface description notations. The TADEUS environment[88] adopts this approach.

3.4.3 A Constraint Oriented View of Composition

The connection-oriented approach to composition is familiar, but it is not the only possible mechanism for composing interactive system software. The approach we appeal to in this thesis views composition as the simultaneous satisfaction of multiple *constraints*. This constraint-oriented view seems a better fit for composing software generated from models as diverse as the MASTERMIND models, but it is not as visually familiar as the connection oriented approach. This section introduces the constraint-oriented by first discussing drawbacks of the connection-oriented approach and then motivating the view in its historical context.

We noted that the PAC and interactor-based approaches necessarily interleave and distribute task, presentation, and application detail among inter-dependent software components. This is feasible when designers are building the components from the bottom up, but it has a consequence which seems inconsistent with the MASTERMIND vision of modeling. Recall that Paterno's interactors have a rigid structure of data paths and connection obligations. MASTERMIND models, particularly the presentation model[25], have features which do not easily map into this world.

We believe that this aspect of MASTERMIND models makes the connection-oriented approach to implementation unfeasible.

MASTERMIND models match a specialized specification paradigm to a particular problem. The presentation model uses concepts and terminology from graphic design and employs a declarative, constraint based, model of interface layout. The task model, on the other hand, expresses explicit sequencing within the global control of the system and employs a process notation. The problem of binding models is actually a problem of multi-paradigm composition[106]. Zave argues that in such situations, conceptual structures other than the familiar inter-language procedure call are necessary for composition.

The conceptual structure we used views the realization of different models as processes and their binding as synchronizing composition. In her work on the CSP specification of concurrent dialogue, Alexander[4] presents a technique for modularizing dialogue through synchronous parallel composition. In her examples, dialogue specifications are separated into presentation and application components which are specified independently and then combined using the || (parallel composition) operator. Under this approach, binding can be thought of as the simultaneous satisfaction of two constraints (the presentation and application in Alexander's case) and can be represented formally as an operator in a process calculus. The constraint-oriented model of composition seems natural for composing models defined top-down (as is done in MASTERMIND); whereas the connection-oriented approach is natural for composing models defined bottom-up (as is done in Paterno's interactor approach).

Of course, constraint oriented composition requires a different way of thinking about software synthesis[106]. Zave[105] suggests synthesizing a program by applying a series of *transformations* to an initial abstract program. Each transformation in the series incorporates a different constraint. Successive transformations enrich a program from an abstract specification into a concrete executable application. Transformations enrich programs without violating their meaning. That is, constraints are additive[104]. We feel this is characteristic of how presentation models should relate to task models, and we contend that model-based environments which support task models should use the transformational constraint-oriented techniques for code generation. We, therefore, designed the MMTK to support this.

Others have looked at using constraint-oriented methods for separating user interface functionality. An interesting application of the constraint view of decomposition comes in the definition of simultaneous constraints over control and data. Suffrin and He[93], for example, combine CSP constructs with the ability to precisely define states using Z[91] schemas. Abowd's agents[1, 2] use a similar mixing of CSP and Z to support usability analysis.

3.5 Virtual Machine Design

Our strategy for designing the MDL virtual machine and the MMTK toolkit is a twist on what has become a popular idea. The popular idea is the use of model checking to validate properties of formal user interface specifications. This idea was first presented by Paterno[79], and has been followed by Abowd et al.[3] and Paterno et al.[67]. The basic idea is that often formal specifications of user interfaces use only regular subsets of formal notations (like CSP and LOTOS) and that many usability properties can be expressed using temporal logic. This opens the door to using a model checker to exhaustively validate that the specification satisfies the temporal properties. We have used model checking to validate compositional properties of software components which *implement* formal notations like CSP and LOTOS. This section introduces model checking, the application of model checking to validate dialogue specifications, the use of model checking to validate compositional properties, and the limit of temporal logic for expressing the semantics of a notation like LOTOS.

3.5.1 Temporal Logic

Temporal logic is an extension of predicate logic with operators that express the time-relation of predicates. An introduction to the use of temporal logic in concurrent systems programming is provided by Schneider [89]. Temporal operators include $\Box$ (read "henceforth"), $\Diamond$ (read "eventually"), and $\bigcirc$ (read "next") to name a few. If P is a predicate, the temporal formula $\Box P$ states that P will hold from now on, $\Diamond P$ states that P will hold at some point in the future, and $\bigcirc P$ states that P will hold at the next moment in time. Different sets of temporal operators are supported by different *models of time*. A model of time is a representation of the instants of time. For example, the linear model of time represents instants of time as a sequence. This means that any instance

has a unique next state and a unique history. Alternatively, the branching model of time represents instants of time as nodes in a state graph. Being a graph, instants of time have multiple futures and indeterminable histories. Dense models of time represent instants as numbers in the set $\mathbb{R}$. With a dense model of time, there can be no $\bigcirc$ operator (because in $\mathbb{R}$, there is no "next" instant. With a branching model of time, there are no past operators because the past is non-determinable in a graph based model of time.

The branching model has recently been incorporated into model checking tools. One particular variant, called Computation Tree Logic (CTL)[27], is a branching time logic with temporal operators and path quantifiers. Since the model of time in a branching time logic is a graph, temporal predicates can be applied to one or all *paths* emanating from the current instant in time. Moreover, CTL allows the existential and universal quantification of a temporal predicates over paths.

3.5.2 Model Checking

Model checking is a technique for validating finite automata for adherence to constraints phrased in temporal logic[26]. A model checker is a tool which inputs a description of a finite state model and a set of temporal constraints, builds a representation of all executions of this model, exhaustively checks these executions for adherence to the constraints, and reports violations. Model checking has become popular as a design tool in software and hardware engineering because of its ability to detect subtle design flaws and report the failures in a way which helps track the flaw. Some classes of failure can be reported in the form of a *counter-example* that demonstrates the violation of a predicate. A counter-example is a sequence of machine behaviors (changes in state or the issuing of events) which serve to demonstrate the failure of a temporal constraint. Recent work in symbolic model checking[21] utilizes an efficient representation of finite state spaces to make the checking of large models (on the order of 10^{20} states) feasible.

Model checking has been used to validate many types of software specifications. Atlee and Gannon[8] verify safety properties of requirements expressed in the SCR notation. Their approach maps a finite subset of SCR onto a state machine which is then fed into the model checker. Jackson[53] has investigated using model checking to validate properties of Z and VDM specifications. Since these specifications often describe infinite state spaces, a finite state abstraction must be applied in order to use the model checker. Jackson takes advantage of equivalence classes

on relations to create this abstraction. In [54], Jackson and Damon apply these techniques to the design of a style editor in a word processor. Wing et al. also demonstrate the need for abstraction by applying model checking in a case study verifying the cache coherency protocol of a distributed file system[103]. Allen and Garlan[5] use the FDR[40] tool to check whether or not one CSP process is a refinement of another. In this thesis, we use the Symbolic Model Verifier (SMV)[66] from Carnegie Mellon University to verify the safe composition of our ordering components.

Many interface usability properties can be expressed concisely through temporal constraints. Abowd et al.[3] lists several, including:

1. task completeness, which states whether or not the user can always accomplish a goal,
2. state inevitability, which states that a user can always find a way into some critical state (like saving a file),
3. reversibility, which states that an action can be reversed (by the next action).

They express dialogue using Olsen's Propositional Production System (PPS)[76] which has been shown to be equivalent in expressive power to the regular expressions. Abowd uses CTL to express these properties and uses SMV to check them. Paterno[67] uses a logic called ACTL[75] which can express equivalence relationships between CCS[68] terms. This relates task and system models which are both defined as LOTOS processes in Paterno's framework. They use an ACTL[42] model checker to carry out these validations. The problem they had to overcome is that, in general, LOTOS specifications denote infinite state processes and are, therefore, not checkable using model checking techniques. However, as noted by Fantechi et. al.[35], subsets of LOTOS operators yield regular languages which can be synthesized into finite state models.

3.5.3 Compositional Model Checking

Model checking is inherently limited by the size of a state space it can analyze. When a state space is too large, a model checker cannot perform its analysis in a reasonable time. This problem is exacerbated by use of concurrency as a means of separating concern in specifications. Given n process specifications each of size m, the parallel composition of these is a process whose state space can be of size m^n. This is commonly referred to as the state explosion problem. Attempts to overcome this obstacle appeal to a technique called **compositional model checking**. The

idea behind compositional model checking is to validate global properties by model checking local properties. In practical terms this means performing n separate model validations on the processes whose state spaces are of size m as opposed to one single model validation on the composed process whose state space is of size m^n. If the different components did not interact, then this is trivial; but in real specifications, components interact.

Compositional model checking takes advantage of the fact that, for many specifications, the size of the composed state space is really much smaller than m^n. As an example, consider two processes, P_1 and P_2, and their composition, $P_1 \parallel P_2$. When P_1 and P_2 are composed, P_1 doesn't observe *every* action of P_2, and P_2 doesn't observe every action of P_1. Clarke et al.[28] suggested abstracting that portion of a processes externally observed behavior into a smaller process called an *interface process*. Say, for example, that I_1 is an interface of P_1 and I_2 is an interface of P_2. Clarke et al. list a set of rules stating when a property ϕ of $P_1 \parallel P_2$ can be checked by checking ϕ in $P_1 \parallel I_2$ and $I_1 \parallel P_2$. Applications of this general approach to the validation of Ada task compositions have been performed by Fischer and Gerber [36] and Bultan, Fischer, and Gerber[20].

We use model checking to validate properties of MMTK component compositions. Our use deviates from the approaches mentioned above in that we handle compositionality without articulating an explicit interface process. Rather, we prove a theorem about testing adequacy for two compositional properties (receptiveness and freedom from divergence). In some sense, this theorem demonstrates the existence of an *implicit* interface process, but it does so without us having to define this process.

3.6 Summary

In summary, the composition of MASTERMIND models requires a strategy for software composition based on synchronization of concurrent processes. We define the MDL language to express task models and binding within a notation that expresses synchronized concurrency as a primitive operator. We then develop a toolkit of reusable components (MMTK) which implement MDL operators and act as an MDL virtual machine. The remainder of this document outlines the MDL language, the MMTK toolkit, our use of model checking to validate the behavior of toolkit components, and the testing adequacy theorem. We conclude with validation of our approach on examples.

Chapter 4

The MDL Language

The MASTERMIND Dialogue Language (MDL) is a notation for specifying interactive system task models and composing these with presentation models. MDL was designed to ease the specification of these models and precisely articulate their composition. MDL has three features which make it novel. The first is a distinction between tasks, presentations, and bindings of tasks to presentations. This distinction shields designers from subtleties which complicate the use of process-oriented notations for interactive system design. The second is a rich set of primitive ordering operators that are useful for specifying interactive systems. The third is a mechanism for declaring dynamically instantiable tasks. Taken together, these features make MDL a suitable foundation for model-based interactive system development environments.

MDL describes systems as a hierarchy of *process* definitions. A process can be thought of as a mechanism for performing actions in a prescribed manner. Processes perform actions and interact with other (concurrent) processes. Process interactions are built up out of atomic units called *events*. Processes which participate in a common event are said to *synchronize*. That is, each process observes the event at the same moment in time. Complex processes may be built by either combining sub-processes through an ordering operator (i.e.- process C is the sequential composition of sub-processes A and B) or by conjoining sub-processes so that they run independently but synchronize on common events. In interactive system design, processes serve as the formal basis for task models and for the *binding* of task and presentation models.

We present MDL in four sections. The first (Section 4.2) gives the basic syntax of MDL and describes the different flavors of process which combine to form an interactive system specification. We then describe the different ways to construct MDL *behavior expressions*. Behavior expressions are the language for specifying how processes decompose into subprocesses and events. We present the control side of behavior expressions first and then talk about adding data. This allows the

Table 1: High Level Syntax of MDL

model	::=	(*event_decl*‖*task_decl*‖*pres_decl*‖*bind_decl*)*
event_decl	::=	**event** identifier { *data_type* } ;
task_model	::=	**task** identifier *process* **endtask**
pres_model	::=	**pres** identifier *process* **endpres**
bind_model	::=	**bind** identifier *process* **endbind**
process	::=	*paramlist* **is** *behavior_expression* submodels **endtask**
paramlist	::=	**parameters** *parameter**
parameter	::=	identifier ':' *type*
submodels	::=	(*task_decl*‖*pres_decl*‖*bind_decl*)*
identifier	::=	['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9''_']*

definition of communications and local parameters. The chapter concludes with a discussion.

4.1 Structure of MDL

MDL allows designers to specify task models and the binding of task and presentation models. The language consists of two kinds of definitions: *event* definitions, which declare atomic user and system actions, and *process* definitions, which declare complex behaviors in terms of sub-processes and events. Table 1 lists the high level syntax of MDL.

There are three kinds of process definitions:

task which encodes the procedure by which a user performs a task,

presentation which is an abstraction of the behavior of code generated by the presentation model, and

binding which connects task and presentation behavior.

To make an analogy with programming languages, task processes are like programs, presentation processes are like external declarations (e.g. the C language `extern` statement), and bindings are the linkage rules which specify how to connect to an external behavior so that it can be used by a program. The general scenario of design and development is:

1. create a task model, which is a collection of MDL task processes;

2. create a presentation model and extract MDL presentation process descriptions from this other model;

3. combine task and presentation processes into binding processes in MDL;

4. invoke the model-based code generator to render an executable user interface from all of these pieces.

The definition of MDL processes factors heavily into this procedure.

After beginning the declaration of a process using either the **task**, **pres**, or **bind** keywords, designers then declare data and event parameters which the processes will manipulate during execution. The **parameter** section of an MDL task description specifies zero or more data values and/or event names. Parameters serve two purposes. They act as local data that can be used during the execution of a task, and they can be supplied when the process is *instantiated*.

After defining any parameters, the designer then describes the behavior of the process. This is done by either composing sub-processes through an ordering operator or prefixing sub-processes with communications. These descriptions are called *behavior expressions*. Behavior expressions order the occurrence of events and sub-processes in time. MDL contains a rich set of operators for defining behavior expressions.

After defining the behavior expression, designers can elaborate the definition of sub-processes referenced in the behavior expression. Sub-process definitions have exactly the same structure as process definitions. Furthermore, the parameters of parent processes may be referenced within sub-processes.

The dynamic aspect of a process is described in its *behavior_expression*. The syntax of these expressions is listed in Table 2. In this table, lower case letters like e represent events; whereas capital letters like B represent behavior expressions. Using this syntax, complex behavior expressions can be built. In the sections that follow, we describe the meaning of each operator by means of an example and a formal definition.

Table 2: Syntax of MDL behavior expressions.

Name	Syntax
inaction	**stop**
action prefix	$e \,;\, B$
event concealment	**hide** $e_1, e_2, \ldots, e_n$ **in** B
choice	$B_1 \mid B_2$
interleaving	$B_1 \mid\mid\mid B_2$
parallel synchronization	$B_1 \parallel B_2$
sequential composition	$B_1 >> B_2$
disabling	$B_1 [> B_2$
mutual disabling	$B_1 \leftrightarrow B_2$
interruption	$B_1 \triangle B_2$
optional	B^{opt}
loop	$B*$

4.2 A Notation for Semantics

We use a formal notation to help describe the semantics of MDL operators. Specifically, we use structural operational semantics[83] (SOS). SOS provides a way to systematically derive the meaning of a behavior expression from the syntax of the expression and is, therefore, a natural mechanism for describing the operators.

In process notations like MDL, progress is made when a process performs an observable action. In order to perform an observable action, two parties must observe the action simultaneously. To make this idea precise, let the *environment* of a process P be the set of processes with which P interacts and an unspecified human observer (the user of an interactive system). Then P performs an observable action e when P and the environment of P both observe the same event e at the same time. We call this mutual observation of an action a *synchronization*. If P is a process, $\alpha(P)$ (read the "alphabet of P") is the set of events which P can observe.

SOS is a vehicle for building *synchronization trees* from the syntactic structure of a process definition. Synchronization trees are tree structures whose nodes are behavior expressions and whose directed edges represent event synchronizations. Given the behavior expression: $e \,;\, ((f \,;$ **stop**$) \mid (g \,;\, h \,;$ **stop**$)$ we can describe all of its possible behaviors using the tree in Figure 12. Edges in the tree are interpreted as event synchronizations which transform the behavior expression named in the source node into the behavior expression named in the target node. When the behavior

expression in our example synchronizes with its environment on event e, it continues to behave as the process $(f \fatsemi \mathbf{stop}) \mid (g \fatsemi \mathbf{stop})$. This process is prepared to synchronize on either f or g, as represented by branching in the synchronization tree. We denote the tree edge which maps a process A into another process B as a result of synchronizing on event e, by: $A \overset{e}{\rightsquigarrow} B$.

Plotkin[83] observed that we can define the meaning of an ordering operator as a pattern for constructing synchronization trees. Furthermore, these patterns can be expressed using axioms and inference rules. The inference rule for the choice operator |, for example, looks like this:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \mid Q \overset{e}{\rightsquigarrow} P'}$$

This rule says that if process P can synchronize on event e to become process P', then process $P \mid Q$ can synchronize on event e to become process P'. This defines the meaning of choice. This rule tells us that the reason there was a branching in the synchronization tree of Figure 12 was that two behavior expressions could be inferred from the expression: $(f \fatsemi \mathbf{stop}) \mid (g \fatsemi \mathbf{stop})$. Inference rules can also contain conditions which are not part of the antecedent. These appear in square brackets to the right of the inference bar. In the rule that follows, C is such a condition:

$$\frac{P}{Q} \quad [\, C \,]$$

We adopt SOS as a means of expressing the meaning of MDL ordering operators.

4.3 The Building Blocks of Processes

MDL includes an *inactive* process called **stop**. Inactive means that **stop** cannot synchronize with any external events. In many notations, **stop** means deadlock. In MDL, however, **stop** is intended as a terminator so that we can define *prefixing* in a standard way. Prefixing, is the act of creating a new process by prepending another process with an event. If P is a behavior expression and e is an event, then $e \fatsemi P$ (read "e prefix P") is another process which first synchronizes on e and then behaves like P. We have already seen an example of building up a process by prefixing in Chapter 2. Recall the task *InstructToLand* was created by prefixing the event *commitToLand* with **stop**. Prefixing a process with an event yields another process.

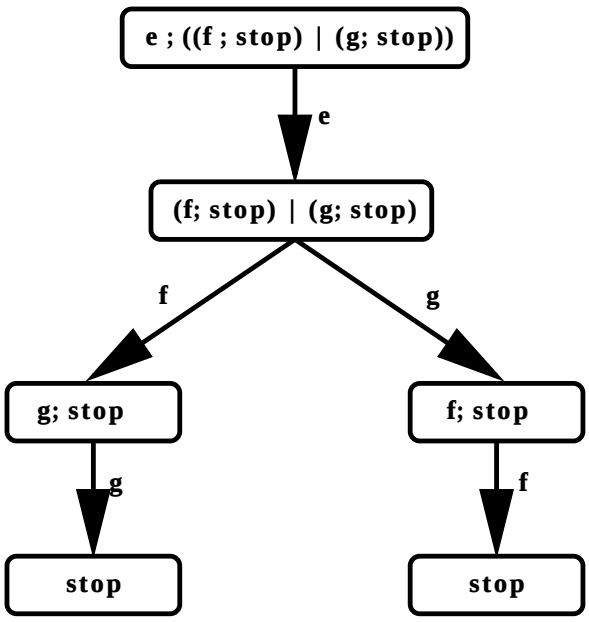

Figure 12: An example process synchronization tree.

The SOS behavior of a prefixed processes is as follows:

$$\frac{}{e \,;\, P \overset{e}{\rightsquigarrow} P} \quad [\, e \in \alpha(P) \,]$$

This says that a process P prefixed by an event e can synchronize on e, and that the resulting process behaves like P.

With prefixing, we can create arbitrarily long event sequences, but as yet, there is no way to express any user choice. For this, MDL provides an operator | (called choice) that combines two behavior expressions into another that behaves like one or the other of them but not both. In MDL, the environment actually decides which choice is taken.

More formally, the MDL choice operator | combines two processes P and Q into another $P \mid Q$ (read "P choice Q") which behaves like *either* P or Q but not both. Furthermore, the committal of a choice is made by the first event synchronization happening in either P or Q. The following

SOS rule formalizes these concepts:

$$\frac{P \overset{e}{\rightsquigarrow} P' \qquad Q \overset{f}{\rightsquigarrow} Q'}{P \mid Q \overset{e}{\rightsquigarrow} P' \qquad P \mid Q \overset{f}{\rightsquigarrow} Q'}$$

This says that if P can successfully transform into P' by synchronizing on event e, then the choice expression $P \mid Q$, upon synchronizing event e will signify that P was chosen and will, thus, transform into P'. Ditto for f and Q. What happens if $P \overset{e}{\rightsquigarrow} P'$ and $Q \overset{e}{\rightsquigarrow} Q'$? That is, what is the meaning of choice if both P and Q can synchronize on the same event e? The choice will be non-deterministic.

4.4 Parallel Composition

The MDL operator which establishes contexts for event synchronization is the parallel composition operator $\|$. This operator conjoins two processes, stating that they may behave concurrently and independently, but that any events in the shared alphabet of the processes must be synchronized. The functionality of this operator is often used to separate complex behavior into a collection of interacting processes using a principle called composition by conjunction [104]. The idea is that complex temporal orderings can be expressed as the simultaneous satisfaction of multiple ordering *constraints*. The synchronization implied by the $\|$ operator is the agent of this simultaneity. This synchronization is the mechanism for the MASTERMIND notion of binding, and as will be discussed, the MDL syntax **bind** ... **endbind** always combines a task and presentation process using $\|$.

The expression $P \| Q$ describes the interleaved execution of P and Q with the exception that if an event occurs in the alphabets of both P and Q, that event can only be observed in one process when it can simultaneously be observed in the other. The practical implication of this idea is that these common events block a process from continuing until it can synchronize with another process. If some event e is in the alphabet of both P and Q, and if P is prepared to synchronize on e but Q is not, then P will block until Q reaches the synchronization point. Events which are not in the alphabets of both processes do not block. The following rules capture the semantics of this

operator:

$$\frac{P \overset{e}{\rightsquigarrow} P' \qquad Q \overset{e}{\rightsquigarrow} Q'}{P \parallel Q \overset{e}{\rightsquigarrow} P' \parallel Q'}$$

This is the usual case. In situations where both P and Q can engage in the same event e, they do so by synchronizing on e. As we will later see, event synchronization may be augmented with the passing of data between the processes. The other rules deal with cases in which P and Q may proceed independently (without synchronization). This is only possible when the event is not in the alphabet of the other process:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \parallel Q \overset{e}{\rightsquigarrow} P' \parallel Q} \quad [\, e \notin \alpha(Q) \,]$$

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \parallel Q \overset{e}{\rightsquigarrow} P \parallel Q'} \quad [\, e \notin \alpha(P) \,]$$

A variant on the $\parallel$ operator is a non-synchronizing version called interleaving ($|||$). Interleaving specifies that two processes run concurrently, and if they have common events, they do not synchronize. The rules for interleaving are:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \,|||\, Q \overset{e}{\rightsquigarrow} P' \,|||\, Q}$$

Of course, since $|||$ is commutative, this rule works for Q without loss of generality. And of course:

$$\frac{\mathbf{stop} \,|||\, P}{P}$$

4.5 Enabling and Disabling

Processes can be ordered using the enabling operator (>>). Given two processes P and Q, we say that P precedes Q by saying $P >> Q$ (read P enables Q). The rules for enabling are:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P >> Q \overset{e}{\rightsquigarrow} P' >> Q}$$

and, of course,

$$\frac{\textbf{stop} >> Q}{Q}$$

Disabling (denoted [>), which we use to implement all of the interruptible operations, is handled as follows:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P [> Q \overset{e}{\rightsquigarrow} P' [> Q}$$

If the left process completes before being disabled, then the result is **exit**:

$$\frac{\textbf{stop} [> B}{\textbf{stop}}$$

Finally, if the disabling process observes activity before the first process completes, then the first is disabled:

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P [> Q \overset{e}{\rightsquigarrow} Q'}$$

In graphical user interface design, disabling is common. During a long operation, user interfaces often provide a cancel button. When a process can be disabled by another process, a symmetric disabling may occur in the other direction. This occurs, for example, with activities governed by an hierarchical radio button panel. With an hierarchical radio-button panel, complex interactions

are enabled by choices in a radio button panel. If a different button is pressed during one of these complex interactions, the activity initiated by the first button is terminated and another enabled. This may happen many times. The User Action Notation (UAN)[49] defines a special operator $\leftrightarrow$ (called "mutual disabling") just for such a purpose. We adopt this operator in MDL. It is defined in terms of disabling as follows:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \leftrightarrow Q \overset{e}{\rightsquigarrow} P' \,[\!>\, (Q \leftrightarrow P)}$$

Note that if the P in this rule ever completes we have:

$$\mathbf{stop} \,[\!>\, (Q \leftrightarrow P)$$

which, by the exit rule of $[\!>$ yields **stop**. The symmetric rule is:

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \leftrightarrow Q \overset{e}{\rightsquigarrow} Q' \,[\!>\, (P \leftrightarrow Q)}$$

4.6 Interruption

It is often the case that certain tasks need to be performed in extra-ordinary situations, but that in the normal case they are not required. The need for these tasks is dictated by some unpredictable asynchronous event like a system failure. Tasks that must be performed in response to such events have a preemptive nature which designers must express relative to other tasks. In CSP, there is an "interruption" operator ($\triangle$) which effectively terminates the interrupted task. We adopt the same notation with a slightly different meaning. In MDL, the $\triangle$ operator establishes an interrupt/resume relationship between two tasks.

Suppose during the normal course of operation, a file browser interface supports the task *BrowseFiles*. If some catastrophe occurs, perhaps a file system corruption, then the user must perform a file system check and repair (fsck) before continuing. Let the task *FileSystemCheck* describe this repair task, and finally, let the event *corruption* denote the file system corruption. Then:

$$BrowseFiles \triangle (corruption \fatsemi FileSystemCheck)$$

describes the interruption of the normal *BrowseFiles* task when the corruption occurs. Interaction with *BrowseFiles* will be suspended until the *FileSystemCheck* task is completed. Note that this use of interruption with an asynchronous guard event preceding the preemptive task is stereo-typical.

There are three rules governing the semantics of the interruption operator. The first says that a completed task cannot be interrupted:

$$\frac{\textbf{stop} \triangle P}{\textbf{stop}}$$

The second rule describes the normal (uninterrupted) behavior of the system. An interruptible task behaves as usual if it is not interrupted!

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P \triangle Q \overset{e}{\rightsquigarrow} P' \triangle Q}$$

The third rule describes how the interruption actually works. Once the interrupting task begins, it must complete, at which point the interrupted task resumes where it left off, and it is again interruptible by Q.

$$\frac{Q \overset{e}{\rightsquigarrow} Q'}{P \triangle Q \overset{e}{\rightsquigarrow} Q' >> (P \triangle Q)}$$

The way this rule is defined, preemptive tasks may, themselves, be preempted. Consider $A \triangle (B \triangle C)$. Designers can prioritize preemptive tasks by daisy-chaining them in this way. Of course, preemptive tasks which contend with each other may be described using alternation (i.e., $A \triangle (B \,\square\, C)$).

4.7 Optional and Looping behavior

In defining tasks, one often identifies subtasks which are optional. If we were building a user interface for a coffee vending machine, for example, the subtasks associated with adding cream and sugar would be optional. In MDL, if P is a task, then P^{opt} (read "opt-P") is a new task whose single sub-task P is optional. Consider, for example, the task *MakeCoffee* in Figure 13.

task *MakeCoffee*
is
    ChooseBrand >> *Enrich* >> *Brew*
where
    task *Enrich*
    is
        $AddCream^{opt}$ ||| $AddSugar^{opt}$
    endtask
endtask

Figure 13: MDL model of a coffee vending machine.

The *Enrich* sub-task decomposes into the interleaving of two optional subtasks *AddCream* and *AddSugar*. According to this model, after the user has completed choosing a brand of coffee, he could then immediately choose to brew the cup, he could choose to add sugar and then brew, add sugar then add cream and then brew, or add cream and then brew.

Another familiar example is the *SendMail* task of an e-mail browser application. Most mailers allow, but do not require, senders to list carbon-copy recipients–users other than the primary recipient who will also receive a copy of the message. Other examples include the specification of voluntary information in forms-based interfaces. Optionality expresses the opportunity for a user to perform a task, but it does not require the performance.

Two rules govern the semantics of ^{opt}. For a given behavior expression P^{opt}, there must be an opportunity for P to be performed and an opportunity for P to be skipped. These two opportunities are captured in the following inference rules. The first allows **stop** to be inferred from P^{opt} without any action on part of the user.

$$\frac{P^{opt}}{\textbf{stop}}$$

That is, if P is optional, then it can be replaced by **stop** without interaction from the user. In the other case, however, the user may choose to exercise his/her option and perform task P. This is captured in the following rule:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P^{opt} \overset{e}{\rightsquigarrow} P'}$$

Note that after witnessing an event, the optional status of process P is lost. This effectively says that an optional task *senses* the intent of the user by the presence (or absence) of the first interactive event within P.

The optional operator specifies that a task may execute zero or one time. Often, however, designers want to express that a task executes zero or more times. In the ATC model, for example, a controller may want to modify the position of a plane many times before instructing the plane to land. We express zero-or-more behavior in MDL using the looping operator $*$. Given a task P, the expression $P*$ (read "loop-P") specifies that P may be executed zero or more times.

The looping operator implies recursion. Process notations in general handle recursion via process *instantiation*. In many cases, recursive process instantiation is well-behaved and may be simulated by a looping construct. In addition, task analysts often express in plans that subtasks should be repeatable. For these reasons we included an explicit looping operator in MDL.

The rules defining the behavior of the looping operator resemble that of the ^{opt} operator. The first rule allows the loop to exit:

$$\frac{P*}{\textbf{stop}}$$

In the other case, the user must be able to participate in the looping task.

$$\frac{P \overset{e}{\rightsquigarrow} P'}{P* \overset{e}{\rightsquigarrow} P' >> P*}$$

Note how this rule unwinds one iteration of the looping task and follows it with a recursive instantiation of the loop.

4.8 Event Hiding

It is often the case that events will need to be concealed to prevent synchronization in undesired contexts. Consider again the definition of binding *ManageFlight*. This binding expresses the parallel synchronization of two events: *commitToLand* and *newPosition*. These events are used to connect presentation activity to task inputs. It is also intended that these events not be available for

Table 3: The syntax of communications in MDL.

input communication	e?x
output communication	
(literal data)	e!a
(data from storage)	e!x
(data from external linkage)	e!#

containing environments to synchronize with as indicated by the **hide** statement. The designer understood that run-time systems would manage multiple instances of this binding (under the **set** constructor) and he didn't wish for these instances to try and synchronize on *each other's* events. The effects of the **hide** statement are described by the following rules:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{\mathbf{hide}\{e_1 \ldots e_n\}\mathbf{in}P \overset{e}{\rightsquigarrow} \mathbf{hide}\{e_1 \ldots e_n\}\mathbf{in}P'} \quad [\ e \notin e_1 \ldots e_n\ ]$$

That is, there is no effect on events which are not in the hidden set. However, if an event e is in the set of concealed events, it appears as the unobservable event i to the outside world:

$$\frac{P \overset{e}{\rightsquigarrow} P'}{\mathbf{hide}\{e_1 \ldots e_n\}\mathbf{in}P \overset{i}{\rightsquigarrow} \mathbf{hide}\{e_1 \ldots e_n\}\mathbf{in}P'} \quad [\ e \in e_1 \ldots e_n\ ]$$

4.9 Data Parameters and Communications

We add data to the MDL notation using a declarative mechanism similar to that found in LOTOS. LOTOS augments behavior expressions with data specifications and a convenient notation for combining data and process behavior. In full MDL specifications, events are never used directly for synchronization. Rather they are augmented with direction and often data flows to yield communications. We saw examples of communications in Chapter 2. Figure 3 lists the syntax of communications. Given an event e, an input communication referencing some storable data location x is phrased $e?x$. Likewise, an output referencing some value y is phrased $e!y$. The definition of parallel synchronization is now augmented to make synchronization require both inputs and outputs. Let $P == e?x\ ;\ P'$ and $Q == e!3\ ;\ Q'$. Then $P \parallel Q = P' \parallel Q'$ with the additional constraint

that $x = 3$ after the synchronization of event e.

Our definition of communication semantics is more restricted than that provided in LOTOS. LOTOS defines a more elaborate definition of *value offering*. During a LOTOS synchronization, for example, values may be passed in *both* directions, as in:

$$(e?x!3 \fatsemi P') \parallel (e!10?y) \fatsemi Q'$$

which would, after synchronization of e, yield $P' \parallel Q'$ with $x = 10 \wedge y = 3$. We chose not to adopt such powerful rules in MDL.

4.10 Dynamic Task Management

Designers may wish to associate the necessity of performing a task with some asynchronous physical phenomenon. The interruption operator ($\triangle$) is one example, but there are others. Consider the model of air traffic controller tasks described above. The number of planes in the airspace of a given airport varies widely during the course of a day. As planes enter the airspace, they request landing clearance. For each plane that requests clearance, the controller has a task to perform. The inherent dynamics of this task make it difficult to model using the primitives we have supplied thus far. To ease the job of designers, we added a special operator to MDL for declaring just such a situation. The $\mathbf{Set}_P^e$ constructor specifies the interleaved execution of an undetermined number of instantiations of process P. Upon every synchronization of event e, a new P process is instantiated with the value passed in to the communication of e.

There will be an instance of the *ManagePlaneInFlight* task for each plane in the airspace. Pilots entering the airspace of an airport communicate flight information and status to the air traffic controller. The controller must then record this information and assign the flight a position in the airspace. These latter activities coincide with the instantiation of a *ManagePlaneInFlight* task. We now see the larger picture of an air-traffic controller's task. He or she must manage many planes in flight, often introducing new ones by providing the proper information. This aspect of their job is easily expressed in MDL as shown in Figure 14.

We now incorporate the **Set** constructor into MDL by defining it formally in terms of operators we already have.

$$\mathbf{Set}_P^e == e \fatsemi (P \mid\mid\mid \mathbf{Set}_P^e)$$

event $accept : \langle \mathbf{string}, \mathbf{int} \rangle$

task *ManagePlanes*
is
    RecordFlightInfo $\parallel \mathbf{Set}^{accept}_{ManagePlaneInFlight}$
where
    task RecordFlightInfo
    is
        $(newFlight?x \,;\, position?y \,;\, accept!\langle x, y \rangle)*$
    endtask
endtask

Figure 14: MDL description of task for manging planes in an airspace.

Since we define this operation in terms of other MDL operators, $\mathbf{Set}^{e}_{P}$ defines another process which may be composed using any of the other MDL operators.

Chapter 5

The MASTERMIND Toolkit (MMTK)

This chapter describes the MASTERMIND Toolkit (MMTK), a run-time infrastructure and collection of reusable C++ components which can be instantiated and aggregated to implement MDL behavioral specifications. MMTK components are designed to easily compose into a run-time implementation of user tasks and task/presentation bindings. Model-based code generators implement an MDL behavior expression E by aggregating MMTK components (representing the various operators within E) into a class and connecting these components according to the syntactic structure of E. Inherent in this approach is a distribution of control policy over many independent components. These components implement orderings by issuing control imperatives to sub-ordinate components and announcing activity and status to parent components in the tree hierarchy. We designed these components around a model of machine execution in which each component is independent and may *message* other components without waiting for them to return. That is, from the code generator's standpoint, MMTK components are concurrent and need only be aggregated and connected in order to implement an MDL behavior expression. The bulk of the complexity in MMTK centers around designing components and infra-structure to support this model.

There are three different kinds of components in the MMTK architecture: orderings, events, and the run-time control scheduler. The scheduler is the arbiter of control in this architecture. There are many ordering and event components, all of whom are designed to be dispatched by the scheduler. In the abstract, ordering components correspond to entities in an MDL behavior expression. The >> and [> operators, for example, correspond to specific classes of ordering component in MMTK, as do input and output communications _?_ and _!_. Ordering components implement MDL orderings. They cooperate using a model of computation which we formalize in Section 5.1.2. Event components implement MDL events. In MMTK, event components are separate entities from input and output communication components, which synchronize through

the events. Event components are independent of the control component tree hierarchy, but they must inter-operate with the components of this hierarchy. Finally, the scheduler is an autonomous entity which rations control out to the ordering and event components so that they may carry out their functionality.

Additionally, MMTK components support data and event augmentation and user-interface toolkit inter-operation. Since such support extends the control behavior of MMTK, we designed MMTK components to be extensible by parameterizing the classes so that specific data, event, and UI toolkit behavior augmentation can be specified by instantiation. The design is novel because it separates these aspects of functionality into well-encapsulated parts which may vary independently but which compose uniformly and seamlessly with the control aspects of components. This chapter describes the components which implement each of these dimensions and how they compose.

5.1 The Control Model

To understand the design of MMTK, we must understand the nature of control which the toolkit must support. MDL code generators expect to aggregate an MMTK component for each MDL ordering operator. This aggregation should be all that is required to implement the control functionality of the MDL expression The run-time execution model assumed by this strategy is one of concurrent components which communicate with other components by sending messages. Messaging in this model is not the same as invoking a method and waiting for that method to return. Rather, a component messages another component and then goes on about its business. This is necessary because some of the components implement concurrency. This section motivates this control model with an example and then states the model formally.

5.1.1 A Run-time Scenario

We now look at precisely what happens at run-time. Recall, the code generated for the air-traffic control interface (Figure 15). The task *ManagePlaneInFlight* is realized by four components: `inputNewPositionThenLand`, `loopInputNewPosition`, `inputNewPosition`, and `inputCommitToLand`. At run-time these components are independent entities that communicate by messaging each other. An example of this messaging is shown in Figure 16. Arrows denote the issuing of a signal from one

```
class Binding_ManageFlight : public MdlParOrdering,  // The binding operator.
                                 // The events.
                             public MMBlockingEvent_NewPosition,
                             public MMBlockingEvent_CommitToLand {
public:
  Binding_ManageFlight();
  virtual ~Binding_ManageFlight();
     // Parameter access methods.
  string& getFlight();
  void    setFlight( const string& );
  int     getPos();
  void    setPos( const int& );
protected:
  //
  // Implementation of the task ManagePlaneInFlight.
  //
  MdlSeqOrdering                                    inputNewPositionThenLand;
  MdlLoopOrdering                                   loopInputNewPosition;
  InputConcealedEvent<string,
                      Binding_ManageFlight,
                      MMBlockingEvent_NewPosition,
                      Binding_ManageFlight>         inputNewPosition;
  InputConcealedEvent<string,
                      Binding_ManageFlight,
                      MMBlockingEvent_CommitToLand,
                      Binding_ManageFlight>         inputCommitToLand;
  //
  // Implementation of the presentation Airplane.
  //
  MdlLoopOrdering                                  loopoutputNewPositionOrLand;
  MdlAltOrdering                                   outputNewPositionOrLand;
  PresOutputConcealedString<MMBlockingEvent_NewPosition,
                            Binding_ManageFlight>        outputNewPosition;
  PresOutputConcealedString<MMBlockingEvent_CommitToLand,
                            Binding_ManageFlight>        outputCommitToLand;
protected:
  virtual void MMenable();
private:
    // The parameters.
  string  flight;
  int     pos;
};
```

Figure 15: C++ header file implementation of ManageFlight binding.

component to another, and the numbers indicate the order in which these signals are issued. At some point `inputNewPositionThenLand` will be instructed by its parent `Binding_ManageFlight` to accept activity (message 0 in the diagram). At this point, `inputNewPositionThenLand` must implement this instruction in a way consistent with the >> ordering. Since the intent of >> is to sequence two processes, `inputNewPositionThenLand` instructs its left component (`loopInputNewPosition`) to accept activity (message 1 in the diagram). At this point, it sends no messages to the right child. `loopInputNewPosition` implements the $*$ ordering, which means that it must support two behaviors:

1. the iteration executes (as sensed by user activity), or
2. it must be skippable (the loop terminates).

To implement the possibility of the iteration executing, the `loopInputNewPosition` component must instruct its child, `inputNewPosition`, to accept activity (message 2 in the diagram). This child implements a leaf, so it enables input in some presentation widget and then responds with an acknowledgement (message 3 in the diagram). To implement the possibility of being skipped (loop termination), the `loopInputNewPosition` component must instruct its parent, `inputNewPositionThenLand`, that it may be skipped (message 4 in the diagram). Consistent with the >> ordering, `inputNewPositionThenLand` must respond to this instruction by instructing its right child, `inputcommitToLand` to accept input (message 5 in the diagram). Presumably, this component will enable input in some presentation widget and then respond with an acknowledgement to its parent (message 6 in the diagram).

This run-time scenario illustrates two facets of control component behavior. First, components communicate point-to-point in the tree topology as opposed to broadcasting signals. Second, components expect to issue signals back and forth using a protocol.

5.1.2 A Formal Model of Control

As the example demonstrates, the control model assumes independent, concurrent, communicating components. Components maintain internal state which manages coordinating messages, and it issues the coordinating messages down well-defined control paths (like parent, left child, right child,

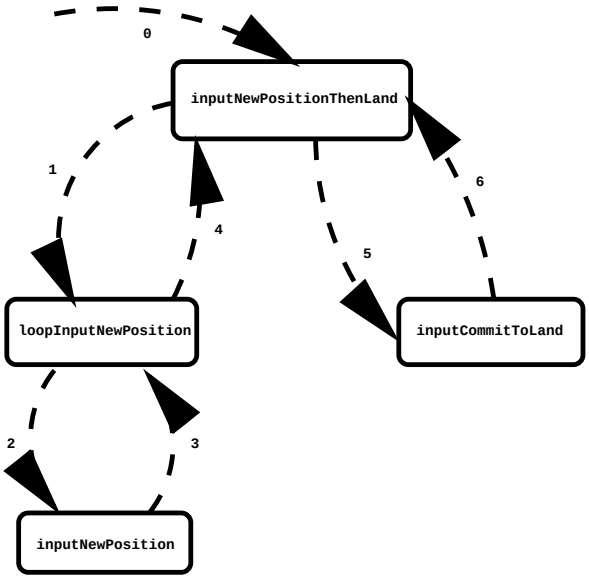

Figure 16: Example run-time communication of MMTK components.

etc). We shall henceforth refer to these messages as *signals* and these control paths as *channels*. Signals communicate a change of status from one component to another, and components often react to signals by issuing signals to other components. Channels, likewise, may be connected to establish point-to-point communication paths between different components. This model of concurrent component signaling through channels is not directly supported in a traditional programming language like C++. We overcame this difficulty by formalizing the control model and then implementing the formalism.

If we ignore event synchronization for the moment, the control issues that come up with MMTK components can be conveniently modeled using Mealy machines. A Mealy machine[52, p. 43] is a finite automaton extended with the ability to produce output. Mealy machines are often used to formalize reactive systems because they are simple, and they support input and output communication. We adapt Mealy machines to our purposes by organizing the input and output alphabets to model point-to-point communication in a tree topology. Whereas formal Mealy machines have input and output symbols from alphabet sets, our adaptation encodes point-to-point channel communication into these symbols. To model point-to-point tree communication via Mealy machines channel/signal communications are encoded as input/output symbols in the alphabet of a Mealy machine. Since there are a fixed number of signals, and since each machine has a fixed number

of channels, we can use the following formal trick. For every signal s which can be input over a channel c, $c?s$ is in the input alphabet of the Mealy machine. Likewise, for every signal s which can be output over a channel c, the symbol $c!s$ is in the output alphabet of the Mealy machine. Channel communications are encoded as input/output symbols. This enables the finite control of a Mealy machine to be implemented in C++ using the case logic of a `switch` statement.

To add event synchronization to the control model, we model events as additional components with data structures that can store pointers to Mealy machines which are *pending* and then, when input communications can be matched to outputs, the event will explicitly activate all components in the pending queue. The inter-connectivity of control components (Mealy machines) and events is illustrated for the binding *ManageFlight* in Figure 17. In this figure, the circles represent Mealy machines, the roundtangles represent event components, the solid lines represent tree connectors, and the dashed lines represent event registry communications.

5.2 Control Components

Control components are those run-time entities which implement MDL operators. In MMTK, these components must behave as modeled by concurrent Mealy machines which communicate in a point-to-point tree topology. There is a different C++ class for particular orderings like $>>$, $[>$, and $e?x$, but all of these have a common structure. This section discusses this generic structure and defers the definition of specific Mealy machine finite controls to Chapter 6. To support this deferment, we defined a generic control component class for each arity of MDL ordering operator. These classes implement the state transition strategy for their respective arity by delegating ordering-specific detail to sub-classes. This allows us to then implement specific ordering machine implementations by subclassing and providing specialized finite control implementations.

Different MMTK components have different tree communication needs based on the arity of the MDL operator they implement. Figure 18 depicts the UML[] model of these communication needs. At the top is a class `Node` from which all ordering classes (`MdlSeqOrdering`, `MdlAltOrdering`, etc) are derived. Subclasses of `Node` specialize it into tree nodes with subordinate components (`NodeWithChildren`) and nodes which can be subordinate components of other components (`NodeWithParent`). As the diagram illustrates, nodes with children contain an attribute called `child`,

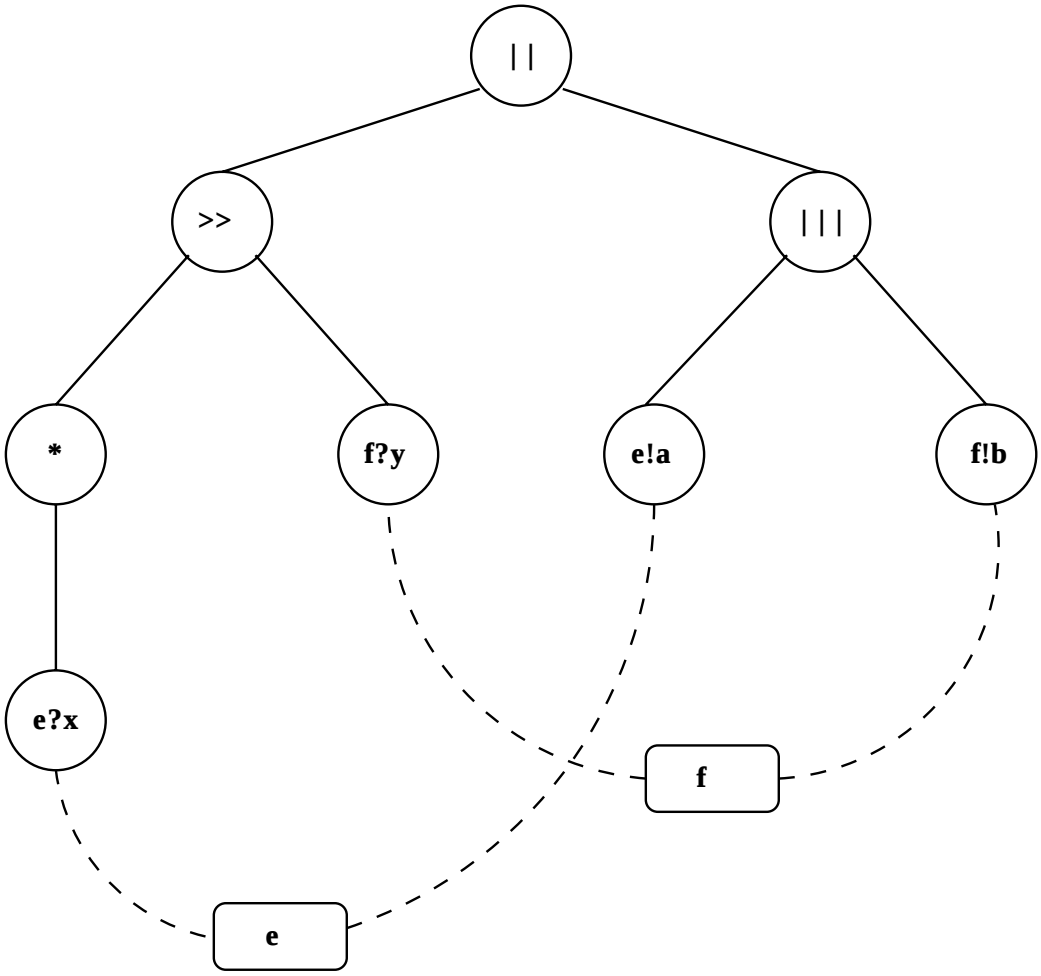

Figure 17: Example MMTK control event component connectivity.

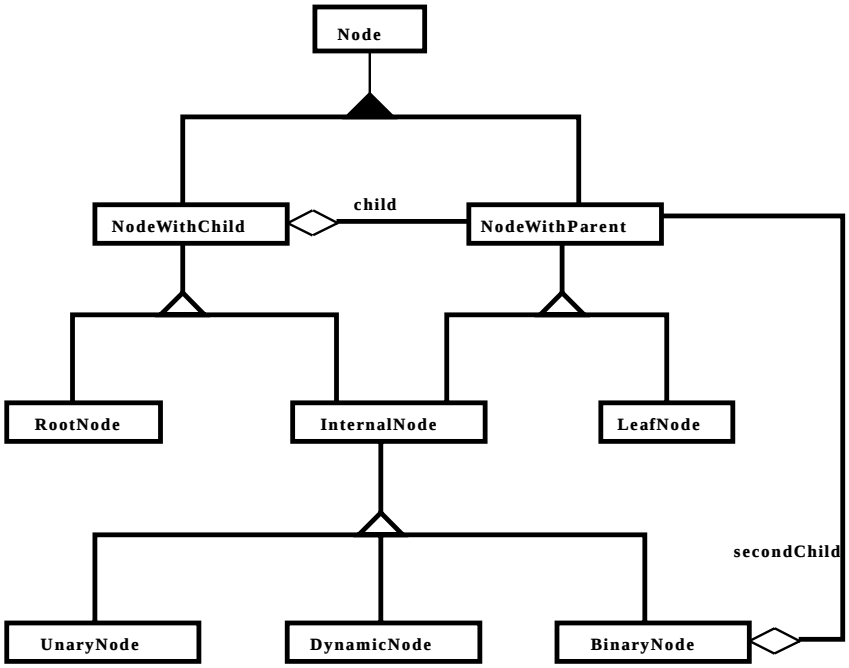

Figure 18: High level design of MMTK control components.

and the type of `child` is `NodeWithParent`. Moving down the inheritance hierarchy, we see classes `RootNode`, `LeafNode`, `UnaryNode`, and `BinaryNode`. These form the immediate base classes from which the classes of the specific ordering components (like `MdlSeqOrdering`) will inherit. Implementations of MDL communications and orderings are grouped by their tree communication needs. The class `LeafNode`, for example, is the base class for atomic control components (those which implement MDL input and output communications), `UnaryNode` is the base class for components which implement unary MDL operators (like ^{opt} and $*$), and `BinaryNode` is the base class for components which implement binary MDL operators (like $>>$, $[>$, $\|$, etc).

By modeling the different tree communication needs of each MDL ordering, we are able to implement the behavior of generic components (like `BinaryNode`), allowing specific strategies (like `MdlSeqOrdering`) to specialize the method with ordering specific behavior. There is, for example, a single abstract method for implementing the transition function of all binary components. Subclasses instantiate this method for a particular ordering by specializing abstract methods. Figure 19 lists the abstract method for state transitioning in binary components. The underlined methods are abstract and are over-ridden by subclasses of `BinaryNode`. The class `MdlSeqOrdering`, for

```
void transitionState(Signal pin_signal, cin_signal, c2in_signal)
{
  int nextState = getNextState(pin_signal, cin_signal, c2in_signal);

  Signal pout_signal = getNextParentOutput(nextState);
  if (pout_signal ≠ nil) issueParentSignal(pout_signal);

  Signal cout_signal = getNextChildOutput(nextState);
  if (cout_signal ≠ nil) issueChildSignal(cout_signal);

  Signal c2out_signal = getNextSecondChildOutput( nextState);
  if (c2out_signal ≠ nil) issueSecondChildSignal(c2out_signal);

  state = nextState;
}
```

Figure 19: The transition algorithm for binary MMTK components.

example, overrides the underlined methods with functionality specific to the behavior of the Mealy machine implementing >>; whereas the class `MdlExclOrdering` overrides the same methods with functionality specific to the behavior of the Mealy machine implementing $\leftrightarrow$. These methods can be generated automatically from a Mealy machine description. The upshot of this approach is that we can design the Mealy machines in a form which supports automated correctness reasoning and then automatically generate classes like `MdlSeqOrdering` from the formal description. Chapter 6 explains this process in detail.

The calls to transitionState are made indirectly by the scheduler. The scheduler dispatches a signal to a particular channel on a particular machine. If, for example, a component instructs its child component to accept activity, then that child will be dispatched an `enable` signal over its `parent` channel. If `c` is a pointer to the child component, and `s` is the signal to dispatch, then the dispatch is accomplished by:

```
c->acceptSignalFromParent(s)
```

This method then invokes the `transitionState` method with the signal `s` passed as the parent input and the empty signal `nil_signal` for the child and second child signals. This may result in new signals being issued (added to the scheduler's queue).

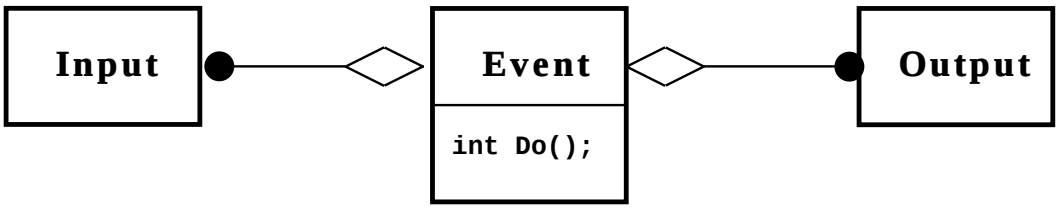

Figure 20: High level design of MMTK event components.

5.3 Event Components

MMTK implements the MDL ordering || for describing concurrency with synchronization. Synchronization is one of the most powerful features of process notations, and consequently, one of the most difficult to implement. To understand the complexity, consider how we might implement a component for ||. Like other components, it instructs sub-ordinates to accept input and to discontinue accepting input, and it announces activity and status information to its parent component. However, communications which must synchronize may not be immediate sub-ordinates of the component that implements the ||. In fact, in the air traffic control example, the || component is many levels above the communications which must synchronize to implement its semantics (as shown in Figure 17). For this reason, we decided to separate the handling of event synchronization from the handling of control components.

In doing this, we created a component called an *Event* which is distinct from input and output communication components yet intimately coupled with them. Event components sense the enabling of associated communication components and, when both inputs and outputs have been sensed, the event component actually *activates* the input and output communication components, passing data as necessary. Whereas thus far, components have been purely reactive, events can actually initiate activity. That is, control components aggregated into tree hierarchies are purely reactive; whereas event components (which are linked to certain leaf components in the tree but which are not a part of the tree itself) initiate activity which percolates through the tree.

5.3.1 Events and Communication

To understand the precise behavior of events and communications, it is instructive to see how they are related. Figure 20 uses the UML notation to describe three classes: *Event*, *Input*, and

Output. Event classes contain collections of *Input* and *Output* objects. The *Input* and *Output* classes represent leaf components in the run-time tree hierarchy, and they correspond to MDL communications. This diagram suggests that input and output communications are related at run time by the event with which they synchronize. Synchronization is triggered when the scheduler calls the `Do` method of the event.

Figure 21 depicts a time series view of the communication requests and eventual synchronization of an event. Let the object **Event** in the diagram correspond to some MDL event e, and let **Input** correspond to an MDL communication $e?x$ and **Output** correspond to an MDL communication $e!y$. Input and output components know the event with which they must synchronize. When an input component is instructed to accept activity, it requests synchronization from the event (Time T_1). The request is made by calling the register method on the event with the object making the request passed as a parameter. If, as shown in the time slice T_2, the `Do` method of the event is invoked by the scheduler, it will return with `false` because at this point, the event has only witnessed synchronization requests from input communication components. Time slice T_3 shows an output component requesting synchronization by registering with the event. At this point, the event has witnessed both input and output communications registering for synchronization. At some later point, the scheduler dispatches the event by calling its `Do` method. What happens as a result of this is shown in time slice T_4. The call to `Do` makes the event look to see if there are both inputs and outputs registered (action 1). Since the event can synchronize, it calls the `Activate()` method of the output component (action 2). This method eventually returns a value which the event component stores in a local variable `k`. When activated, however, the output communication announces activity to its parent in the tree. This is done by pushing a signal onto the scheduling queue and invoking the scheduler (action 3). The scheduler processes the request and modifies the state of the system. When the scheduler returns, the call to the output's `Activate()` method returns, and the event then calls `Activate(k)` on the input (action 4). This likewise invokes the scheduler (action 5), and upon return, the event component returns `true` (action 6).

The behavior of the event and communication components is tightly coupled with that of the scheduler. At any given time, there may be many events which wish to activate synchronizing communication components. The scheduler services them by calling their `Do()` methods in strict sequence. Incidently, when the `Do()` method of an event returns `false` the scheduler puts it on a

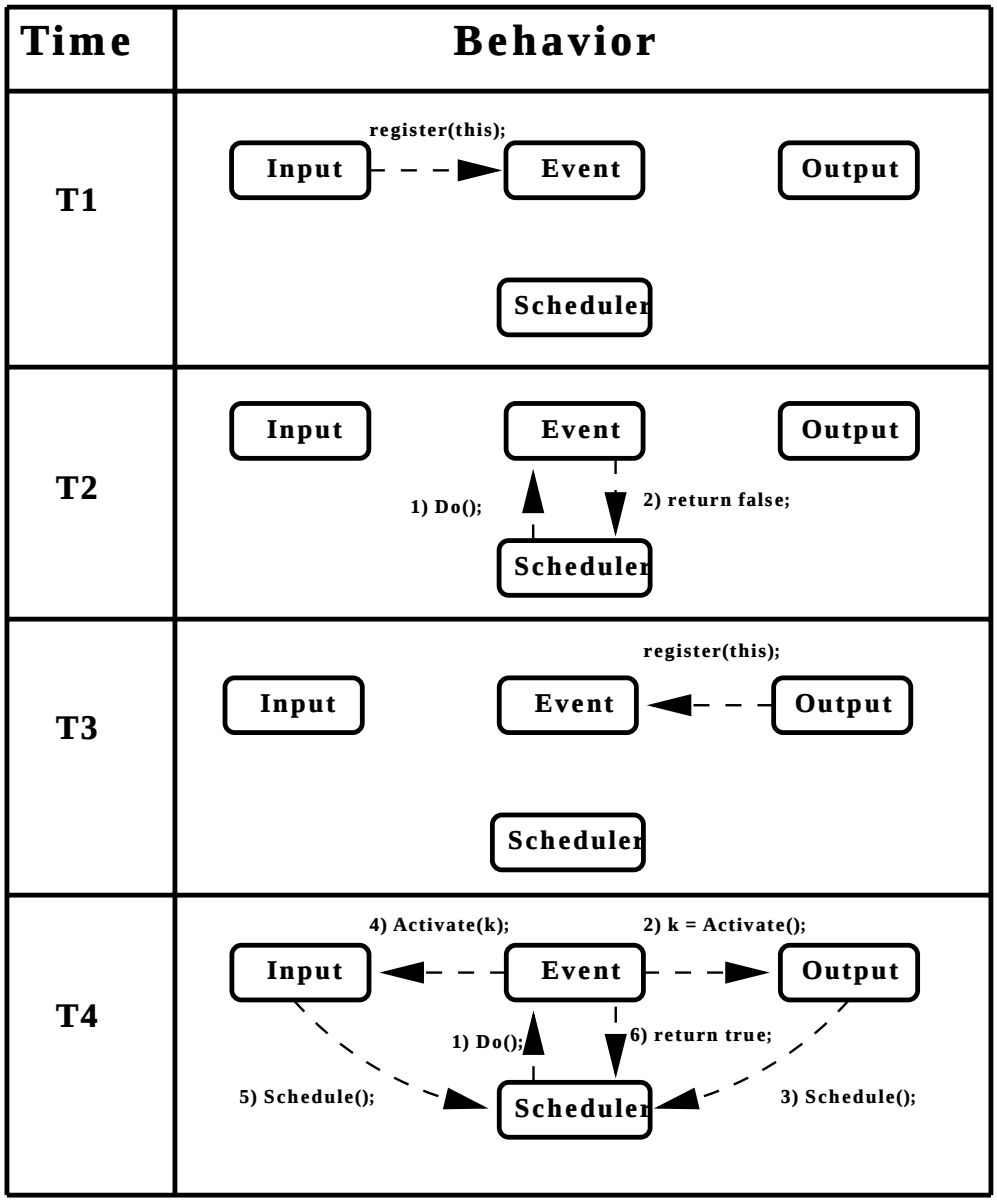

Figure 21: Time series diagram of event synchronization.

special queue called the blocking queue. Such events remain in the blocking queue until a communication registers a synchronization request.

To summarize, synchronizing events are MMTK components distinct from communications but intimately coupled with them. Moreover, activity within the control components is initiated by the synchronization of an event. As a result of these activations, the control components will communicate with each other, the state of the system will change to reflect the activation of the communications, and other events may be queued up for synchronization.

5.3.2 Events with Feedback

Events as described so far synchronize by activating input and output communications as soon as both enable activity. Let $P = e?x \;;\; P'$ and $Q = e!y \;;\; Q'$ for some event e. By the way we have defined events so far, if $P \parallel Q$, then the component associated with $e?x$ will register an input request with the component associated with event e, and the component associated with $e!y$ will register an output request with the same event component. The next time the scheduler is invoked, event e will be synchronized because it has pending input and output communications. However, this strategy does not work for implementing output communications that are attached to presentation functionality.

An output communication of the form $e!\#$ is understood to be connected to a presentation component that actually senses activity from the user. For such components, the enabling of activity is not the same as the committing of such activity. Rather, the enabling of the activity might be interpreted as the *ungreying* of a presentation component like a button in radio-button panel; whereas the committing of such activity only occurs when the user clicks his mouse over said button. To accommodate this distinction, events need to distinguish between the potential for output synchronization and the recognition of synchronization as a result of the user interacting with a presentation. For this we designed a class called `FeedbackEvent` which is shown in Figure 22.

The class `FeedbackEvent` subclasses `Event` which means it inherits the aggregate attributes *Input* and *Output* of input and output communication components respectively. In addition,

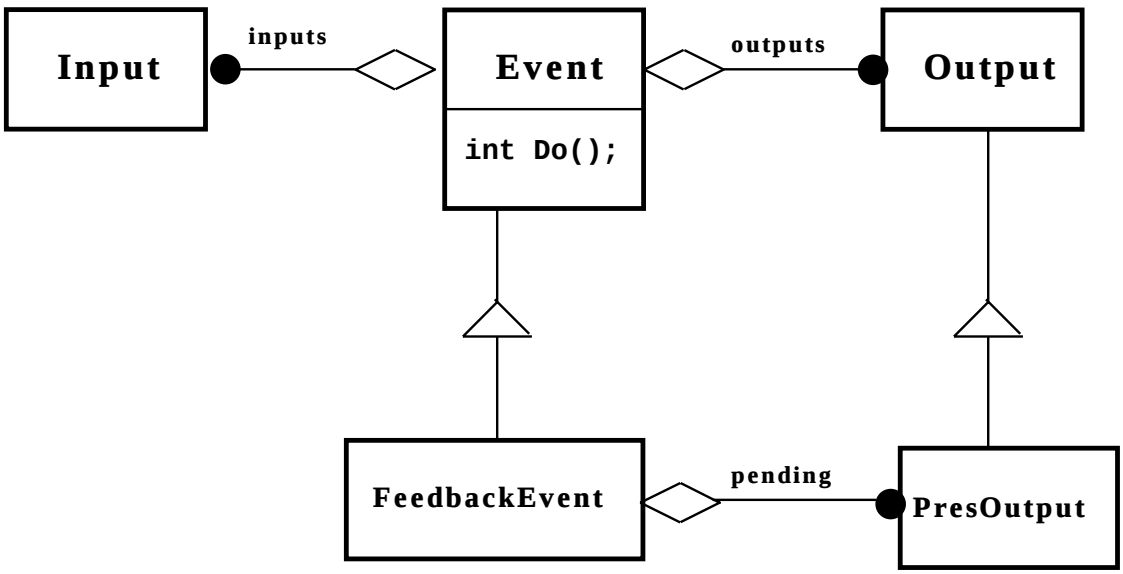

Figure 22: High level design of MMTK feedback-event components.

`FeedbackEvent` has an association called `pending` which aggregates a special kind of Output component. This special output component is called `PresOutput`, and it is understood that these components will be connected to user interface toolkit widgets.

For events with feedback, the synchronization algorithm is slightly more complex than for other events. Figure 23 demonstrates the extra steps involved. Since `FeedbackEvent` subclasses `Event` it behaves in exactly the same fashion with respect to input events. So we can interpose this scenario at time slice number 3 of Figure 21. That is, assume that the input has already registered a request to synchronize with the feedback-event. When a component of class `PresOutput` is instructed to accept activity (first time-slice of Figure 23), it issues a `registerFeedback(this)` message to the feedback-event. Unlike the `register()` message which is a request to synchronize, `registerFeedback()` announces the potential for a request to synchronize. Feedback-events queue up such requests. When the `Do()` method of a feedback-event is called (second time slice), the event checks to see if there are inputs requesting synchronization and outputs with the potential to synchronize. If this is the case, the feedback-event invokes the `Synchronize()` method on the corresponding pres output events (2). The pres output event responds to this by enabling the presentation command to which it is attached (3). Since no synchronization occurred, the event

returns false, which results in it being placed in the blocked queue of the scheduler. But the presentation is enabled and awaiting user input. If this input arrives (third time slice), the presentation command bundles any input data `k` and invokes the pres output component with `SetByCommand(k)`. The pres output event now registers to synchronize with the event using `register(this)`. At this point, things continue with the fourth time slice of Figure 21.

To summarize, `FeedbackEvent` components and `PresOutput` components work in tandem to connect UI toolkit commands into the run-time system of task implementations.

5.3.3 An Example

Returning to the example in Figure 16 of Section 5.1, consider what happens when the component `inputNewPosition` is instructed to accept input. Before acknowledging his parent, this component registers an input request with the event
`MMBlockingEvent_NewPosition`. This event queues up the input request and adds itself to a pending event queue in the scheduler (described below). At around the same time, the `outputNewPosition` component is instructed to accept activity and registers a feedback request to the event. When the scheduler services pending events, it calls the `Do` method of this event which instructs the `outputNewPosition` component to enable its corresponding presentation. This presentation is a drag interactor, and when enabled, it allows the user to drag the plane widget around the display and drop it into a new position. When such a drag occurs, `outputNewPosition` registers with the event, and the next time the `Do` method of the event is called, synchronization occurs.

5.4 The Runtime Scheduler

The concurrency assumed by components is actually simulated by a small control scheduler. At run-time, components signal other components by submitting a signal request onto a scheduling queue and returning. At various points, the scheduler pops requests off this queue and dispatches them to the target components. The scheduler actually maintains three queues:

signals holds inter-component signal requests.

pendingEvents holds events which are ready to synchronize as indicated by the existence of an input and corresponding output communication component, which are both ready to accept

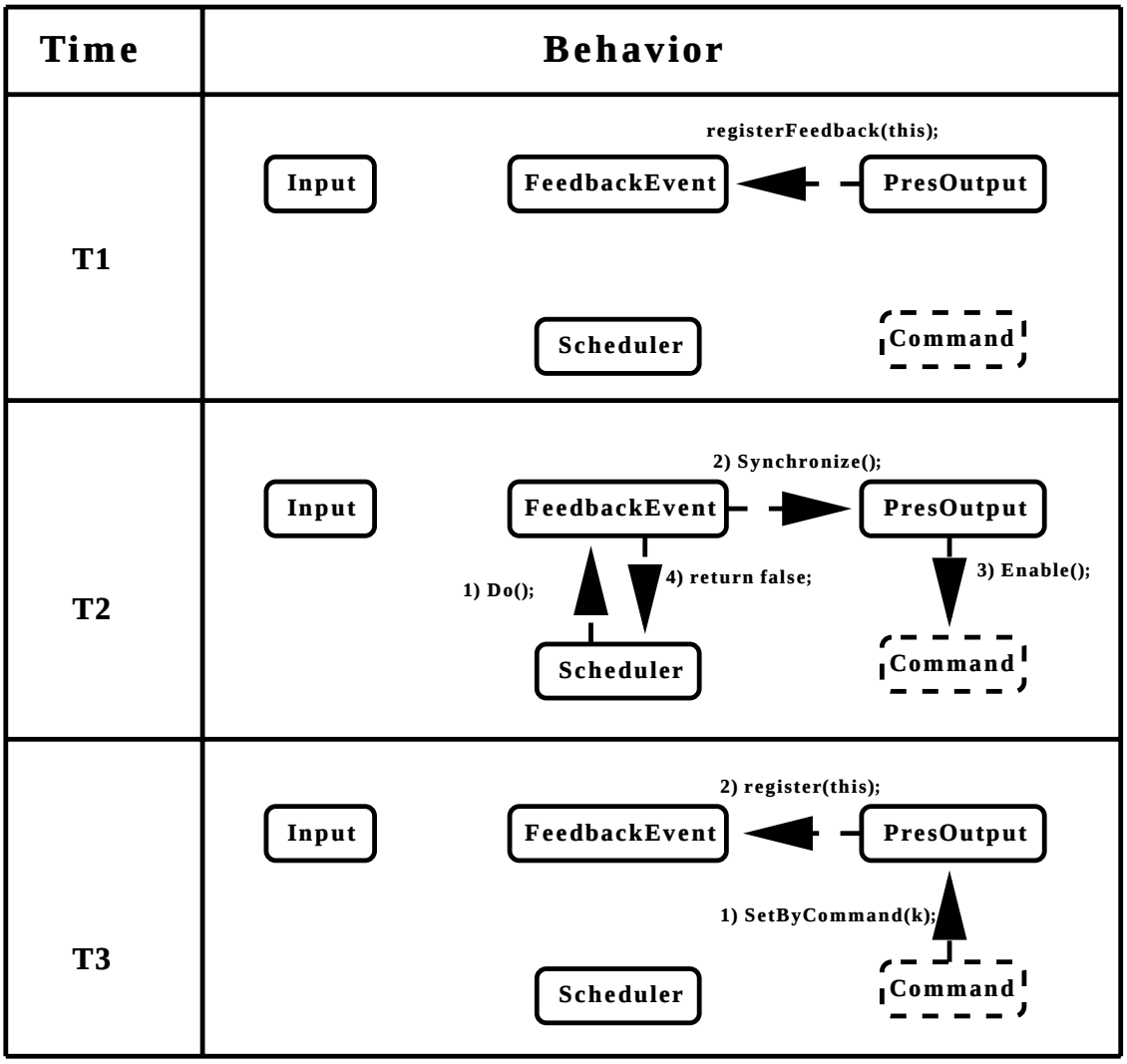

Figure 23: Time series diagram of feedback-event synchronization.

activity.

blockedEvents holds events which are only partially ready to synchronize as indicated by the existence of either an input or an output ready to accept activity, but not both.

When the scheduler is invoked, it services all the requests in the signals queue. When this queue is empty, the scheduler checks the pendingEvents queue, and, for each pending event, it invokes the `Do` method of the event and, based on the result of this method, puts the event back in the queue.

5.5 Data and Event Management

We have alluded to the data forwarding capability of MMTK components in the execution scenarios of event synchronization. Recall that MDL associates data types with events, and communications synchronizing via these events are required to have compatible types. MDL does not have a rigorous data type language (like LOTOS's use of ACT ONE) because we expect task models and presentation models to interoperate with an *application model* which defines, among other things, data types. We assume that any data types in an MDL expression are legal C++ primitive or class types. This assumption is of great practical significance because it allows the parameterization of MMTK components by data type. We have already seen the use of this parameterization in the code generated for the ATC interface. Recall that the input and output communications were instances of instantiated C++ template classes. In general, input and output communications are parameterized by data type and event type. This section discusses this design in more detail.

5.5.1 Data Extension

There are two obstacles to overcome when adding data forwarding to an implementation of a control engine. First is the notion of where data actually lives. In an MDL behavior expression, communications can reference any datum within its block scope. This means that, at run-time, the data location needed by a communication may actually reside in a different component. Second is the complexity of parameterizing a tightly coupled association like that which governs the interoperation of communications and events. It is not enough to parameterize just the event class or just the communication class. Both classes must be parameterized, and, unfortunately, C++ templates

```
template < class DATA,
           class CONTAINER,
           void (CONTAINER::* SET_METHOD)(const DATA&) >
class Input : public Input_impl {
public:
    Input( const char* s ) : index(s) {}
    ...
    void Activate( class DATA& d )
        {
          CONTAINER* location = walkTree(this, index);
          location→ *SET_METHOD(d);
          Input_impl::activate(); // Invokes the Mealy machine.
        }
private:
    const char* index;
};
```

Figure 24: C++ template parameterization of input component.

are not nearly as expressive as we would like them to be to govern this multi-class parameterization. We formulated a solution which solves the data forwarding problem and correctly handles the multi-parameterization of C++ classes.

5.5.2 Data Forwarding

To do data forwarding, we take advantage of the tree structure of our systems. Though a datum may not always reside in the same place, it always resides within some component in the tree hierarchy. In the ATC interface, for example, the `flight` and `pos` data live in the tree component of class `Binding_ManageFlight`. The insight into data forwarding is that the communications which use these data can find them by walking up or down the tree structure. Moreover, since the structure is a tree, we can encode the exact *path* to the data as a sequence of the tree-walking operations: *parent*, *leftChild*, and *rightChild*. In fact, we can encode these path expressions as strings of the symbols p, l, and r. These strings, which we shall henceforth call *indices*, can be passed to the constructor of the communication components so that they will know how to find data. Recall the constructor for class `Binding_ManageFlight` passed strings like this to the constructors of its aggregate communication components.

With this in place, communications may be parameterized. Figure 24 shows how the parameterization of the input component works. The mechanism for parameterizing classes in C++ is the template. The template in Figure 24 has three parameters: DATA, CONTAINER, and SET_METHOD. The parameter DATA represents the type of data this communication component receives. In the ATC example, the data type is either **string** (if the datum of interest was `flight`) or **int** (if the datum of interest was `pos`). The parameter CONTAINER represents the class of the component in which the real data reside. In the ATC example, the CONTAINER is the class `Binding_ManageFlight`. The parameter SET_METHOD is a pointer to a CONTAINER member function which takes a DATA parameter. Instantiations of class `Input` should supply for this parameter the name of the set method for the desired datum.

Returning to the ATC example, suppose we wanted to declare a communication component which inputs the `flight` datum. The following instantiation of the `Input` template would accomplish this:

```
Input < string,
        Binding_ManageFlight,
        &Binding_ManageFlight::setFlight >
```

Now suppose we want to declare an input component for the `pos` datum. The following instantiation would accomplish this:

```
Input < int,
        Binding_ManageFlight,
        &Binding_ManageFlight::setPos >
```

Now assuming the constructors of these instances are passed proper data location indices, this code, when activated by a synchronizing event, sets the given datum with the value passed to `Activate()`.

The design of the input template is novel because of its generality. We can put as many data in a container class as we desire and declare input and output communication components which query and set these data as appropriate. Unfortunately, though the definition of `Input` provided here is legal C++, we could not find a compiler that could handle passing pointers to member functions as template parameters. The compromise we used in building our demos throws out this parameter, and assumes the existence of methods called `Set()` and `Get()` in class CONTAINER.

This is, of course, a poor compromise because it effectively limits the number of data a container class may contain (in fact the limit is 1).

5.5.3 Event Locators Through Parameterization

With a facility for data forwarding, it is realistic to parameterize communications by a data type. Recall, however, that in MDL, there is a strong typing relationship between events and communications which synchronize over those events. The code for class `Input` in Figure 24 only deals with the data forwarding aspect of communications. The other aspect deals with registering synchronization requests with an `Event` component. Now here is the problem. For practical reasons, we do not wish for the code which builds component trees to have to build and link the communication components to event components. So instead, we adopted a mechanism for attaching events to communications using template parameterization.

In some sense events are like global variables. They are names which communications use to express a common locus of synchronization. The key observation here is that if we forget about hidden events for the moment, there will be exactly one instance of an event for each different event name. There is an object-oriented design pattern called **Singleton** [41] which applies when there must be exactly one instance of a class which must be accessible to clients from a well-known access point. Clients of event components are input and output communication components, and the well-known access point is the name of the event. More precisely, the access point is the class name of the event.

Our implementation of events assumes that there will be exactly one instance of unconcealed events and an arbitrary number of instances for concealed events. Communications which need to access a non-concealed event do so by invoking the static method `Instance()` of the event class. That is, if e is an MDL event, there will be an MMTK class called `MMEvent_e`. So, for example, when a communication component wishes to register a synchronization request with an event e, he invokes the following code:

```
MMEvent_e* event = MMEvent_e::Instance();
event->register(this);
```

Since all access is done through the class name, we can abstract this class name into a template parameter as shown in Figure 25. So now, when we instantiate the Input or Output communication

```
template < class DATA,
           class CONTAINER,
           void (CONTAINER::* SET_METHOD)(const DATA&),
           class EVENT >
class Input : public Input_impl {
public:
...
    void MMEnable() {
        EVENT* event = EVENT::Instance();
        event→register(this);
    }
...
};
```

Figure 25: C++ template parameterization of event locator logic.

class with a data parameter, we can also supply the classname of an event with which it should synchronize. This makes the connection between events and communications very declarative and simplifies code generation.

For concealed events there may be multiple independent instances of the event. This is much like local data, and that is how we handle it. Recall from Figure 8 that communications referred to concealed events. Recall also that the class `Binding_ManageFlight` subclassed event classes: `MMBlockingEvent_NewPosition` and `MMBlockingEvent_CommitToLand`. This is an example of aggregation inheritance. It is used to give local copies of these events a place to reside within the tree. This is nice because, augmented with an *event locator index*, communications can walk up the tree to the component which holds the concealed events and then cast that component into the appropriate event class. An example of this is shown in the declaration of parameterized class `InputConcealedEvent` in Figure 26. This requires that the constructor take, in addition to a data index, an event index (called `eventIndex`) so that the instance of the concealed event may be located by walking up the tree. When this class is instantiated, the parameter CONCEALMENT will be the class which aggregates a local copy of the event. In the ATC example, this will be the class `Binding_ManageFlight`.

```
template < class DATA,
           class CONTAINER,
           void (CONTAINER::* SET_METHOD)(const DATA&),
           class EVENT,
           class CONCEALMENT >
class InputConcealedEvent : public Input_impl {
public:
...
    void MMEnable() {
        CONCEALMENT* eLoc = (CONCEALMENT*) walkTree(this,eventIndex);
        EVENT* event = (EVENT*) eLoc;
        event→register(this);
    }
...
};
```

Figure 26: C++ template parameterization of concealed event locator logic.

5.5.4 Summary

To summarize, we augmented the control aspect of MMTK with data and event location management through use of class parameterization. The net effect is the ability to declare input and output communication components which specify:

1. the type of data they manipulate,
2. the location of that data,
3. the name of the event with which they synchronize, and
4. if the event is hidden, the location of that event.

All of this is done using template instantiation parameters and data/event locator indices which are passed to the constructors of these components.

5.6 UI Toolkit Interoperation

Output communications defined in **pres** ... **endpres** processes may contain wild-card data indicated by the symbol #. These symbols represent a linkage obligation that resolved by a presentation

component. Presentation components are generated from a presentation model, and are, therefore, not expressible in MDL. The link obligation is non-trivial.

1. `PresOutput` components must issue enable and disable presentation widgets according to whether or not the components can legally synchronize.

2. The specific presentations need to be able message `PresOutput` components to indicate user activity and data flow.

As we alluded to earlier, MMTK components resolve these linkage obligations using a UI toolkit device called a **command object**. Command objects are run-time presentation objects which encapsulate the action performed as a result of graphical user-interface interaction. They represent the enabledness of a presentation interactor and, therefore, are convenient vehicles for explicitly enabling and disabling interaction. The also have a special method called `DO()` which is invoked whenever an interaction completes. We use command objects in Amulet[72] as a means for connecting MMTK components to bound presentations. This section describes the details of this connection.

5.6.1 The Amulet Input Model

The Amulet toolkit provides a solution to this problem in the way it handles interaction. Whereas, in toolkits like Xt, interaction functionality is tied directly to graphical objects, in Amulet, interaction functionality is encapsulated in one of a small set of behavioral entities called interactors. Interactors encapsulate protocols of gesture like Selection, Text Input, MoveGrow, etc. What is remarkable about Amulet is that these interactors compose very easily with graphical entities. This is a subtle point, but it basically means that any graphical entity can be made interactive by adding to its parts-list an interactor object. To support such a general technique of composition, the Amulet designers needed a convenient way to express the actions which are performed as a result of interaction.

Amulet interactors aggregate a special kind of object called a **command object**. When an interaction completes, the interactor invokes the `DO` method of its command object. Recall the direct manipulation that we used in the ATC interface. This was accomplished by attaching an `Am_Move_Grow_Interactor` object to the airplane widgets. This interactor manages the moving of the graphic while a user is dragging it to a new location. When the user drops the graphic (by

```
Am_Define_Method( Am_Object_Method,
                  void,
                  callback_with_string,
                  (Am_Object self))
{
  Am_Value leaf = self.Get(MM_TASK_LEAF);
  ...
  PresOutput¡string¿* c = (PresOutput¡string¿*)((Am_Ptr) leaf);
  Am_String astr = (Am_String) self.Get(Am_VALUE);
  ...
  string str( (const char*) ((Am_String) astr ));
  c→SetByCommand(str);
}
```

Figure 27: Method that activates an MMTK PresOutput component. This method is invoked by an Amulet interactor when in response to user input.

releasing the mouse), the interactor invokes the `DO` method of its command object. The benefit within Amulet of command objects comes from the fact that new types of behavior to draggable objects without having to change a monolithic call-back function. We found them to be equally useful for connecting MMTK components to presentation components.

5.6.2 Command Objects Satisfy Linkage Constraints

Amulet command objects contain a number of data slots within which we store all of the necessary MMTK/Amulet linkage information. We built a prototype object called `MM_Callback_Command` which is an instance of the more generic Amulet `Am_Command` object. These objects have three data slots:

Am_ACTIVE which if set to `true` enables interaction,

Am_VALUE which holds the value of the interaction (the name of the button that was selected, mouse position, etc), and

MM_TASK_LEAF which contains a pointer to the `PreOutput` component containing this command object for use in calling back upon interaction.

To understand how the data in these slots, work, consider the `DO` method for commands which can communicate strings in Figure 27. This method gets called when an interactor observes the end of an interaction. The first thing this method does is to retrieve the `PresOutput` which is to

be notified of interaction (stored in the `MM_Task_Leaf` slot. Then the actual string which needs to be passed is retrieved (stored in the `Am_VALUE` slot). Finally, the `PresOutput` is messaged with the string.

5.6.3 Connecting MMTK and Amulet Components

The method of connecting MMTK and Amulet components then, is to connect an `MM_Callback_Command` object to a given `PresOutput` component and then to install this object into the `Am_COMMAND` slot of a particular interactor. The connection of object ot MMTK component can be done in the constructor of the MMTK component. Making the object a part of an interactor, however, must be done explicitly when the graphical widget is created.

Command objects are useful as a means of *multiple aggregation inheritance*. Buttons, and in fact any interactive object in the Amulet toolkit, use Command objects not only to implement interactions, but also to represent so-called interface state like the status of being greyed-out. When a button is created in Amulet, it actually instantiates a prototype Command object and constrains its visual attributes (like color) over attributes of the Command object (like Active). This indirection of state and behavior through command objects allows us to treat commands as a linkage mechanism between MMTK components and presentation components.

5.7 Discussion

Run-time systems embody more implementation detail than do model-based specifications. This embodiment of detail results from design decisions which codify assumptions and give structure to abstract entities. In designing MMTK, we made two fundamental design decisions concerning run-time control policy, and user interface toolkit architecture.

5.7.1 Run-time Control Policy

The run-time control policy of a system governs the delegation of control among independent components of the system. On sequential machines, concurrency must be simulated. In a classic paper[50], Hoare identified three distinct classes of concurrent processes:

1. Competing processes, which require the exclusive use of some global resource during certain phases of their execution,

2. Cooperating processes, which independently make contributions to some desired result, and

3. Communicating processes, which exchanges information at intermediate stages in their progress.

Process notations like CSP and LOTOS support the specification of processes having all three characteristics. It is not clear, however, that a given domain requires all three characteristics. We chose not to implement support for the competing process aspect of concurrency because it does not seem to be needed in interactive system architecture and because its presence can lead to poorly designed interfaces[32]. Without this form of process interaction, concurrent run-time control policies can be simulated quite efficiently on a sequential machine.

In defining the ESTEREL[13] notation, Berry and Gonthier proposed a model of concurrency which is appropriate for task-based concrete architectures. They observed that notations which utilize concurrency forces systems to be either concurrent or deterministic, and that the reason for this was that concurrency was based upon an *asynchronous* implementation model. In an asynchronous model, designers can express non-determinism by process *competition* for limited resources. Asynchronous notations, therefore, often contain two different forms of the choice operator: external (which is the form we supply in MDL) and internal (which is a choice made by the system and not the user). Though user interfaces should not be overly deterministic, the system should also not non-deterministically make choices with regard to user task performance[32]. Berry and Gonthier propose overcoming this dilemma by applying a *synchronous* model of concurrency.

The essence of synchronous concurrency is the so-called synchrony hypothesis[13], which states that interactions take no time with respect to their environments. The synchrony hypothesis reflects the way current GUI toolkits are implemented in the following sense: when an interaction event is dispatched, it is not preempted by other events. In the X-window system, for example, mouse and keyboard events are *queued-up* and call-back functions are dispatched in sequence so that one function completes before the next event is dispatched. Even though more modern toolkits like Amulet[72] and subArctic[33], support multi-threading, their behavior is consistent with the synchrony hypothesis because methods dispatched by events are not *interrupted* by other events. Programmers familiar with toolkits understand this and deal with it by keeping callstacks shallow

in event handling code.

In addition to being appropriate for the UI domain, the synchrony hypothesis greatly simplifies the implementation of process notations by precluding the need to resolve race conditions resulting from multiple overlapping device interactions. In fact, synchronous languages are often compilable into deterministic finite automata. The Squeak[24] language of Cardelli and Pike assumes the synchrony hypothesis and is able to create single-threaded implementations from a concurrent notation. The ESTEREL language also shares this property.

Design Decision 1. *The MMTK architecture adopts a synchronous model of concurrency.*

5.7.2 Presentation Communication

Another influence on MMTK design is the nature of user interface toolkit architecture. There are two important concerns: input and output. On the input side, progress in an interactive system is made in accordance with user actions. These actions are *sensed* by interaction with graphical presentation objects. Without exception, user interface toolkits combine a policy of input with support of graphical presentation objects. On the output side, the state of a run-time task must be made apparent to the user. Usually this takes the form of special presentation markings to identify actions which are legal or illegal at various points in the performance of a task. Our decisions in this regard leverage abstractions provided by the Garnet[34] and Amulet[72] toolkits.

The Amulet toolkit[72] provides an **interactor model** which purports to be a true solution to the independence of input and output. In Amulet, a special type of object called an interactor serves as the controller. Interactors are freely composable with graphical objects (views) and, using an abstraction called **command objects**, with models. Interactors abstract complete gesture interactions into a single software entity. These are quite useful in bottom-up design because they relieve programmers from a large implementation task and are quite reusable over a wide range of applications. Command objects encapsulate everything about an action, including the status of the action being enabled, the mechanism for performing an action (the DO method), and support for UNDOing the action.

Design Decision 2. *Presentation interaction entities will be those provided by the Amulet[72] user interface toolkit.*

Chapter 6

Control Correctness of MMTK

MDL behavior expressions describe control in a reactive system. MDL programs compile into a communicating hierarchy of components in which each component implements a single operator or communication. In Chapter 5, we pointed out that the MMTK components have many states and that, consequently, they are difficult to design and test. We overcame this problem by formally modeling the control components, analyzing their behavior in composition, and then generating the MMTK component implementation directly from the models. The formal model we chose to use is called a Mealy machine. This chapter describes our use of Mealy machines to implement control components and validate their correctness.

6.1 Mealy Machines

Inherent in our implementation of MDL behavior expressions is a policy for distributing control over the many independent components. Creating a component for an MDL operator means expressing declarative ordering invariants as a protocol of operational inter-component commands. Examples of these commands include sending a message to a subordinate component to accept activity, sending a message a subordinate component to no longer accept activity, and sending a message to a parent component to announce activity. To take an example, consider a component A implementing the $>>$ operator, whose left subcomponent B implements an input communication and a whose right subcomponent C implements the | operator. When A is instructed (by its parent component) to accept user activity, it must interpret this command in a way consistent with its ordering. Since the intent of $>>$ is to order two processes, A instructs B to accept activity but does not instruct C to do anything yet. Now if user activity causes the input communication associated with B to fire, then B must announce activity to A. At this point, A will propagate this announcement

to its parent. When B announces completion, A will instruct C to accept activity. Since C implements an ordering, it will forward this command to subordinates in a way that is consistent with its ordering semantics. Specifically, it will instruct both of its children to accept activity and then, when one announces activity, C will instruct the other to forbid activity. Components maintain internal state which manages these coordinating messages. The internal state changes in response to a *signal* which communicates a change of status from one component to another. Finally, components often react to signals by issuing their own signals to other components. A formal machine model which can express this kind of behavior is a *Mealy machine*[52].

Mealy machines are finite automata with output. Mealy machines react to an input symbol by changing state and issuing an output symbol all in one step. MMTK components can be modeled using Mealy machines because the components maintain a finite state, and they communicate with other components using a fixed alphabet of signals. Mealy machines are more convenient for reasoning about control issues than are C++ objects. This section describes the Mealy machines which implement the various MDL orderings. In doing this, we introduce two notations. The MTREE notation names the Mealy machine that implements each MDL operator and is used for expressing the connectivity of these machines in a run-time configuration. The MM notation is used for expressing the finite control of a given class of Mealy machine.

6.1.1 The MTREE Notation

MTREE is a notation for expressing instances of Mealy machines and their inter-connectivity. MTREE names a Mealy machine that implements the functionality of an MDL control operator (as shown in Table 4). Note that there is no Mealy machine implementing the **stop** process or the **hide** operator, input and output communications are both implemented by the same Mealy machine (leaf), and $\|$ and $|||$ are both implemented by the same Mealy machine (par). Instances of these Mealy machines are connected in MTREE using the binary operator $\lhd \_ \rhd$ and the unary operator $\_ \rhd$. The $\_$ in these operators will contain one of the named Mealy machines. So, for example, $\lhd$ seq $\rhd$ is a binary connector that implements the seq machine functionality and loop $\rhd$ is a unary connector that implements the loop machine functionality. A simple syntactic transformation maps an arbitrary MDL expression into an MTREE expression.

For example, the MDL expression: $(e?x \fatsemi \mathbf{stop}) >> ((f?y \fatsemi \mathbf{stop}) \mid (g?z \fatsemi \mathbf{stop}))$ corresponds

Table 4: Mealy machine implementations of MDL operators.

MDL Operator	Mealy Machine	Mdl Operator	Mealy Machine
$\Box$	alt	$[>$	dis
$\leftrightarrow$	excl	$\triangle$	int
$e?x,\ e!x$	leaf	$*$	loop
$^{\text{opt}}$	opt	$\|\|,\ \|\|\|$	par
$>>$	seq		

to the following MTREE expression:

$$\mathsf{ceiling} \rhd (\mathsf{leaf} \lhd \mathsf{seq} \rhd (\mathsf{leaf} \lhd \mathsf{alt} \rhd \mathsf{leaf}))$$

The reader should observe two things about this mapping. First, the **stop** processes map to nothing, and parentheses are retained under the mapping. Second the MTREE representation applies the unary operator ceiling $\rhd$ to the image of the MDL expression. Recall that Mealy machines must communicate signaling information to subordinates and parents in the tree hierarchy. This means that all the Mealy machines that implement MDL control operators will expect to be connected to a parent in the tree hierarchy. The special ceiling is used to **top** an Mealy machine tree so that there are no unresolved communications.

In order to reason formally about Mealy machine composition, the MTREE language must be formally defined. Let $\hat{\Sigma}$ define the set of Mealy machines from Table 4. Let Σ denote the set $\hat{\Sigma} \cup \{\mathsf{ceiling}\}$, Let $\hat{\Sigma_0} \subset \hat{\Sigma}$ be the set of null-ary machines (machines with no children), $\hat{\Sigma_1} \subset \hat{\Sigma}$ is the set of unary machines (except for ceiling), and $\hat{\Sigma_2} \subset \hat{\Sigma}$ is the set of binary machines. The set of all trees of Mealy machines is then the *language* generated from this set $\mathcal{L}(\hat{\Sigma})$. $\mathcal{L}(\hat{\Sigma})$ is defined inductively as follows:

1. $\forall\, m \in \hat{\Sigma_0} \bullet m \in \mathcal{L}(\hat{\Sigma})$.
2. $\forall\, m \in \hat{\Sigma_1} \bullet \forall\, \sigma \in \mathcal{L}(\hat{\Sigma}) \bullet m \rhd \sigma \in \mathcal{L}(\hat{\Sigma})$, and
3. $\forall\, m \in \hat{\Sigma_2} \bullet \forall\, \sigma_1, \sigma_2 \in \mathcal{L}(\hat{\Sigma}) \bullet \sigma_1 \lhd m \rhd \sigma_2 \in \mathcal{L}(\hat{\Sigma})$.

Now we define $\mathcal{L}(\Sigma)$ as the set of all strings in $\mathcal{L}(\hat{\Sigma})$ which are "topped" by a ceiling component. More formally:

$$\mathcal{L}(\Sigma) == \{\mathsf{ceiling} \rhd \sigma \bullet \sigma \in \mathcal{L}(\hat{\Sigma})\}$$

We call any string in $\mathcal{L}(\Sigma)$ a **configuration** of Mealy machines and often use the symbol σ to represent a particular configuration. Within a configuration σ, a given class of Mealy machine like, say leaf, may occur in many places. We call these instances *occurrences* and use middle-of-the-alphabet letters like m and n to represent them in formal arguments. The following configuration, for example, contains three occurrences of the leaf machine:

$$\mathsf{leaf} \lhd \mathsf{seq} \rhd (\mathsf{leaf} \lhd \mathsf{alt} \rhd \mathsf{leaf})$$

Occurrences may also be used to select adjacent occurrences through channels. If m is an occurrence, then $m.p$ is the occurrence of m's parent (if it exists), $m.l$ is the occurrence of m's left subordinate (if it exists), and $m.r$ is the occurrence of m's right subordinate (if it exists). Consider, for example, the configuration $m \lhd n \rhd o$. Occurrence m has a parent channel connected to the left child channel of occurrence n. We can express this with the following equalities: $m.p = n$ and $n.l = m$. We say that two occurrences m and n are *incident* if either $m \lhd n$ or $m \rhd n$ is a subconfiguration of σ.

The syntactic similarity of MDL and MTREE allows us to reason about the mapping from MDL expressions into MMTK components. Recall the MDL definition of the task *ManagePlaneInFlight*. The task had the following behavior.

$$(newposition?pos \fatsemi \mathbf{stop})* >> (commitToLand? \fatsemi \mathbf{stop})$$

Using MTREE, we can model the MMTK implementation of this expression:

$$(\mathsf{leaf} \lhd \mathsf{loop}) \lhd \mathsf{seq} \rhd \mathsf{leaf}$$

For purposes of reasoning about behavior, this is much more tractable than the corresponding C++.

6.1.2 Detailed Design of Mealy Machines

With MTREE in place, we now must describe the inner workings of the machines. Mealy machines communicate by issuing signals over parent and child channels. In the current implementation, there are twelve such signals[92]. These signals communicate the control commands of one machine to another. When a machine m is issued one of these signals, it changes state, and it may, in turn, issue a signal to another machine.

The global state of an interactive system can be thought of as a sequence of *equilibrium* states.

When in these states, the system awaits a *perturbing* signal from the user or the application functionality. When in any equilibrium state, say s, a user event may come along which causes a flurry of activity among the components. During this flurry the system will not be in an equilibrium state, but rather, will be in a series of transient states. Eventually the system settles into another equilibrium state (usually different from s) and awaits the next perturbing signal. The synchrony hypothesis, which we adopted in Section 5.7.1, states that no external events occur during these transient flurries of *internal* signals. The distinction between equilibrium and transient states percolates down to the level of Mealy machines. In fact, the states of each Mealy machine may be strictly partitioned into two sets: *Equilibrium* and *Transient*.

Of the *Equilibrium* states, there are six possibilities:

notready the machine has either completed or has not been enabled,

active the machine (or one of its subordinates) has observed user activity.

committed the machine has not witnessed activity but is able to witness activity.

completable the machine has witnessed activity, and may complete on its own, or it may be completed implicitly by the activity of other machines.

interrupted the machine has been interrupted.

skippable the machine has not witnessed activity but may be completed implicitly by the activity of other machines.

We use these general categories for initially designing the detailed finite control of our Mealy machines. Actually describing this detailed control requires another notation.

6.1.3 MM: A Notation for Mealy Machine Finite Control

The MM notation graphically describes the transition relationship of a Mealy Machine using a two-dimensional ASCII representation of state machine graphs. The definition of a particular machine often consists of many such descriptions joined together by unifying the names of states. An example of the notation appears in Figure 28. The reader may immediately observe three things about this notation:

Table 5: Mealy machine (MM) syntax.

```
State       ==  Equilibrium | Transient
Equilibrium ==  Identifier
Transient   ==  '@'+ { Identifier }
Identifier  ==  ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9''_']*
Transaction ==  Action { Reaction }
Action      ==  { ~ } Channel '?' Signal
Reaction    ==  '[' Channel '!' Signal ']'
Channel     ==  'p'                          /* parent */
            |   'l'                          /* left subordinate */
            |   'r'                          /* right subordinate */
Signal      ==  ['a'-'z']+
```

1. machine transitions are represented as `s -- c1?e1[c2!e2] --> t`, indicating that a transition takes place from state s to state t upon receipt of signal e_1 on channel c_1, and results in the issuing of signal e_2 down channel c_2,

2. the transitions can be laid out in two dimensions, thus simplifying input and understanding of the state topology, and

3. states are given either mnemonic names like `notready` or they are unnamed and represented with the `@` placeholder. Unnamed states are taken to be transient states; whereas named states are equilibrium states.

The diagram in Figure 28 demonstrates how the `seq` machine behaves in reaction to an `activate` signal from its left child in the tree (`l?activate`). The signal coerces `seq` into a transient state with the side effect of issuing an `activate` signal to its parent.

Figure 5 describes the syntax of MM. Transitions are described as directed edges from one state to another with an intervening *transaction*. Transactions describe the action that stimulates the transition and any resulting reaction. States and transactions are connected using transition edges which appear as ASCII lines in the MM files. A line may be vertical, horizontal, slanted left or right, or bent with the junction symbol `+`. Transitions are always directed with a unique source and target state and a unique transaction. To specify the direction of a transition, one end must be affixed with an arrow point (either `>`, `<`, `^`, or `%`) depending on whether the edge is flowing to the right, to the left, up, or down respectively. Slanted edges cannot be given a direction, and so they must be bent into a non-slanted edge which can then be affixed with an arrow point. In the

example of Figure 28, `@ --- l?ack[p!ack] --> lcommit`, denotes a transition whose source is the transient state `@`, whose target is the state `lcommit`, and whose transaction is `l?ack[p!ack]`.

6.1.4 Generating MMTK Components from MM

The ultimate output of a Mealy machine finite control specification is a reusable component in the MASTERMIND toolkit (Section 5). After having designed and tested the interoperation of the Mealy machines, we invoke a tool called the Mealy Machine Compiler (mmc) to create the C++ implementations. Mmc creates two files–a C++ header (.h) file and a C++ implementation (.cc) file–for each class of Mealy machine.

Consider, for example, the MM definition of the `seq` machine (which constitutes a number of .mm files). The command:

```
$ mmc -c -Mseq seq*.mm
```

creates two files: `MdlSeqOrdering.h` and `MdlSeqOrdering.cc`. The file `MdlSeqOrdering.h` defines the class `MdlSeqOrdering` and declares it to be a subclass of class `BinaryNode`. `MdlSeqOrdering` inherits a method called `getNextState` from `BinaryNode`. This method implements the Mealy machine transition relationship. It is defined in the file `MdlSeqOrdering.cc`.

Since there are ten MDL orderings, there will be ten header and ten implementation files that make up the finite control constituent of the MASTERMIND toolkit.

6.1.5 Carrying On

Having identified the set $\hat{\Sigma}$ of machines and having created a notation for expressing the behavior of the individual machines, we then embarked on the design of each machine. Because these machines have so many states, it was difficult to arrive at correct designs. The most difficult problem was making sure that the machines would cooperate with other machines. We sensed this difficulty early and investe in a formal and mechanical correctness validation before committing to a detailed design. In the next section, we discuss the theoretical underpinnings of this validation infrastructure.

6.2 Mealy Machine Inter-Operation

From the discussion in the previous section, it should be clear that Mealy machines do not implement orderings in isolation. Rather, they issue control directives to parent and subordinate machines in the tree hierarchy. Any two machines connected in the tree hierarchy must understand and implement a *protocol* of communication.

The protocol nature of communication is unavoidable because the signals which MMTK components use to instruct subordinates affect their internal (unobservable) states. A component a, for example, never expects to field a **resume** signal unless it has received a prior **interrupt** signal. If another component, say b, expects a to field a **resume** signal before b issues an **interrupt** signal to a, then b and a could have trouble communicating. In designing the Mealy machines, we discovered that these protocols were generally simple to discuss informally but were very difficult to precisely articulate. The problem is that the protocols are a function of the states of connected Mealy machines, and since our Mealy machines have hundreds of states, the sheer size of the protocols becomes an obstacle to comprehension.

In order to establish correctness of protocols, we took a different approach. Rather than try to formally specify all inter-machine protocols, we instead formulated two properties–**receptiveness** and **freedom from divergence**–which all protocols must respect. These properties are necessary in order for a distributed collection of Mealy machines to effectively communicate control, and they are easier to specify than the myriad of specialized protocols to which they apply. In addition, specific configurations of Mealy machines may be subjected to a tool called a model checker which will exhaustively validate these properties over the configuration. We then prove a theorem that identifies a finite set of test cases which, if all tests succeed, guarantees that these properties hold in arbitrary configurations of Mealy machines. We now describe and formalize the correctness properties. The testing theorem is the subject of Chapter 7.

6.2.1 Property 1: Receptiveness

The first property of component interactions is that one component never receives a signal which it is not prepared to handle. This property is often called **receptiveness** meaning that a machine is always able to receive a signal. A receptiveness failure is always the result of a design flaw. If a

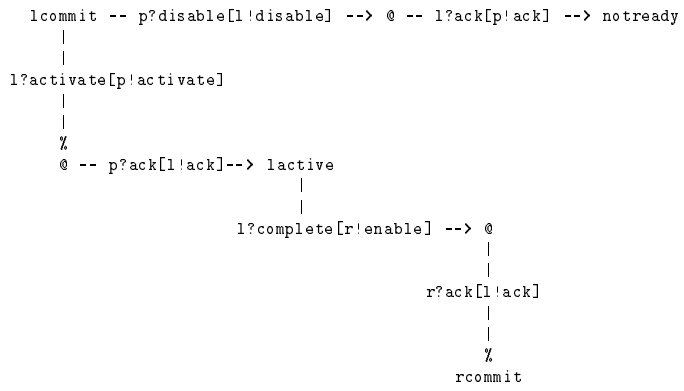

Figure 28: MM description of seq machine transitions.

machine receives a signal it is not prepared to handle, then it cannot react according to the wishes of the sender of the signal. Consider the snippet of machine logic in the partial definition of the seq component of Figure 28. When in the state `lcommit` this machine is prepared to witness activity in the left child, and it is prepared to be disabled by its parent. The left child announces activity by issuing an `activate` signal to its parent (the seq machine). If somehow, the designer forgot to describe that, from within state `lcommit`, the machine could receive an `activate` signal from the left child, then the machine would not be receptive to this event. To see how the effects of this can be felt by the user, consider that the machine is in the `lactive` state, but the designer forgot to add the transition out of `lactive` in reaction to a `complete` signal by the left child. If this happened, then the right child would never be enabled, and the system would appear to deadlock.

We would like to encode a property which says "machines are always receptive". We do this by introducing into each Mealy machine a special state called `error`. Now for each non-`error` state v, consider the signals $e_1, e_2, \ldots e_k$ which can initiate a transition out of v. To the complement of this set $\overline{E} = E - \{e_1, e_2, \ldots, e_k\}$, add a transition $(v, \overline{e}, \texttt{error})$ for all $\overline{e} \in \overline{E}$. If we extend machines in this way, we can state the receptiveness property by saying that a machine never enters the `error` state.

6.2.2 Property 2: Freedom from Divergence

The next property is called freedom from divergence, and it states that when a component enters a transient state, it necessarily eventually enters an equilibrium state. There are two (equally

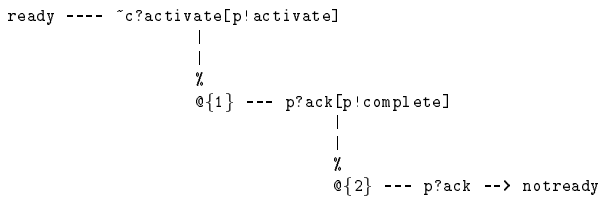

Figure 29: MM description of leaf machine transitions.

undesirable) scenarios which may cause this property not to hold. In the first scenario, a component gets *stuck* in a transient state awaiting a signal that never arrives. This might happen because machines must acknowledge the receipt of control signals. Consider the snippet of functionality from the leaf machine if Figure 29. In the `ready` state, upon recognition of user input, this machine issues an **activate** signal to its parent and goes into the transient state `@{1}`. The machine expects an acknowlege signal (`ack`) from its parent in order to issue the `complete` signal. If the parent machine fails to issue the `ack`, then this machine is stuck. In the second scenario, a configuration of machines enters a signaling cycle which effectively prevents the system from reaching an equilibrium state. Both of these scenarios are undesirable. The freedom from divergence property asserts that they never occur.

To state this property, we take advantage of the disjoint partitioning of machine states into two sets: *Equilibrium* and *Transient*. The property is true if, whenever a machine enters a state in the *Transient* set, it necessarily eventually enters a state in the *Equilibrium* set. This predicate takes care of both the scenario in which machines get stuck waiting for an event and that in which a configuration of machines enters an infinite signaling cycle.

6.2.3 Formalizing The Properties

Protocols are expressions of temporal behavior. We formalize the correctness properties as invariants over protocols. Since we want to state properties like "after receiving signal **interrupt** from machine x, machine y is prepared for signal **resume**" we need a language which can express constraints over time. The standard way to do this is to use a temporal logic. We want to supply these properties to automated state space analysis tools, and so chose Computation Tree Logic (CTL)[26, 27], as a language for expressing the properties.

CTL expressions have three parts: a path quantifier, a temporal quantifier, and a predicate.

A path is a sequence of machine states. Different execution orderings yield different paths. Path quantifiers identify whether an invariant should apply to all future paths ($\forall$) or at least one future path ($\exists$). For each quantified path, a temporal quantifier asserts that a condition will hold ($\Box$) in every state along a path, at some point along a path ($\Diamond$), in the next state ($\bigcirc$) along a path, or will hold in every state along a path until (`[ U ]`) some other condition becomes true. These conditions are logical propositions about the state of machines. Path and temporal quantifiers are combined syntactically in CTL. So, for example, the combined quantifier $\forall\Box$ means the following predicate holds globally on all possible paths, $\forall\bigcirc$ means the subsequent condition holds in the next state of all paths, and $\forall[x\mathtt{U}y]$ means that condition `x` holds in all future paths until some other condition `y` becomes true. CTL expressions conveniently and succinctly express the Mealy machine composition properties.

6.2.3.1 Receptiveness

If we assume that machines have been extended with the special `error` state and all of the accompanying transitions, then the receptiveness property can be stated in CTL as follows. Let m be the machine in question, then:

$$\forall\Box\neg(\mathtt{m.state} = \mathtt{error})$$

specifies that for all execution paths machine m is never in the error state. Since we know by construction that m can only enter the error state when it receives a signal it cannot handle, we can infer that this CTL property checks receptiveness.

6.2.3.2 Freedom From Divergence

To specify freedom from divergence, we take advantage of the disjointness of transient and equilibrium states. For any Mealy machine m, let:

$$Active(m) == \bigvee_{\mathtt{s} \in Trans(m)} (\mathtt{m.state} = \mathtt{s})$$

where $Trans(m)$ is the set of transient states of m. $Active(m)$ is a predicate which is true whenever m is in a transient state and false whenevenr m is in an equilibrium state. Assuming such a predicate makes the CTL constraint for this property simple to express:

$$\forall\Box(Active(m) \Rightarrow \forall\Diamond(\neg Active(m)))$$

This says that, for all execution paths, if a machine m is in a transient state ($Active(m)$), then for all subsequent execution paths, m will eventually get into a non-transient (equilibrium) state. This captures both forms of divergence. If a machine m is stuck in state s waiting for a signal that never arrives, then there will be no subsequent paths in which m is in an equilibrium state because m will never transition out of s. If a machine m is involved in an endless cycle of transient signaling, it may go through many different transient states $s_1, s_2, \ldots, s_n$, but from any of these states, there will be no subsequent path in which m enters an equilibrium state.

6.3 Testing Machine Inter-Operation

Having formalized the Mealy machine representation of the MDL orderings and correctness properties of their composition, we now look to the problem of checking that the machines preserve these properties when composed. Since, as machine designers, we have the ability to define states and signals, machines can issue signals which are interpreted differently by different classes of receiving machines. This means we cannot test the machine associated with, say, the `alt` ordering in isolation because it may issue signals that have one behavior when received by `leaf` machines and another when received by `seq` machines. On the other hand, machines connected in this tree hierarchy do not broadcast signals to all machines, but rather they receive and propagate events in a strict hierarchical fashion.

This indicates that we can test much of the inter-operation of machines by testing an exhaustively constructed set of small configurations. That is, we ought to find most of the bugs in machine design by testing small configurations of machines. To test the `seq` machine, for example, we run tests with every possible Mealy machine as children. The next obstacle to overcome is to build the scaffolding required to test these machine configurations. We take advantage of automated technology to build this scaffolding and run these tests.

6.3.1 Model Checking

Rather than build a large suite of tests for our machines, we took advantage of state space analysis tools which can verify properties of state machines. Specifically, we used a tool called the Symbolic Model Verifier (SMV)[66] to test our Mealy machines. Model checkers validate state machines for

conformance to properties specified in temporal logic.

The idea behind model checking is simple. A problem is modeled in terms of a state machine, and that model is exhaustively searched to verify that it upholds certain properties. Model checkers are useful for testing because they exhaustively verify properties and, for some classes of property, the model checker can demonstrate failure with a counter-example. As a result of employing an exhaustive path analysis, when the model checker discovers a constraint violation, it can demonstrate a signal trace that makes the machines violate the constraint! This property is crucial to giving design feedback on how a proposed Mealy machine fails to inter-operate with other machines.

We found the model checker to be more reliable and easier to build and maintain than a large suite of test cases. The tool was so reliable that we incorporated it into our development process much like a compiler is used to develop programs. The development is iterative. Given a set of machines to test, we encode them in the input language of SMV and assert that the correctness properties hold. We then run SMV over this input. If SMV detects a failure, it lists counter-examples. We then use these counter-examples to trace the design flaws, fix them, and iterate. The design process converged on the set of Mealy machines which we use to generate MMTK components.

6.3.2 Mealy Machine Closures

One problem with testing small configurations of machines is that many of the Mealy machines expect to be connected to parent machines *and* child machines. The obvious solution is to "top" machines expecting a parent with the ceiling machine and "ground" all machines expecting a child with leaf machines, but this inflates the state space which we must test, and, unfortunately, the model checker becomes prohibitively expensive when we apply it to configurations of more than three machines. To overcome this practical difficulty, we defined a closure operation over Mealy machines. This operation tops and grounds machines which occur at communication fringes.

To understand the need for closures, consider what is involved in model checking configurations of Mealy machines. Since machines are reactive, they must be tested in configurations. Otherwise, they will always get stuck waiting for signals that cannot arrive because we forgot to incorporate the machines which generate these signals! Unfortunately, this problem does not go away when we just incorporate incident machines in a testing configuration becuase these incident machines

(unless they are leaf or ceiling) will get stuck waiting on signals from an unincorporated machine. It is not clear how to test configurations containing fringes like this.

What is needed is a **closure** operator that we can apply to machines whose channels are not connected to other machines to force them to exercise any and all functionality that could have been provided by these unincorporated fringes. The name closure is motivated by its meaning in topology. In a very liberal sense, configurations with unconnected fringe machines are like open sets in $\mathbb{R}^N$ because they do not have discernable **boundary points**. Of course, for configurations of state machines, boundary points are signals which a configuration lacks because of unresolved channel connections. In topology, the closure of a set A is defined as a set which contains all of its boundary points. Interpreting boundary points as fringe signals, this suggests defining a closure for configurations and testing closures. Clearly, since closures contain their boundary points, they may be tested in isolation.

Definition 6.3.1. *Let $\sigma \in \mathcal{L}(\Sigma)$ be a configuration. The* **closure** *of σ is another configuration, denoted $\overline{\sigma}$, in which all actions and reactions which involve communication over fringe channels have been replaced by λ.*

It may seem counter-intuitive that closure, which is supposed to incorporate boundary signals, removes information. To understand this, note that we encode transitions guarded by λ as non-deterministic Mealy machine transitions. SMV includes non-deterministic transitions in the state graph of a model. Since the transition is non-deterministic, there will always be paths in which it is chosen. However, if a transition is guarded with an input, the transition will only be taken if the input guard is satisfied (that is, when some other Mealy machine issues a signal). But note that what is being removed are guards. If a transition $s \xrightarrow{i[o]} t$ is closed, then there will be an execution scenario in which this transition is taken.

6.3.3 Generating SMV Input from MM Descriptions

The SMV input files are automatically synthesized from MM finite control specifications. The tool is again the mmc compiler. Consider, for example, the MM definition of the seq machine (which constitutes a number of .mm files). The command:

```
$ mmc -Mseq seq*.mm
```

creates a file called: `seq.smv`. This file contains an SMV readable description of the aggregate finite control expressed in all of the .mm files associated with the `seq` ordering. The compiler can also compute arbitrary machine closures through command line switches.

This is convenient because it allows us to create large regression testing engines which synthesize testing configurations using mmc, invoke the smv tool on these configurations, and then report failures if any are found. This degree of automation supported the design and validation of the Mealy machines and also makes MDL language extension a less daunting task.

6.3.4 Testing Adequacy

The testing suite we built for our Mealy machines only tests specific configurations of machines. While this is highly useful for debugging, it begs the question of testing adequacy:

Question 6.3.4.1. *Does there exist a finite set of testing configurations which, if validated, guarantees that machines compose correctly in* any *arbitrary configuration?*

Since machines delegate control commands, the answer to this question is not immediately clear. In the next chapter, we prove a testing adequacy theorem (Chapter 7) which identifies a set of configurations which sufficiently tests machine composition to guarantee correctness in arbitrary configurations. This is a key result, and we use it to construct the configurations we applied when designing the Mealy machines.

6.3.5 Summary

To summarize, we developed finite control descriptions for the ten Mealy machines needed to implement MDL behavior expressions. These descriptions were written using the MM notation. We then validated these specifications for adherence to the receptiveness and freedom from divergence properties using the SMV model checker. The input descriptions to the model checker were generated automatically from the MM descriptions. This validation technology was applied to a collection of machine configurations. We proved that this collection sufficiently tests Mealy machine behavior in composition. Finally, the MMTK components were generated automatically from the MM descriptions. This lends a high degree of confidence to the correctness of implementation of the MDL orderings and enables an non-intrusive and highly precise implementation of task modeling

languages.

Chapter 7

The Composition Adequacy Theorem

Necessity ... cannot be derived from experience. – Immanuel Kant, *Critique of Pure Reason*

We now present the **Composition Adequacy Theorem**, which states that the model checking we performed in Chapter5 adequately tests the MMTK components for composition in arbitrary configurations. This theorem plays a critical role in simplifying code generation from abstract task models. Since every structuring operation in MDL maps to a component in MMTK, and since by the theorem, all connections behave correctly regardless of the components being connected, model based code generation from task models can be realized by a simple syntactic transformation.

In general, proofs of software composition live in danger of being swamped with unwieldy implementation detail. Software component source code is difficult to manipulate in formal deductions. This obstacle is typically overcome by reasoning about a *model* of the component source. In Chapter 6, we described the Mealy machine model of MMTK components and pointed out that machines have hundreds of states. Since these machines interact, the state space of a given task grows as the product of the state spaces of the individual components. With this much state inherent in the problem, proofs of composition are still unwieldy. We made the proof tractable by delegating much of the proof tedium to the model checker[66]. The proof of the Composition Adequacy Theorem describes how to construct adequate testing configurations for submission to the model checker. The proof, when augmented with successful model checking, demonstrates that MMTK components compose correctly in arbitrary tree configurations.

The insight behind the proof is that a well-behaved projection of run-time component behavior is faithful to the correctness properties in the sense that any failing behaviors will be observable as failures in the projection. We formalize this idea to get the **Sufficiency Theorem**. Our application of the technology is a construction of testing configurations which are then fed to the model checker.

The correctness of these configurations is proved by the **Coverage Theorem**. These theorems are used to prove the Composition Adequacy Theorem.

7.1 Formal Statement of the Theorem

The Composition Adequacy Theorem states that any syntactically well connected component hierarchy behaves well at run-time in the sense that components never receive a signal they are not prepared to receive, they never get *stuck* waiting for control signals which cannot arrive, and they never enter an infinite cycle of signaling which never reaches equilibrium.

To formalize and prove this theorem requires some precise notation. Recall that these properties are defined in CTL with reference to a particular occurrence m. We now adopt the notation:

$$\phi_r(m) == \forall\Box\neg(\texttt{m.state} = \texttt{error})$$
$$\phi_d(m) == \forall\Box(Active(m) \Rightarrow \forall\Diamond(\neg Active(m)))$$
$$\phi(m) \;\; == \phi_r(m) \wedge \phi_d(m)$$

where $Active(m)$ is the disjunction $\bigvee_{s \in Trans(m)}(\texttt{m.state} = \texttt{s})$. $\phi(m)$ is just a short-hand for expressing the composition properties of m. $\phi(m)$ is a formula of temporal logic. Its truth or falsehood is judged relative to one or more *models*. Models in this sense are mathematical structures which behave according to the temporal formulae. We want to ask if the behavior of a given configuration σ is a model of a temporal formula. Traces are natural models of temporal formula. A trace is a sequence of state snapshots which identify the state of machines in the hierarchy and any signals issued and/or received. For a given configuration σ, let $tr(\sigma)$ to be the set of traces which could be observed of σ. Occasionally we will need to refer to a specific trace sequence within $tr(\sigma)$. We usually use the symbol ς (read "varsigma") for this purpose. Counter-examples generated by the model checker are traces in $tr(\sigma)$, and so we often will use ς to refer to a counter-example.

Theorem 7.1.1 (Composition Adequacy Theorem). *Let σ be any configuration in $\mathcal{L}(\Sigma)$. For all machine occurrences m in σ, $tr(\sigma) \models \phi(m)$.*

This says that any machine occurrence m in any configuration σ composes correctly in the sense that m is always receptive and always divergence free. The proof will consider all possible

configurations of Mealy machines ($\mathcal{L}(\Sigma)$, an infinite set) and demonstrate that $\forall \sigma \in \mathcal{L}(\Sigma)$, each occurrence m within σ respects the properties.

7.2 Proof Technique: Symbolic Model Checking

Model checking can be thought of as a limited form of theorem proving. Whereas theorem provers deduce consequent truths from antecedents using rules of logic, model checkers "check" to see if a given model violates the claims made by a theorem. That is, given a model M and a theorem T, a model checker constructs an efficient representation of all executions of M and then checks each execution for violations of T. If no violations are found, then M is claimed to be a model of the T (sometimes written $M \models T$).

We structure the proof of the adequacy theorem so that it utilizes a finite set of *lemmas* of the form: $tr(\sigma) \models \phi(m)$ for some $\sigma \in \mathcal{L}(\Sigma)$ and some machine occurrence m within σ. Lemmas of this form can be validated using the SMV model checking tool as we demonstrated in Chapter 6. With these lemmas, the proof of the adequacy theorem establishes a finite set of obligations which Mealy machine finite control design must satisfy. We think of these obligations as regression tests which can be applied whenever a modification is made to the finite control of any Mealy machine in our collection. This view suggests the following strategy:

1. design the finite control of the Mealy machines for each MMTK component,
2. use the model checker to try and prove the lemmas about their composition and find counterexamples,
3. use the counterexamples to fix the problem in the Mealy machine finite control logic, and
4. repeat the process until no further violations of are detected.

We used this strategy to get the design of our Mealy machines correct.

In order for the strategy to work, there must be only a finite number of these lemmas, and they must be clearly articulated. The problem with checking configurations for satisfiability of $\phi(m)$ is that there are too many configurations to check! The key to making this strategy work is to identify some finite basis of configuration behavior and apply the model checker to elements in this basis.

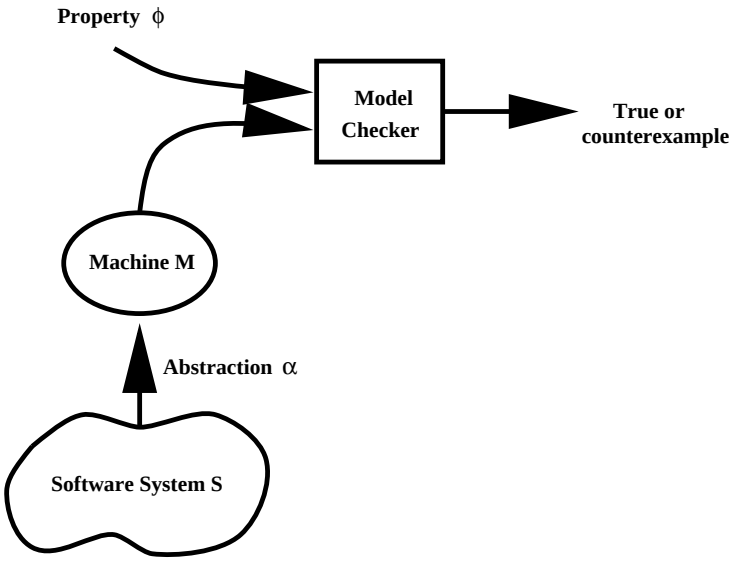

Figure 30: Use of abstraction to incorporate model checking into the proof of a theorem.

7.2.1 A More Realistic Use of Model Checking

In order to identify a finite set of model checking lemmas, we appealed to an insight suggested by Wing and Vaziri-Farahani[103] in their 1995 case study. They suggested model checking a finite state *abstraction* of an infinite state space (see Figure 30). This requires developing a mapping α from the (possibly infinite) state space of the real system to a finite state machine model, and then feeding this model and a property ϕ to check into a model checker. The abstraction function α must preserve enough of the structure of the real state space so that the property ϕ does not become vacuously true. Moreover, any counterexamples of ϕ in the real system must be preserved in the abstract space. Otherwise, the debugging benefit of model checking will not be fully realized.

Wing and Vaziri-Farahani suggest exploiting the nature of the property ϕ in order to formulate abstractions. One method is to exploit *domain specific knowledge*. Our domain (Mealy machines under a synchrony hypothesis) exhibits highly regular run-time behavior. Machines do not run concurrently, but rather pass control back and forth between adjacent machines in a tree hierarchy. This behavior is formalized in Observation 7.3.1 below. Using this property, we chose a carefully crafted projection of behavior as the abstraction.

The software system S of Figure 30 is a representation of possible Mealy machine execution traces. We observed that certain projections of these traces induce a finite number of equivalence

classes and, moreover, that representatives of these classes preserve the properties we are interested in proving. That is, if $\varsigma \in tr(\sigma)$ is a counterexample of $\phi(m)$ that the projection of ς is also a counterexample of $\phi(m)$. The sufficiency theorems prove this point.

7.3 Formalizing the Abstraction

Mealy machines in this domain exhibit a highly regular run-time behavior. By the synchrony hypothesis, machines do not run concurrently. Furthermore, the direct communication influence of a machine is limited to machines which are adjacent to it in the tree hierarchy. This regularity makes the model amenable to an abstraction of the run-time behavior. This abstraction, which is formally defined as a projection, preserves counter-examples of the correctness properties as we demonstrate in Section 7.4. Furthermore, this abstraction can be realistically computed by a symbolic model checker as we demonstrate in Section 7.5. We now formalize the domain specific machine properties and the projection abstraction.

7.3.1 Domain Knowledge

Mealy machines connected in a tree topology under a synchrony hypothesis exhibit very regular execution traces. A large class of traces, which would be legal in absence of either the synchrony hypothesis or the tree connectivity, cannot occur in their presence. This aspect of run-time behavior is useful for proving theorems about abstractions.

The synchrony hypothesis introduces the disjoint partitioning of machine states into either the *Equilibrium* or *Transient* class (Section 6.1.2). Consider, for example, the configuration: $\mathsf{ceiling} \rhd (\mathsf{leaf}_1 \lhd \mathsf{alt} \rhd \mathsf{leaf}_2)$. When the system is in an equilibrium state, either leaf occurrence (leaf_1 or leaf_2) might recognize an external user event (like a key press or mouse-click) and begin a transient flurry of control signals. Suppose this happens and the occurrence leaf_1 recognizes the external activity. Then, by the synchrony hypothesis, this occurrence (and, in fact, the other machines in the configuration) will reach an equilibrium state before leaf_2 could recognize an external user event. While in this transient flurry, control (in the form of signals) must propagate by a single thread because Mealy machines can output only one signal at a time.

Recall that for any configuration σ, we can define the set $tr(\sigma)$ of *execution traces* which could

be observed of σ. Since our computational model does not admit true parallelism, execution traces are sequences of machine state transitions. Let m be an occurrence of a Σ-machine in a particular configuration σ. We can mark Mealy machine transition tuples by the occurrence m_i to get a quint:

$$(s \xrightarrow{c_1?e_1[c_2!e_2]} t)_{m_i}$$

which defines a transition in machine occurrence $m_i \in Occurrences(\sigma)$. This transition takes state s to state t in response to event e_1 on channel c_1 and reacts by issuing event e_2 down channel c_2. A σ-execution trace is a sequence of these quints, and the set $tr(\sigma)$ contains all such traces. In the arguments that follow, we will refer to set of all such quints as *Quint* and a particular quint as an *occurrence-m* transition.

Behavior traces of Mealy machines can be represented as a unique sequence of quint sub-sequences as follows. Recall that states in a Mealy machine fall categorically into one of two disjoint classes: *Equilibrium* or *Transient*.

Definition 7.3.1 (Transient Sub-sequence). *Let $s \in tr(\sigma)$ be a valid behavior trace of σ. Let $t : \mathbf{N} \longrightarrow Quint$ be a sub-sequence of s. t is a* **transient sub-sequence** *of s if:*

1. $source(head(t)) \in Equilibrium \wedge target(head(t)) \in Transient$,
2. *if* $\operatorname{dom} t$ *contains an upper bound b, then:*

$$source(t(b)) \in Transient \wedge target(t(b)) \in Equilibrium$$

3. *If, for any other quint $q \in \operatorname{ran} t$, $source(q) \in Equilibrium$, then q is the head of a proper transient sub-sequence of t.*

Transient sub-sequences begin in an equilibrium state, and if they end, they end in an equilibrium state. Furthermore, all intermediate states in the sequence are transient unless there is a contained transient sub-sequence.

Transient sub-sequences are a convenient representation of behavior traces because for every pair of consecutive quints q_j, q_{j+1} in a transient sub-sequence, the machine occurrences of q_j and q_{j+1} are incident in σ. That is, transient sub-sequences of execution traces exhibit a locality in direct correspondence to the tree topology. The following observation captures these properties.

Observation 7.3.1 (Transient Adjacency). *Let s be any execution trace in $tr(\sigma)$, and let t be a transient sub-sequence of s. Then for all consecutive quints q_i, q_{i+1} in t, either:*

1. $occurrence(q_i) \rhd occurrence(q_{i+1})$
2. $occurrence(q_{i+1}) \lhd occurrence(q_i)$
3. $occurrence(q_{i+1}) \rhd occurrence(q_i)$
4. $occurrence(q_i) \lhd occurrence(q_{i+1})$

Now let $S : \mathbf{N} \longrightarrow (\mathbf{N} \longrightarrow Quint)$ be a sequence of quint sequences. Then $\operatorname{dom} S$ is the set of indices into S, and $\operatorname{ran} S$ is the set of indexed quint sequences. If each sequence in $\operatorname{ran} S$ is a transient sub-sequence, then we call S a **transient partition** of s. The definition uses the distributed concatenation operator $\frown\!/$ which maps a sequence of sequences to a single sequence by concatenation.

Definition 7.3.2 (Transient Partition). *Let $s \in tr(\sigma)$ be any trace of σ, and let $S : \mathbf{N} \longrightarrow (\mathbf{N} \longrightarrow Quint)$ be a sequence of quint sequences. The sequence S is a* **transient partition** *of trace s if:*

1. $\frown\!/\, S = s$, *and*
2. *each $S_i \in \operatorname{ran} S$ is a transient sub-sequence of s.*

The definition implies that each S_i is a maximal containing transient sub-sequence. If this is not the case, then either S_{i-1} or S_{i+1} would not be a transient sub-sequence. The following Corollary formalizes this observation:

Corollary 7.3.1. *Let $s \in tr(\sigma)$ be any trace of σ. There is a unique transient partition S such that $\frown\!/\, S = s$.*

7.3.2 Projections

The correctness properties in which we are interested express temporal behaviors over the state-space of individual machines. Counter-examples of these properties are particular traces (denoted ς) in this state space. That is, for any configuration $\sigma \in \mathcal{L}(\Sigma)$, any counter-examples ς are legal execution sequences of σ ($\varsigma \in tr(\sigma)$). This means that counter-examples will demonstrate the same

transient sub-sequence locality that any other trace in $tr(\sigma)$ would demonstrate. With this locality, it is likely that only some portion of a given counter-example will be relevant to a flaw in the design of a component. This suggests that an appropriate abstraction is sequence projection.

We first define the function π_J over quints.

Definition 7.3.3. *Let $\sigma \in \mathcal{L}(\Sigma)$ and let J be a set of occurrences in σ. Then for any quint q of the form: $(s \xrightarrow{c_1?e_1[c2!e_2]} t)_m$, define:*

$$\pi_J(q) = \begin{cases} q & \text{if } m \in J \text{ and } m.c_1 \in J \text{ and } m.c_2 \in J \\ (s \xrightarrow{c_1?e_1} t)_m & \text{if } m \in J \wedge m.c_1 \in J \wedge m.c_2 \notin J \\ (s \xrightarrow{\lambda[c_2!e_2]} t)_m & \text{if } m \in J \wedge m.c_1 \notin J \wedge m.c_2 \in J \wedge t \neq \texttt{error} \\ (s \xrightarrow{\lambda} t)_m & \text{if } m \in J \wedge m.c_1 \notin J \wedge m.c_2 \notin J \wedge t \neq \texttt{error} \\ \bot & \text{otherwise} \end{cases}$$

The function π_J strips a quint of any functionality that is not associated with an occurrence in J. Given a quint q, if both the source of the input channel and the target of the output channel are in the occurrence set J, then q is unaltered. If, at the other extreme, no occurrence aspect of q (either the occurrence which witnesses q, or it's input channel occurrence, or its output channel occurrence) is in J, then q is filtered out by being mapped to $\bot$. The cases in between these extremes require some justification. Suppose the q is defined as above, and m is in the set J, but, either the source of q's input channel or the target of its output channel (or both) are not in J. Then, under π_J, these communications are filtered out and replaced by a λ-transition. We do this because a projection should not reference any occurrences not in J.

The operation (Π_J) over execution traces is the distributed application of the operation π_J over quints. When applied to an execution trace $\varsigma \in tr(\sigma)$, Π_J filters ς by the set of non-$\bot$ images of quints under π_J. The formal definition is a bit thick. It uses the filter operator $\upharpoonright$ which filters a sequence by a set. This basically projects ς into a new sequence which does not contain any $\bot$ elements. More formally:

Definition 7.3.4. *Let $\sigma \in \mathcal{L}(\Sigma)$ and let J be a set of occurrences in σ. Then forall $\varsigma \in tr(\sigma)$*

$$\Pi_J(\varsigma) == \varsigma \upharpoonright \{(q \in \operatorname{ran} \varsigma) \wedge (\pi_J(q) \neq \bot) \bullet \pi_J(q)\}$$

So, for example, if $\sigma = (m_1 \lhd m_2 \rhd m_3) \lhd m_4 \rhd m_5$, and $\varsigma \in tr(\sigma)$, then $\Pi_{\{m_2, m_4\}}(\varsigma)$ will be ς with the quints associated with occurrences m_1, m_3, and m_5 removed, and with all communication with said occurrences replaced by λ-actions and reactions.

Clearly, if a quint exists in an execution trace, and the occurrence machine associated with the quint is in the projection set, then the quint will not disappear under projection. Some information, however, like input or output signals, could disappear. If this information is critical to a counterexample, then the projection fails to be an abstraction as defined by Wing et. al. because it loses detail that is required in the proof of the theorem. As an example, consider a trace in which an occurrence m goes into the `error` state (introduced in Section 6.2.1). If the occurrence initiating the signal which carries m into the `error` state is not in the projection set, then the source of the error will not be present in the projection. Without this, the counterexample will be incomplete. We now prove that, if a trace is a counterexample of a correctness property, one of a small set of its projections will also be a counterexample of that property.

7.4 The Sufficiency Theorems

This section introduces two theorems which relate counterexamples of correctness properties to projections. Before delving into these theorems, we establish some additional notation for talking about Mealy machine make-up. If m is an occurrence of a Mealy machine in some configuration ς, then:

1. $States(m)$ denotes the set of states in m;
2. $Trans(m)$ denotes the transitions of m. Transitions take the form $s \xrightarrow{i[o]} t$ for $s, t \in States(m)$ and input action i and output reaction o;
3. $Inputs(m)$ denotes the set of input signals which can cause transitions in m; and
4. $Outputs(m)$ denotes the set of output signals which m can issue to other machines.

7.4.1 Receptiveness

Let $\sigma \in \mathcal{L}(\Sigma)$ be a configuration in which $p \rhd (o \lhd m \rhd n)$ holds of occurrences m, n, o, p. Let $\phi_r(m) = \forall \Box \neg(\texttt{m.state} = \texttt{error})$. Let $\varsigma \in tr(\sigma)$.

Theorem 7.4.1. *If ς is a counterexample of $\phi(m)$ (that is, $\varsigma \models \neg\phi(m)$), then:*

$$\Pi_{\{m,n\}} \models \neg\phi_r(m) \vee \Pi_{\{m,o\}} \models \neg\phi_r(m) \vee \Pi_{\{m,p\}} \models \neg\phi_r(m)$$

Proof. The property $\neg(\texttt{m.state} = \texttt{error})$ is asserted to hold globally over all paths (traces). Suppose there exists a ς like the one above. There is a sequence index j such that

$$\varsigma(j) = (s \xrightarrow{c_1?e_1[c_2!e_2]} \texttt{error})_m$$

for some non-error state s. Since an input action $c_1?e_1$ made this occur, the *cause* of the error is the quint $\varsigma(j-1)$. Let $x = occurrence(\varsigma(j))$ be the machine occurrence associated with this prior quint. Since it caused the error transition, x must be one of the occurrences n, o, p (by Observation 7.3.1), Neither $\varsigma(j)$ nor $\varsigma(j-1)$ is lost under projection because both are transitions of occurrences within J. It should now be clear that $\Pi_{\{m,x\}} \models \neg(\forall\Box\neg(\texttt{m.state} = \texttt{error}))$ □

7.4.2 Freedom From Divergence

The next theorem identifies projections which preserve counterexamples of the *freedom from divergence* property for a given class of incidence. Let $\sigma \in \mathcal{L}(\Sigma)$ be a configuration for which $(p \rhd (o \lhd m \rhd n)) \mid \sigma$. The theorem is applicable to all traces $\varsigma \in tr(\sigma)$ which are counter-examples to the inevitability of equilibrium. In the lemmas that follow, we assume that the occurrence of interest is m. Let $\phi_d(m) = \forall\Box(\texttt{m.ACTIVE} \Rightarrow \forall\Diamond\neg(\texttt{m.ACTIVE}))$. The theorem is concerned with traces $\varsigma \in tr(\sigma)$ for which ς is a counter example of the property ϕ ($\varsigma \models \neg\phi$). The theorem demonstrates that if any such ς exists, then one of $\Pi_{\{m,n\}}(\varsigma)$, $\Pi_{\{m,p\}}(\varsigma)$, or $\Pi_{\{m,o\}}(\varsigma)$ is also a counterexample of ϕ.

There are two ways in which a given trace might be such a counter-example. Either a machine gets stuck in an infinite cycle of transient state transitioning, or it it gets to a point where it expects a signal from an incident occurrence and this expected signal never arrives.

Observation 7.4.1. *If $\varsigma \models \neg\phi_m$, then either:*

1. *there exists some $k \in \mathbb{N}$ for which $\varsigma(k)$ is the last* occurrence-m *quint in ς, or*
2. *no such k exists.*

In the former case, there is a discernible point at which *occurrence-m* transitions cease to occur; whereas in the latter case, there is no such point. When the former occurs, we say the machine m is *stuck* waiting for a signal that never comes, whereas in the latter case, we say the machine m is *thrashing* in transient states.

The following lemma deals with the thrashing case.

Lemma 7.4.1 (Thrashing). *Let* $\varsigma \models \neg\phi$ *as described, and assume there is no last* occurrence-m *quint in* ς. *Then* $\Pi_{\{m\}}(\varsigma) \models \neg\phi$.

Proof. We want to show that all *occurrence-m* quints in ς are in $\Pi_{\{m\}}(\varsigma)$. By the definition of $\Pi_{\{m\}}$, the only way a quint will not be preserved is if it maps to $\perp$ under $\pi_{\{m\}}$. Let q be any *occurrence-m* quint in ς. Then, by the definition of a quint, $q = (s \xrightarrow{c_1?e_1[c_2!e_2]} t)_m$ for some states $s, t \in \mathit{States}(m)$, some channels $c_1, c_2 \in \{l, r, p\}$, some $e_1 \in \mathit{Inputs}(m)$ and some $e_2 \in \mathit{Outputs}(m)$. Since $m \in \{m\}$, clearly, one of the first 4 cases applies. The disjunction of the channel occurrences in the 4 cases is *true*, so as long as t is not the `error` state, $\pi_{\{m\}}(q) \neq \perp$.

Suppose $t =$ `error`. The `error` state is in the set $\mathit{Equilibrium}(t)$. Recall by construction that there is no transition out of the error state. So, if there was such a q, q would be the *last occurrence-m* quint in ς. But if this is the case, then $\varsigma \models \phi$ which contradicts the assumption that $\varsigma \models \neg\phi$. So t cannot be the `error` state; consequently $\pi_{\{m\}}(q) \neq \perp$; consequently $\Pi_{\{m\}}(\varsigma) \models \neg\phi$. □

Theorem 7.4.2. *Let* $\sigma \in \mathcal{L}(\Sigma)$ *be a configuration in which* $(p \rhd (o \lhd m \rhd n)) \mid \sigma$. *Let* $\phi_m = (\forall\Box(\texttt{m.ACTIVE} \Rightarrow \forall\Diamond\neg(\texttt{m.ACTIVE})))$. *If* $\varsigma \in tr(\sigma)$ *is a counterexample of* ϕ_m *(that is,* $\varsigma \models \neg\phi_m$*), then:*

$$\Pi_{\{m,p\}}(\varsigma) \models \neg\phi_m \vee \Pi_{\{m,n\}}(\varsigma) \models \neg\phi_m \vee \Pi_{\{m,o\}}(\varsigma) \models \neg\phi_m$$

Proof. Consider such a trace ς. By Observation 7.4.1, there are two cases to consider. Either there is some positive integer $k \in \mathbb{N}$ such that $\varsigma(k)$ is the last *occurrence-m* transition in ς, or there is no such k.

Case I There does not exist a k which is the last *occurrence-m* transition in ς. By the Thrashing Lemma 7.4.1, $\Pi_{\{m\}} \models \neg\phi_m$, which implies in this case that all of $\Pi_{\{m,n\}}(\varsigma)$, $\Pi_{\{m,o\}}(\varsigma)$, and $\Pi_{\{m,p\}}(\varsigma)$ model $\neg\phi_m$.

Case II There is such a k, so consider the quint $q = \varsigma(k)$. By the definition of a quint, $q = (s \xrightarrow{c_1?e_1[c_2!e_2]} t)_m$ for some states $s, t \in States(m)$, some channels $c_1, c_2 \in \{l, r, p\}$, some $e_1 \in Inputs(m)$ and some $e_2 \in Outputs(m)$. Since q is the last *occurrence*-m quint in ς, it must be the case that q is stuck in state t. By Transient Adjacency (Observation 7.3.1) q can only transition out of t by a signal issued from occurrence n, o, or p. Suppose it could transition out of t by a signal from n. Then clearly any quints involving m or n are preserved in the projection $\Pi_{\{m,n\}}(\varsigma)$. Furthermore, $\Pi_{\{m,n\}}$ will be a counter example to ϕ because it will demonstrate the communication which causes machine m to get stuck. The same holds for $\Pi_{\{m,o\}}(\varsigma)$ and $\Pi_{\{m,p\}}(\varsigma)$.

□

7.5 The Coverage Theorem

Recall the definition of Mealy machine closures from Section 6.3.2. The coverage theorem demonstrates that sufficient projections (as defined by the **Sufficiency Theorem**) occur in the execution representation which the model checker creates for configuration closures. This is the last step in the proof because it directly relates finite representations (projected execution traces) to finite models (closed configurations).

Theorem 7.5.1 (Coverage Theorem). *Let $\sigma \in \mathcal{L}(\Sigma)$ contain occurrences m and n such that $m \rhd n$. Then for all $\varsigma \in tr(\sigma)$, there exists an $S \in tr(\overline{m \rhd n})$ such that $\Pi_{\{m,n\}}(\varsigma)$ prefix S.*

This theorem basically tells us how to construct closure configurations which, when supplied to the model checker, will sufficiently test the given machines for the composition properties.

In order to prove the theorem, we had to formalize the effect of configuration closures on the finite control of occurrences within the closed configurations. That is, if occurrences m and n occur in a configuration $\sigma = m \rhd n$, we need to understand the effect of $\overline{\sigma}$ on m and n.

Clearly, $\overline{m \rhd n} \neq \overline{m} \rhd \overline{n}$ because, in $\overline{m \rhd n}$, the right channel of m is connected to the parent channel of n; whereas $\overline{m} \rhd \overline{n}$ is an illegal channel connection because the $\overline{m}$ closes all channels of m and $\overline{n}$ closes all channels of n. This issue comes up in the proof of the coverage theorem, and so we need a notation which relates the closing of a configuration to the specific channel closings of a given

occurrence in the configuration. If m is a machine occurrence, let $m_{\overline{c}}$ denote the non-deterministic Mealy machine arrived at by closing all channels *except* c in m. Given this, then the following table defines $\overline{m \rhd n}$ for all all m and n:

$$\overline{m \rhd n} = \begin{cases} m \in \hat{\Sigma_1} & m_{\overline{l}} \rhd n_{\overline{p}} \\ m \in \hat{\Sigma_2} & m_{\overline{r}} \rhd n_{\overline{p}} \end{cases}$$

Our definition of closure does not forget transitions, only the communication information (guards and reactions) associated with transitions. The following lemma follows from the definition of machine closure.

Lemma 7.5.1. *Consider an incidence $m \rhd n$ in a configuration σ. Let q be the transition $(s \xrightarrow{i[o]} t)$. If $q \in Trans(m)$, then $\pi_{m,n}(q) \in Trans(m_{\overline{r}})$.*

This says that any transition in an occurrence m corresponds to a transition in $m_{\overline{r}}$ under the projection.

The proof of the coverage theorem now follows:

Proof. We prove this by induction on the length of $\Pi_{\{m,n\}}(\varsigma)$.

Base $length(\Pi_{\{m,n\}}(\varsigma)) = 1$. Let q be the transition in ς which maps to $(\Pi_{\{m,n\}}(\varsigma))(1)$. Assume without loss of generality that $q \in Trans(m)$. Clearly q must be the first *occurrence-m* transition in ς, for if not, then $length(\Pi_{\{m,n\}}(\varsigma)) > 1$. Since q is the first such quint in ς, $source(q)$ must be the initial state of m. Clearly q is an occurrence m transition, and so by Lemma 7.5.1, $\pi_{\{m,n\}}(q) \in Trans(m_{\overline{r}})$. Moreover, since $source(q)$ is the initial state of m, $source(\pi_{\{m,n\}}(q))$ is the initial state of $m_{\overline{r}}$. Therefore $\langle \pi_{\{m,n\}}(q) \rangle$ is a prefix of a trace in $\overline{m \rhd n}$. This establishes the induction base.

Hypothesis for all $\varsigma \in tr(\sigma)$ with $length(\Pi_{\{m,n\}}(\varsigma)) < K$, then there exists an $S \in tr(\overline{m \rhd n})$ such that $\Pi_{\{m,n\}}(\varsigma)$ prefix S.

Step Let q be the quint in ς which maps to position K under $\Pi_{\{m,n\}}$. Let p denote the quint immediately preceding q in ς. By Lemma 7.3.1, we can assume the reaction of p causes the transition q. Assume without loss of generality that q is an *occurrence-m* transition. Let $c?e$ be the action which initiates q. Again, we assume this action is initiated in ς by the

reaction of p. Let $s = source(q)$. Let $t = target(q)$. Let P be the prefix of length $K-1$ supposed to exist by the induction hypothesis. If we could prove that there is at least one trace S in $tr(\overline{m \rhd n})$ in which $P \leq S$ and the m occurrence is in state s at $S(K-1)$. then, Lemma 7.5.1 guarantees that the occurrence $m_{\overline{r}}$ contains a transition $\pi_{\{m,n\}}(q)$. With this, let $S = P^\frown\langle\pi_{\{m,n\}}(q)\rangle$. $S \in tr(\overline{m \rhd n})$, and clearly, $\Pi_{\{m,n\}}(\varsigma)$ prefix S. This will complete the induction step.

So now to prove the existence of a trace $S \in tr(\overline{m \rhd n})$ in which P prefix S and the m occurrence is in state s at $S(K-1)$. The length of P is less than K, so by the inductive hypothesis, there is a trace in $tr(\overline{m \rhd n})$ of which P is a prefix. To demonstrate that the m occurrence of $\overline{m \rhd n}$ is in state s after P, there are two cases to consider: Either there is no *occurrence-m* transition in P, or, since P is of finite length, there is a *last occurrence-m* transition in P, say at position j.

Case I If there is no *occurrence-m* transition in P, then there could not have been an *occurrence-m* transition in ς (until q of course), in which case $source(q) = init(m)$. But if $\overline{m \rhd n}$ contains a trace of length $K-1$ with no *occurrence-m* transitions, then the m-occurrence will be in the initial state, which we just argued was the source of q.

Case II Suppose on the contrary that, in trace S, the m occurrence of $\overline{m \rhd n}$ is in a state $s' \neq s$ after $K-1$ steps. Then there must be some last *occurrence-m* transition q' in S in which $s' = target(q')$. But note that the position of this transition would have to be less than K, so by the inductive hypothesis, this transition exists at the same position in $\Pi_{\{m,n\}}(\varsigma)$. Herein lies the contradiction. This could not be because that would imply this *occurrence-m* transition occurs *after* $P(j)$ which contradicts our choice of j.

□

The utility of the coverage theorem lies in its assertion that closures of incidences observe any behaviors the incidences would observe in a configuration. The property was part of the motivation for calling these things closures, as they can be thought of as incidence configurations in *all* configurations.

Chapter 8

Validation

We observed that approaches which base generation on task models do a good job at generating stereo-typical form-based user interfaces but fail to generate more graphical. direct manipulation interfaces. The MASTERMIND vision postulates a remedy to this situation through the incorporation of first class task and presentation models. It is the thesis of this work that for this postulated remedy to be successful, three things must hold simultaneously:

1. Task and presentation modeling languages must be expressive enough to precisely model the relevant aspects of the interface,

2. The binding of task and presentation models must be explicitly articulated (preferably via a formal notation), and

3. The code generated from each model must compose *easily* with the code generated by the other models.

We set out to validate this thesis by defining a task modeling language (MDL), formalizing presentation binding within this language, and building a toolkit of reusable components which can be aggregated to implement task models and which compose easily with presentation entities.

In this chapter, we critically analyze the degree to which these results validate the claims laid out in the thesis. The validation proceeds in two steps. First, we question whether or not our solution demonstrates a remedy in the sense postulated by the MASTERMIND vision. We check this by showing that in fact, these techniques can be used to generate interfaces which support direct manipulation and are more graphical than forms based interfaces (Section 8.1). Then we test the claim that the code generated from MDL models is highly composable. In support of the claim we provide a formal model of composition, a proof that the generated code cooperates, and

an argument about the simplicity of code generation (Section 8.2). This chapter concludes with a reflection of this research, lessons learned, conclusions, and avenues of future research.

8.1 The Quality of Generated Interfaces

MASTERMIND aims to increase the richness of applications generated from task models by defining an environment in which multiple models contribute to the design and implementation of an interactive system. This vision manifests itself in the use of both task and presentation models. The most visible validation of our approach comes from the end-to-end application of a task model to generate example interfaces. We chose our examples to demonstrate:

1. the practical adequacy of MDL in modeling both common and domain-specific tasks.
2. the ability of our framework to generate better interfaces than existing model-based approaches.
3. the efficiency of systems generated using the framework.

We tested the quality of user interfaces generated by our approach on two examples (one small and one large). The examples demonstrate three functional quality attributes:

1. the adequacy of MDL for common, highly reusable, tasks (Section 8.1.1),
2. the ability to implement complex, domain specific tasks, and (Section 8.1.2),
3. the ability to generate direct manipulation interfaces (Section 8.1.2).

We measured the performance of generated code using two metrics. The first metric, which we call the Δt method, measures the wall-clock time between when a initiates an interaction (by clicking on a button or dragging and dropping an icon) and when the system has completely reacted to that interaction. Δt is measured as a fraction of a second, and the benchmark we hope not to exceed is a tenth of a second. We found, in fact, that under all test cases, the system never consumed a tenth of a second per interaction. Δt is a satisfactory bottom-line metric because it relates easily to usability, but it does not help us model conceptually how much work the system is doing per interaction. To get a handle on this, we adopted a more intrinsic metric that captures the number

of MMTK component interactions which constitute a user interaction. We call this metric $\#S$ to represent the "number of signals" passed around to implement an event synchronization. We chose to base this metric on event synchronizations rather than complete interactions because often a single interaction employs many event synchronizations. The examples and the associated timing metrics are collected in a testbed which can be recreated with the MASTERMIND environment, thus adding credence to our claims.

8.1.1 Test I: The Save/Print Task Model

This application is intended to validate the use of MDL/MMTK for stereo-typical tasks that occur in modern interfaces. Saving to a file and printing are common tasks in drawing tools, web browsers, and word processing applications. Though a simple application, this example provided our first challenge to linking MMTK components with standard toolkit widgets (as opposed to widgets we designed using a presentation model).

The task model for this example is shown in Figure 31. The *PrintSave* task has two parameters: *portOrLand*, which represents the type of page layout (portrait or landscape) and *saveFileName*, which holds the name of a file in which to save the document. This task decomposes the mutual disabling, which means that the user may alternate back and forth between the *Print* and *Save* tasks arbitrarily until he finishes one of these tasks. The presentation for this task model contains two radio-button panels and a text entry box.

When we ran this example, the dialogue was implemented correctly in the sense that buttons were enabled and disabled as prescribed by the MDL model. We observed no time delays between interactions, but to check ourselves, we instrumented the source code to measure two resource consumption properties of the implementation. The maximum time taken during an interaction was 0.04 seconds, well below our benchmark of 0.1 seconds. This interaction involved 20 signals among the MMTK components.

8.1.2 Test II: The Air Traffic Control Task Model

The objective of this experiment is to validate the end-to-end performance of MDL and MMTK on a deeply domain sensitive task model. We have been using this example for illustrative purposes throughout the document. The presentation appears in Figure 32.

```
task PrintSave
parameters
      portOrLand : string; saveFileName : string
is
      Print ↔ Save
where
      task Print
      is
            choosePrint? ; orientation?portOrLand ; stop
      endtask

      task Save
      is
            chooseSave? ; fileName?saveFileName ; stop
      endtask
endtask
```

Figure 31: MDL task model for print/save dialogue.

8.1.2.1 Qualitative Analysis

This example demonstrates the ability of MDL/MMTK to support direct manipulation interfaces. As flight numbers are keyed in to the text entry box at the top corner of the display, buttons labeled with the flight numbers appear in the airspace at carefully spaced intervals. This is shown in Figure 32. There are three planes in the airspace at this time: `US 1155`, `NOR 316`, and `DEL 111`. As the presentation demonstrates, the latter two flights are in a closer approach pattern than `US 1155`. As more planes come into the airspace, the controller will key in their flight number in the text-entry box at the top left corner of the display.

When the controller decides to change the position of a plane, he does so by dragging the plane to a new location on the canvas. As soon as he presses and holds the mouse button, a feedback object shaped like an airplane appears and follows the mouse to the new location. When the mouse is released, the plane icon moves to the newly selected location. The presentation is actually quite sophisticated. It performs gridding so that airplane presentations are always uniformly placed within the lanes, and it provides feedback objects. Binding with our task model clearly does not impede these features. Figure 33 shows another shot of the display. At this time, the controller has changed the positions of the planes and has added a new plane (`DEL 265`) into the airspace. Figure 34 shows the display after the controller has instructed `DEL 111` to land, has moved the remaining

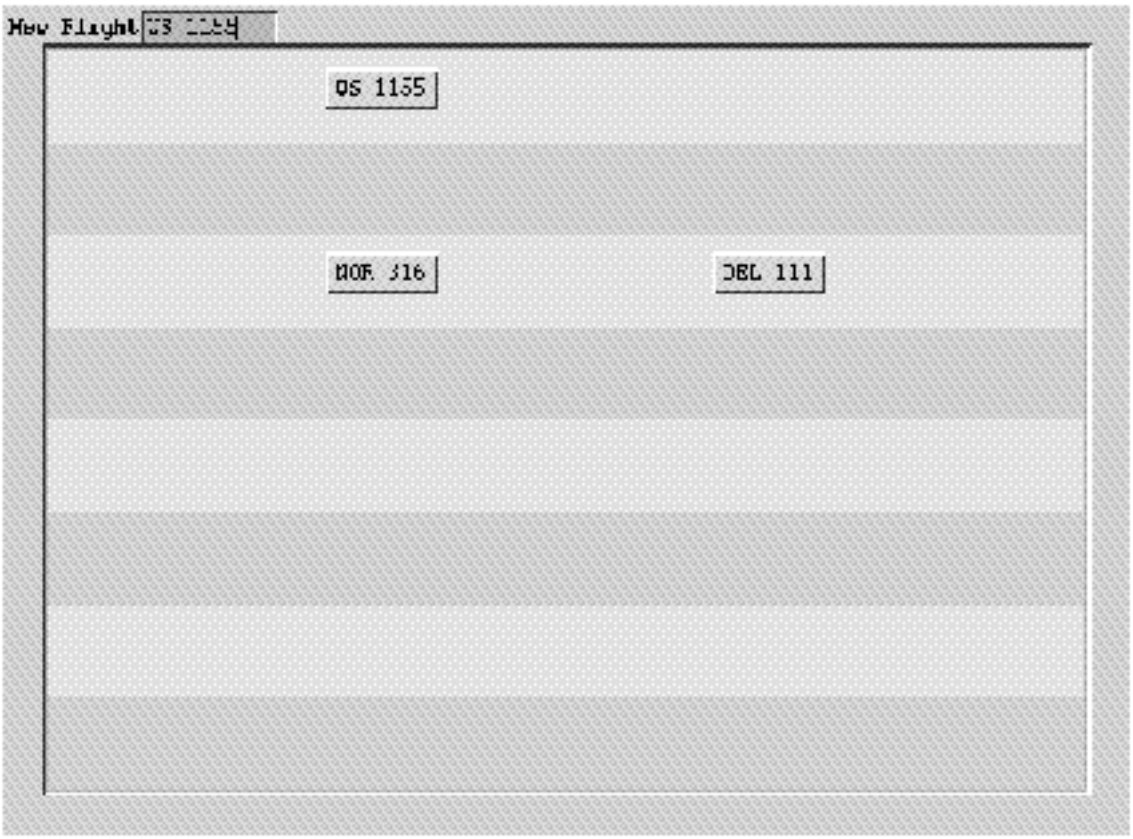

Figure 32: The ATC user interface.

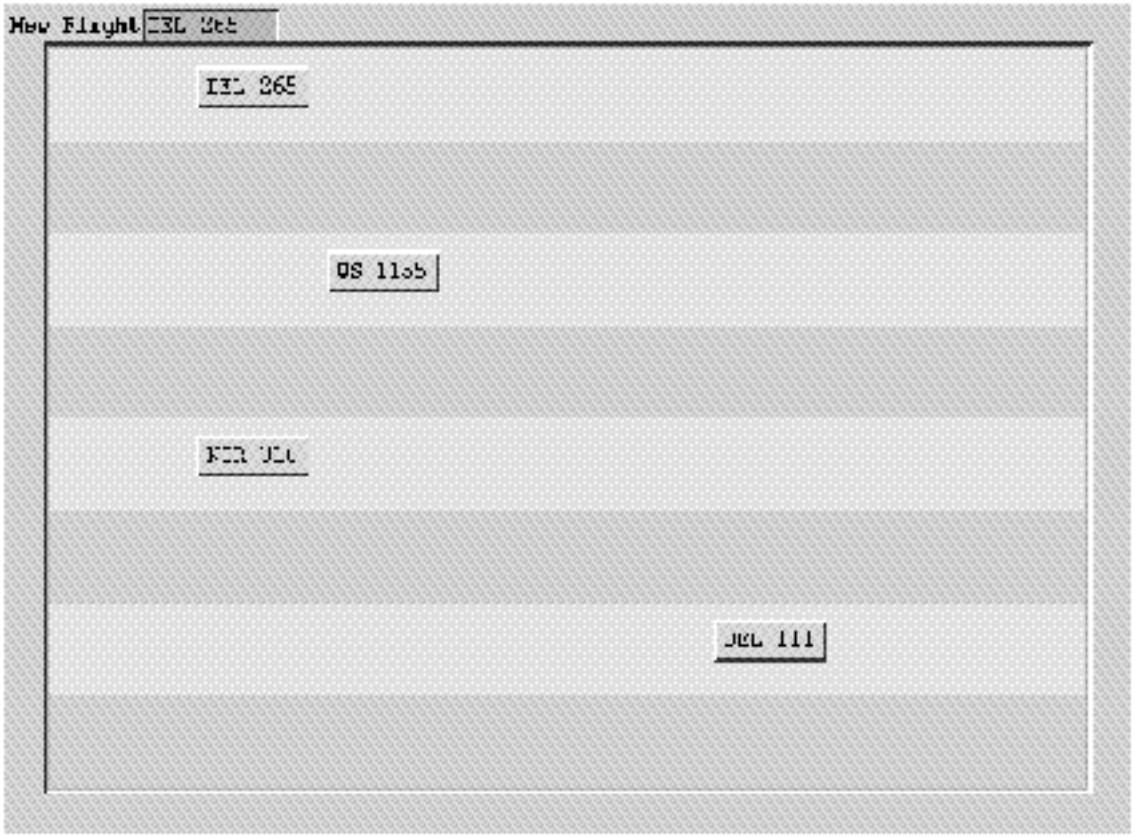

Figure 33: The ATC user interface after adding a new plane.

planes into a closer pattern and has recorded a new flight (`TWA 211`) into the airspace.

The display in this example can hold up to forty plane widgets. It seemed a good test to load up the airspace with widgets and see if we could force the system to commit a noticeable delay. In spite of increasing the load, when we moved planes or double clicked to land the planes, we did not observe a delay.

8.1.2.2 Quantitative Analysis

On the quantitative side, we measured three attributes of the system in execution: executable image size, MMTK signal behavior under three different loads, and the average time of interaction. We conclude that each is reasonable.

The size of the executable file is 958Kbytes. This is not uncommonly large. As a point of

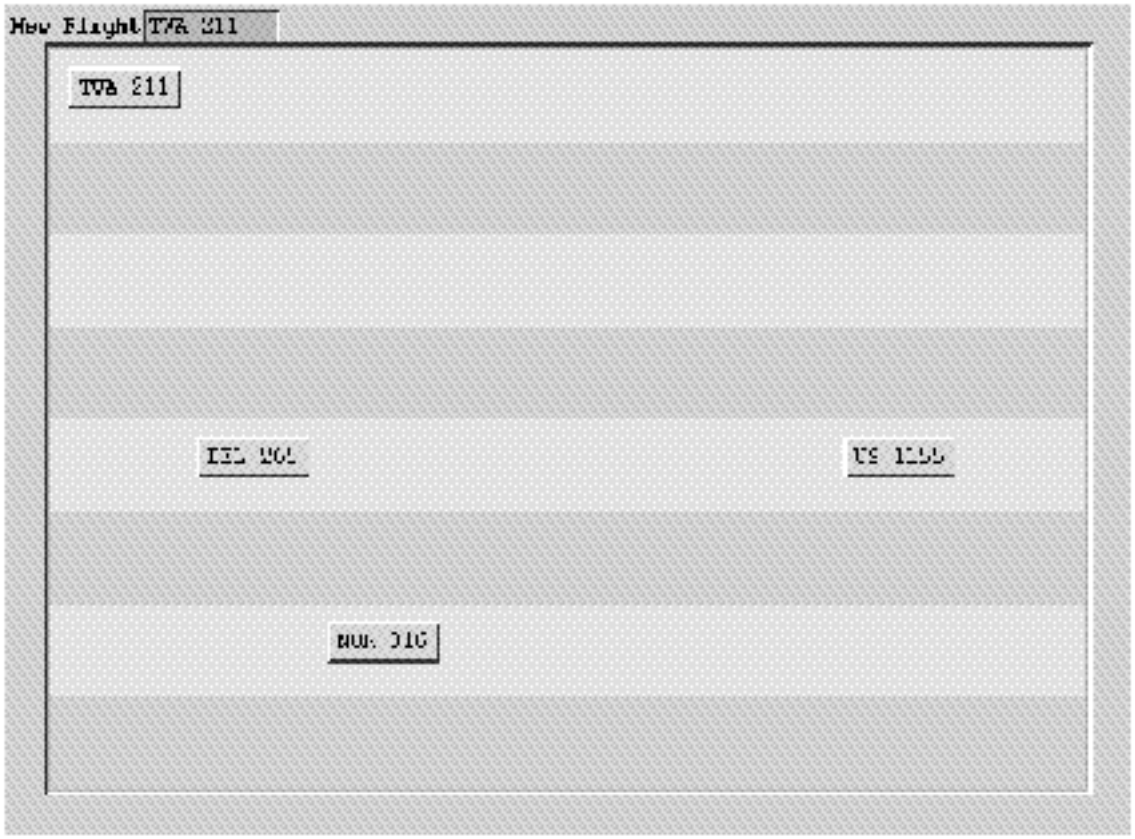

Figure 34: The ATC user interface after landing a plane.

reference, we looked in the Amulet distribution and found a sample program of similar graphical make-up (the Amulet checkers program). This program has a size of 680Kbytes. To understand the difference in size, consider that our executable linked, in addition to the Amulet library, the LEDA library, and the MMTK library.

This example utilizes the dynamic features of MDL/MMTK. This added dimension of complexity could introduce performance degradation. We measured the behavior of the example under two different loads and two different patterns of interaction. The load difference is eight vs. twenty four planes in the airspace. The two patterns of usage are:

1. Load up the airspace to the given number of planes and then perform all position changes and landings, and

2. Interleave position changes and landings with the incorporation of new planes into the airspace up to the maximum.

As in the print/save example, we measure run-time performance using the system clock to measure time differentials around interactions (Δt), and the number of MMTK communications per interaction ($\#S$). Unlike the print/save example, however, there are enough interactions to plot the behavior over time and reason about performance degradation.

The experiments demonstrated two behavior regularities relative to performance degradation. First, the MMTK component trees exhibit a great deal of control passing locality as measured by the signals-per-interaction metric. As tree size grows, the number of signals needed to implement a given interaction remains fixed. This is significant because it demonstrates a locality of behavior in the MMTK component tree. Recall that when new planes are added, an MMTK component tree grows. This data shows that, even though the tree gets deeper, the instantiated sub-processes have *local* behavior. That is, when interaction is occurring only a small number sub-trees are involved. Second, even complex event synchronizations like those which result from a controller creating dynamic *ManageFlight* subtasks do not consume unreasonable resources to implement. We define reasonable here as "the entire interaction takes less than a tenth of a second." As the Δt graphs below demonstrate, no interaction in any of our tests consumed more than nine one-hundredths of a second.

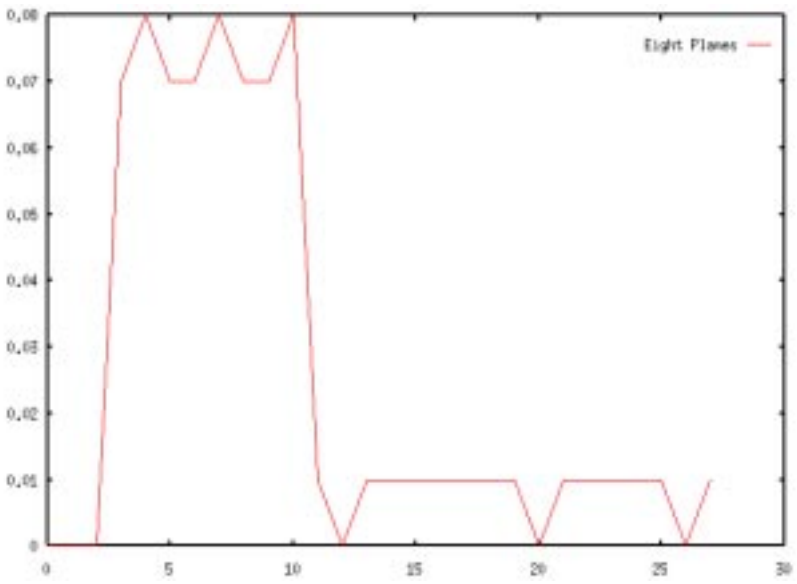

Figure 35: Δt interaction graph with a load of 8 planes.

Eight Planes Figure 35 shows the Δt time series of the ATC interface with a maximum of eight planes at any one time and an usage scenario in which all eight flights are entered before any are landed (hereafter called the *up-front* usage scenario). Points along the x-axis are interactions (entering a flight number, dragging a plane to a new position, and double clicking on a plane to instruct it to land). Points along the y-axis are times as measured by the C library getrusage(1) function. The scale of the y-axis is in hundredths of a second. The most complex signaling occurs when flight numbers are entered. When this happens, three separate events synchronize. Note that in spite of this, the time required for any given interaction remains under a tenth of a second.

Figure 36 shows the signal behavior of the interface ($\#S$) as a function of the event synchronizations. Points along the x axis are distinct event synchronizations. Points along the y-axis are the number of signals issued between MMTK components in reaction to these synchronizations. The high spikes occur when the user enters flight information. There are thirty-six MMTK component communications every time this happens. The reason for the high number is that, when a new flight number is entered, a new *ManageFlight* task is instantiated and enabled for the first time. Process instantiations tend to require more signal propagation than the use of these processes. What is interesting is that no matter how many planes are in the airspace, adding a new one always takes only 36 communications.

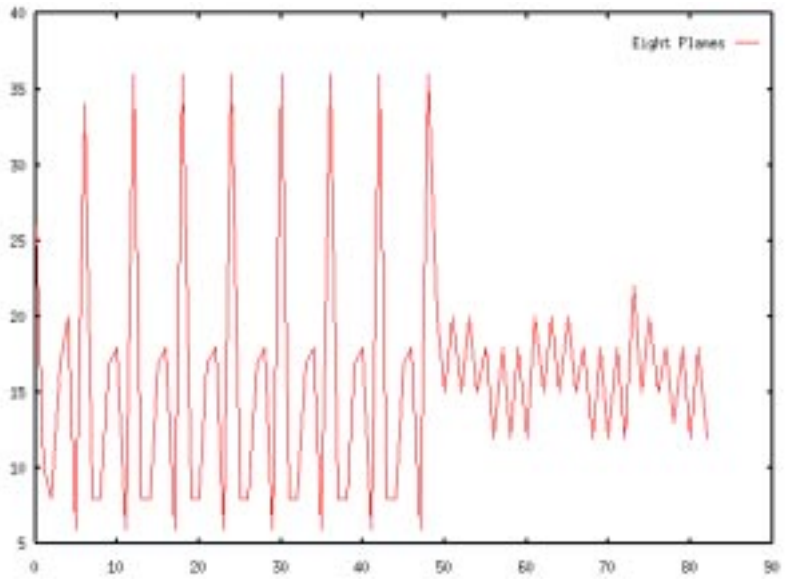

Figure 36: $\#S$ graph with a load of 8 planes.

Twenty Four Planes Figure 38 shows the $\#S$ behavior of the interface with a maximum of twenty four planes (triple the load of the previous example) at any one time and an up-front usage scenario. Points along the x-axis are interactions. Points along the y-axis are the number of signals that get issued between MMTK components in reaction to the interactions. The graph demonstrates that a larger load does not affect the locality of signaling behavior. This is consistent with the clock-time measurements as well.

We tried one more load test with a different scenario of usage. In this scenario, we manipulate and land planes before, during, and after the load limit (still twenty four planes) has been reached. Figure 39 shows the performance of the system under this scenario according to the Δt metric. The high spikes again correspond to the creation of new flights. The times are still under a tenth of a second.

Figure 40 shows the results of the varied usage scenario according to the $\#S$ metric. Note that interleaving flight manipulations within flight creation does not affect the control signaling. Flight creations still consume thirty-six MMTK component signals and move/land interactions.

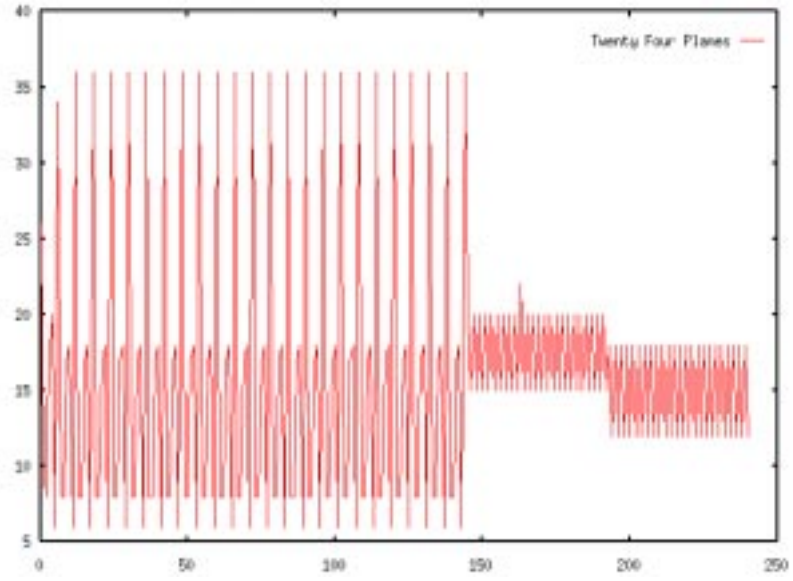

Figure 37: Interaction signal flurrying with 24 planes, up-front usage.

Figure 38: $\#S$ graph with a load of 24 planes.

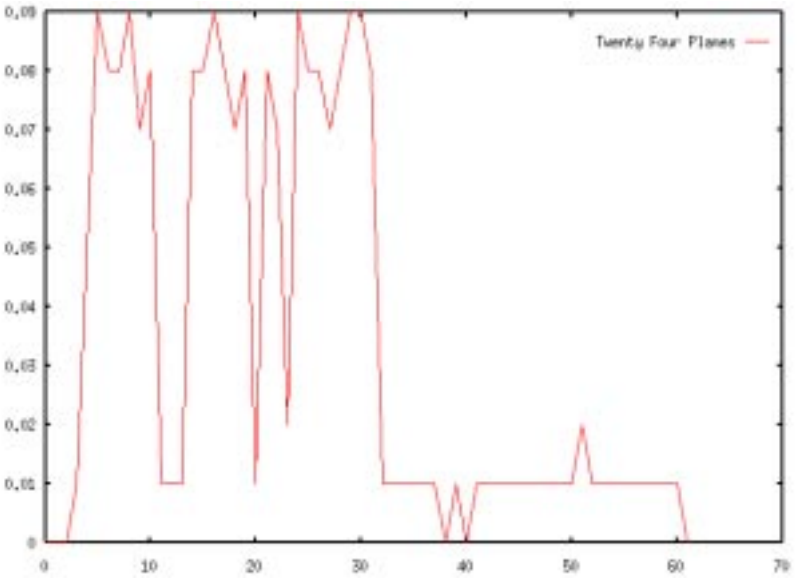

Figure 39: Δt graph with a load of 24 planes (varied usage).

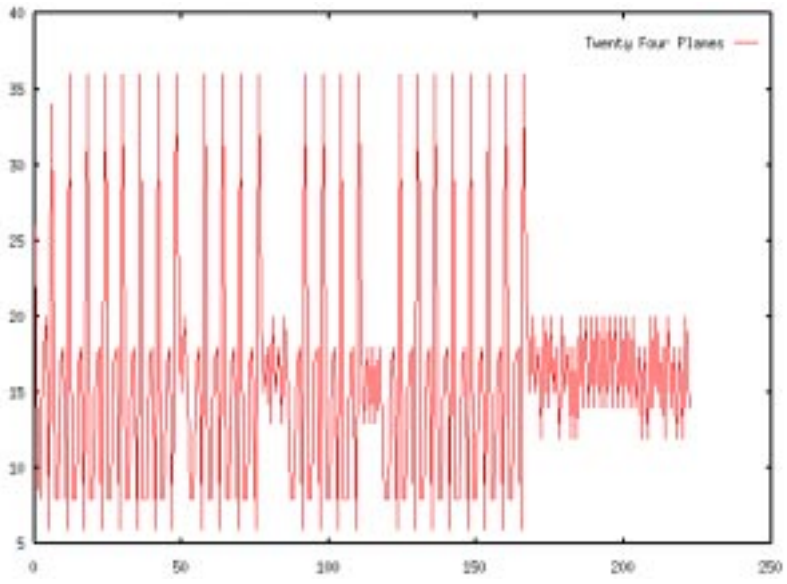

Figure 40: $\#S$ graph with a load of 24 planes (varied usage).

8.2 Multi-model Compositionality

The biggest challenge to effective model-based code generation is to make the code generated by separate models composable. MMTK was designed to support seamless integration with code generated by other models. To validate this claim, we have identified two ways in which the composition might be impeded:

1. the tree component algorithm is incorrect, or
2. the presentation linkage strategy over the target toolkit (Amulet) is incomplete.

8.2.1 Correctness of Control Component Interoperation

To argue the correctness of our control model, we submitted four pieces of evidence:

1. Formal semantics of the machine definitions (Chapter 6).
2. Two correctness properties of component inter-operation (Chapter 6).
3. Proof of a theorem relating the satisfaction of these correctness properties to a suite of configuration tests performed by a model checker (Chapter 7.

We now discuss how each of these items is validated.

We identified two correctness properties–receptiveness and freedom from divergence–whose satisfaction cannot be judged by looking at the definition of a single component. We tested the components under composition using the SMV model checker. When the model checker discovered failure cases, we modified the machine designs and reiterated. The set of Mealy machines defined in this dissertation have all passed the model checking.

Using the model checker, we were able to validate the two properties for all MMTK ordering components in configurations of depth two. We then validated the claim for configurations of arbitrary size by proving the **Composition Adequacy Theorem**.

The next item is the collection of specific Mealy machine definitions. In the examples we have applied, we have not yet discovered a design flaw. The detailed designs appear in[92].

One final point to consider. We utilized a great deal of automation in translating MM definitions to SMV input configurations. The details of this translation are provided in the tool documentation [92]. We did not formally prove that the correctness of the MM compiler.

8.2.2 Completeness of Presentation Linkage

Inherent in our definition of binding was an assumption about the complexity of linking MMTK components to presentation components. Specifically, we assumed that all presentation/task communication could be implemented by event synchronization and that the initiation of a communication could be implemented using Amulet Command[71] objects. Clearly things worked out for the examples we tried, but it is reasonable to ask if there exist situations whose complexity exceeds the facilities we have laid down. That is, is our presentation linkage strategy *complete* with respect to the target toolkit. We believe, in fact, that our approach is complete with respect to the way the Amulet toolkit handles interactor actions[71].

In general there are three obstacles to MMTK/presentation linkage:

1. Techniques for connecting UI functionality vary widely depending upon the presentation toolkit and programming language. Connecting with presentation entities often requires setting values in toolkit objects and being invoked by asynchronous call-backs.

2. The connection device should not be bound to a specific MMTK component class. A radio

button panel, for example, under some circumstances might be attached to a single MMTK input component, and in others, each individual button in the panel might be attached to a different MMTK input component.

3. Presentation entities may already be parts of *other* aggregation hierarchies. This can complicate their linkage with MMTK components.

We have not provided a toolkit-independent linkage strategy (which rules out any strong claims about the first point). However, if we limit our focus to those presentation behaviors expressible in the Amulet toolkit, we can make some precise statements about points two and three.

We believe that our choice of linkage via Amulet command objects takes care of point two. That is, by design Amulet command objects are generic over presentations and interactors. This makes them trivially "pluggable" in different contexts. In fact, in [71], Myers and Kosbie argue that by virtue of command objects and the interactor model, input in Amulet is completely separable from output in the sense of the model-view-controller (MVC) paradigm. The "late-binding" of our linkage strategy is directly related to the degree to which Myers' claims are true.

Point three is a bit more subtle. Aggregation hierarchies are used extensively in modern object-oriented UI toolkits for purposes of managing layout and visibility. This packaging can complicate the style of linkage we assume by muddying the question of who owns, creates, and destroys a part. Consider, for example, the radio button panel widgets in which the panel widget code actually creates the individual button widgets. To link each button in the panel to an MMTK component, a complex run-time walk of the panel aggregation tree would be required. This walk would find the buttons and associated with various `PresOutput` communication components and then extract the `MM_Callback_Command` objects from these components and make them parts of the interactors associated with the buttons. This is clearly difficult to express declaratively. Moreover, it implies run-time linkage. This is an unavoidable consequence of combining models with different hierarchical decompositions. On the bright side, this is the kind of operation a model-based code generator can perform.

8.3 Summary and Future Work

The model-based approach is a new paradigm for interactive system development. Like any new paradigm, it presents new challenges. Current model-based approaches have been criticized because the code they generate is not highly graphical, does not support direct manipulation, and does not allow the designer to express all of the salient detail of an interface. We have argued these deficiencies are not inherent in the model-based approach, but rather, are a consequence of applying traditional software composition techniques to a new paradigm of software composition. The proper mechanism for model-based generation is a constraint-oriented composition. MDL and MMTK demonstrate that the constraint-oriented view of interactive-system composition is both theoretically sound and practically feasible.

Though the work in this thesis is still new, the results demonstrate the potential for model-based development environments to become competitive tools in the software development circle. In addition, this research has made technical contributions on two levels. First, we identified the model-composition problem as one which could be solved using simulated concurrency and synchronization. By itself, this solution is not that novel, as researchers have been thinking about concurrency and synchronization in this domain for years. The difference is that we took this idea beyond the realm of specification and into the realm of implementation. Second, we identified the primary complexity of designing a temporal constraint system as one of correctness. This understanding led to a formulation of the problem in which we dealt with design correctness from the beginning rather than trying to prove it at the end. We did this by formulating correctness of composition properties and then adapting a tool to check these properties automatically. This freed us from having to worry about composition correctness and allowed us to concentrate on the interesting side of implementing MDL operators in Mealy machines.

8.3.1 Lessons Learned

We learned many lessons during the course of this research. It began with an investigation into the generation of code from task and presentation models. Early in the project, we applied formal methods to try and uncover the semantics of task models, presentation models, and model composition. Unfortunately, in the beginning, we applied the methods poorly. We took the approach of

picking formalism X and trying to model all of the desired task and presentation functionality in X. Chronologically, formalism X was StateCharts[44], then Petri-nets[86], and finally Mealy machines. The problem was we failed to identify acceptance criteria before plunging into the modeling phase. We spent weeks investigating Petri-nets and even began building a Petri-net simulator before realizing they did not compose well with our intended model of presentations. Had we identified clear and simple presentation composition as an acceptance criteria, we would have been able to allocate intellectual resources more wisely.

The other lesson has to do with using theoretical knowledge to simplify tedious effort. When we began designing these Mealy machines, we knew the model checker could check temporal properties of state machines, but we did not have a clear view of the composition properties. These had to be discovered by trial and error. It was clear, however, that we needed a bound on the size of testing configurations. Before proving the adequacy theorem, we thought this bound was much larger than we have shown. This meant that each configuration contained more machines and, therefore, more state, which meant that model checking a single configuration took hours rather than minutes. Had we invested more energy into the theorem early on, we could have saved a great deal of effort and compute resources.

8.3.2 Future Work

We see future work proceeding in three areas: MDL extension, integration into the MASTERMIND design environment, and generalization of the Composition Adequacy Theorem.

MDL needs an operator similar to the guarded expression operator of LOTOS. In LOTOS, designers can specify: $[cond] \rightarrow P$ which will enable P if the condition *cond* evaluates to true and disable P otherwise. Currently, we lack an MMTK operator that can handle this, but it should be a straight-forward extension to the framework. With this operator in place, we can define **Stack** and **Queue** dynamic process constructors similar to the **Set** constructor we have now.

Currently MDL and MMTK comprise stand-alone tools and run-time libraries. We want to see them integrated into the full MASTERMIND design environment so that presentation binding can be done graphically. The reason this has not been done already is a lack of resources. It should be a straight-forward integration.

In Chapter 7, we proved the Composition Adequacy Theorem, which demonstrates useful composition properties of Mealy machines under the synchrony hypothesis. The proof is not restricted to either MASTERMIND or the user-interface generation problem. Nevertheless, we would like to generalize the Composition Adequacy Theorem over other properties. The **receptiveness** and **freedom from divergence** properties take advantage of a class of CTL constraint. We would like to axiomatize the theorem so that it clearly states which other CTL composition constraints can be checked in arbitrary configurations.

Bibliography

[1] Gregory D. Abowd. *Formal Aspects of Human-Computer Interaction.* PhD thesis, University of Oxford, 6 1991.

[2] Gregory D. Abowd and Alan J. Dix. Integrating status and event phenomena in formal specifications of interactive systems. In *Proceedings of the ACM SIGSOFT'94 Symposium on the Foundations of Software Engineering*, New Orleans, Louisiana, 12 1994.

[3] Gregory D. Abowd, Hung-Ming Wang, and Andrew F. Monk. A formal techniqe for automated dialogue development. In Gary M. Olson and Sue Schuon, editors, *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods & Techniques*, pages 219–226. ACM Press, August 1995.

[4] Heather Alexander. Structuring dialogues using csp. In *Formal Methods in Human-Computer Interaction[45]*, pages 273–295. Cambridge University Press, 1990.

[5] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–248, 7 1997.

[6] J. R. Anderson. *The Architecture of Cognition.* Harvard University Press, 1983.

[7] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[8] Joanne Atlee and John Gannon. State-based model checking of event driven systems requirements. *IEEE Transactions on Software Engineering*, 19(3), January 1993.

[9] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szczur. The arch model : Seeheim revisited, 4 1991. User Interface Developer's Workshop Report.

[10] L. Bass and C. Unger, editors. *Engineering for Human Computer Interaction.* Chapman & Hall, 1996.

[11] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface.* SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, 1991.

[12] D. Benyon and P. Palanque, editors. *Critical Issues in User Interface Systems Engineering.* Springer-Verlag, Berlin, 1995.

[13] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language; design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[14] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, and J. Vanderdonckt. A model-based approach to presentation: A continuum from task analysis to prototype. In *EuroGraphics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'94)*, 1994.

[15] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, J. Vanderdonckt, and G. Zucchinetti. Key activities for a development methodology of interactive applications. In *Critical Issues in User Interface Systems Engineering [12].* Springer-Verlag, Berlin, 1995.

[16] Barry Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[17] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Network ISDN Systems*, 14(1), 1987.

[18] Thomas Browne, David Davila, Spencer Rugaber, and Kurt Stirewalt. Using declarative descriptions to model user interfaces with MASTERMIND. In Fabio Paterno and Philippe Palanque, editors, *Formal Methods in Human Computer Interaction*. Springer-Verlag, 1997.

[19] H. J. Bullinger and B. Schackel, editors. *Human Computer Interaction - INTERACT'87*. North Holland, Amsterdam, 1987.

[20] T. Bultan, J. Fischer, and R. Gerber. Compositional verification by model-checking for counter-examples. In *International Symposium on Software Testing and Analysis*, pages 224–238, 1996.

[21] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the* 5^{th} *International Symposium on Logic in Computer Science*, June 1990.

[22] S. K. Card, T. P. Moran, and A. Newell. The keystroke-level model for user performance. *Communications of the ACM*, 23:394–410, 1980.

[23] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human Computer Interaction*. Lawrence Erlbaum, 1983.

[24] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. *Computer Graphics*, 19(3), 1985.

[25] P. Castells, P. Szekely, and E. Salcher. Declarative models of presentation. In *IUI'97: International Conference on Intelligent User Interfaces*, pages 137–144, 1997.

[26] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Workshop on Logics of Programs*, pages 52–71, 1981.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[28] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, 1989.

[29] J. Coutaz. PAC, an object-oriented model for dialog design. In *Human Computer Interaction - INTERACT'87 [19]*, pages 431–436. North Holland, Amsterdam, 1987.

[30] D. Diaper, editor. *Task Analysis for Human Computer Interaction*. Ellis Horwood, 1989.

[31] D. Diaper. Analysing focused interview data with task analysis for knowledge description (takd). In *IFIP INTERACT'90: Human-Computer Interaction*, 1990.

[32] Alan Dix. Non-determinism as a paradigm for understanding the user interface. In *Formal Methods in Human-Computer Interaction[45]*, pages 97–127. Cambridge University Press, 1990.

[33] W. Keith Edwards, Scott Hudson, Roy Rodenstein, Thomas Rodriguez, and Ian Smith. Systematic output modification in a 2d user interface toolkit. In *UIST'97: ACM Symposium on User Interface Software Technology*, 1997.

[34] Brad A. Myers et. al. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, 11 1990.

[35] A. Fantechi, S. Gnesi, and G. Mazzarini. How expressive are LOTOS behaviour expressions? In *Formal Description Techniques, III. Proceedings of IFIP TC/WG 6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'91)*, pages 17–32, 1991.

[36] J. Fischer and R. Gerber. Compositional model-checking of ada tasking programs. In *IEEE Ninth Annual Conference on Computer Assurance (COMPASS'94)*, 1994.

[37] J. Foley, W. Kim, S. Kovacevic, and K. Murray. Uide - an intelligent user interface design environment. In *[96]*, pages 339–384. Addison-Wesley, Reading, Massachusetts, 1991.

[38] J. D. Foley. The structure of interactive command languages. In R. A. Guedj et. al., editor, *Methodology of Interaction*, pages 227–234. North Holland Publishing Company, Amsterdam, The Netherlands, 1980.

[39] James D. Foley, Victor L. Wallace, and Peggy Chan. The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications*, 4, 11 1984.

[40] Formal Systems (Europe) Ltd., Oxford, England. *Failures Divergence Refinement: User Manual and Tutorial. 1.2β*, 1992.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[42] S. Gnesi and S. Larosa. A sound and complete axiom system for the logic actl. In *Fifth Italian Conference on Theoretical Computer Science*, 1996.

[43] James Gosling and Frank Yellin. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets*. Addison-Wesley, 1996.

[44] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[45] M. Harrison and H. Thimbleby, editors. *Formal Methods in Human-Computer Interaction*. Cambridge University Press, 1990.

[46] H. R. Hartson, A. C. Siochi, and D. Hix. The uan: a user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, 7 1990.

[47] H. Rex Hartson and D. Hix, editors. *Advances in Human-Computer Interaction*, volume 2. Ablex, Norwood, 1988.

[48] H. Rex Hartson and Deborah Hix. Human computer interface development : Concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, 3 1989.

[49] H. Rex Hartson and Kevin A. Mayo. A framework for precise, reusable, task abstractions. In *Eurographics Workshop: Design, Specification and Verification of Interactive Systems*, pages 147–164, 1994.

[50] C. A. R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, 1(2):151–160, 6 1975.

[51] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall International, Englewood Cliffs, New Jersey, 1985.

[52] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company, 1979.

[53] Daniel Jackson. Abstract model checking of infinite specifications. In *Formal Methods Europe*, Barcelona, 10 1994.

[54] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *International Symposium on Software Testing and Analysis (ISSTA'96)*, 1996.

[55] Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating user interfaces from data models and dialogue net specifications. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, 1993.

[56] P. Johnson. *Human Computer Interaction: Psychology, Task Analysis, and Software Engineering*. McGraw-Hill, 1989.

[57] P. Johnson. *Human-Computer Interaction, Psychology, Task analysis, and Software Engineering*. McGraw-Hill, London, 1991.

[58] P. Johnson, S. Wilson, P. Markopoulos, and J. Pycock. Adept - advanced design environment for prototyping with task models. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, 1993.

[59] S. C. Johnson. Yacc–yet another compiler compiler. Technical Report C.S. Technical Report # 32, Bell Telephone Laboratories, Murray Hill, Ne Jersey, 1975.

[60] D. Kieras and G. Polson. An approach to the formal analysis of user complexity. *International Journal of Man Machine Studies*, 22, 1985.

[61] David E. Kieras, Scott D. Wood, Kasem Abotel, and Anthony Hornof. Glean: A computer-based tool for rapid goms model usability evaluation of user interface designs. In *ACM Symposium on User Interface Software and Technology*, 1995.

[62] G. E. Krasner and S. T. Pope. A cookbook for using the model view controller user interface paradigm in smalltalk. *Journal of Object Oriented Programming*, 1(3), 8 1988.

[63] P. Markopoulos. On the expression of interaction properties within an interactor model. In *Design Specification and Verification of Interactive Systems (DSV-IS'95)*, 1995.

[64] P. Markopoulos, J. Rowson, and P. Johnson. On the composition of interactor specifications. In *Formal Aspects of the Human Computer Interface, BCS-FACS Workshop*, 1996.

[65] P. Markopoulos, S. Wilson, and P. Johnson. Representation and use of task knowledge in a user interface design environment. *IEE Proceedings–Computers and Digital Techniques*, 141(2), 1994.

[66] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[67] M. Mezzanotte and F. Paterno'. Verification of properties of human-computer dialogues with an infinite number of states. In *Formal Aspects of the Human Computer Interface BCS-FACS Workshop*, pages 29–39, 1996.

[68] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[69] Thomas P. Moran. The command language grammar: a representation for the user of an interactive computer system. *International Journal of Man–Machine Studies*, 15:3–50, 1981.

[70] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, 1990.

[71] Brad A. Myers and David S. Kosbie. Reusable hierarchical command objects. In *CHI'96: Human Factors in Computing Systems*, 1996.

[72] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 6 1997.

[73] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *SIGCHI'92: Human Factors in Computing Systems*, 5 1992.

[74] R. Neches, J. Foley, P. Szekely, P. Sukaviriya, P. Luo, S. Kovacevic, and S. Hudson. Knowledgeable development environments using shared design models. In *Intelligent Interfaces Workshop*, pages 63–70, January 1993.

[75] R. "De Nicola", A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–78, 2 1993.

[76] Dan R. Olsen. Propositional production systems for dialogue description. In *Empowering People–CHI'90 Conference Proceedings*, pages 57–63, 1990.

[77] P. Palanque, R. Bastide, and V. Sengès. Validating interactive system design through the verification of formal task and system models. In *Working Conference on Engineering for Human Computer Interaction*, 1995.

[78] P. Palanque, F. Paterno', R. Bastide, and M. Mezzanotte. Towards an integrated proposal for interactive systems design based on tlim and ico. In *Third Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'96)*, 1996.

[79] Fabio Paterno'. Detection of properties of user interfaces. In *5th International Conference on Software Engineering and Knowledge Engineering (SEKE'93)*, 1993.

[80] Fabio Paterno'. A theory of user-interaction objects. *Journal of Visual Languages and Computing*, 5:227–249, 1994.

[81] Fabio Paterno', Maria Sabrina Sciacchitano, and Jonas Löwgren. A user interface evaluation mapping physical user actions to task-driven formal specifications. In *Second Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'95)*, 1995.

[82] G. Pfaff and P. ten Hagen, editors. *User Interface Management Systems: proceedings of the Workshop on User Interface Management Systems, held in Seeheim FRG.* Springer–Verlag, Berlin, 1985.

[83] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[84] A. Puerta. The mecano project: Comprehensive and integrated support for model-based user interface development. In *[99]*, pages 19–36. Namur University Press, 1996.

[85] A. R. Puerta, H. Eriksson, J. H. Gennari, and M. A. Mussen. Beyond data models for automated user interface generation. In *People and Computers IX HCI'94 Conference Proceedings*, pages 353–366. Cambridge University Press, Cambridge, 1994.

[86] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs on Theoretical Computer Science*. Springer Verlag, 1982.

[87] Egbert Schlungbaum. Model-based user interface tools: Current state of declarative models. Technical Report GIT-GVU-96-30, Graphics, Visualization and Usability Center (GVU), Georgia Institute of Technology, 1996.

[88] Egbert Schlungbaum and Thomas Elwert. Dialogue graphs: A formal and visual specification technique for dialogue modelling. In *Formal Aspects of the Human Computer Interface, BCS-FACS Workshop*, 1996.

[89] F. B. Schneider. *On Concurrent Programming*. Springer, 1997.

[90] J. L. Sibert, W. D. Hurley, and T. W. Bleser. Design and implementation of an object-oriented user interface management system. In *Advances in Human-Computer Interaction: Volume II [47]*. Ablex, Norwood, 1988.

[91] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1992.

[92] Kurt Stirewalt. The design and implementation of the MASTERMIND toolkit (mmtk). Technical Report GIT-GVU-97-23, Graphics, Visualization and Usability Center (GVU), Georgia Institute of Technology, 1997.

[93] Bernard Suffrin and Jifeng He. Specification, analysis and refinement of interactive processes. In *[45]*. Cambridge University Press, 1990.

[94] Piyawadee "Noi" Sukaviriya and James D. Foley. Coupling a UI framework with automatic Generation of context-sensitive animated help. In *ACM Symposium on User Interface Software and Technology*, 1990.

[95] Piyawadee "Noi" Sukaviriya, James D. Foley, and Todd Griffith. A second generation user interface design environment: The model and runtime architecture. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, 1993.

[96] J. Sullivan and S. Tyler, editors. *Architectures for Intelligent User Interfaces*. Addison-Wesley, Reading, MA, 1991.

[97] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher. Declarative interface models for user interface construction tools : The mastermind approach. In L. Bass and C. Unger, editors, *Engineering for Human-Computer Interaction [10]*. Chapman & Hall, 1996.

[98] Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, pages 383–390. Addison Wesley, April 1993.

[99] J. M. Vanderdonckt, editor. *Computer Aided Design of User Interfaces*. Namur University Press, Namur, 1996.

[100] J. M. Vanderdonckt and F. Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Bridges Between Worlds: Human Factors in Computing Systems: INTERCHI'93*, 1993.

[101] Visual Edge Software Ltd., Cupertino, CA. *Extending and Customizing UIMX*, 1993.

[102] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, and P. Markopoulos. Beyond hacking: A model based approach to user interface design. In J. L. Alty, D. Diaper, and S. Guest, editors, *People and Computers VIII, Proceedings of the HCI '93 Conference*, pages 217–231, 9 1993.

[103] Jeannette M. Wing and Mandana Vaziri-Farahani. Model checking software systems: A case study. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

[104] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):371–411, 1993.

[105] Pamela Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2), 2 1984.

[106] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Computer*, 9 1989.

Vita

Richard Erick Kurt Stirewalt was born September 3, 1968 in Atlanta, Georgia. He attended elementary school at Rock Chapel Elementary and secondary school at Lithonia High School. As a senior in high school, he took joint-enrollment courses at the Georgia Institute of Technology and remained at Georgia Tech after high-school graduation to study computer science. In 1989, he received the Bachelor of Science degree with honors in Information and Computer Science. He decided to stay at Georgia Tech to pursue a Ph.D. in computer science. Stirewalt's area of interest in 1989 was high-performance computing. He did work in program analysis and compilers and published a number of papers in the area. This interest led to a collaboration with researchers at the MITRE Corporation, and this collaboration opened the door to his current interests: formal methods for software analysis, design, and generation.