

.An Integrated Event-Driven Framework Supporting MDD

Lizhang Qin, Hongming Liu, Chris Jones, Xiaoping Jia
DePaul University
School of Computer Science
Chicago, IL
{jordan, lqin, cjones, xjia}@cs.depaul.edu

Abstract

Model-Driven Development (MDD) defines an approach to developing software systems that doesn't rely on programming. MDD emphasizes the use of models for defining software specifications and uses automatic tools to generate the implementations for specific platforms. In the MDD, different views and models are specified for different parts of the system. ZOOM, which is a formal notation we propose to enhance existing modeling notations, defines the structural, behavioral and UI models as the three most critical components for a software model. In this paper, we present a pre-defined event model, processed through an event-driven framework, which integrates these views, and provides the run-time execution model for ZOOM.

1 Introduction

Model-Driven Development (MDD) is a methodology where all vital parts and aspects of the system under consideration are described with formal models. Instead of requiring developers to detail a system's implementation using a programming language, MDD lets them use models to describe the system's functionality and overall architecture [1,2]. MDD enables reuse at the business domain level, increases quality as models are successively improved, reduces costs by supporting automated software development processes, and increases software longevity. Because of the potential to change the way we develop applications, many researchers have been working on methodologies to support MDD [1,2].

ZOOM (Z-based Object-Oriented Modeling) is a formal foundation we propose to support MDD [16]. Unlike UML-2, which separates a system into several loosely related views, ZOOM provides a mechanism to merge those views together along with their formal semantics. This approach helps overcome some of the problems with UML-2, specifically, problems of intra- and inter-model consistency. While UML-2 provides many different views of a software system, it does not inherently enforce the consistency between these views. Such tasks are left to the modeler or the modeling software. In contrast, a ZOOM model has three integrated parts: structural models, behavior models and user interface (UI) models. Ultimately, an event model, processed through an event-driven framework, integrates these views.

This event-driven framework satisfies the consistency requirements between different views of the system, and works as a mechanism to integrate the structural, behavioral and UI models together. The event-driven framework also defines the run-time execution model for ZOOM, which can easily be implemented on specific platforms. In this paper, we present the overall design of the event-driven framework and how it works to integrate different views for ZOOM.

2 An Approach to MDD

Model-driven development (MDD) is the notion that we can construct a model that can then be transformed into a real system. MDD moves the development focus from third generation programming language code to models, specifically models expressed in version 2.0 of the Unified Modeling Language (UML-2) and its profile definitions. Because of MDD's potential to dramatically change the way we develop applications, companies are already working to deliver supporting technologies [3]. However, there is no universally accepted definition of the requirements for an MDD infrastructure and many of the requirements for MDD support are unclear or even unspecified. To overcome this lack of standardization, we argue that an MDD infrastructure requires precise, analyzable, transformable and executable models. MDD's defining characteristic is the focus on building models rather than programs [4]. However, the concept of a model requires clarification. Mellor [5], suggests that a model is "a coherent set of formal elements describing a system built for a purpose that is amenable to a particular form of analysis." Seidewitz [6] defines a model as "a set of statements about some system under study." In our research, we define a model to be "a consistent and complete set of formal elements describing a system that is amenable to analysis." Models depict one or more aspects of a system's structure and behavior at some level of abstraction. At the lowest level, models use implementation technology concepts.

To overcome the obstacles associated with the existing modeling notations and to realize our vision of MDD, we propose to enhance the existing UML-2 models and metamodel by including support for formal syntax and semantics. This new notation is called ZOOM.

ZOOM stands for Z-based Object-Oriented Modeling. It is based on the formal specification notation Z [8, 12, 13], which is in turn based upon set theory and mathematical logic. Although widely used to specify software systems, one of Z's deficiencies is that its specification is limited to mathematical logic and does not provide useful mechanisms to support OO modeling such as classes or inheritance. ZOOM extends Z to support these object-oriented concepts. The ZOOM notation has three separate specification languages to formally describe these three parts: ZOOM-M for the structural parts, ZOOM-FSM for the behavioral parts and ZOOM User Interface Description Language (ZOOM-UIDL) for the user interface parts. Significant work has already been done on the ZOOM notations resulting in a stable grammar for the ZOOM specification languages along with their formal syntax and semantics.

3 ZOOM Models

3.1 Overview

Figure 1 shows the overall structure of the ZOOM models for a software system. The functional requirements derive the structural and behavioral models and are the only requirements for a modeler to consider. Extra-Function Requirement (XFRs) and Environmental Requirements are two other dimensions that must be adequately addressed before an application can be successfully realized from a model. The XFRs stipulate the minimum service levels that an application must provide while exercising its functional requirements. The environmental requirements are capture the requirements of the target runtime environment. User interface design is derived from the functional and user interface requirements. Those UI requirements are incorporated in the profiles for the knowledge-based UI generation. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to bind the structural, behavioral, and UI models together. The integrated ZOOM model will be processed by the

Knowledge-based Model Compilation Tools resulting in different implementations of the software system based on the specific platform and / or knowledge base.

The separation of a system into its structural, behavioral and UI models is an application of the well-known paradigm in software engineering of “Separation of Concerns”, which formally separates the system based on special purpose concerns [14]. This separation allows each aspect of the system to be specified separately, making each aspect easier to write, understand, and change with less affect on the other aspects. For example, under this separation, modelers can modify the user interface based on profiles and user preferences without changing the structural or behavioral models. The other advantage of this separation is that we use different specification languages to describe different aspects of the system. The structural, behavioral, and UI models describe different parts of a system, each of which has its own distinguishing characteristics.

Each ZOOM model has dual representations including a textual specification and a graphical view. The graphical representations are consistent with common UML-2 diagrams, such as class, use case and state diagrams, but also include formal semantics and syntax. This enables the use of popular tools to design and maintain ZOOM models. Modelers will appreciate the ability to use available tools to construct their models and to then add formal specifications to those models[15].

3.2 Structural Model

The structural model defines the functional part of the system using object-oriented concepts. These models provide not only the class relations and hierarchies, but also a precise specification of the functionality of each entity.

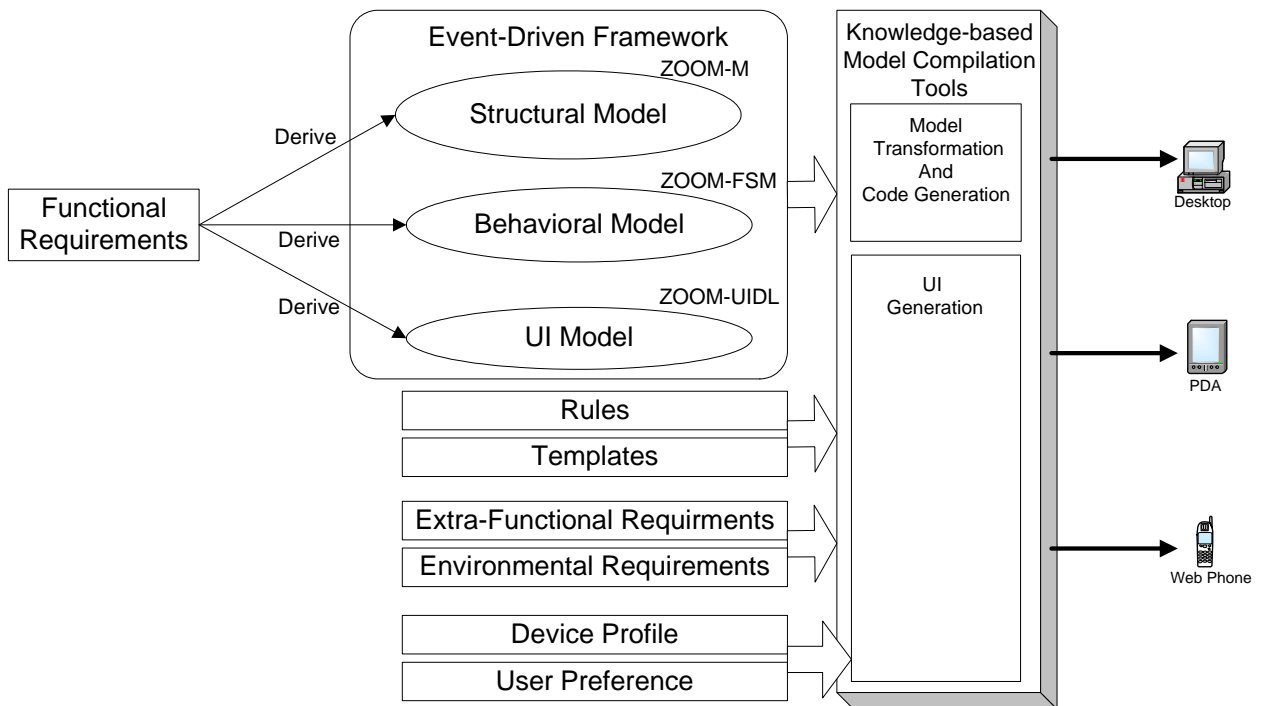


Figure 1. ZOOM Models

The textual specifications of the structural models are specified by the ZOOM-M language. ZOOM-M is based on Z and incorporates object-oriented concepts. ZOOM-M is fully developed and implemented. It provides the ability to formally specify application functionality, while using a Java/C++ like syntax to make it easy to adopt by practitioners.

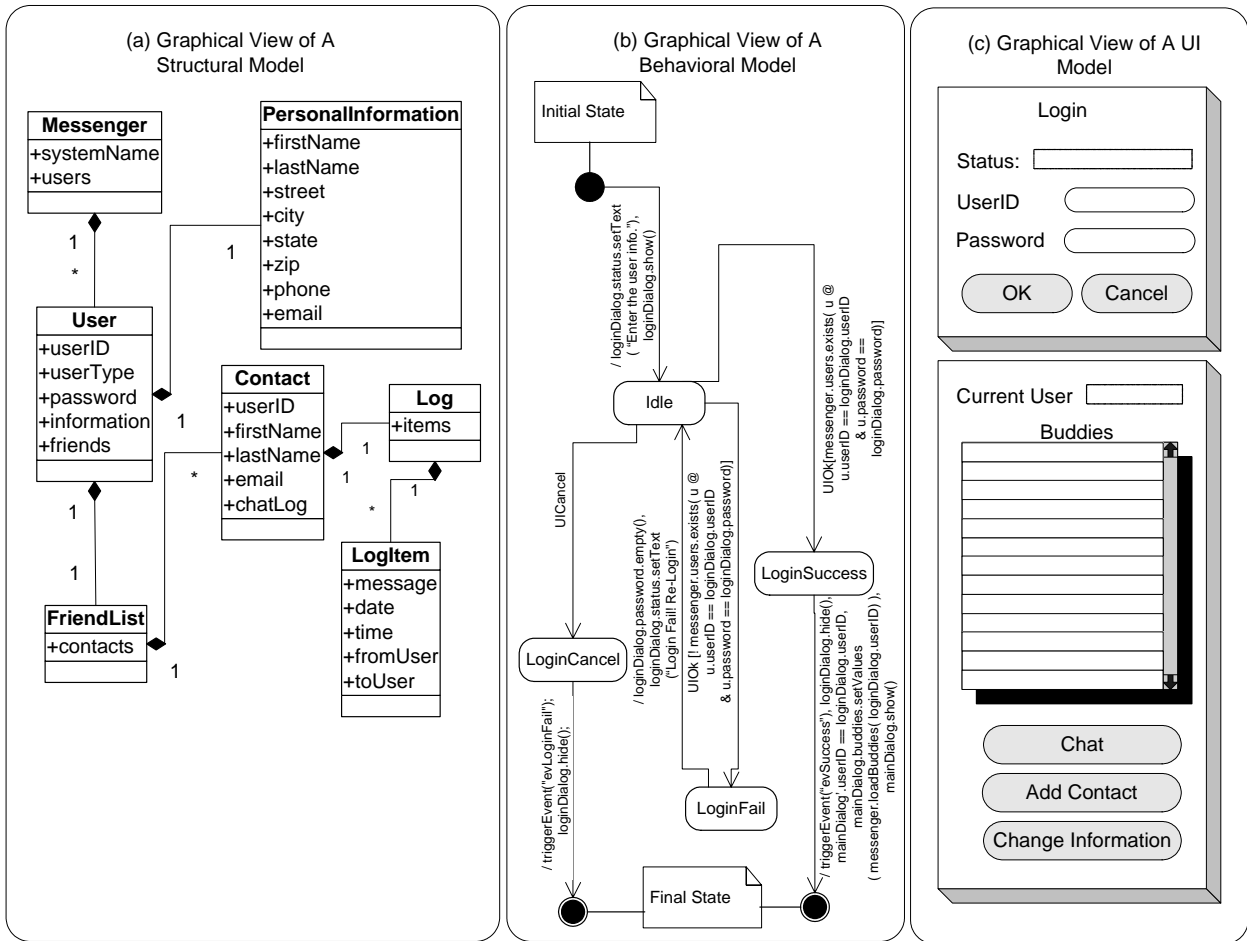


Figure 2. A ZOOM Model Example

The graphical views of the ZOOM structural models are consistent with UML-2 package/class diagrams along with the formal specifications including class invariants and operation pre- and postconditions. These constraints are a key element of Bertrand Meyer's principle of design by contract [17, 18], which provides elements of the ZOOM foundation. The use of such constraints can improve the quality of software documentation as well as its precision [19].

Structural model design in ZOOM separates the behavioral and user interface aspects from the core static functionality. Structural models formally define the functionality of the system without describing the user interface or how the structural components work together. In this Instant-Messenger example, User, FriendList, Contact, Log, PersonalInformation and Messenger are the basic entities in the system. As with typical object-oriented design approaches, these entities should be specified as autonomous structs with proper attributes and operations. Each struct and each operation is precisely described by a formal specification. Part (a) in Figure 2 shows the graphical view of a simplified structural model for the above scenario. The

specifications in Figure 3 are parts of the textual specification for User and FriendList. They show how modelers can use ZOOM-M to formally specify the functionality.

```
1: typedef FriendList = List[Contact];
2:
3: public struct User {
4:   invariant { userID != null }
5:
6:   public:protected String userID;
7:   public      UserType userType;
8:   protected String password;
9:   public      PersonalInformation information;
10:  public      FriendList friends;
11:  public      boolean loggedIn = false;
12:
13:  public User(String userID)
14:    requires { userID != null }
15:    ensures  { this.userID == userID };
16:
17:  public void login(String maskedPassword)
18:    ensures {
19:      this.loggedIn == ( encrpt(this.password) == maskedPassword )
20:    };
21:
22:  public boolean hasFriend(String friendID) const
23:    ensures {
24:      if (friendID == null) then
25:        hasFriend! == false
26:      else
27:        hasFriend! == friends.collect( x @ x.userID ).contains(friendID)
28:      endif
29:    };
30: }
```

Figure 3. A Structural Model Example

3.3 Behavioral Model

The behavioral model is the central communication mechanism that links the structural models with the UI models. It uses a formalized state diagram to specify the dynamic aspects of a system. The textual specifications are specified by ZOOM-FSM. Graphically, the representation of the behavioral model is consistent with a UML-2 state chart. The key characteristics of ZOOM-FSM include a rich syntactical grammar with formal semantics, an intrinsic mechanism to use the structural models specific by ZOOM-M, and a strong compatibility with the UML-2 state chart. ZOOM-FSM shares the same type and expression system as ZOOM-M, with a grammar specifically designed to model the behavioral aspects in a system. Compared with the MDA approach, which specifies the dynamic aspects of a system using UML-2 state charts combined with OCL, ZOOM-FSM provides complete and easy to use representation capabilities. The ZOOM behavioral model keeps the key functions of UML statechart[20]. Compared with Executable UML (xUML) [21], which is an MDA compliant methodology that aims to make a limited set of UML state chart models executable, the ZOOM behavioral model can specify more general and complex system interactions.

For the Instant-Messenger example, the behavioral model should include several finite state machines for different user behaviors. For instance, the main FSM will stay in the Idle state after login, waiting for the user's command, and change states to specific FSMs based on the user's input. Some essential FSMs might be Main, Login, AddContact, Chatting and

ChangeInformation. Part (b) in Figure 2 shows part of a simplified behavioral model for the login process. The specifications in Figure 4 are a simplified version of the textual specification for Login FSM. They show how modelers can use ZOOM-FSM to formally specify the behavioral aspects.

3.4 User-Interface Model

Unlike UML-2 and MDA notations, ZOOM separates UI models from other aspects. This comes from the intrinsic characteristics of the UI aspect being separated from the structural and behavioral aspects.

```
1: fsm Login() {
2:   state Idle;
3:   state LoginSuccess;
4:   state LoginFail;
5:   state LoginCancel;
6:
7:   transition initial to Idle : {
8:     loginDialog.status.setText("Enter the user info.");
9:     loginDialog.show();
10:  };
11:
12:  transition Idle to LoginSuccess : UIOk [
13:    messenger.users.exists( u @ u.userID == loginDialog.userID
14:      & u.password == loginDialog.password)
15:  ];
16:
17:  transition Idle to LoginFail : UIOk [
18:    ! messenger.users.exists( u @ u.userID == loginDialog.userID
19:      & u.password == loginDialog.password)
20:  ];
21:
22:  transition Idle to LoginCancel : UICancel;
23:
24:  transition LoginSuccess to finalState : {
25:    triggerEvent("evSuccess");
26:    loginDialog.hide();
27:    mainDialog.userID == loginDialog.userID;
28:    mainDialog.buddies.setValues(
29:      messenger.loadBuddies( loginDialog.userID ) );
30:    mainDialog.show();
31:  };
32:
33:  transition LoginFail to Idle : {
34:    loginDialog.password.empty();
35:    loginDialog.status.setText("Login Fail! Re-Login");
36:  };
37:
38:  transition LoginCancel to finalState : {
39:    triggerEvent("evLoginFail");
40:    loginDialog.hide();
41:  };
42: }
```

Figure 4. A Behavioral Model Example

The UI typically has a tight coupling to a specific platform while other aspects are typically platform-independent[22]. ZOOM uses a specific UI modeling notation to achieve this separation of concerns rather than mixing the UI design as part of programming. One advantage of this separation is that it is easy to change a UI view by only changing the UI models, thus leaving other aspects the same. This is especially important while developing multi-platform

software systems. Another advantage is that it is convenient for a modeler to model the software system from the bottom-up. We use ZOOM-UIDL to specify UI models in a formal way. Syntactically, ZOOM-UIDL is a hierarchical description framework containing a set of predefined UIDL schemas. The graphical view of the UI model provides a visual representation and is easy for designers to learn.

UI models are important components when modeling a software system with ZOOM. In the Instant-Messenger example, all the visible parts belong to UI models. They may include Login, Main, AddContact, Chatting and ChangePersonalInformation dialogs. Part (c) in Figure 2 shows part of a simplified UI model in its graphical view, including the Login and Main dialogs. While the structural and behavioral models satisfy the functional requirement for an Instant-Messenger system, such UI models design the logical view for user interface, and the same model can be implemented in different platforms, such as a desktop computer, a mobile device or even an embedded system. The specifications in Figure 5 are parts of the textual specification for Login dialog. They show how modelers can use ZOOM-UIDL to formally specify the user interfaces.

```
1: <Window id="String" name="LoginDialog" show="true">
2:   <Panel>
3:     <Label text="Status"/>
4:     <TextBox name="status" type="text" editable="false" columns="2"/>
5:     <Label text="UserID"/>
6:     <TextBox name="userID" type="text" editable="true" columns="2"/>
7:     <Label text="Password"/>
8:     <TextBox name="password" type="password" editable="true" columns="2"/>
9:
10:    <Button name="okButton" whenclick="UIOk" text="OK"/>
11:    <Button name="cancelButton" whenclick="UICancel" text="Cancel"/>
12:  </Panel>
13: </Window>
```

Figure 5. A UI Model Example

4 An Event-Driven Framework for Model Integration

The pre-defined event model, which is processed by an event-driven framework, coordinates the activities between the structural, behavioral and UI models. Events are used in the behavioral model to specify the dynamic nature of the system. Upon receiving specific types of events, the behavioral model's finite state machine (FSM) will perform pre-defined operations by triggering changes in the various models. In the rest of this section, we define the event-driven framework in detail with an example of the Login scenario of the Instant-Messenger we described above, and explain how the event-driven framework works as a run-time mechanism to support the execution of integrated ZOOM models.

The example scenario is shown in Figure 6. First, the initial Login dialog appears, waiting for the user to login as shown in Figure 6(a). When the user provides valid credentials, the Main dialog is shown as Figure 6(c), with the current user id and buddy list set. If the user fails to provide valid user account information, the system will ask the user to re-login as shown in Figure 6(b) with the preservation of last entered user id. If the login process succeeds, an *evSuccess* event is generated. If the login process is cancelled by the user, an *evLoginFail* event is generated. These event generations are shown as line 25 and line 39 in Figure 4.

4.1 The Components of Event-Driven Framework

Figure 7 shows the structure of the event-driven framework in ZOOM. The shaded parts indicate the run-time components of the framework while the unshaded elements indicate the design-time components. Both of these components are discussed in greater detail below.

The UI models are specified by ZOOM-UIDL. Each UI model represents a logical view in a tree form and is used at run-time to instantiate UI model instances. Each node of the logic tree is an instance of a UIDL schema, which in turn defines that at run-time each UI model instance contains a tree structure of User Interface Structural Model instances.

At run-time, structural model instances are instantiated from the structural models, which contain instances of both User Interface Structural Models and Requirement Structural Models. Similarly, behavioral model instances and UI instances are instantiated from behavioral models and UI models respectively.

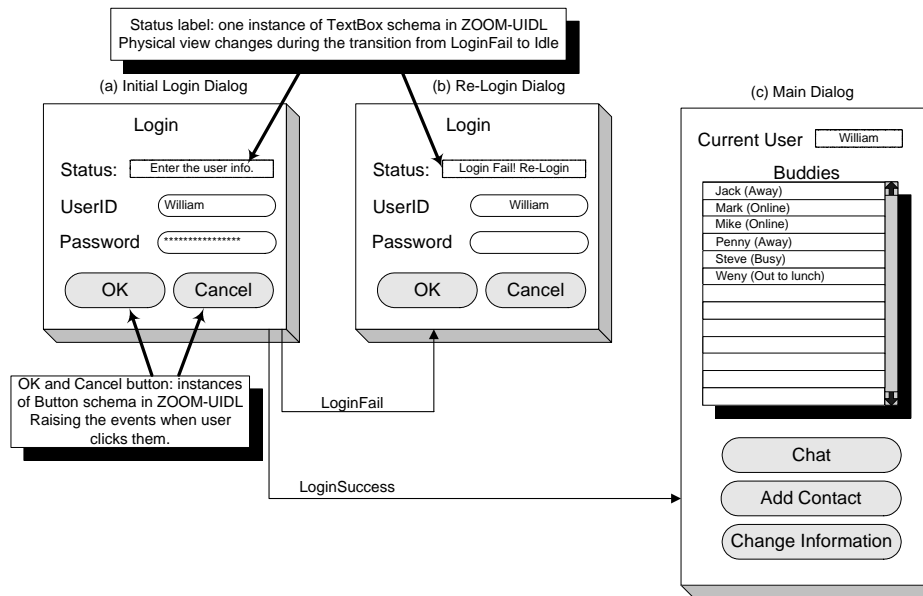


Figure 6. The Login Scenario for Instant-Messenger Example

Each UI instance is the logical representation of a physical view in a tree structure. Each tree node has a corresponding User Interface Structural Model instance. Each User Interface Structural Model instance reflects the status of corresponding physical component.

When the user triggers UI events on UI instances, these events are transmitted to behavioral model instances, which are a set of FSMs. The FSMs perform state transitions based on these events. Simultaneously, FSMs may trigger more events and send them back to behavioral model instances.

Behavioral model instances interact with structural model instances. Since user input is reflected by User Interface Structural Model instances, behavioral model instances can get those inputs by accessing those instances. Conversely, since behavioral model instances can manipulate User Interface Structural Model instances, the state changes will be reflected by the UI physical views automatically.

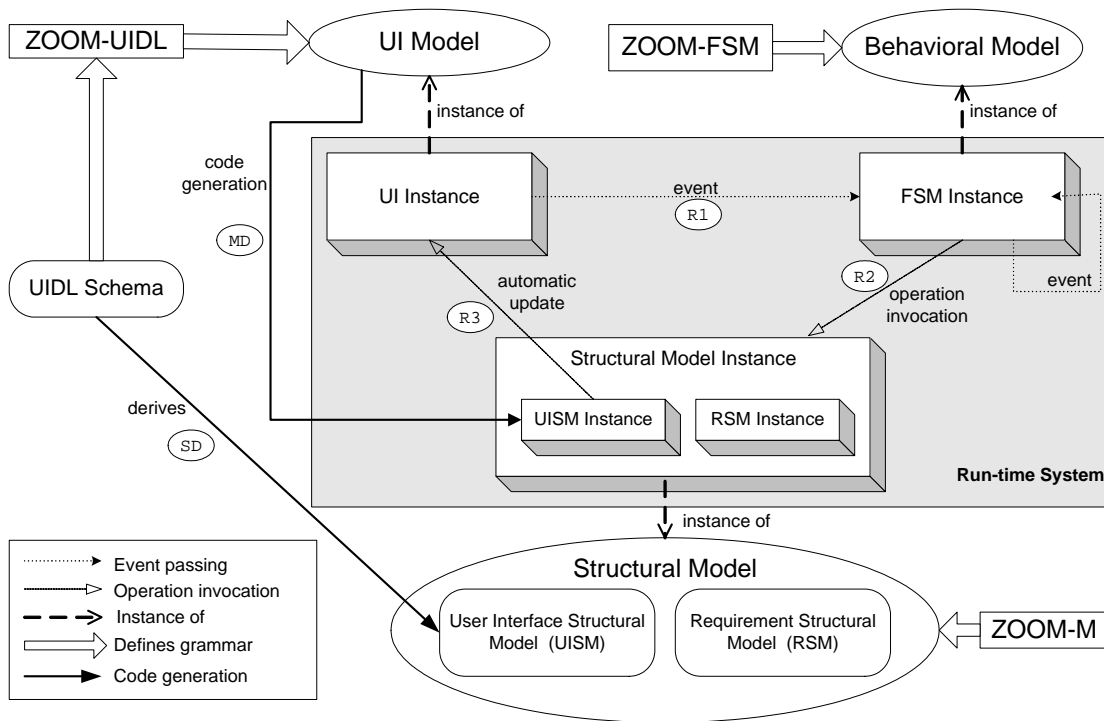


Figure 7. The Integration for ZOOM Models

4.2 The Design Time Code Generation

Before the specified model can be executed, two static steps for code generation are conducted at the design time. These two steps are specified as *SD* and *MD* in the Figure 7.

SD stands for the code generation for the schema definition and *MD* stands for the code generation for the model definition.

Step SD: The code generation for UI schemas

As described in section 3.4, each UIDL schema has a corresponding pre-defined struct in the structural models. This process is completed when we define the event-driven framework, and is only conducted once. These generated structural models can not be modified by the modelers, and unless the UIDL schemas change, they remain fixed. To demonstrate the result of this step, we use the *Button* schema as an example. Figure 8(a) shows a simplified specification for the *Button* schema, Figure 8(b) shows the corresponding generated *UIButton* struct specifications.

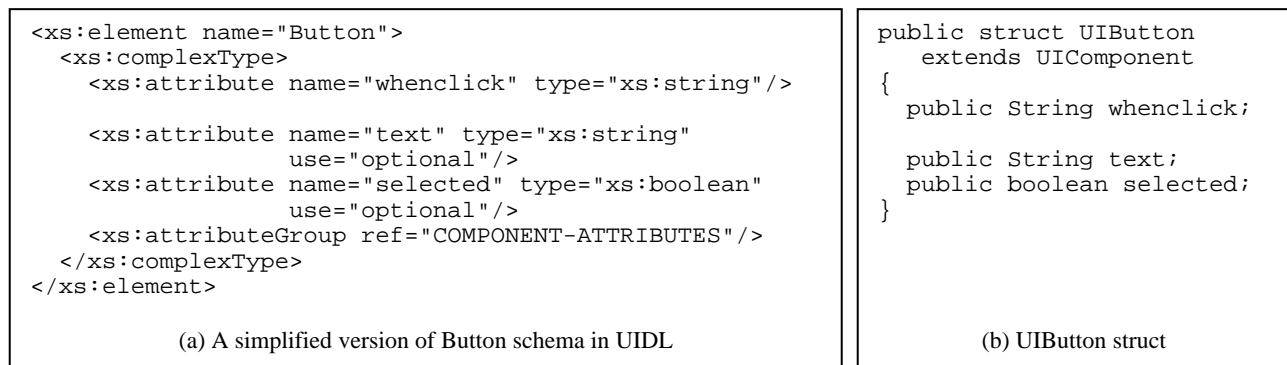


Figure 8. Button schema and UIButton struct

Step MD: The code generat

The other design time code generation is for UI models. Unlike Step SD, which is only executed once, Step MD is executed multiple times when a UI model is put into the event-driven framework. For each UI scenario defined by modelers as the UI model, an implicit struct is automatically derived in the structural models, and an instance will be automatically instantiated during the run-time which can be used in all FSMs without declaration.

For example, for the *LoginDialog* model as specified in Figure 5, a corresponding struct type *TLoginDialog* is implicitly generated as part of the structural models. Figure 9 shows a simplified version of *TLoginDialog* struct based on the model defined in Figure 5. Since the *LoginDialog* UI model includes two buttons, each of which is a reference to the *Button* schema in UIDL, the automatically generated *TLoginDialog* struct has two variables of type *UIButton* as shown above.

```
public struct TLoginDialog {
    public UITextBox status;
    public UITextBox userID;
    public UITextBox password;
    public UIButton okButton;
    public UIButton cancelButton;
}
```

Figure 9. Automatically generated struct *LoginDialog*

A system variable called *loginDialog* with type *TLoginDialog* will be instantiated automatically as follows:

```
public TLoginDialog loginDialog;
```

This *loginDialog* variable is a global variable and can be used in all FSMs.

4.3 The Execution Model

The execution model for the Event-Driven Framework includes three steps. These steps are shown as the numbered links starting with 'R' in Figure 7, and are only applied during execution time. They form an execution cycle and drive the system. More detailed discussions for those steps are described below. In the rest of this section we focus on the login fail scenario as the example.

Step R1: Interaction from the UI instances to the FSM instances

Step R1 carries out the interaction from the UI instances to the behavioral model instances. This interaction is conducted using events. There are several different levels of events defined in the system. Hardware events have the lowest level and have a tight coupling to the specific run-time platform. When the user interacts with the user interface, hardware events are generated. These hardware events are turned into pre-defined UI events based on the UI component type. This process is done by a platform specific virtual machine. The type of UI events that can be triggered for each UI construct is defined as part of the UIDL schemas. Ultimately, UI events are turned into FSM events and feed into the behavioral model instances. The conversion from UI events to FSM events are defined as part of UI models.

For example, the initial Login dialog is shown to the user as in Figure 6(a). After the user enters the login information and clicks the *OK* button using the mouse, a mouse-click event is triggered. This process is handled by the operating system. The mouse-click event is a low-level event, and is turned into the *ButtonClick* event, which is defined as one UI event for the *Button* schema in ZOOM-UIDL. The *Button* schema defines a set of supporting events such as

ButtonClick. Finally, this *ButtonClick* UI event will be turned into *UIOk* event by the following definition and sent to the FSMs. This is part of the *LoginDialog* specification in Figure 5:

```
<Button name="okButton" whenclick="UIOk" text="OK" />
```

Step R2: Interaction starting from the behavioral model instances

Step R2 carries out the interactions starting from the behavioral model instances. There are two destinations: the behavioral model instances themselves and the structural model instances.

In the behavioral models, the modeler could specify the actions that raise events. For example, the following code in the behavioral models raises a new FSM event called *evLoginFail*:

```
triggerEvent("evLoginFail");
```

Such action specifications can appear in the state action or the transition action in the behavioral models. The example code above is line 39 in Figure 4 and is executed when the transition from the *LoginCancel* state to the *finalState* state is triggered.

The behavioral model instances also interact with the structural model instances. This interaction is established by two steps. First, at the design time, modelers can declare the reference to struct types in the behavioral models. These references are called state variables, and will be instantiated at the run-time. For example

```
public Messenger messenger;
```

declares an instance called *messenger* for the struct type *Messenger*. Second, when the transition is triggered during the execution process, an FSM can alter the states of these structural model instances by operation invocations that are specified as transition or state actions. This mechanism ensures the separation of structural model instance changes from the FSM, and makes the status of an FSM stable.

As an example, we continue the execution process illustrated in Step R1. When then *UIOk* event generated in Step R1 arrives at the behavioral model instances with current state *Idle*, the FSM will try the following transition.

```
transition Idle to LoginFail : UIOk [  
    ! messenger.users.exists( u @ u.userID == loginDialog.userID  
        & u.password == loginDialog.password)  
];
```

This transition specification declares a transition guard within the square brackets when *UIOk* event is received. The guard utilizes functions provides by *Messenger*, *User* and *LoginDialog* to verify whether the user entered valid user information or not. If the user entered incorrect login information, the active state will move from *Idle* to *LoginFail*. According to the behavioral model in Figure 4, the FSM will perform the transition from *LoginFail* to *Idle* automatically with the companion action as follows.

```
loginDialog.password.empty();  
loginDialog.status.setText("Login Fail! Re-Login");
```

These actions do two jobs by operation invocations. The first is to clear the password textbox in *loginDialog*, and the second is to set new status label information.

Step R3: Interaction from the structural model instances to UI physical views

Step 3 carries out the update process of UI physical view. Since we separate the UI models from the behavioral models, we need to design a mechanism to communicate from the behavioral models to UI models. As described in Step R2, the behavioral model instances can alter the status for structural model instances. When the operation applied to the User Interface

Structural Model instances, the status changes automatically affect the corresponding physical views. This process is conducted by platform specific virtual machines, and is developed when ZOOM is applied to a specific run-time environment.

We continue the login fail example. As described in the last paragraph of Step R2, the password textbox and status label variables are altered. Those changes are automatically reflected in the physical view as shown in Figure 6(b). Similarly, If the login process succeeds, the action with the transition from LoginSuccess to Idle will make the UI physical view to Figure 6(c).

5 Prototype Development

We have completed significant work on designing the notations and implementing the tools to support modeling in ZOOM. We have a stable grammar for ZOOM-M language along with its formal semantics. We have designed the ZOOM-M library using ZOOM-M itself. The initial grammars for ZOOM-FSM and ZOOM-UIDL have been established. ZOOM-UIDL is based on XML and the basic UIDL schemas are defined in XML schemas.

We have implemented the fundamental language support tools for the ZOOM modeling notations. Modelers can use ZOOM-M, ZOOM-FSM and ZOOM-UIDL to specify the software system, do type checking and animate some of the partial models.

In order to illustrate the feasibility and advantages of event-driven framework, an experimental prototype was developed, which includes two main components, a translation engine for the behavioral models and a set of rule-based tools for UI generation. The translation engine, which is for the code generation and animation of the behavioral models, takes the behavioral models and action semantics as inputs, and generates executable Java code. The translation engine also provides an underlying architecture for this executable code with built in support for persistence, generation of signals and events, transition and state priorities, threading and concurrency. The rule-based UI tool is developed with two separate rules for Java Swing and SWT frameworks. This translation tool takes the UI models as input, generates the executable java programs in Swing or SWT framework with different physical representations.

6 Conclusion

We introduce an event-driven framework to integrate three loosely coupled views in ZOOM, which is a new formal and object-oriented approach to support MDD. ZOOM separates software modeling into three components: the structural, behavioral and UI models. This separation of concerns allows each aspect of the system to be specified separately, and makes it possible for us to use an appropriate formal specification language to specify each of these unique aspects.

Modelers use the three loosely coupled components in ZOOM to model the software system. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to coordinate the activities between the structural, behavioral and UI models. The event-driven framework satisfies the consistency requirements between different views of the system, works as a mechanism to bind those views together at design time, and provides the execution model for run-time execution. The event-driven framework approach defines an adequate solution for both design needs and run-time execution needs. It provides a simple solution for modelers to integrate different aspects of the software system, while allowing them to specify each aspect separately. This work provides a solid foundation to continue solving the issues of implementing the event driven framework.

References

- [1] Object Management Group (2001) Model Driven Architecture (MDA), 9 July 2001 draft, edited by J. Miller and J. Mukerji, available at <http://doc.omg.org/ormsc/2001-07-01>
- [2] D.S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, OMG Press, 2003
- [3] J. Poole Model-Driven Architecture: Vision, Standards And Emerging Technologies ECOOP 2001
- [4] J. Mukerji and J. Miller, Model Driven Architecture www.omg.org/cgi-bin/doc?ormsc/2001-07-01
- [5] S. Mellor, A. Clark and T. Futagami Model-Driven Development IEEE Software No.5. Volume 20, 2003
- [6] Ed Seidewitz. What models mean. IEEE Software, 20(5):26–32, 2003.
- [7] J. Woodcock, J. Davies, Using Z Specification, Refinement, and Proof, Prentice Hall Europe 1996
- [8] J. B. Wordsworth Software Development with Z Addison-Wesley 1992
- [9] H. Liu, L. Qin, C. Jones, X. Jia, A Formal Foundation Supporting MDD --- ZOOM Approach, CTIRS03 2003
- [10] Paul D. Johnson & Jinesh Parekh Multiple Device Markup Language A rule approach
- [11] N. V. Carlsen , N. J. Christensen , H. A. Tucker, An event language for building user interface frameworks, Proceedings of the 2nd annual ACM SIGGRAPH symposium, p.133-139, 1989
- [12] J. M. Spivey, The Z Notation: A Reference Manual, 2nd Ed., 1992
- [13] Edmund M. Clarke and Jeannette M. Wing, Formal methods: state of the art and future directions, ACM Computing Surveys, 28(4): 626-643, 1996
- [14] C. Lopes, W. Hrsh, Separation of Concerns, technical report, Northeastern University, 1995
- [15] William E. McUmber and Betty H. C. Cheng, A general framework for formalizing UML with formal languages, Proceedings of the 23rd International Conference on Software engineering, 433 – 442, 2000
- [16] X. Jia, The ZOOM Notation - A Reference Manual, Technical Report, DePaul University, 2004
- [17] Bertrand Meyer. Object-Oriented Software Construction, 2nd Ed. Prentice Hall PTR, Upper Saddle River, NJ, 1997.
- [18] Bertrand Meyer. Applying ‘design by contract’. 25(10):40–51, oct 1992.
- [19] Jos Warmer and Anneke Kleppe. The Object Constraint Language. Addison Wesley, Boston, MA, 1999.
- [20] Object Management Group, UML 2.0 Superstructure Specification
- [21] Mellor, Stephen and Balcer Marc, Executable UML: A foundation for model-driven architecture, Addison Wesley, Boston, MA, 2002
- [22] Paulo Pinheiro da Silva and Norman W. Paton, User Interface Modeling in UMLi, IEEE SOFTWARE, 20[4]:62-69 2003