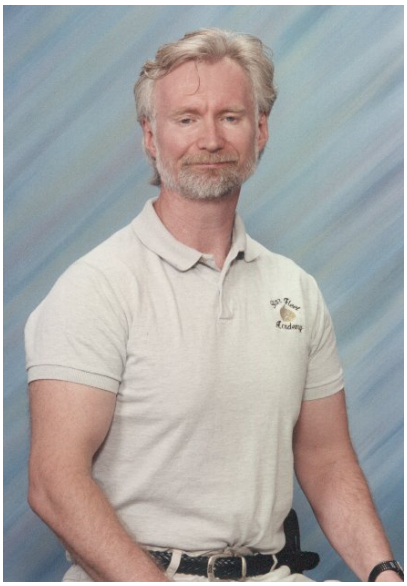
UML for Executable Specification



Bruce Powel Douglass
Chief Evangelist
I-Logix

Abstract

The Unified Modeling Language, or UML for short, is by far the most common modeling language in use today in software development. It is routinely used in firmware and real-time and embedded systems development, and more and more, being used by systems engineers to specify the architectural structure of the overall system. The primary benefits of the use of UML for system specification are two-fold. First, for systems that are software-centric, the use of UML allows easy transition from the system engineers to the software development teams. This means less work and fewer translation defects, because software engineers are used to dealing with the UML and have tools that allow them to do so. The second primary benefit comes from the fact that the behavioral model of the UML is based on Harel Statecharts, a well-defined executable language. This means that a specification of a system in UML can be executed and evaluated for consistency, accuracy, and fidelity before it is even decomposed into mechanical, electronic, and software technologies. This power aids the system engineer in ensure that they have a consistent and buildable specification long before personnel or hardware is design, lessening the risk of new development.

Why Executable Specifications?

Why specifications at all? Why don't we just wire together parts (whether with a soldering iron or by writing code) until we have want we want? The problem is that with today's highly complex systems, it isn't clear what we want, nor that what we want is consistent, reliable, or buildable. Producing specifications allows us to be unambiguous about what the system is, the features it supports, consistency among those features. It allows us a forum to capture, analyze, and understand what system provides to the user prior to actually building it. So what are the things we want from a specification. A good specification should be

- Unambiguous; that is, it should clearly state the requirements so that it is obvious what the system has to do
- Clear and understandable to all the various stakeholders (engineers, marketers, customers, managers, documentation specialists, etc)
- Correct; the requirements should accurately reflect what the system should do

- Consistent; different parts of the specification must not contradict each other in ways great or small
- Testable; that is, we should be able to, in a straightforward way, validate that the delivered system meets the requirements
- Implementation-free; design decisions should be left to design – the specification should identify how it behaves from a black-box perspective
- Translatable into design; the transition from requirements specification into hardware and software design should be as easy and straightforward as possible

For complex systems, specifications can reach hundreds or thousands of pages, so meeting these goals can be a daunting task. Typically, specifications are reviewed in detail by a large number of experts to ensure that it meets these goals, but this is an expensive, highly labor-intensive, and error-prone task. Another approach in common use in certain industries where the cost of project failure is particularly high (e.g. automotive and aerospace) is to write specifications in a more formal language so that some of these properties can be checked either through the application of formal methods (e.g. proofs or "model checking" rules automatically applied) or via testing of some kind. The success of this approach is obvious from the widespread use of tool such as Statemate[1] from I-Logix.

Clarity, correctness, consistency and testability are all closely related aspects of a specification. The most common way of specifying systems – the so-called "Victorian Novel Approach" – uses imprecise natural language to specify precise things. The problem with this was identified (admittedly in a different context) by Immanuel Kant in *The Critique of Pure Reason* – it is called the "analytic-synthetic dichotomy." Simply put, the analytic-synthetic dichotomy states "that which is known (the spec) cannot be real; that which is real (the envisioned system) cannot be known." This is a serious problem for teams developing systems. How do we know that the specifications reflect what we really want and what we actually deliver?

I will argue that the only way to demonstrably achieve these varied goals for specification is to write the specification is such way that it is testable; and, in order to be testable, a specification must be executable. If a specification is not directly executable, then you are "hand-waving" because you can't test something you can run. And it becomes all too easy to miss expensive defects in the specifications that will come back to haunt the engineers later.

Even though the specification ought to be, *in principle*, executable, it should also be implementation-free. That is, the ones who REALLY know how to do the mechanical engineering are the mechanical engineers, the ones who REALLY know how to do the electrical engineering the electrical engineers and the ones who REALLY know how to do the software are…. the managers. Oops, I mean the software engineers. And these engineers should not be handicapped at the outset from making sounding engineering

[1] Statemate is a state machine-based specification tool that has been widely adopted in automotive and aerospace industries. See www.ilogix.com for information and white papers.

decisions as more design information is discovered. That is, the specifications should clearly identify *only* the things which are actually required and not how these things ought to be designed and implemented. This is a very common failing of engineers who end up writing specifications – they are so used to doing design that their requirements specifications are really nothing more than disguised design specs.

In order to facilitate the last goal of good specifications, translatability, the use of a common or similar language for specifying requirements is a great aid to that. It is common for system engineers to use one set of concepts and one language and this is "thrown over the wall" to the development engineers who must manually interpret and translate the specification into their own language. One way to minimize this translation effort is to use a common language for specification and design. Particularly when the system is software-intensive, using a software specification/design language makes a lot of sense, provided that the language is sufficiently rich to represent all the things needed.

The Unified Modeling Language (UML) meets all these needs. It is a common language with broad tool support, it is in use by an increasing number of system engineers, supports both requirements and design capture, and it is, at its core, an executable language.

What UML Offers For Specification

System requirements can be divided into two broad categories: functional and quality of service requirements. Functional requirements define what the system does, such as "The system shall compute a flight path and smoothly adjust the flight surfaces to achieve that flight path." The other kind of requirements is constraints that affect the functional requirements. These might be performance (e.g. worst case performance, average case performance, deadlines, periods, jitter, and so on), schedulability, time boundaries, weight, size, reliability, safety, maintainability, development cost, time to market, etc.

Organizing Requirements

Use cases are the primary means provided by the UML to organize captured requirements. A use case is a named capability of a system that does not reveal or imply implementation. A use case associates (i.e. exchanges messages with) actors, entities in the system's environment with which the system must interact. A use case, by itself, is inadequate to capture the semantic details of the capability, but it does provide a placeholder. A common rule of thumb is that a use case represents 3 – 10 pages of a written specification.

The UML represents use cases on diagrams as a named oval. Actors are shown as stick figures. Lines connecting the two are associations indicating the messages come from or go to the associated actors during the execution of the use case. Figure 1 shows an example Use Case diagram from a secure 2-way radio with a satellite tracking capability.

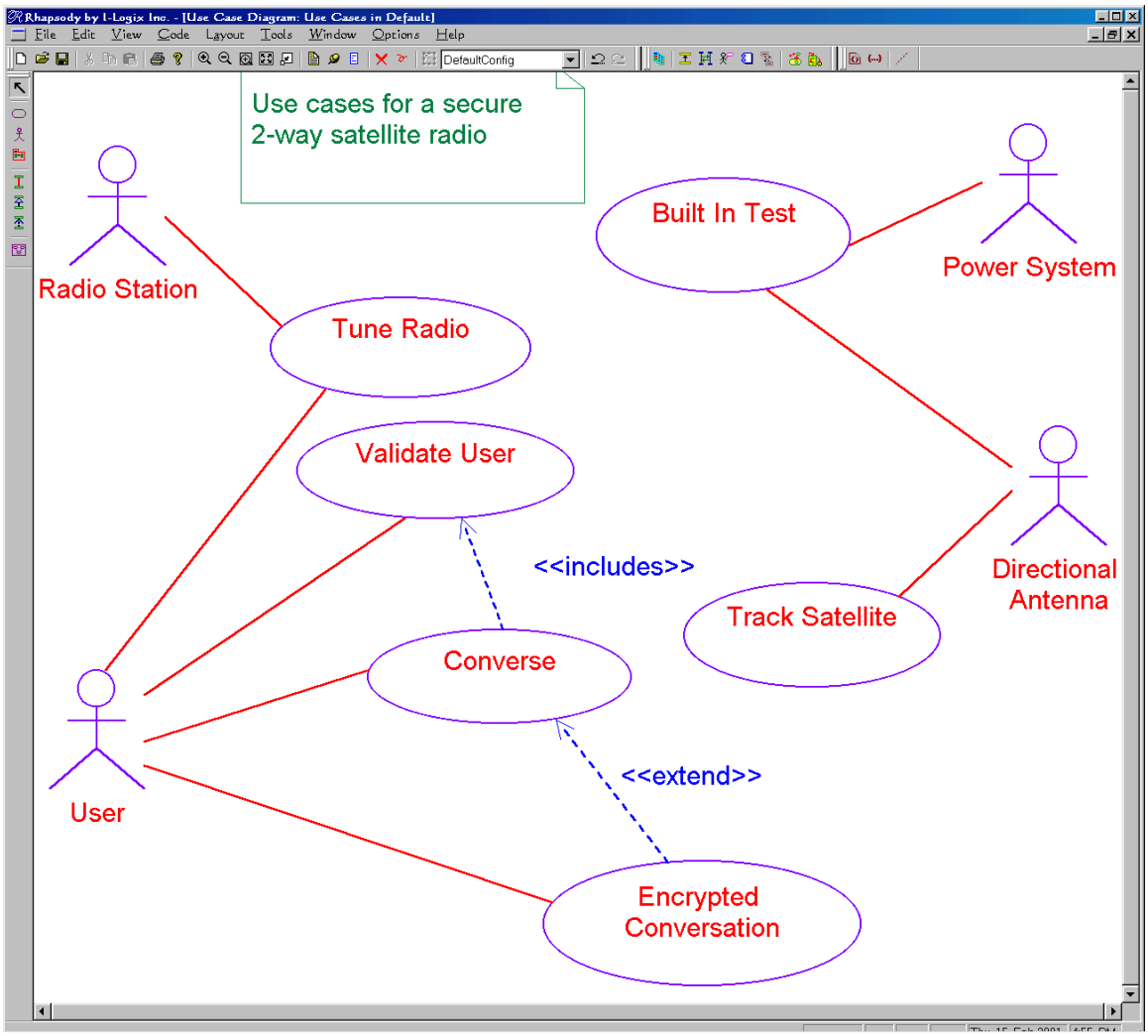


Figure 1: Secure 2-Way Radio Use Cases

Each of the use cases obviously needs to have more substance – what does it *mean* to have an encrypted conversation anyway? Clearly, we must add and manage the detailed requirements; in fact, this is commonly called *detailing the use cases*. We will discuss that in a moment.

There is another means that the UML provides to manage things as well – packages. A package is a "bag" which holds model elements. For large systems with dozens of use cases, it is more effective to package the use cases into related capabilities, or perhaps capabilities of architectural aspects of the system (e.g. guidance and navigation, power, attitude control, communications, etc).

Detailing the Use Cases

There are two approaches to capturing the detailed requirements of a use case: by example and by specification. Both have pros and cons, but they can be used together as

an effective means to capture requirements and ensure that, during design, the system meets those requirements.

Scenarios

The most common way to detail a use case is to provide example scenarios – system-actor interactions – that illustrate what we mean. Scenarios capture functional requirements in terms of the

- Operations or behaviors provided to the actors by the system,
- Protocols of interaction (sets of operations) between the system and the actors
- Quality of service constraints on operations and sets of operations

The scenario shown in Figure 2 shows such as scenario. The vertical lines are called *instance lines*; they represent the "players" in the scenario. The actors are shown using hatched lines and the use case is shown with a normal line. As an alternative to a use case instance line, you can use the System as a line here, since the system is executing the use case during the scenario. What we *don't* want to have is the internal structural pieces (e.g objects, subsystems, or components) shown on this diagram, because the purpose of this diagram is to show the requirements of the system as a whole, with respect to a specific use case, *not* the system's internal design. Later, as we add design elements, we can elaborate this diagram by adding those internal design elements but we don't want to pollute our requirements at this early stage with design, as we've discussed before.

The horizontal lines are messages or events sent between the actors and the system. A set of messages exchanged in a particular order is called a *protocol*. It is not enough to merely specify the individual operations that may be invoked during a scenario, but to understand any order dependencies among them as well.

There are also constraints in the scenario as well. In the UML, a constraint is a user-defined well-formedness, or rule of correctness, that applies to one or more model element. Constraints are normally expressions, in formal or informal languages, that add special rules about some aspect of the model elements involved. Two of the constraints specify maximal periods of time between message pairs, while a third indicates that a particular sequence of messages is *required*.

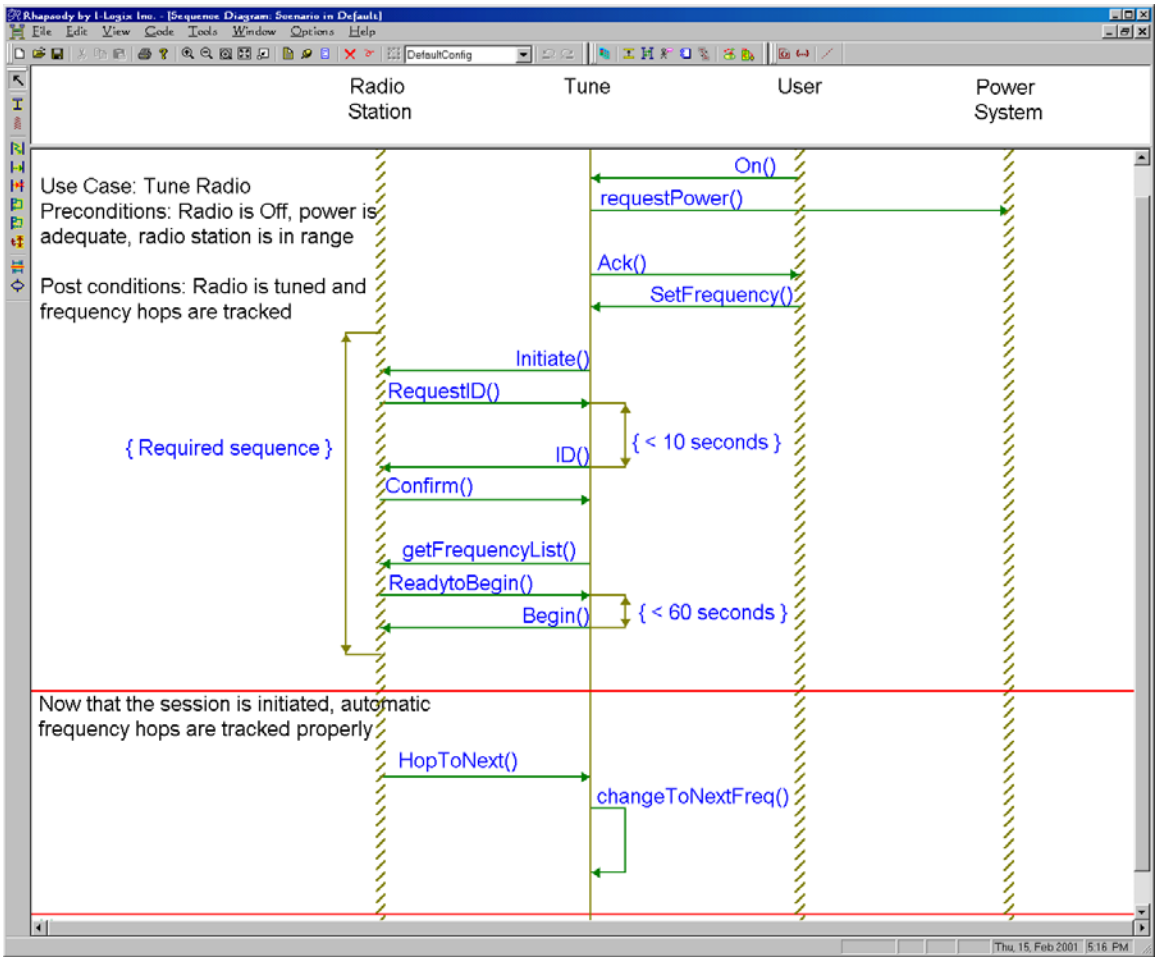


Figure 2: A Scenario for Tune Radio Use Case

Scenarios are popular, largely because they are examples of system execution. It is very easy for non-technical stakeholders to follow the interaction of the system with the actors. This allows the exploration of this interaction without requiring the readers to understand formal specification languages or arcane mathematics.

Scenarios are not without their cons as well. First, they are examples, and as such, are inherently incomplete. There is, in fact, an infinite set of such scenarios. One difficulty of the requirement analyst is to find a useful set of scenarios that capture all the relevant detailed requirements. To identify the requirements, you must look at the entire set of scenarios and understand them in toto.

Another difficulty is that there is no way to specify negative requirements, such as “The system shall *not* …”.

Specifications

The other approach to detailing use cases is to define or specify them, rather than give examples of them. In this approach a language is used to capture the definition of what the feature is and how it works (from an external viewpoint). The language used to

express the specification may be a natural language, such as English, or a formal language, such as statecharts, or a combination of the two. The UML provides a formal language (Statecharts) that may be used for the specification of requirements. Closely aligned with statecharts, the UML also provides activity diagrams for specifying behavior.

Natural languages have an obvious advantage – they are *natural* – easy to understand, easy to use. They are, however, imprecise, and this can lead to serious problems during design and implementation. The system will always ultimately do something very precise, but if the specification is imprecise, the system may not provide the desired behavior. Further, because the specification is not precise, it is not executable, and this has problems all its own, as we shall see later.

When you detail a use case with natural language, it is common to use a template in which textual descriptions of words are entered. Different authors suggest different templates, but a common one is

---

Name:
Use case name

Preconditions:
Statements about the condition of the system and its environment prior to the execution of the use case

Postconditions:
Statements about the condition of the system and its environment after to the execution of the use case

Description:
Description of the use case (the specification itself)

Constraints:
Limitations on the behavior or the system related to the use case, including various qualities of service (worst case performance, reliability, safety, etc)

Some authors add other fields such as "Actors" and "Related Use Cases" but I prefer to use diagrammatic representations for these things.

As mentioned above, this is a common approach, very similar to DeMarco's "minispecs" of Context and Data Flow Diagrams popular in the 1970s. The approach is not as precise as a more formal engineering specification, but it can be effectively used.

Formal specifications, to my mind, are simply specifications written using a precise and formal language. Statecharts and activity diagrams are both formal languages in this sense, and have the advantages not only of precision, but also of executability.

A statechart is a kind of extended finite state machine; the extension are designed with the intent of circumventing the limitations and problems with traditional Mealy-Moore (M&Ms) state machines (a subset of statecharts). The primary problem with M&Ms has to do with scalability. There is a well-known problem called *state explosion* that occurs with state machines – they simply get too complicated to handle. In order to address the state explosion problem, statecharts introduce a number of extensions, such as

- Hierarchical nesting of states
- Concurrency regions of state machines
- History (persistence of state)
- Conditional event processing (guards)
- Conditional branching (conditional pseudostates)

as well as many others. It is beyond the scope of this article to describe in great detail these extensions[2], but a few words about the primary extensions are in order.

[2] For details see the author's *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Patterns and Frameworks* (Addison-Wesley), 1999 or *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004); downloadable whitepapers on statecharts are available at www.ilogix.com.

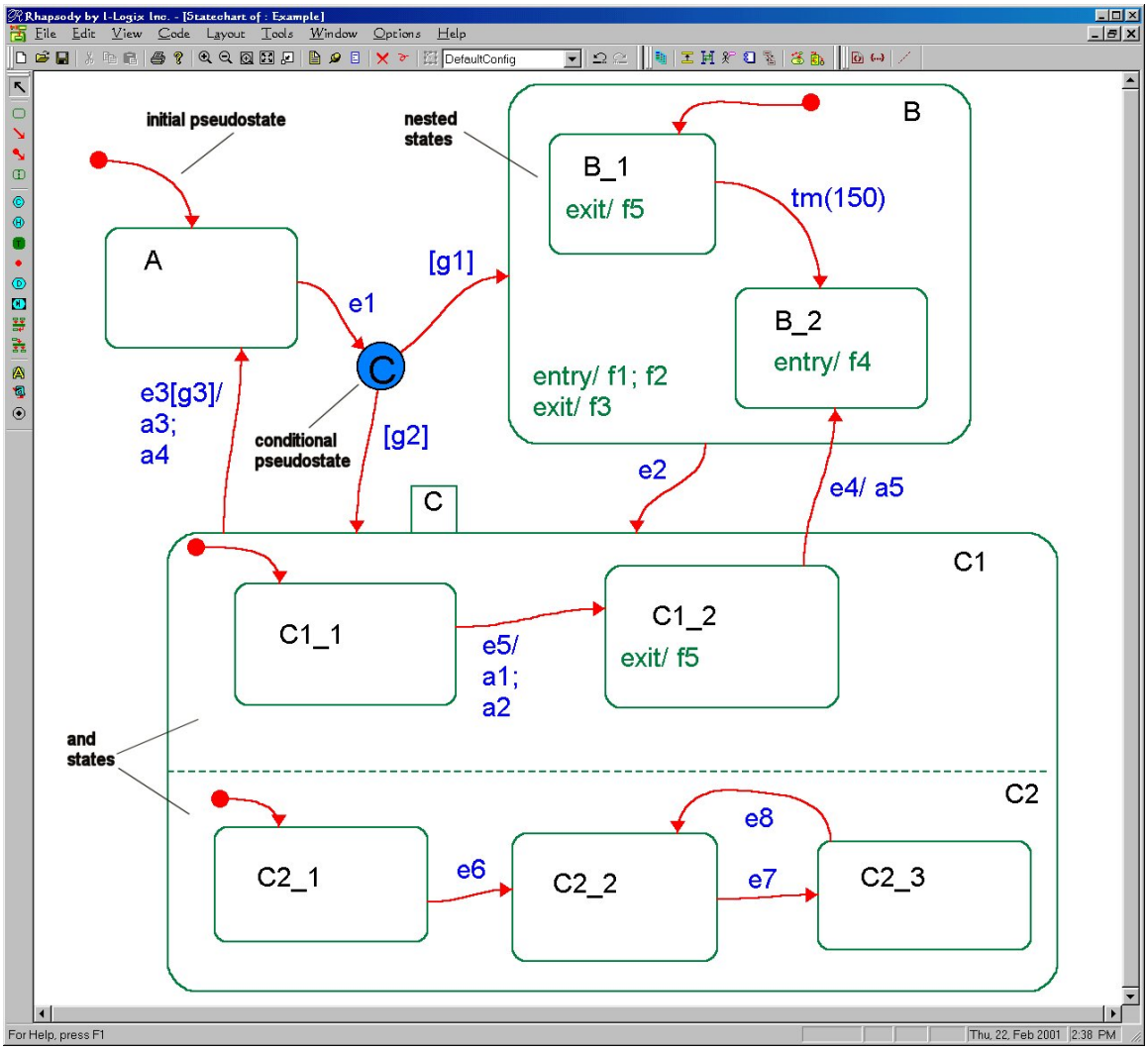


Figure 3: Statechart Extensions

The most noticeable thing about the statechart in Figure 3 is the fact that states (shown as rounded rectangles) are nested inside other states. State B, for example, as two *substates* B_1 and B_2. The semantics of this nesting is that white the system is in state B, it may be in either state B_1 or B_2; it must be in one of those states while it is in state B, but cannot be in more than one. This is equivalent to saying, for example, "When the system is ON, the light shall be either GREEN (OK) or RED (FAILURE).

The second most notable thing about the figure is the dashed line in state C. This dashed line separates the state into two concurrent regions, C1 and C2. C1 and C2 are called *and-states*, because while the system is in state C, it *must be* exactly one substate of C1 AND exactly on substate of C2 *at the same time.*

The figure uses two so-called "pseudostates." The first is a transition with a ball on one end. This indicates the starting state when the system is created or when a state with nested states is entered. The second is the conditional pseudostate, the © mark. When the event triggering the transition occurs (such as e1 in Figure 3), the *guards* for each

transition segment exiting the pseudostate are checked. The guards are the Boolean condition inside the square brackets. If one of them evaluates to TRUE, then that branch is take; if none of them evaluate to TRUE, then the event is discarded and the transition to a new state is not made. If more than one guard is true, then one of the true branches will be taken, but you can't, in principle, know which one.

Actions are defined to be "run-to-completion behaviors" that may be executed when a state is entered or exited, or when a transition is taken. On transitions, actions are listed following a '/' character, such as the list of actions following event e5 on Figure 3. They may also be put as entry or exit actions on the state, as in state B in the figure.

Statecharts excel in specifying reactive behavior – that is, behavior where the system waits for an event and then reacts to it. For systems where transitions proceed primarily because actions complete, activity charts are preferred. Activity charts are a kind of state machine where the progression from state to state is done after the work done in a state completes. They may be thought of as a kind of concurrent flowchart and is useful for specifying sequential and iterative algorithms and work flows of various kinds.

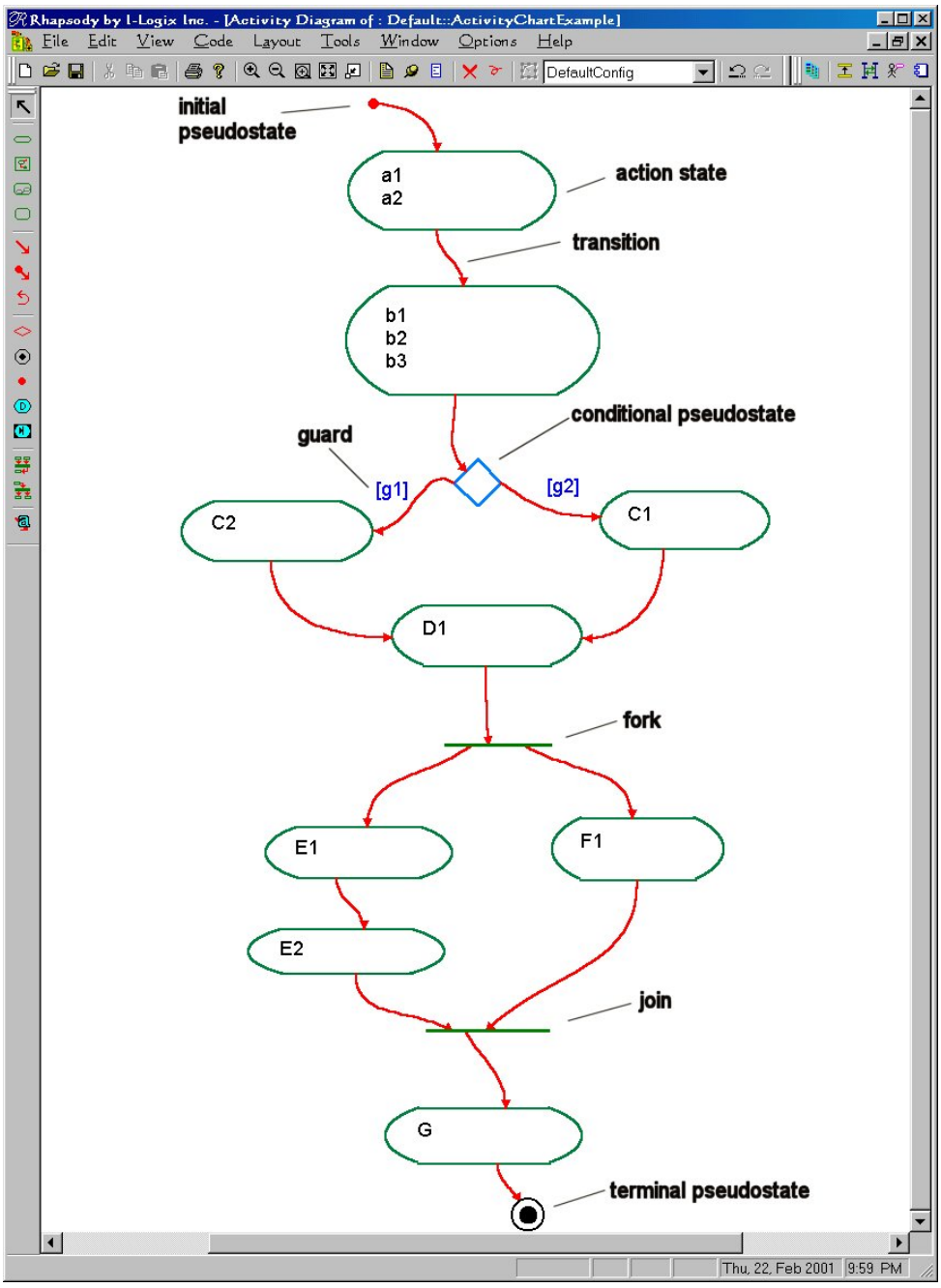


Figure 4: Activity Chart

The rounded rectangles in Figure 4 are action states and they contain action lists that they execute when entered. The transitions fire as soon as the actions in the predecessor action state completes. The conditional pseudostate works identically to how it works in statecharts. The fork and join specify concurrent action state flows (i.e. E1 and E2 execute concurrently with F1 in the figure). The terminal pseudostate marks the end of the behavior.

Statecharts and activity diagrams are useful because they specify exactly how a behavior takes place, in all possible scenarios. This formal definition is in one place allowing the

design full access to the details of the specification whereas scenarios scatter requirements among multiple diagrams.

In the UML, statecharts or activity diagrams can specify the behavior of a *Classifier*. Two important Classifiers for our discussion here are Use Cases and Objects. You can define the behavior of a use case with a statechart or activity diagram or you can specify the behavior of a system or subsystem (which are, after all, big objects no matter how you cut it).

When you use statecharts to specify behavior for use cases, you use the messages from the actors as events to the statechart and messages from the system or use case as actions on the statechart. The statechart thus specifies what the system does when in various conditions (states) when different messages are received. When using activity diagrams, since these are primarily driven by completion of actions, you may either use normal states with incoming events (just as on statecharts) or use can use actions to send messages from the system to the actors, such as to acquiring data for processing.

Figure 5 shows a statechart specification for the Tune Radio use case for our secure radio system.

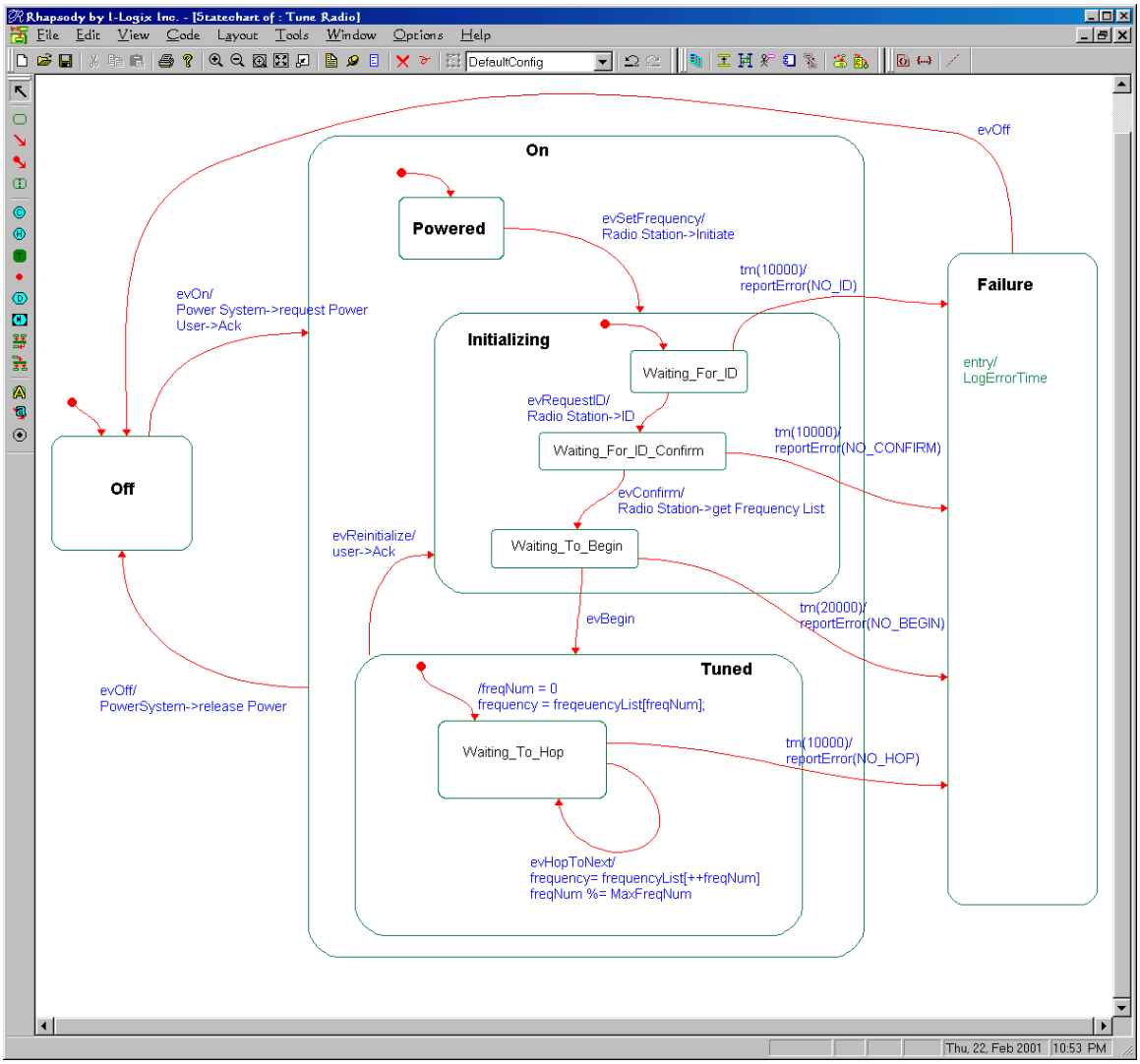


Figure 5: Statechart for Tune Use Case

You can trace the scenario in Figure 2 easily through the statechart in Figure 5. However, you can also see additional requirements, such as error handling when waiting for various messages. If you wait too long, then the timeout event occurs (see tm() on the statechart) and the use case transitions to the Failure state. One can also see *all* of the allowable sequences of messages received. If, for example, an ID event is received out of order, it is discarded, since it isn't explicitly handled. If desired, the out-of-order message could have triggered a failure or an elaborate error recovery, depending on the exact requirements being modeled.

Executability of Specifications

Because statecharts are a precise specification of behavior, they can be executed without having to actually design and construct the actual system. This is very useful for complex specifications into which conflicting or erroneous requirements can occur without being obvious. Executing the statechart is useful for both show the specification is correct as

well as constructing a set of test vectors that can be applied to the design system to ensure that it meets those requirements.

Statecharts are executable, which means that specifications done using statecharts may be executed. You can easily answer questions such as

- "What happens if the user attempts to reinitialize the system when the system is waiting for a Confirmation?"
- "What happens if there is no ID Confirmation sent?
- "Under what conditions can I shut the system down"

and so on. To be fully executable, not only must the statechart be well-formed (i.e. not violate any syntactic or semantic rules for correctness), the actions specified must also be written in an formal action language. The action language is the language in which actions are written; it may be informal, such as English, or formal, such as C, C++, Java, or Z. The formal action languages are, of course, executable as well. It is common, when writing statecharts for an object design, to use the same language that the UML model will be implemented in as the action language. So if the system is to be implemented in Java, actions on the statecharts are also written in Java. This is a highly effective approach, but it does make the migration of the system to another target language, such as C++, more difficult. For this reason, a small set of developers use an abstract action language that can easily be translated into any target language. However, the vast majority of developers use the same action language as source language because it simplifies their life overall.

For systems engineers who are not software experts, learning an action language can take time and effort. If the language of implementation is known and unlikely to change, it might be advantageous for the systems engineers to learn enough of that language to use in the action statements. In any event, however, if executability of specifications is a desired goal, the systems engineer writing the specifications must use some formal action language. Tools such as Rhapsody™ from I-Logix generate C, C++, or Java code from UML models that execute after being compiled, and are an effective way to create and evaluate specifications as well as pass them on to the software development staff.

It should be emphasized that executable specifications are just that – specifications. The system must still be designed and implemented. However, they do provide the standard for correctness and accuracy.

Aside from the obvious benefit that executability of specifications brings to the table, it is also useful to expedite the passing of the specifications to the engineering development staff in such a way as to facilitate the development of the system, and to the testing staff to facilitate the testing of the completed system.

There is an old saw in the software development industry that the best way not to have defects in a system is not to put defects in the system in the first place. In practice, the best way to achieve this is to continuously test that the parts you are working meet their specifications and continue to meet their specifications as you add more to the system and

elaborate its functionality. This is easy to achieve using a combination of scenarios and formal specifications.

It works like this: Once a formal specification of a use case is developed, a set of scenarios are derived from it. In the case of a statechart, you write one scenario for every non-looping path through the statechart, and every looping path exactly once. This covers all paths in the statechart, although not every combination of paths. This is enough to generate a basis set of test vectors with the actors and the use case.

The implementation ("realization" in UML-speak) of a use case is a *collaboration*. A collaboration is a set of objects working together to realize a use case. How do we show during development that our collaboration is adequate?

Some of you may be thinking that we already have a statechart specifying the detailed requirements of the use case, can't we translate this directly into design object model? The answer is "No, you can't." Creating a good collaboration is a creative effort and cannot be automated. The collaboration must, as a group, provide the state behavior defined in the statechart, but it is inobvious how to map this to the behavior of individual objects. The most obvious way to do this – mapping states directly to objects, results in massively *bad* object models. So let's try a different way.

A collaboration is adequate when it implements *all* of the scenarios defined for the use case. If we derive a good basis set of scenarios and use them to test the collaboration as it develops, and if we do this continuously throughout development, then we are almost assured of having a good solution at the end. How do we do this?

Remember the vertical lines on the sequence diagram are called *instance* lines. As we develop a collaboration, we add the objects on the collaboration to the sequence diagram and show, by drawing the messages among these internal objects, how they achieve the scenario. Once the collaboration can meet all of the scenarios derived from the formal specification, then our collaboration is "adequate." Of course, this means meeting not only the functional requirements, but also the quality of service requirements as well.

It turns out, that this elaboration of scenarios by adding design detail and testing the elaborated collaborations early and often has the effect of identifying defects early, when they are still easy and cheap to remove. The key to achieving that effectiveness is creating the specifications properly in the first place. The UML provides the tools – use cases, sequence diagrams, statecharts, and activity diagrams – that allow you to create high quality specifications that easily translated into high quality designs.

About the Author

Bruce Powel Douglass has over 20 years experience designing safety-critical real-time applications in a variety of hard real-time environments. He has designed and taught courses in object-orientation, real-time, and safety-critical systems development. He is an advisory board member for the Embedded Systems Conference, UML World Conference,

and Software Development magazine. He is a cochair for the Real-Time Analysis and Design Working Group in the OMG standards organization. He is the Chief Evangelist at I-Logix, a leading real-time object-oriented and structured systems design automation tool vendor. He can be reached at bDouglass@ilogix.com.