Overview

1. Introduction to Rhapsody
2. Basic Usage of Rhapsody
3. Important Points
4. ROPES
5. Example: Answering Machine
6. Personal Experiences using Rhapsody
7. Conclusion
Introduction to Rhapsody

- Rhapsody is software for UML-based design and simulation
  - Activity Diagrams
  - Collaboration Diagrams
  - Component Diagrams
  - Deployment Diagrams
  - Sequence Diagrams
  - Statecharts
  - Structure Diagrams
  - Object Model Diagrams
  - Use Case Diagrams
Introduction to Rhapsody

- Generates C, C++, Ada, or Java code
- Allows for reactivity, multiple threads, real-time environments
- Allows user to “roundtrip”, i.e. alter code directly and then update visual model
- For this presentation, “Rhapsody in J” version 6.0 for Windows used (“Rhapsody in C++” examined)
Browser

Shows models, model items
Used to edit features and create/delete model items
Statechart Drawing Toolbar

Class with Statechart icon

Active Code Viewer
Choose Sequential or Active for each class.
Type choices include language-agnostic and user-defined types.
Operation choices include Primitive (manually coded), Event Reception, and Triggered Operations.
Features can vary greatly for different model items.
Choose classes to be instantiated at runtime
protected int rootState_subState;
protected int rootState_active;
public static final int class_1_Timeout_state_0_id = 1;

public RiJThread getThread() {
    return reactive.getThread();
}

public void schedTimeout(long delay, long tmID, RiJStateReactive reactive) {
    getThread().schedTimeout(delay, tmID, reactive);
}

public void unschedTimeout(long tmID, RiJStateReactive reactive) {
    getThread().unschedTimeout(tmID, reactive);
}

Finishes/System.out.println("Done!");

tm(10)/setMyInt(getMyInt() + 1);
System.out.println("Hello World");
if (getMyInt() >= 10) {
    gen(new Finish());
}
Finish/System.out.println("Done!");

tm(10)/setMyInt(getMyInt() + 1);
System.out.println("Hello World");
if (getMyInt() >= 10) {
    gen(new Finish());
}

getThread().schedTimeout(delay, tmID, reactive);

public void unschedTimeout(long tmID, RiJStateReactive reactive) {
    getThread().unschedTimeout(tmID, reactive);
}
Basic Usage of Rhapsody

1. Create classes (in Browser, OMD, Sequence Diagram, etc.)
2. Create Statechart or Activity Diagram of one or more classes
3. Set up default Component and active Configuration
4. Generate code for active configuration
5. Compile and run active configuration
Important Points

- **Model-Based Design**
  - Sequence Diagrams are created in Analysis or Design Mode
    - **Design**: Messages, classes realized on insertion into diagram. Messages deleted from diagram on deletion from model
  - “Roundtripping” is only allowed at code locations between special comment markers
    - “---+[ <Type> <Name> ]” and “---+” in Ada
    - “/[*# [ <Type> <Name> */’] */’” and “/[*#] */’” in C
    - “/[*# [ <Type> <Name> ]” and “/[*#]” in C++, Java
  - Rhapsody’s Internal Reporter and ReporterPLUS can generate reports in RTF, DOC, HTML, PPT, etc. (examples on slides)
Answer

Report on Configuration DefaultConfig

Packages

AnsMachine

Constraints:
blankTapeNonNeg

Specification
blankTape >= 0;

Anchors:
- Anchor to Recorder

oneCallAtATime
Only one call can be received at one time.

Anchors:
- Anchor to Recorder

Requirements:
requirement_0
- Must save all calls
recvMsg Operation

Type: void
Visibility: public
Implementation: java.io.BufferedReader myIn = new java.io.BufferedReader(new java.io.InputStreamReader(System.in));
String msg = "";

//Receive input
try {
    msg = myIn.readLine();
}
catch(Exception ex) {
}
//-

//Set it
setMsg(msg);
//-

getMsg Operation

Type: String
Visibility: public
Implementation: return msg;

setMsg Operation

Type: RhpVoid
Visibility: public
Implementation: msg = o_msg;
Important Points

- **Object/Instance Behavior**
  - Each class may have one “state” diagram
    - Activity Diagram
    - Statechart
  - Classes can have one of two thread behaviors
    - Sequential, running in main thread
    - Active, running in own thread
  - Code added to states/actions must be written in the language of the active program ("Rhapsody in C", "C++", "Ada", "Java")
  - Configurations determine which classes will be instantiated into objects at runtime (at least one class required).
Important Points

- **Simulation**
  - Runs can be viewed through any number and combination of the following animated diagrams
    - Activity Diagrams
    - Sequence Diagrams
    - Statecharts
  - User can use Rhapsody's built-in, text-only tracer to advance through runs
  - During runs, user can manually send events to system through Event Generator

- **Third-Party Software**
  - Rational Rose models can be imported
  - Rhapsody models can be exported to DOORS
ROPES Development Process

• ROPES = “Rapid Object-oriented Process for Embedded Systems”
  by Bruce Powel Douglass (of I-Logix)
  - Iterative development process
  - Works for both elaborative and translative development, better with translative

• Outline
  1. Analysis
  2. Design
  3. Translation
  4. Testing
ROPES: Analysis

• Requirements Analysis
  – Create use cases, scenarios
  – Discover necessary constraints
  – Discover external factors that affect system
  – Discover possible system hazards, risks
  ▪ Sequence Diagrams
  ▪ Statecharts
  ▪ Use Case Diagrams

• Systems Analysis
  – Separate system into functional segments
  – Design high-level algorithms for these segments
ROPES: Analysis

- Categorize system functions as software, electronics, or mechanics
- Test the segments
  - Activity Diagrams
  - Component Diagrams
  - Sequence Diagrams
  - Statecharts

- Object Analysis
  - Design required classes and objects for system
  - Test the classes and objects
    - Activity Diagrams
    - Collaboration Diagrams
ROPES: Analysis

- Component Diagrams
- Object Model Diagrams
- Sequence Diagrams
- Statecharts
ROPES: Design

- **Architectural Design**
  - Determine number and usage of threads
  - Utilize design patterns for error handling, safety, fault tolerance
    - Activity Diagrams
    - Collaboration Diagrams
    - Component Diagrams
    - Object Model Diagrams
    - Sequence Diagrams
    - Statecharts

- **Mechanistic Design**
  - Utilize design patterns to facilitate collaboration
ROPES: Design

- Collaboration Diagrams
- Component Diagrams
- Object Model Diagrams
- Sequence Diagrams

- Detailed Design
  - Specifically design internals of classes, associations with other classes
  - Activity Diagrams
  - Object Model Diagrams
  - Statecharts
ROPES: Translation & Testing

- **Translation**
  - Transform model information into source code
    - **Rhapsody Code Generation**

- **Testing**
  - Follow a planned testing document
  - Add one component at a time during integration testing
  - Run validation tests (black box)
  - Run safety tests (white box)
Example: Answering Machine

- Requirements Analysis
  - Use cases
    - Recording/Playing back messages
    - Recording outgoing message
    - Displaying number of recorded messages
    - Recording incoming messages
    - (2\textsuperscript{nd} Iter.) Keep track of message lengths, blank tape
  - External factors
    - Length of tape/Amount of memory in answering machine
Example: Answering Machine

• Systems Analysis
  – Split Answering Machine components into groups
    ▪ AnsMachine (Software)
    ▪ Hardware
    ▪ (2\textsuperscript{nd Iter.}) Caller & Owner Agents

• Object Analysis
  – Design classes and objects, interactions between them
    ▪ (Collaboration Diagram on next slide)
Example: Answering Machine

• Architectural Design
  – Design threads
    1. Main system thread
    2. (2nd Iter.) Caller agent thread
    3. (2nd Iter.) Owner agent thread
  – Use Design Patterns for error handling, safety, fault tolerance
    ▪ No safety or fault tolerance concerns
    ▪ Found patterns generally not applicable to example

• Mechanistic Design
  – Use Design Patterns to aid in collaboration
    ▪ Found patterns generally not applicable to example
Example: Answering Machine

- Detailed Design (OMD in two slides)
  - AnsMachine
    - Contains one instance each of Chronometer, Microphone, Recorder, Speaker
    - Takes events from Caller and Owner
  - Microphone
    - Receives incoming messages from Caller and outgoing message from Owner
  - Recorder
    - Saves message data
    - Discards data if “blankTape” is less than message length
  - Speaker
    - Plays outgoing message to Caller
    - Plays incoming messages to Owner
Example: Answering Machine

- Plays informational messages
  - *(2nd Iter.)* Caller
    - Makes calls and leaves incoming messages by sending events to AnsMachine
  - *(2nd Iter.)* Owner
    - Sets outgoing message
    - Hears, deletes incoming messages by sending events to AnsMachine
  - *(2nd Iter.)* Chronometer
    - Tracks lengths of messages as Caller “speaks” into microphone
  - *(3rd Iter.)* MsgTuple
    - Contains string data “msg”
    - Contains integer data “msgLength”
Example: Answering Machine

Demonstration
Personal Experiences using Rhapsody

• Ease of Use
  – Appears intuitive, but surprises can confuse new users
    ▪ Some models and model items generate code while others do not
    ▪ Setting up a Default Configuration incorrectly can cause compilation errors
  – New user will probably consult Rhapsody manual often, but it is often lacking
    ▪ Manual is C++-centric
    ▪ Manual does not discuss model to code translation
  – Few Java example projects, many C++ examples; but very few use Activity Diagrams
  – Auto-realization of operations, events very useful
Personal Experiences using Rhapsody

• Extent of Model-Driven Design
  – Depends on user
    ▪ Statecharts cannot have sub-activity diagrams and vice versa. This limits extent of model-driven design
    ▪ Round-tripping may allow user to greatly ignore model

• Stability of Rhapsody
  – Glitches
    ▪ Animated Sequence Charts must be opened from menu to animate properly
    ▪ Collaboration Diagrams can improperly number messages if new messages are inserted
  – Occasionally, .rpy files corrupted while saving
  – Occasional, inexplicable crashes occur
Conclusion

- **I-Logix Rhapsody**
  - has
    - simple model-drawing, model-defining tools
    - useful thread control mechanisms
    - a variety of report generation styles
    - powerful animation/simulation and debugging tools
    - the ability to generate code in several programming languages
  - but lacks
    - a language-agnostic User Guide that answers the questions new users will have
    - the stability it should have (at least, in my experience)
    - language-agnosticism as its focus, a little idealism(?). Rhapsody is strictly utilitarian

I-Logix. “Getting Started with Rhapsody”.


I-Logix. “Rhapsody List of Books”.


I-Logix. “Rhapsody List of Books”.


I-Logix. “Rhapsody List of Books”.