

## Executable Systems Design with UML 2.0

---

By Scott Niemann  
scottn@ilogix.com

### Introduction

Document driven approaches to software development have been a predominant approach to design in the past. This approach has had some success, but in the days of ever increasing complex systems, this type of approach is falling short. The issue with a document driven approach, is that the document itself is always up for interpretation. The requirements document usually consists of several “shall” statements of what the system is supposed to do, and usually that is thrown over the wall to the software developers to implement. In this approach there are several areas of potential miscommunication. The first is how do you know the requirements document is correct? Does it really capture what the customer wants? How do you know? The fact that it may be incorrect, or if there is a misinterpretation of the document, is only going to propagate itself throughout the development cycle, through high level design, implementation, and test. Hopefully the defect in requirements will be found much sooner than the testing stage, but you can only hope.

This problem has attempted to be solved with formal specifications for requirements modeling. The formal models have defined semantics that are more difficult to misinterpret than a textual specification. However even with formal methods, the only way for you to be sure that your models are correct is to test those models. The fact we can take formal models and execute those models, creates for us a new paradigm of systems and software engineering. Emerging Model-Driven Development (MDD) technology based on the Unified Modeling Language (UML) 2.0, such as Rhapsody by Telelogic, can seamlessly tie the activities of systems and software engineers into one environment, allowing execution and deployment of the model unto the target system. This paper will take a look in detail to how executable UML models using MDD technology can reduce the risk of finding errors sooner than later, creating a higher quality system that meets the customer expectations.

### Rhapsody MDD Overview

Rhapsody is referred to as a MDD environment for embedded real time systems. Its modeling aspects are based on the UML 2.0. It supports the concept of building the

entire systems application from concept to deployable code. This means the end result from Rhapsody, will be an executable, which is generated for the actual target system.

Rhapsody can execute the model throughout the entire product lifecycle. In the requirements phase, you can define through models, the functional requirements of the system through use case diagrams, and through formal methods describe the scenarios of those use cases or functions. The formal method in this case is a state machine, and in UML this is referred to as a statechart or activity diagram, which contains semantics such as concurrency, history, deferred events, and many more special semantics designed for the UML 2.0. Because these diagram have formal semantics they can generate an executable model, and this model can be debugged at run time. Meaning these diagrams can be stimulated by injecting an event stimulus, such as a signal or simple method call, and this process can even be automated.

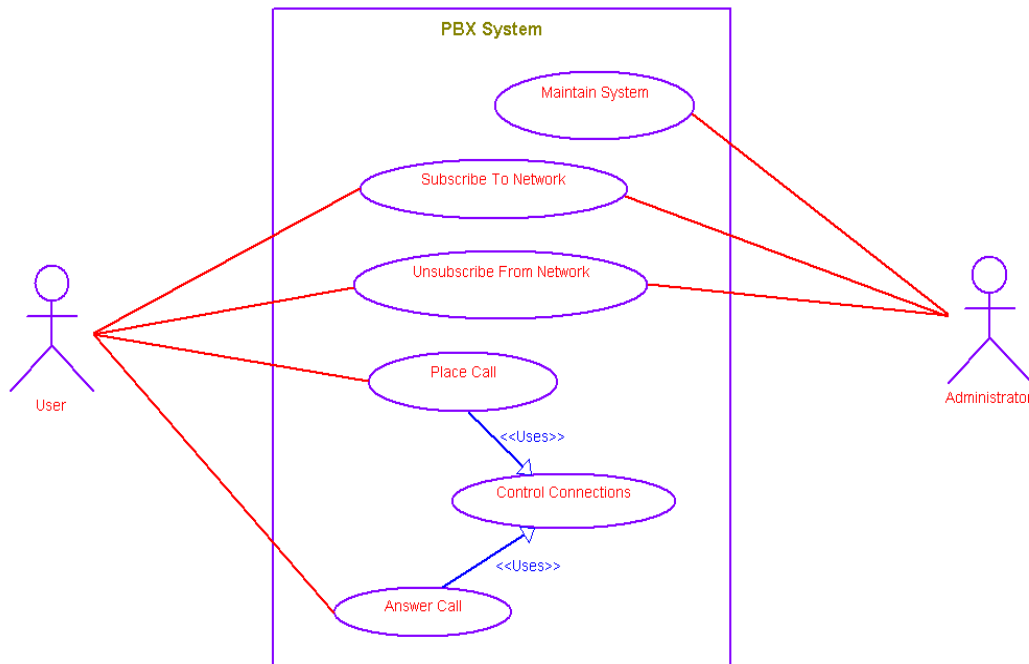
We will discuss in depth on the value this technology within model-based requirements engineering. This activity is traditionally thought of as a pencil and paper activity, but through MDD technology, these requirements can be caught graphically, and life can be brought to the pictures through execution, so they are no longer simply two-dimensional graphics.

## **Case Study: PBX System**

The traditional requirements process has worked in a unidirectional fashion. The requirements of the systems are identified as a series of “shall” statements that seemingly cover the customers needs, and are handed over to the next phase of development, which is high-level design. However there is an assumption that the requirements are correct, and without formal methods to check that the requirements cover all possible scenarios of the system, you can only guess whether the requirements are correct.

Executable MDD technology solves this problem through execution of scenarios. What this provides is feedback to the requirements team as they define the requirements, not after the requirements are implemented, and defects are found in system test. The concept of executable models works well, when the formal models can be executed at any point in requirements capture in order to receive immediate feedback as to the correctness of the design. This prevents the common problem of cascading errors within the development process, because requirements engineers have now fully executed and validated the models before the models are expanded upon in high-level design.

To better illustrate the power of executable models in the terms of requirements analysis, let’s take a look at a typical PBX system. In UML, the idea is to define the functional requirements of the system through use cases, then define the external entities of the system or actors that will communicate with the system.



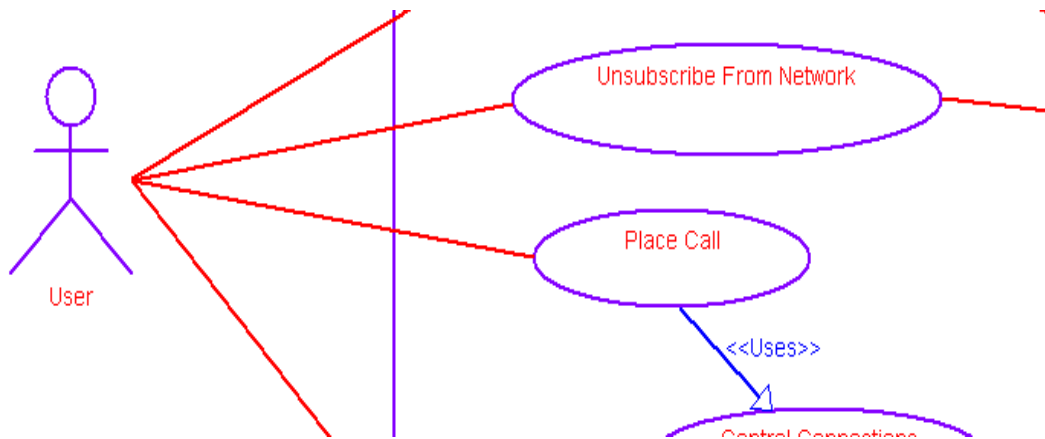
**Figure 1 Use Case Diagram**

The ovals represent the main function points of the system: place call, answer call, subscribe to network, unsubscribe to network, and maintain the system. Note that these function points represent what the system should do; it does not describe how the system should do it. At the requirements level, through a model-based development paradigm, we are graphically capturing the same information on what the system should do as we would have done previously with pencil and paper. Secondly, in figure 1 we also capture the User and the Administrator that serve as actors to the system, in other words entities that interact with our PBX system.

By capturing the system visually, it is easy for anyone in the organization, as well as the customer, to understand the intent of the system. Model based requirements development through UML provides a mechanism for clarity and communication between peers and customers, without having to know the details of the system. Unfortunately this does not yet give us deep look into the scenarios that make up these function points, and does not yet provide a mechanism for visually executing the model so that we can see all the behaviors of each function point.

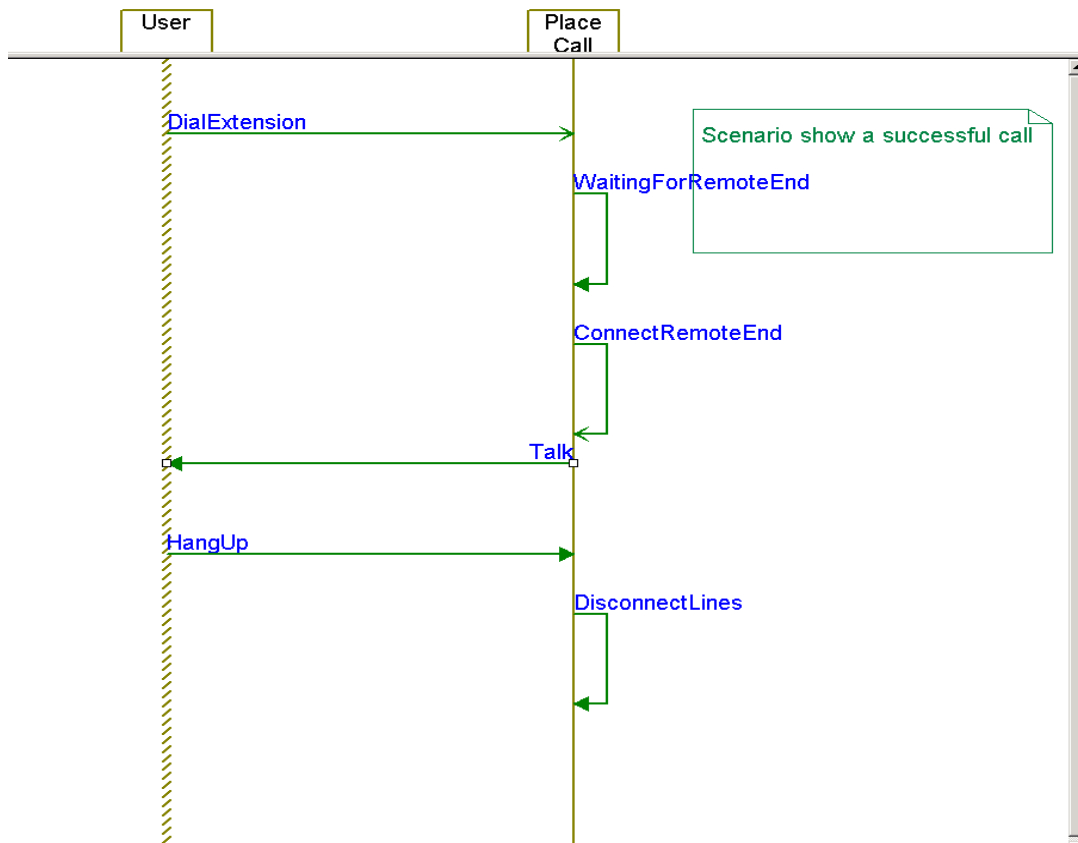
To continue with the model-based approach let's focus on the use case, "Place Call". There may be several scenarios, which we want to describe formally through other means that the UML provides us, so that we can get a better understanding of the

system. The next step within requirements process within MDD technology would be to define the scenarios, either using sequence diagrams, or collaboration diagrams, which give us a sequence flow of the interactions of the typical and non-typical interactions of each use case.



**Figure 2 Place Call use case**

This use case interacts with only the User actor. There are many scenarios, which we now need to define in order for us to visually understand the use case Place Call, because this can consist of many things. The User can place a call and connect with the remote end, the User can place a call but get a busy signal, the User can place a call and not get a line connection, etc. For each scenario we capture this information through a sequence diagram, which is a diagram we use to define a scenario for the requirement Place Call through message flows:



**Figure 3 Sequence Diagram**

You can see that in this diagram, the identified collaborators that realize the scenario of a successful call are the User and Place Call use case. We have not decomposed the system yet, because at the requirements level, we define how the system interacts with the environment as black box. Once we capture all the interesting scenarios, and formal semantics of the functional points of the system are defined, system developers will have enough knowledge to begin system level decomposition to describe the high level design.

The advantage of using model-based development tools in requirements capture, is because these diagrams are more easily understood by a variety of individuals, rather than just those who specialize in software engineering and UML. It is natural to look at a picture description of the system behavior as opposed to reading textual specifications, this is just human nature, but now with executable models MDD can bring these diagrams to life, to better aid in the understanding and capture of the design concepts.

How can this diagram be executed? If we look at figure 3, we have two entities or lifelines, which are receiving and injecting messages. This means that these lifelines have behavior, and through our UML model based tools, we can define behavior formally in one of two ways, using either Statecharts or Activity Diagrams. The semantics of Statecharts define states and transitions, where a transition is taken from

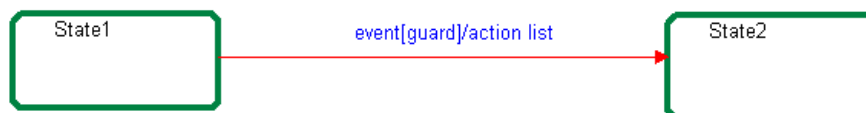
one state to another based upon an event stimulus. This follows the same concept as the classical state machine. Activity Diagrams contain what is referred to as action states, and the transition from one action state to the next is done upon the completion of a set of actions within the state. The difference is that the transition between states is not done necessarily by the event stimulus, but by the completion of the set of actions within the state.

Statecharts and Activity Diagrams are the essence of executable models. The other diagrams allow the capture of requirements in a graphical form. However in order for us to benefit from the value of iteratively executing and learning from models, we need to define the behavioral pieces through Statecharts or Activity Diagrams. If we take a look at the “why this is”, we can see that each state has inputs to states, transitions to and from states, and exit and entry of the states.

All machines (activity and state) have the concept of a default state. For a model to execute, the execution engine must know in which state to initially start in. On the default transition itself, actions can be performed, and those actions are formal and are referred to as an action language. What the action language is depends on what MDD tool you are using, for example, if it is Rhapsody it is simply C, C++, or Ada code. The execution engine must be able to interpret the actions correctly, whether the interpreter is a simulator or an actual compiler. Upon entry of a state of a UML statechart the following applies:

- Perform more actions upon entry of the state
- Perform more actions upon the exit of the state

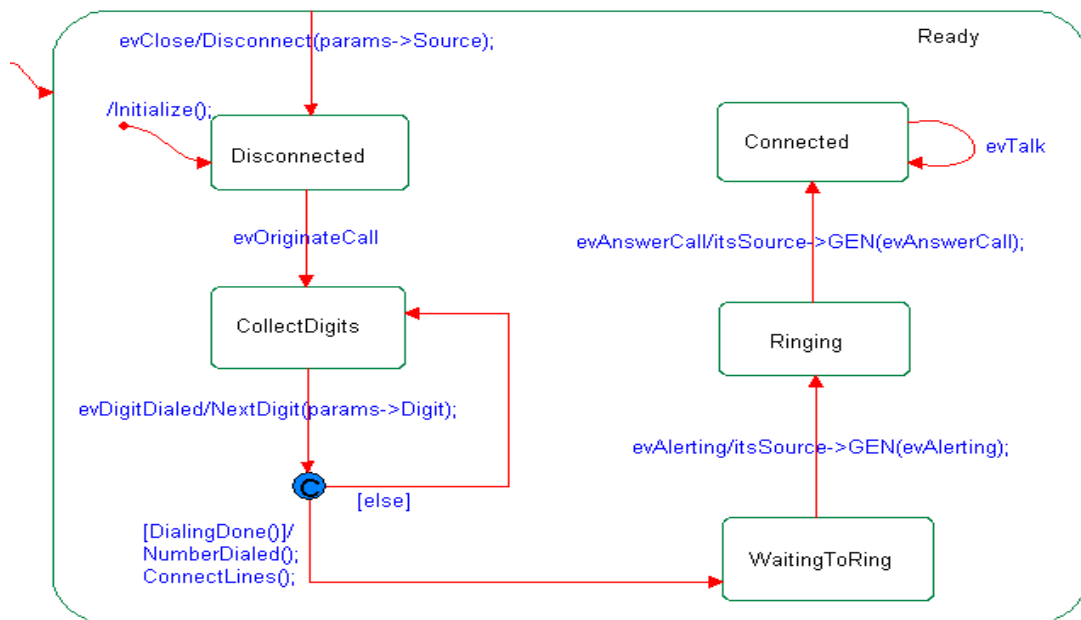
Again this allows the model to execute, since we are formally defining the actions in a language, which can be executed. The semantics on the transitions take the following form:



**Figure 4 Transition semantics**

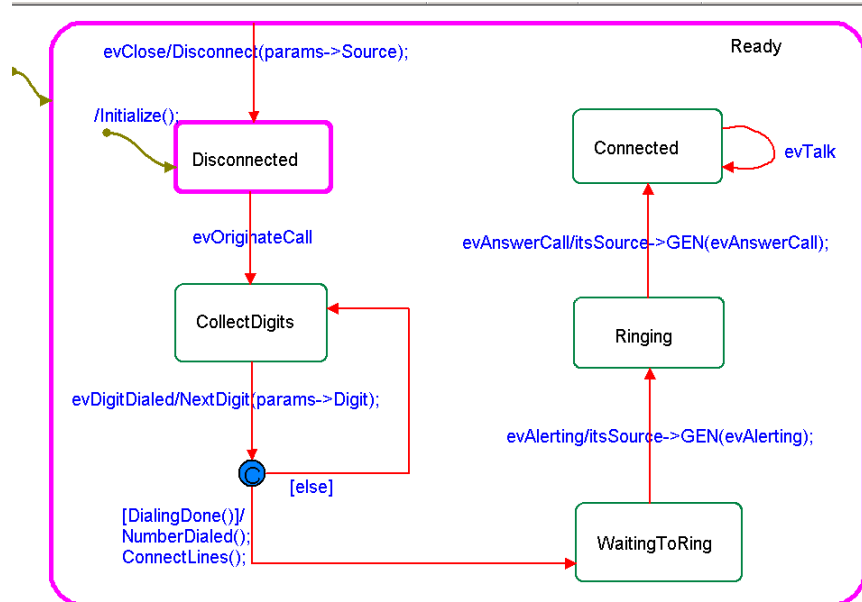
To transition from State1 to State2 in a UML statechart, you need to take a transition. The transition consists of an event (which is synchronous or asynchronous), a guard condition, which is a condition that evaluates to true or false, and a set of actions (code or other language).

You can take these formal semantics and completely define the behavior of a use case using the behavioral semantics of UML statecharts, with a complete set of states and transitions as such:



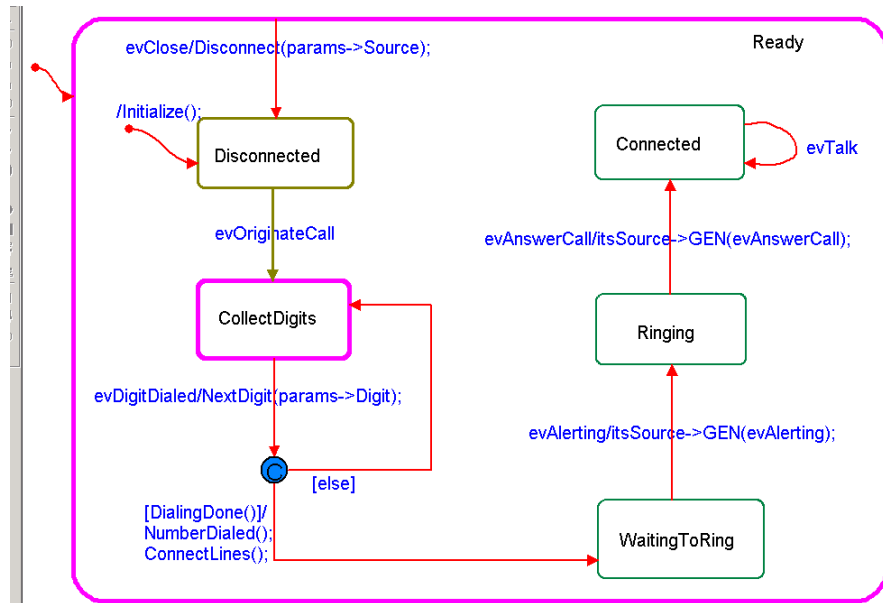
**Figure 5 Formally defined behavior through Statecharts**

The statechart illustrated in figure 5 has a set of well-defined states that have well-defined inputs and outputs. Through the power of executable MDD technology, you can now take this design and simulate the system, to see if the design specification for the use case lifeline meets the requirements.



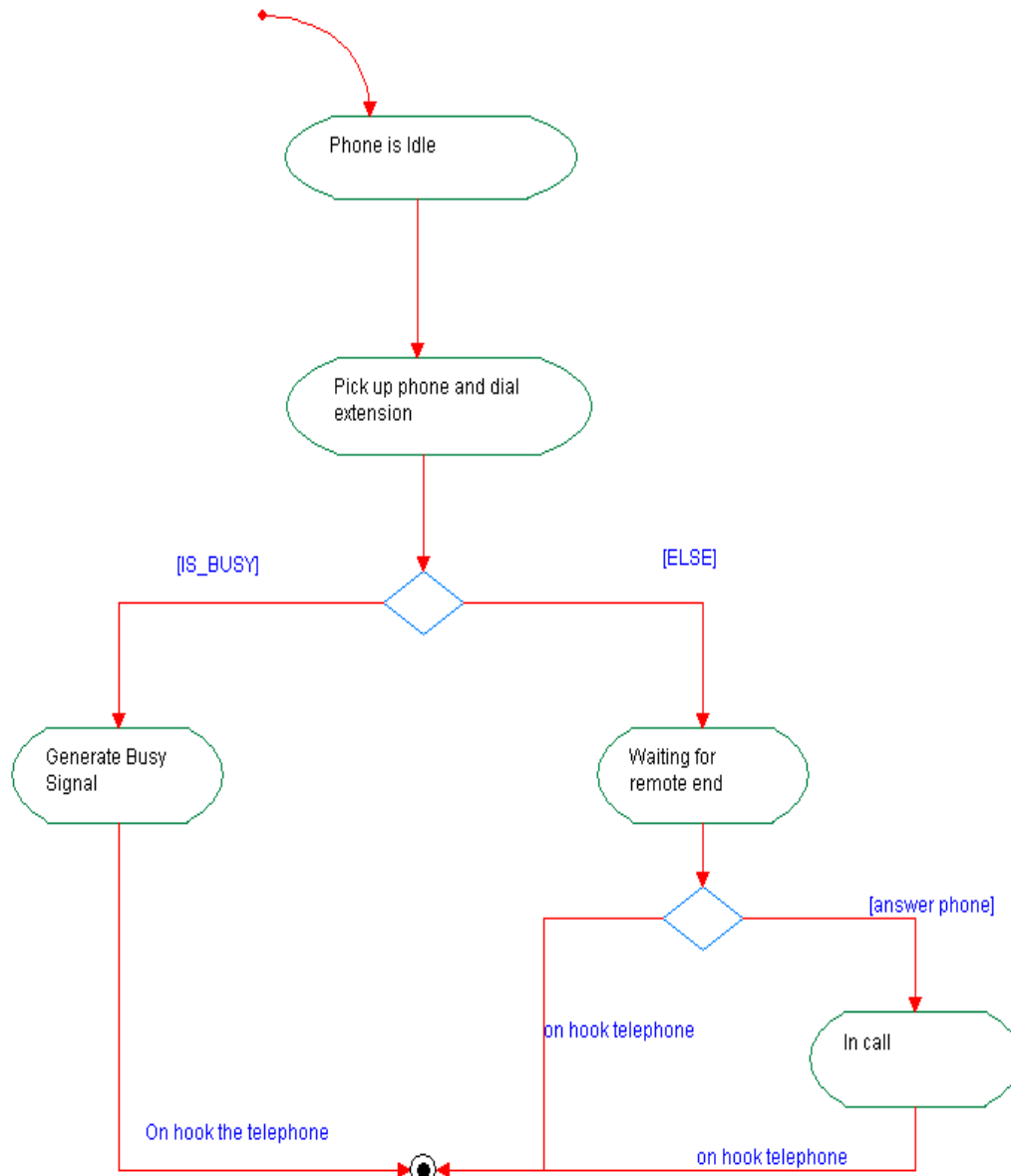
**Figure 6 Executing Statechart**

Looking at the executable mode displayed in figure 6, you can clearly see the system behavior as the model executes. Executable MDD gives you a visual indication of where you are within a certain components behavior of the system. Because we initially drew our specification so that the default state would be Disconnected, shown through the default connector, you can see at runtime the use case is in the Disconnected state which is within the super state Ready. To stimulate the model you can manually inject event stimulus. So if you wanted to check the model by transitioning the use case Statechart from Disconnected to CollectDigits, I would simply inject the event, evOriginateCall, which is defined on the transition.



**Figure 7 Stimulated the model by injecting events**

However at this point, it is not clear what this entirely buys you. The idea of requirements modeling is to capture the system functional requirements, define the relevant scenarios that make up the functional requirements, and define key behaviors of the functions, so that the functions can be modeled within high-level design. Looking back at figure 3, up to that point, we had defined the main function points of the system, and detailed one sunny day scenario of the use case Place Call. If you want to formally define the requirement Place call, you could look at modeling its behavior through a statechart or an activity diagram. The below figure illustrates how one might define the behavior of this use using formal models.



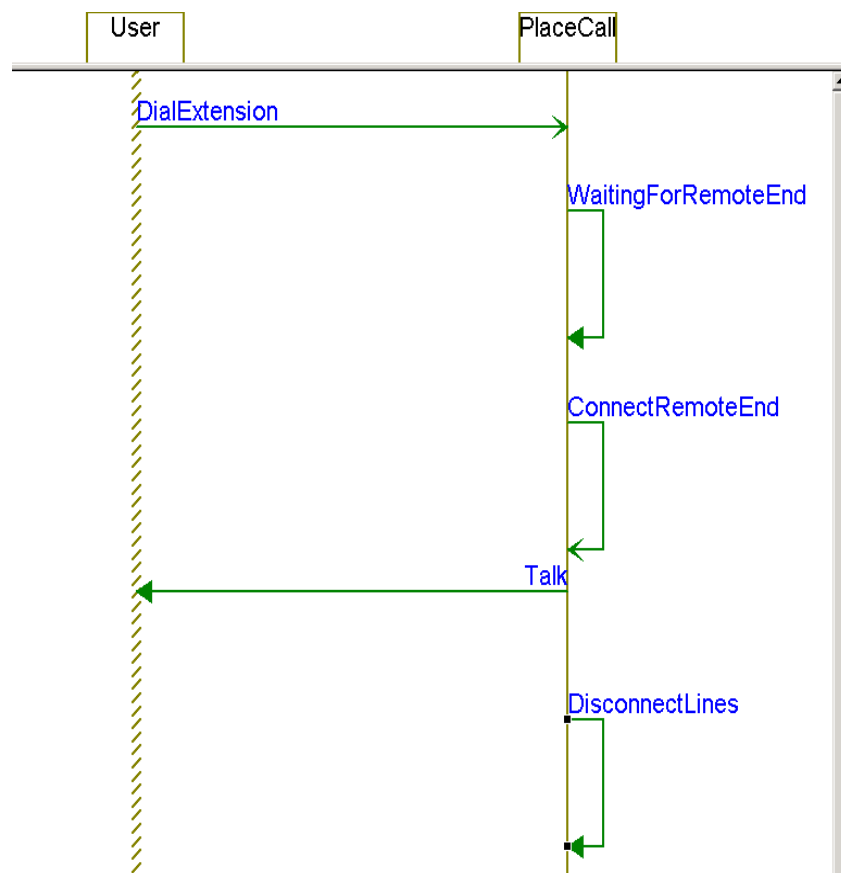
**Figure 8 Activity model of Place Call functionality**

This type of model looks very much like a flow chart. For purposes of this paper, the activity diagram model for this use case is of very simplistic nature, and does not use the fraction of the available semantics available to us through UML 2.0. But reflecting back a moment, in model-based design today, many organizations are formally creating models to capture requirements to a certain point. Describing systems functions with use cases and defining scenarios is not that all of an uncommon practice. However those models are not executable. Typically what many existing technologies support is the notion of textually describing the specification of a use case, either through a word document or other media, which is linked into the design, or the text is typed right within the MDD tool itself. This is a mix of modeling and non-formal methods before high-level design begins. This gets us away from a pure model-driven approach where models are

the artifacts produced in each design phase, and these models are validated via execution.

Figure 8 represents the formal model (the activity diagram) of replacing the text to describe the functional requirement.

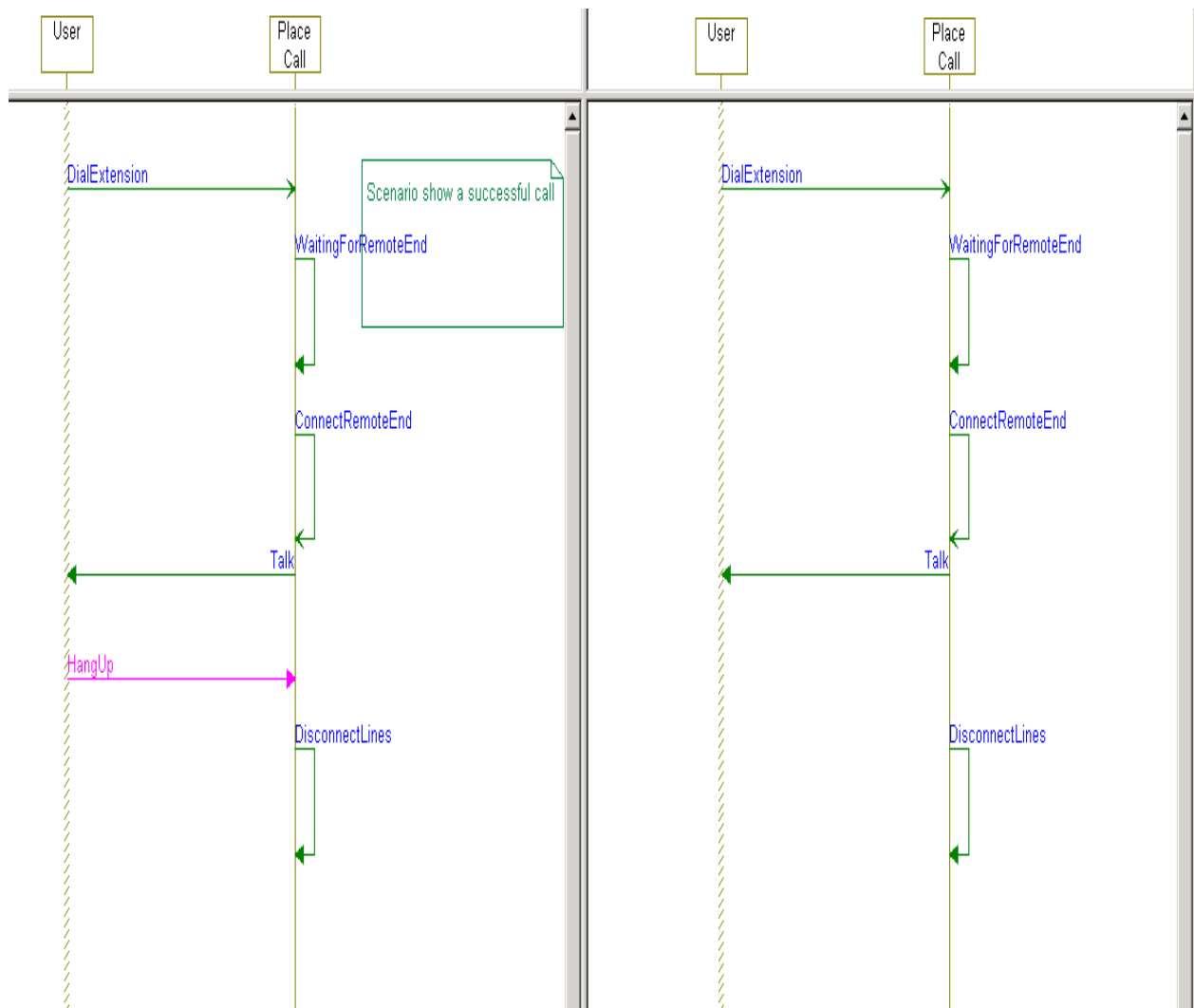
Now that the formal model of Place Call is created, we can execute the PBX system, and observe the lifeline collaboration between the User actor and the use case Place Call. The execution results can give us something similar to what is depicted in figure 9.



**Figure 9 Executed scenario**

This is the execution based upon the formal model created for Place Call. So how do you know it is correct? You know whether it is correct or not based upon the expected scenario shown in figure 3. The scenario in figure 3 is the expected behavior of the system; because this is the way the user has modeled this particular scenario by hand. To understand the functionality of this use case we then modeled the use case with formal semantics and executed our model to see if the formal requirements model

matched the expected behavior. MDD technology for executable models can record and compare the execution results.



**Figure 10 Comparison of execution against expected results**

The captured scenario on the left is the scenario that we defined manually, our requirement. The value on the right is the scenario at which we executed. Rhapsody gives us a color-coding scheme so that we can quickly identify problems within our formal models. We can clearly see that in our formal model, the use case Place call is disconnecting the line before receiving the HangUp signal from the User. We now have immediate feedback back, that the formal model is not designed to specification. With

this MDD technology, you can quickly modify the behavior of the Place Call use case, re-execute the model, and compare the results to see if the problem has been corrected. This is constant validation of the model as you build it.

## Advantages of Model Execution

The case study on executable models was written to show the clear advantage of formal executable models, over a traditional waterfall type document driven approach. Let's do a short recap on how we use model-driven development to quickly receive validation of the requirements through executable models. Rhapsody allows you to define the system requirements using standard UML 2.0 notations. The real power however is being able to execute the formal models, so that the realization of the use case functionality is clearly understood, and formal descriptions of functional behavior can be used to create the high level design of the system.

The clear differentiator here over non-executable models, is that at any point in the modeling of the requirements, you can check your formal models through simulating the system, getting a graphical indication of its behavior. This means you can find the defects in the models as you develop them, sooner rather than in system test. The main advantage is that now you can find the errors early, up front. Since you have an environment in which you can execute behavioral models, and compare the executed results of your scenarios, you naturally introduce an iterative approach to requirements capture. This type of formal methods gives the requirements engineer powerful tools to ensure system concepts are captured appropriately.

As a result of this process, this type of technology's intent is to provide a higher quality product that meets specifications when shipped. The tools that support this technology such as Rhapsody, build on the concept of Model-Driven Development, by allowing models to be formally specified and executed as they are built, but they also allow another type of execution, which makes model-driven development a truly valuable technology, and that is execution for the embedded target system.

The concept behind execution of the system on the target essentially means that the end product of these tools is a binary executable, which can run on an embedded target with a real-time operating system. The intent of these technologies is to ensure that the model is correct by the time they get to the target. Once the target system is available, they can in theory take the same model, which has been validated throughout the development process through iterative execution, and generate the executable model for the target system as the model being developed should be platform independent.

## Conclusion

Through MDD technology, enabling executable models, we can clearly get a grasp of what the system is doing early on, and find flaws in the design that can be immediately corrected. The clear advantage to executable models is that they allow validation of the

system early on, rather than later when it is too costly to find them. MDD technology in its entirety is still relatively new, however the most advanced tools on the market such as Rhapsody, which take advantage of modeling through execution, can even validate and execute on the embedded target system, bringing forth a whole new paradigm of test. This technology can definitely lead to higher quality systems, with faster time to market. The majority of any development budget goes into the testing phase of the product. Executable models enables testing of the system as the system is built, essentially enabling true concurrent engineering, where test is done up front and throughout.

## References

**Simona Bernardi, Susanna Donatelli, Jose Merseguer.** *From UML Sequence Diagrams and Statecharts to analyzable Petri Net models.* 2002

**Bruce Powel Douglass.** *Capturing Requirements for Real-Time and Embedded Systems.*

**Bran Selic.** *Requirements Specification Using Executable Models*

©2004 Telelogic. All rights reserved. Information provided here is subject to change without notice. Rhapsody, StateMate, and iNotion are registered trademarks of Telelogic. Telelogic and the I-Logix logo are trademarks of Telelogic. OMG marks and logos are trademarks or registered trademarks, service marks and/or certification marks of Object Management Group, Inc. registered in the United States. Other products mentioned may be trademarks or registered trademarks of their respective companies.

## I-Logix

### **Telelogic.**

3 Riverside Drive  
Andover, MA 01810  
Tel: 978-682-2100  
Toll Free: 888 8ILOGIX  
Fax: 978-682-5995  
E-mail: [info@telelogic.com](mailto:info@telelogic.com)  
<http://www.telelogic.com>

### **European Headquarters Telelogic UK Ltd.**

1 Cornbrash Park  
Bumpers Way  
Chippenham  
Wiltshire SN14 6RA  
England  
Tel: +44 1249 467-600  
Fax: +44 1249 467-610  
E-mail: [info\\_euro@ilogix.com](mailto:info_euro@ilogix.com)