Rhapsody:
A Complete Lifecycle Model-Based Development System

Eran Gery, David Harel and Eldad Palatshy

I-Logix, Inc.

1 Introduction

Model-based development has been a central theme in the software development industry almost since its inception. Nevertheless, model-based development is not yet mainstream practice. The emergence of the unified modeling language (UML) [7] as a ubiquitous practice across the software industry brings awareness to modeling and abstraction a step forward. Still, only a small percentage of software developers practice model-based development with UML.

A common usage pattern of models is the informal use-case, by which concepts and ideas are sketched in a model, and then followed by traditional implementation, without any formal consistency between the modeling artifacts and the implementation artifacts. As we explain below, despite its common practice this approach exposes some major weaknesses and actually limits the ability to benefit from the main advantages of model-based development.

Other paradigms follow a translative approach, whereby implementation artifacts are generated from models but are not practically accessible to the developers. This approach represent the other extreme. On the one hand it is formal, meaning that model-semantics is fully compiled into the implementation artifact. But on the other hand it poses significant difficulties as a mainstream technique in the software industry [reference to ROOM?, SDL], since it ignores the role and importance of interfacing the implementation platform itself.

Recently, the OMG emerged with the Model Driven Architecture initiative (MDA), which combines a set of technologies to achieve the construction of systems in a highly reusable, platform-independent manner. In it, model-based development serves as the key facilitator in achieving platform independence and a high level of reuse. The model-based development approach proposed in the MDA is consistent with what we describe here, in that it offers a formal and automated approach on the one hand, while realistically addressing implementation constraints the other.

In this paper we describe the concepts behind Rhapsody [3, 4], a system designed to facilitate an effective model-based development lifecycle. In retrospect, Rhapsody realizes many of the ideas outlined in MDA, while focusing its applicability on real-time embedded systems. It addresses those deficiencies that prohibit the effective usage of model-based development, turning it into a mainstream tool for industry. Rhapsody is based on the idea of executable models, as described in [3], and it is fair to say that both of these pioneered the idea of formal model-based development with UML.

2 Iterative development

Iterative development is the current mainstream philosophy for software development. In the UML context it is described as a generic process in [5]. A more specific process suited for the development of embedded systems is described in [1]. Iterative development addresses several key issues that can be viewed as the legacy of waterfall-based processes:

- Reduces risk by early detection of analysis, design and implementation issues.
- Is able to effectively trade off schedule and functionality in order to address competition and/or schedule slippage.
- Facilitates concurrent development to shorten development lifecycles.

It is crucial that a model-based development process effectively support iterative development. Iterative development is based on incremental steps, each of which goes through a complete analyze-design-implement-test cycle. Completion of increment n is the starting point for increment $n + 1$. Therefore, effective iterative development requires strong bi-directional traceability between all development artifacts. In addition, it is important to facilitate rapid iteration throughout the development lifecycle.

In the sequel we will analyze the key enabling technologies required to facilitate effective iteration and traceability.

3 Problems with model-based development

There are several key inhibitors for the adoption of model-based development by software developers. Each of them results in a workflow deficiency that inhibits the synergy between iterative and model-based development.

A common deficiency is lack of *implementation automation*, resulting in the need for manual coding of the implementation or significant parts of it. This creates traceability gaps between the modeling and implementation artifacts, as they are inherently out of synch in the first place. The main reason for this deficiency is lack of support for model executability. Model executability requires complete semantic interpretation of the model, accompanied by algorithms for translating the model into artifacts that can be executed.

Another common deficiency is the *discontinuity* between the implementation artifacts and modeling artifacts representing the analysis and design stages. Iterative lifecycles require that the implementation artifacts created in the nth iteration are augmented and refined during the $n + 1$ iteration. In the absence of proper traceability support, changes to the implementation during integration of iteration n, invalidate the model for iteration $n + 1$.

The next inhibiting factor is *disintegration* with the implementation platform. Modeling languages offer rich semantic abstractions of structure and behavior, but they do not normally provide rich sets of operations at the detailed level. In contrast, implementation languages offer rich constructs at the detailed level, such as arithmetic, memory manipulations and other kinds of detailed level operations, but lack proper abstractions for concepts above these. The lack of proper integration between the two results in a compromise, either of usability or of efficiency of the resulting implementation.

4 Rhapsody's key enabling technologies

Rhapsody embodies several principles that are instrumental for effective model-based development, and which address the common automation and traceability deficiencies. The key enabling technologies are *model-code associativity, automated implementation generation, implementation framework, model execution and back animation*, and *model-based testing*.

We now describe each of these in some detail.

4.1 Model-code associativity

Model-code associativity is a key enabler for software developers to effectively leverage the benefits of model-based approaches, but without compromising the benefits of direct access to the implementation language/platform. The fundamental principle here is that the model does not replace the implementation language, but rather being augmented by it in a synergistic manner, where abstractions are described and viewed by models, but detailed implementation is carried out by an implementation language.

To achieve this in Rhapsody, the model and code are viewed as two viable development artifacts of the system. This is done using the following design principles:

- Detailed behaviors (also known as "actions") are written in the target implementation language. This avoids unnecessary translation of detailed computational expressions from one language (the "action language") to another, although the two are essentially isomorphic. Using the implementation language as the action language also contributes to the readability of the resulting code, as well as to the expressiveness of low-level manipulation.
- The implementation language is augmented by an execution framework (see subsection 4.3 below), which provides additional semantic constructs not supported by the implementation language. The framework is essentially the interface between the implementation language and the modeling language abstractions.
- Code resulting from the model is intuitive and self-explanatory. This is achieved by using common translation patterns for UML's abstract constructs, and by attaching notes and constraints to the resulting code. In addition, using the implementation language as the action language, and the implementation framework itself, contribute to the readability of the implementation source code.

- Rhapsody supports online navigability between the model artifacts and their code counterparts, and vice versa. It is always possible to trace the code resulting from certain model elements and to trace the model element corresponding to a particular section of the code.
- Model and code are always synchronized: changes in the model are instantaneously reflected in the code view, and many kinds of changes in the implementation code are "round-tripped" back, to be reflected in the model. This facility is instrumental to the entire idea of model-based development. It addresses the inherent discontinuity problem, whereby changing the code invalidates the model and thus also the entire model-based development workflow.

4.2 Automated implementation generation

The core of Rhapsody support for model executability is its implementation generator. The generator generates fully functional, production-ready implementations, employing all the behavioral semantics specified in the UML model. These include system construction, object lifecycle management (construction and destruction) , object behavior as specified by statecharts or activity-graphs, as well as methods directly specified by the implementation language.

The implementation generator maps every model artifact into a set of implementation artifacts (source code, make-files) based on generation rules and a set of predefined parameters for each model element type (metaclass). The generation parameters may specify translation tradeoffs (size/speed, size/modularity) as well as code style. In addition, the parameters specify implementation domain objects to be reused in the translation. These may come from the Rhapsody execution framework or be defined by the user. All this enables the user to choose from a wide range of implementation strategies that realize the model semantics.

The implementation generator supports various implementation languages (C++, C, Java) and component frameworks (COM, CORBA).

4.3 The execution framework

The execution framework is an infrastructure that augments the implementation language to support the modeling language semantics. Our framework based approach is an open architectural mode of work, providing a set of architectural and mechanistic patterns to support modeling abstractions like active objects, signal dispatching, state based behaviors, object relationships and lifecycle management.

The execution framework is given in several forms of APIs, based on the implementation language. These APIs are used by the modeler to perform manipulation at model abstraction level, including sending signals, creating composite objects, creating object relationships, and interacting with the state model.

The Rhapsody framework consists of 3 domains of architectural and mechanistic patterns:

- The active objects base framework: a set of architectural patterns that supports the active object set of semantics, including concurrency, signal dispatching, synchronization, and lifecycle management.
- The operating system abstraction layer (OSAL): a layer that encapsulates a set of basic services typically supported by an operating system, such as creation of threads, event queues, semaphores, memory management, etc. The OSAL needs to be implemented specifically for every targeted operating system. It allows easy retargeting of models to different platforms without any change to the model.
- Utilities for mechanistic design: a set of containers to implement the various relationships and lifecycle management between objects in the system.

The execution framework API is also a protocol that is used by the code-generator to facilitate model semantics in the context of a particular implementation language. Developers may specialize and augment the basic semantics by specializing or changing the implementation of the framework pattern. The latter is achieved by changing the open code generator mappings to use a different set of framework classes.

The set of patterns provided by the framework comprises common structural and behavioral patterns used by applications based on the active object paradigm; specifically, real-time embedded applications. The framework implies a high level of reuse of these validated pattern implementations, which contribute to the modularity and quality of the generated application. A fairly detailed description of the framework can be found in [6].

4.4 Model execution (model-based debugging)

Model execution is a key enabler for effective model-based development, as it facilitates the ability to effectively *test* what has been constructed. The key theme behind model execution is expanding the specification model with a runtime model that provides the ability to trace and control the execution of the system at the same level of abstraction as that of the system specification.

There are two main approaches two model execution. The first, which is often referred to as *simulation*, is to construct an interpreter that executes the model based on the runtime semantics. The other one, which is manifested in Rhapsody, is to provide a runtime traceability link between the implementation execution and the runtime model. This technique is also called *model-animation*.

The fact that the model execution is linked to the actual implementation execution has several advantages. From a methodological point of view, it enables shorter iterations of model-implement-debug cycles, while maintaining full consistency between the model and its implementation. Another advantage is the ability to have concurrent source-level and model-level debugging by allowing the use of a code level debugger in the process. Such a combination also allows an easier mapping between the UML design and its source code implementation.

Figure 1 is a screen-shot of the runtime model, instantiated during a model execution session. The model consists of all instances of objects, their links and internal states. It also includes event queues and call-stacks for active objects, and also a trace that logs all the events in the system.

As shown in the figure, one can view individual instances, including their current state configuration, attributes values and association references. The system trace is reflected as instantiations of animated sequence diagrams. Other available views are an output window that can be used to textually trace the run, call-stack and event queue windows that relate to a specified "focus" thread, a browser view showing the instances of each class, and more. As for control capabilities, Rhapsody provides a set of animation commands that can be invoked using the animation toolbar. These include injecting events, setting focus threads, running step-by-step in different granularities (for example, "go event", which means run until the application is finished dealing with the current event in the queue) setting filters for the output window, logging the trace to external files, and more.

Fig. 1. Model execution using Rhapsody

The architecture behind our model-execution technique consists of the following components: (1) Source code instrumentation hooks, inserted by the implementation generator; (2) A runtime trace framework, implemented in the implementation language, which provides a set of services to trace and control the application; (3) A trace and control communication protocol between the application and Rhapsody (implemented over TCP/IP), which allows Rhapsody to debug applications running on remote embedded devices using embedded operating systems; (4) A runtime model maintained within Rhapsody; (5) A set of runtime views, as described above.

4.5 Model-based testing

Model-based testing provides the ability to specify and run tests based on the specifications in the model, as well as the ability to pinpoint defects from the test results by visualizing points of failure within the model. A good model-based testing component also facilitate a trace between the modeled requirements and the constructed system.

Currently, most testing cycles have the following characteristic steps: write or record test scripts based on the specification documents and existing prototypes; run the scripts; review the results; report defects and/or confirm fixes. From the developer's perspective, handling such defects consists of the following: try to reproduce the defect by repeating the reported scenario; debug the application and try to diagnose the source of the failure; fix the defect and ... hope for the best .... Usually, towards new releases, this cycle seems to repeat itself endlessly, where the number of defects constantly fluctuates, until the system meets the required level of quality according to the available set of tests. Another well-known fact is that using traditional approaches, most of the bugs are introduced during the early stages of the development, but are found toward the release. This observation is one of the main motivations for incremental and iterative approaches.

Rhapsody's model-based testing features the following canonical usage sequence:

1. Specify tests using scenarios specified in the model.
2. Run the tests. Rhapsody will drive and monitor the model execution.
3. Review results and pinpoint failures, by having the tool show the actual trace rendered as a sequence diagram that highlights where the scenario that was expected was violated.
4. Fix the defect and verify by rerunning the test.

The scenarios in clause 1 were specified explicitly as requirements throughout the analysis and design stages, or may have been recorded using the model execution feature described earlier to serve as a baseline for regression-testing. The scenarios are specified as extended sequence diagrams, following concepts proposed in the live sequence charts of [LSC]. Scenarios can be referred to as monitors or also drivers. Driver scenarios also provide stimuli to the system during test execution.

This approach to testing has the following main advantages:

- Abstract modeling concepts can be reused to specify tests, as an alternative to script files.
- The requirements from the model and execution traces canbe reused as tests.
- Both black-box and white-box scenarios canbe used for testing, unlike the traditional approach that executes only black-box scenarios. Also, one can easily elaborate a black-box into a white-box scenario by adding the relevant instances during the run, or offline, as part of the test specification.
- It is easier to diagnose the source of the defect, since the exact point of failure is shown by the tool using sequence diagram notation.
- Testers and developers alike can specify and execute tests: no specialized tools or knowledge of scripting languages is required.
- Earlier detection of bugs throughout the development process is possible: tests can be defined and executed as part of the development effort and then routinely executed from that point on.
- Consistency is maintained between requirements and tests: updating requirements automatically updates test criteria and in many cases the other way around too.

5 Conclusion

In this brief paper we have attempted to discuss the rationale behind the approach taken in designing the Rhapsody tool [3, 4], and its main advantages in model-based system development. We talked about the problems Rhapsody was intended to solve, and its key enabling technologies. The two main ones worth repeating here are model execution and model-based testing.

Model execution addresses two of the main inhibiting factors of Section 3: *discontinuity* between the modeling artifacts and the implementation artifacts, and *disintegration* with the implementation platform. Model-based testing contributes to effective model-based development by maintaining consistency between model requirements and tests, by facilitating early testing throughout the iterative lifecycle, and by increasing the productivity of specifying tests and detecting defects.

References

1. Doing Hard Time: Bruce Douglass
2. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1), 2001. (Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (*FMOODS'99*), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pages 293–312.)
3. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, pages 31–42, July 1997. Also, *Proc. 18th Int. Conf. on Software Engineering*, Berlin, IEEE Press, March, 1996, pp. 246–257.)
4. www.ilogix.com
5. RUP book by IVAR
6. Octopus White Paper (on the WEB site)
7. UML Specification
8. MDA
9. Action Semantics