

Design-Level Debugging

Soon after source-level compilers were introduced, software designers realized that assembler-level techniques were no longer the most effective way to debug systems that were defined in high order languages (HOLs). Designers no longer wanted to debug by determining whether the system should have branched on zero or non-carry. They wanted to understand if the FOR-LOOP had been executed the appropriate number of times. Eventually, it became necessary to debug systems in the same level of abstraction in which they were defined.

Over the past several years, the level of abstraction of software definition has risen again. Model-based design is the newest methodology to gain a foothold in real-time design. The idea behind it is simple: "a picture is worth a thousand words." The benefits this visually oriented paradigm offers are analogous to those offered by HOLs. By enabling designers to work at a more natural level, model-based design lets them concentrate on the problem they are trying to solve, rather than the code. However, designing at the model-level is only part of the picture. To achieve all of the benefits of a model-based development process, a product should be debugged against its definition, in this case the model. Model-based -- or design-level -- debugging, consists

of driving and monitoring the debugging process from a design model viewpoint. The designer executes the compiled system while a debugger environment provides visual feedback of the dynamic aspects of the system in the context of the design model. In an object-oriented system this consists of highlighting modeling constructs such as states and transitions in Statecharts and capturing inter-object communication in the form of messages and events in Sequence Diagrams.

MODEL-BASED DEVELOPMENT

In a model-based design paradigm the process phases classically referred to as "analysis and design" are performed by creating and evaluating a model. The

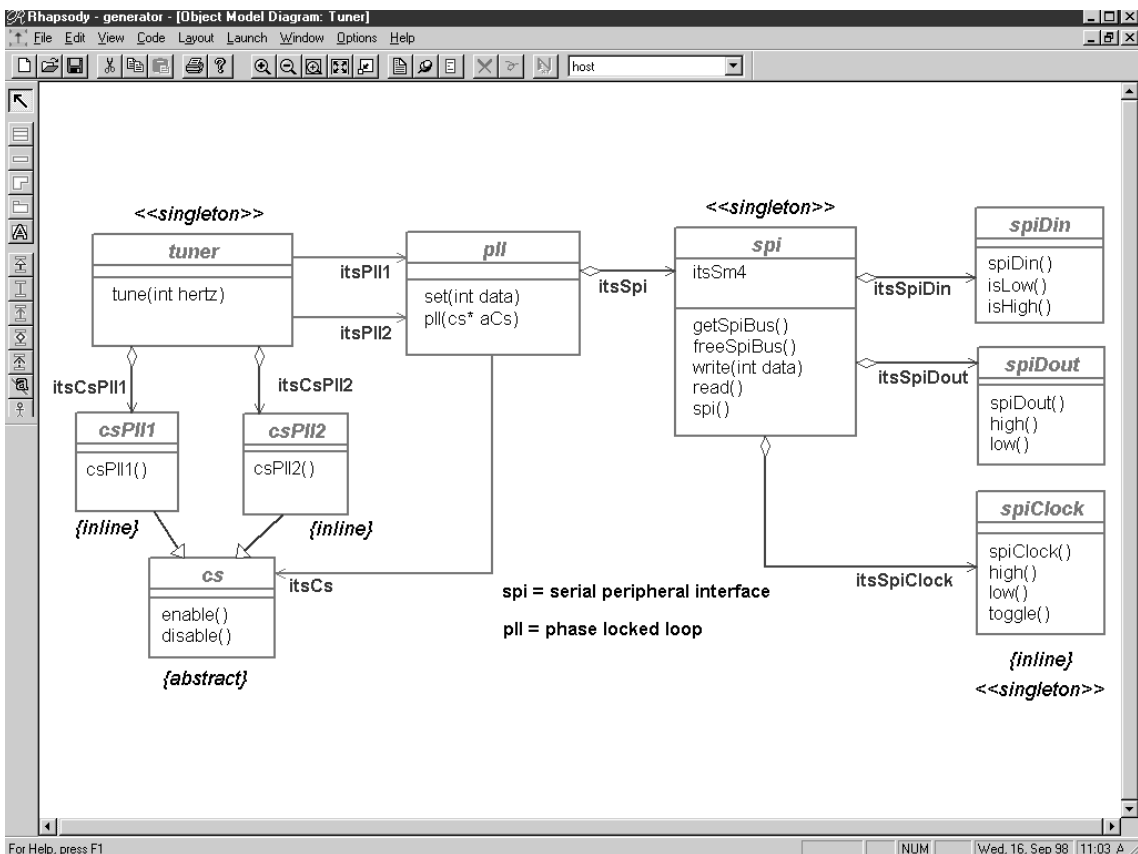


Figure 1. Top-level Object-Model Diagram for a PBX system

evolution of model-based development has gone through several stages. Starting with Structured Design and Functional Decomposition, the practice is now moving toward object-oriented (OO) techniques. Recently, OO modeling languages have become standardized with the Object Management Group's (OMG's) publishing of the Unified Modeling Language (UML).

UML provides a number of modeling views that are useful for real-time software design.

- **Object Modeling Diagrams** - Object Model Diagrams define the architecture of the system in terms of objects. An object is an identifiable bundle of data coupled with the methods used to operate on the data. These "Class Diagrams" define the components of the system and the relationships between them.
- **Statecharts** - Statecharts define the lifecycle behavior of objects in the system. A Statechart is a form of state machine that has been extended to support hierarchical as well as concurrent state behavior.
- **Sequence Diagrams** - Sequence Diagrams specify inter-object or collaborative behavior. Sometimes referred to as Message Sequence Charts or Bounce Diagrams, they define how objects interact to perform a task by showing sequences of messages across time.
- **Use Cases** - Use Cases define the major functions that the product needs to perform and the dependencies and relations between them. These function points are decomposed into scenarios that often are used as part of the test definition for the system.

MODEL-BASED DEBUGGING

Testing and debugging are highly inter-related. However if testing is defined as "making sure that the system operates correctly" then debugging must be defined as "making sure that the system does not operate incorrectly."

Any effective debugging environment, whether source-based or model-based, must provide a number of key capabilities:

- The ability to step through the system in a variety of granularities is needed.
- Visual feedback must be provided within the system definition during debugging.
- It must be possible to interrupt the execution of the application and set breakpoints to trigger on the occurrence of user definable circumstances.
- The ability to input stimuli to the system both interactively and from a 'script' is needed.

- The ability to capture results from the application in a manner that they can be reviewed, played back, and analyzed is needed.
- The user must be able to treat the system as both a black box and, more importantly, a white box.
- It must be possible to insert errors and override correctly or incorrectly operating portions of the system.

In addition to general debugging issues, a number of issues specific to a model-based environment exist:

Visualization

Visualization of system behavior during execution is key to model-based debugging. In an object-oriented system the Statechart and the Sequence Diagram are the primary behavioral views. At a minimum, a good model-based debugging environment must provide the user with visual feedback within these views while the actual software is being executed in the target environment. Typically, states and transitions are highlighted in the Statecharts, and message sequences are shown in Sequence Diagrams during debugging. Another important visualization technique is the presentation of state information in the captured Sequence Diagram.

UMPS

UNIVERSAL

MICROPROCESSOR

PROGRAM

SIMULATOR

The better choices in embedded tools ...

VIRTUAL MICRO DESIGN

Sales: 33 559-438-458

Fax: 33 559-438-401

E-mail: sales@vmdesign.com

Technopole Izarbel

64210 BIDART - FRANCE

<http://www.vmdesign.com>

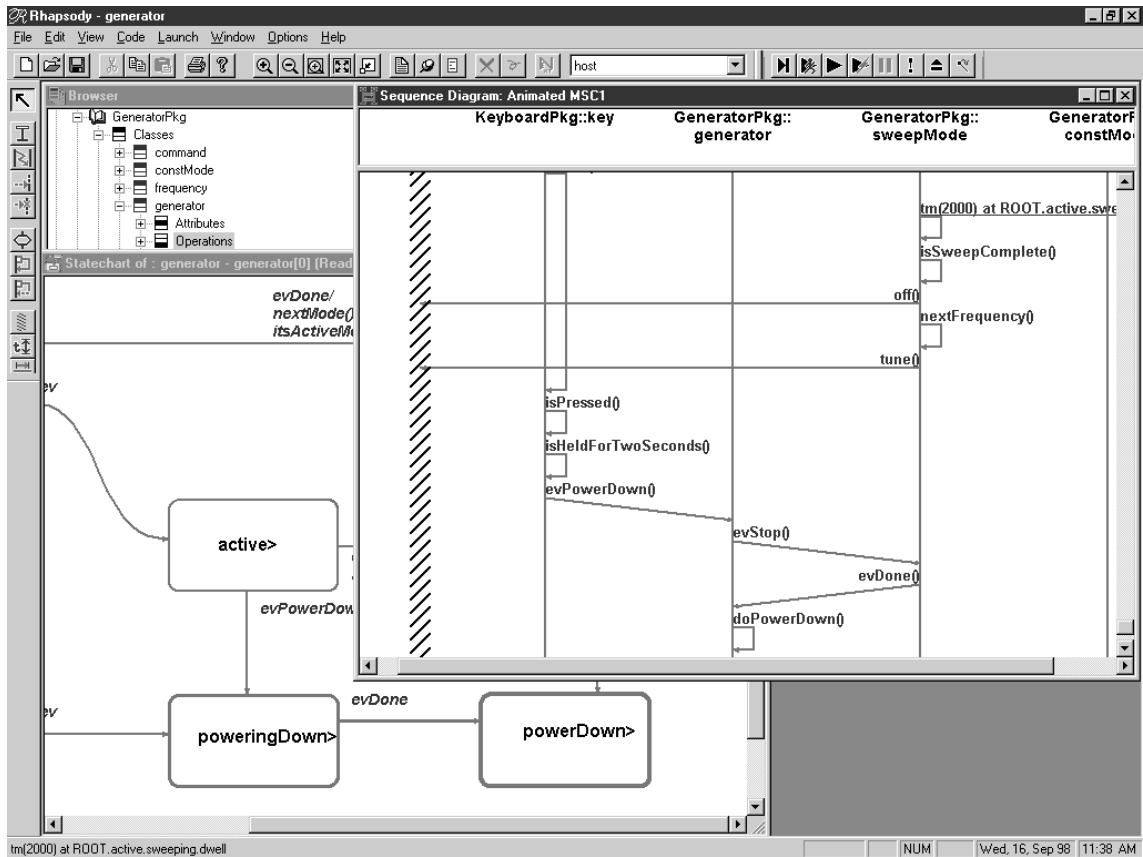


Figure 2. Visualization within model showing Sequence Diagram (upper right) and Statechart (lower left).

In addition to showing the user what is happening in the system, several mechanisms to analyze the debugging information are needed. The ability to analyze the differences between Sequence Diagrams enables a comparison of expected behavior to actual behavior. Color-coding threads enhances the ability for designers to find resource problems especially in systems with a high degree of interdependency between threads.

How does the design model relate to the implementation code?

An important consideration to model-based debugging is how the implementation was created. In a traditional design process the code is often hand-crafted. In such a process, the model is still useful as a debugging map. However, the value of the map is limited by how precisely the manually created code matches the design model and whether it was updated when design changes were made during implementation. Even if the model was rigorously followed during coding, in a manual process the design model serves as a general map only, not a precise implementation-level debugging guide.

If code is created via a code generation tool, the relation between the model and the code is key in being able to achieve any automated aid. If the tool generates code frames or code skeletons, the infrastructure and the behavior of the classes must be created by hand. This situation is essentially the same as that for hand-written code.

Another approach is based on a run-time engine. In this case, functional code is created but the code is generated in a black box fashion. The behavior is intended to match the model but the code is built in a form conducive to the tool's code generation techniques. In this case some automation is possible but the debugger's view is limited solely to the model. No implementation-level debugging aids are possible.

The final technique is one in which the model and the code are treated as views of the design. Called "model/code associativity," this approach produces code that is always in sync with the design model. A high degree of automation aids are possible with this approach. Viewing the executable simultaneously from both the model view and the implementation view enables the design model to act as a precise debugging map of the running system. Additionally, problems can be identified and corrected in either view and the corresponding view is automatically updated.

When an error is discovered using any of the first three techniques, the problem could either be a design flaw, a coding error, or an error in translating the design to implementation. When the same problem is found using model/code associativity, it's readily apparent where the problem exists in the design.

Animation versus simulation

At the highest level, there are two different major technologies for executing the design model, simulation and animation. Simulation is based on an artificial or

REAL TIME MAGAZINE

SUBSCRIPTION SERVICE

REAL-TIME CONSULT,
Rue de la Justice 23,
1070 Brussels,
Phone. 32-2-520-55-77, Fax. 32-2-520-83-09
E-mail: info@realtime-info.com

BOOKSTORES

EBSCO Subscription Services

P.O. Box 1943, Birmingham,
Alabama 35201-1943,
USA
Phone. 1-205-991-6600
Fax. 1-205-991-1479

Standaard Boekhandel

Industriepark Noord 28 A
9100 St.Niklaas,
Belgium
Phone. 32-3-760.32.11
Fax. 32-2-777.92.63

SWETS Subscription Service

P.B. Box 830,
2160 SZ Lisse,
Netherlands
Phone. 31-25.213-51-11
Fax. 31-25-211-58-88

Dawson France

B.P. 57,91871
Palaiseau Cedex,
France
Phone. 33-1-69-10-47-00
Fax. 33-1-64-54-83-26

Lavoisier Abonnements

14 rue de Provigny
94236 Cachan Cedex,
France
Phone. 33-1-47-40-67-00
Fax. 33-1-47-40-67-02

READMORE

22 Cortlandt Street
New-York 10007-3194
USA
Phone. 1-212-349-5540
Fax. 1-212-233-0746

BLACKWELL'S Periodicals

PO BOX 40
Hythe Bridge Street, OX1 2EU Oxford
Great-Brittain
Phone. 44-1865-79-27-92
Fax. 44-1865-79-14-38

FRANCE Publications

112 Rue Reamur
75002 Paris
France
Phone. 33-1-45-08-55-68
Fax. 33-1-45-08-50-19

Schweitzer Sortiment

Lenbachplatz 1
D 80333 Munchen
Germany
Phone. 49-89-551-340
Fax. 49-89-551-34100

Garmar Libreria Internacional

Fundadores, 5-1. 5
28080 Madrid
Spain
Phone. 34-361-1388
Fax. 34-361-07-50

BTJ AB

Box 1302
SE 11183 Stockholm
Sweden
Phone. 46-8-723-6900
Fax. 46-8-723-0038

Hoepli Ulrico Casa Editrice Libreria SPA

Via Hoepli 5
20121 Milano
Italy
Phone. 39-2-865-446
Fax. 39-2-805-2886

emulated environment in which the model language is somehow executed. Animation consists of enabling the application code to communicate with the design environment and highlight the model at run-time.

How greatly animation exceeds simulation in both performance and fidelity to the target implementation depends on the efficiency of the simulation technique. Regardless, simulation only tests the design, not the implementation -and this is only part of the debugging task. Once the design is implemented, more tests still need to be run, some to verify that the implementation meets the design. Others will be required based on the specific implementation.

Animation, on the other hand, tests the actual implementation. Animation based on a model/code associative approach enables the design model and the implementation to be tested simultaneously since they are same design.

Mixed design-level and source-level debugging

The most effective technique of debugging systems developed using model-based design is to do most of the debugging at the model level. This provides the best understanding of the problem and the most rapid solution. However, situations will arise where being able to dive down to the implementation and perform source-level debugging is the better approach.

Consequently, it is important that any design-level debugging environment also support implementation-level debugging. It must be possible to be able to step through the running executable either in the design view or the source view. It must be possible to interrupt the executable and set breakpoints in both views. Design and source-level information must be provided to the designer in sync and in the correct form for the related level of abstraction.

Integration with IDEs

Synergism between the source-level and model-level debugging environments is important for efficient debugging. A good model-based debugging environment should smoothly and seamlessly interface to source-level IDEs. The designer must be able to easily move from a model-level of abstraction to a source-level of abstraction. Where appropriate, such context switching should be automatic. At either level, debugging information should be shown in a form consistent with that level of abstraction. Common operations should be performed consistently and be available at either level. For example, setting a breakpoint should be possible at both levels, but setting it at either level should create the break at both levels.

Debugging on the host versus debugging on the target

Another consideration is the availability of the target hardware. Often, out of schedule necessity, software development starts in advance of target hardware selection or existence. An effective model-based design and debugging environment must provide a means to debug the software prior to the target hardware's availability.

Two primary techniques -- OS emulation and application retargetting -- provide the capability to execute the software on a different platform other than the unavailable target. OS emulation allows one to run the executable in a simulated OS environment. The downside of this approach is reduced performance within the emulator as well as the financial cost of the emulator and supporting tools. Hardware emulation is also an option but availability of new processors is still an issue and the cost is even higher than software emulation approaches.

Retargetting consists of porting the application to the host platform or even to a different target. The cost of this approach is the programming labor of doing the port itself. This can vary greatly depending on the application. An effective debugging tool set will support automatic retargetting of the application so that it can run on different targets or on the design host itself. This provides the ability to verify the software behavior in advance of the hardware availability without the cost of expensive emulation products.

THE I-LOGIX APPROACH TO DESIGN-LEVEL DEBUGGING - A FULL IMPLEMENTATION ENVIRONMENT

In order to provide the necessary model-based debugging capabilities for embedded real-time systems, I-Logix has created a Visual Programming Environment with full production code generation capabilities. This product, Rhapsody, is based on the OMG-standard Unified Modeling Language and provides an iterative method of developing software throughout the analysis, design, implementation and test phases.

Rhapsody enables software developers to design, implement, and debug at the model-level of abstraction. Through its unique model/code associativity, the product generates complete, deployable C++ code for embedded and real-time applications. Through model-based debugging it provides debugging and test capabilities at the design level and enables software development as a truly iterative process. An application can be tested and debugged while it is being designed -- delivering a system with a design known to be correct by the time it reaches the test and integration phase. The result is a 66% savings in debugging and integration time, an overall reduction in time-to-market, and an application that works right - from the start. ■

John Stanglewicz is product marketing manager for I-Logix' Rhapsody product line, an object-oriented visual programming environment for embedded, real-time software development. Prior to joining I-Logix, Stanglewicz worked on a variety of avionics systems and software projects for the Hamilton Standard Division of United Technologies and then for McDonnell Douglas Aircraft Company. Stanglewicz graduated from Northwestern University in 1985 with BS in electrical engineering.