Graphical Programming Using UML and SDL

The author describes a two-language merger that combines UML's expressive power and SDL's coherence/semantics strengths to provide a modeling paradigm for visual software engineering that is more effective than either language alone.

Morgan Björkander
Telelogic AB

Since its introduction a few years ago, the Unified Modeling Language (UML) has captured industrywide attention for its role as a general-purpose language for modeling software systems. Although it does a good job in the early phases of the development process, UML does leave some things to be desired in the systems design and implementation phases because it is lacking in structural and behavioral constructs. We propose a merger between UML and the Specification and Description Language (SDL) that would enhance UML's usefulness as a graphical programming language.

SDL, a language that models the architecture and behavior of event-driven, distributed systems in real-time environments, originated as a specification language within the telecommunications industry approximately 20 years ago. Today, SDL is often used as a full-blown programming language. Although UML is headed in a similar direction, combining the two languages provides a modeling paradigm for *visual software engineering* that is more robust and effective than either language alone.

From a language standardization perspective, this kind of bootstrapping process spans international standardization organizations, including the Object Management Group and the International Telecommunication Union. The individual maintenance of these languages influences their evolution to the extent that both languages benefit. Also, combining the two languages seamlessly leads to the natural conclusion that, at some point, it will be impossible to distinguish between them.

COMBINING UML AND SDL

A modeling language must evolve to keep abreast with technological changes. This developmental process can occur either in isolation or by building cumulatively on other modeling language results. As a language, SDL[1] is more coherent than UML[2] and also has a more sound semantics foundation. On the other hand, UML is far more expressive and widely accepted. The objective for combining the two languages is to permit the expressive power of UML to coalesce with SDL's strengths of coherence and semantics.

To combine the languages, we use UML's extensibility mechanisms to create profiles. These profiles are primarily intended to tailor UML toward a specific domain by giving modelers access to common model elements and terminology from that domain. However, we can also use profiles to create language mappings, each of which we can then use as an intrinsic part of UML. After creating the appropriate profile, it would be fair to say that SDL *is* UML.

SDL revisions

The latest SDL revision takes into account the alignment of UML and SDL, particularly with respect to annotating models. Modeling passive data graphically using class diagrams is the most important addition; other additions have, however, further improved SDL's visual aspects. Most graphical symbols have additional stereotyped versions that bear UML's look and feel, while a large part of core UML is incorporated directly into SDL, along with parts of UML model management.

0018-9162/00/$10.00 © 2000 IEEE

The International Telecommunication Union Recommendation Z.109 titled "SDL Combined with UML"[3] incorporates the principal work on combining these languages. This recommendation specifies the UML subset that has a semantics meaning in SDL. UML programmers can use this subset, a common part of any UML profile, to create a model that they can directly interpret as SDL. Ordinarily, this capability by itself is too limiting to be useful; therefore, modeling virtually all of SDL using UML becomes feasible only when we combine it with proper SDL modeling constructs. The ITU recommendation already includes some profile parts, and others are expected to follow suit.

UML action semantics

The work on precise action semantics for UML[4] is ongoing, and includes a mapping to SDL. The action semantics provide definitions of actions that can occur in state machines or class operations that are more detailed and precise than the definitions currently available in UML.

The intention is to create an abstract behavior specification at a detail level that enables the execution and verification of UML models. Action semantics specify, for example, what it means to make an assignment and how an if-statement works. However, using action semantics in concrete programming is difficult because the action semantics do not give a notation. Also, the action semantics intentionally do not define the execution engine required to drive the specified behavior.

Notation is omitted from the action semantics for several reasons—most importantly, it would be nearly impossible to agree on a common notation. We expect tool vendors to create mappings or profiles to concrete programming or action languages to make the action semantics generally available—in effect, turning UML into a de facto programming language. Developers have created one such mapping for SDL.

In addition to the definition of an SDL profile for UML and the work on action semantics, the SDL community is actively participating in developing UML 2.0.[5] Structure modeling plays a central role in this work, as does behavior modeling in the form of state machines and interactions—namely, sequence diagrams.

HIERARCHICAL STRUCTURE DECOMPOSITION

One essential advantage of SDL compared with UML is that SDL can perform hierarchical decomposition of a system's internal structure to easily model arbitrarily complex structures.

Creating a profile

The basic mechanisms for creating a UML profile include stereotypes (a key construct), tagged values, and constraints. In its simplest form, creating a UML stereotype is akin to renaming an existing UML model element and using it as a base for the new model element. The most frequently stereotyped model elements are *class* and *classifier*. The stereotype provides a model element that relates directly to the problem domain. Tagged values, like attributes, extend the stereotype's properties. A constraint, on the other hand, restricts existing properties, for example, by setting an attribute to a specific value or by disallowing certain associations.

To define a context for the stereotypes that are used as part of a profile and show how they interact with each other, we use a virtual—or conceptual—metamodel. Although this is not a proper UML metamodel, for all practical purposes we can use it as if it were. As Figure 1 shows, this conceptual model uses the agent and gate as UML constructs with the same standing as, for example, class. Further, each stereotype appropriates all properties of its base model element, including its attributes and associations.

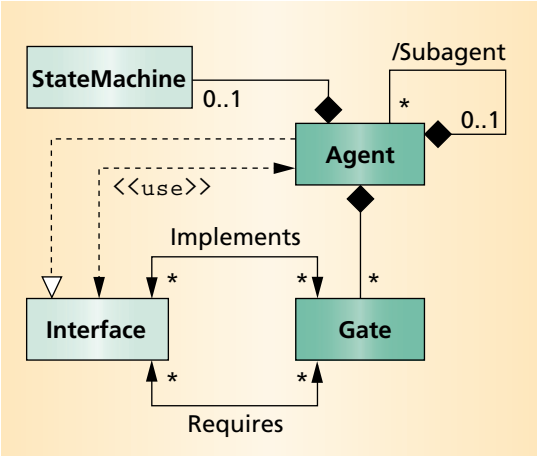

Figure 1. Part of an SDL conceptual metamodel. The model uses agent and gate as UML constructs. In SDL, the agent is a logical component with bidirectional interfaces and can be viewed as a black box encapsulating structure and behavior. The agent's behavior is reflected through the interfaces it implements and, to some extent, by the interfaces it requires. Inside the agent, a state machine delegates its behavior to an internal substructure of nested agents.

Logical components and interfaces

In SDL, the agent is essentially a logical component with bidirectional interfaces. By far the most important construct for modeling structure, we can view the agent as a black box encapsulating structure and behavior. The agent's behavior is reflected through the interfaces it implements and, to some extent, by the interfaces it requires. A state machine gives the agent's behavior or it delegates the behavior to an internal sub-

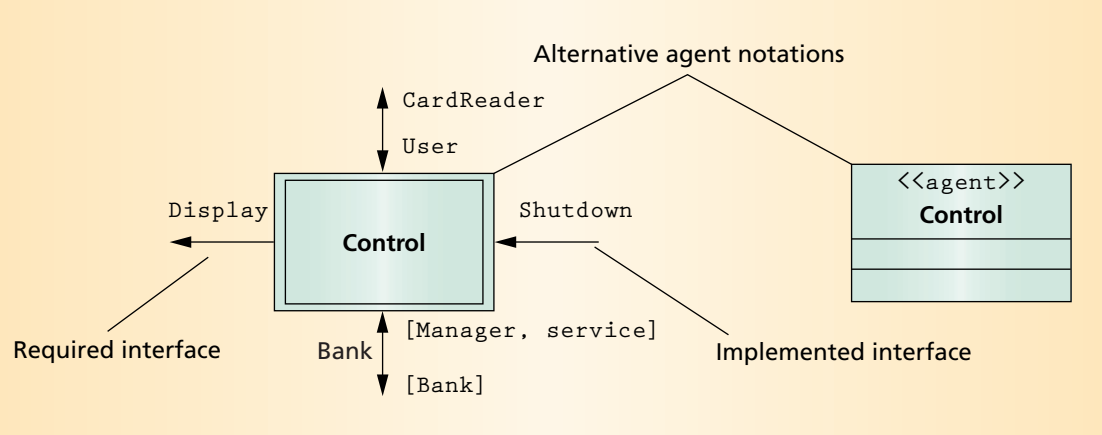

Figure 2. Gates support bidirectional interfaces that specify contracts between agents. SDL only allows communication between pairs of implemented and required interfaces. Each gate can implement several interfaces. The lollipop notation ordinarily used for interfaces in UML corresponds with the instance when a gate only implements one interface. The arrows emphasize the gate's role in establishing communication paths and give a visual cue about what is happening. Using gates on agents makes each agent self-contained.

structure. The internal substructure can be hierarchically decomposed of other agents, but a state machine always implements the innermost agent.

This kind of hierarchical decomposition is ideal for creating arbitrarily large, complex systems. From a process point of view, developers can either decompose logical components into more fine-grained agents in a top-down approach or use them as building blocks to form even larger building blocks in a bottom-up approach.

The profile defines agent as a stereotype of the UML model element *class*, but it constrains agent so that it is always active. However, because class does not have sufficient attributes to model an agent, the profile adds several tagged values to the stereotype. One such tagged value is *kind*, which specifies the internal substructure's runtime characteristics, indicating whether the contained agents will execute concurrently. Another tagged value is *virtuality*, which indicates how an agent can redefine another agent during specialization.

Gates support bidirectional interfaces that specify contracts between agents. Contracts are an important part of the structure; the profile allows communication only between pairs of implemented and required interfaces. Each gate can implement several interfaces, which must be realized by the agent owning the gate. The gate also can require other agents to implement interfaces. Figure 2 shows an example of the notation for agents and gates. We use this notation to assign alternative icons to stereotypes to make them stand out and to make the diagrams less cluttered.

The lollipop notation ordinarily used for UML interfaces corresponds with the special case in which a gate implements only one interface. However, we do not recommend this notation because it is not suitable for showing the direction of interfaces. Instead, we use arrows to emphasize the role of gates in establishing communication paths and to provide an immediate visual cue to what is happening. We can easily imagine the sample agent interacting with other agents only hinted at in the graphics. Using gates on agents makes each agent self-contained because the agent does not need to know anything about the environment in which it works; the interfaces contain all the information the agent needs.

Runtime structure

The runtime structure describes how agents interact with each other. An agent can contain a state machine and a substructure consisting of other agents. To see how the state machine and the agents it contains are interconnected—the roles they play as part of the agent—we need to look beyond the agent's black box properties to see its inner contents. Figure 3 shows an agent's internal structure. In this case, we disregard the state machine's behavior except for the way it interacts with and controls the agents that form part of the substructure.

BEHAVIOR VISUALIZATION

Treating a modeling language as a programming language requires providing a level of detail that currently does not exist in UML. The work on action semantics is a necessary first step toward providing this kind of detail.

A transition-centric view

In UML's action semantics, the core model element is the *procedure*, which consists of several actions. In most programming languages, these actions corre-

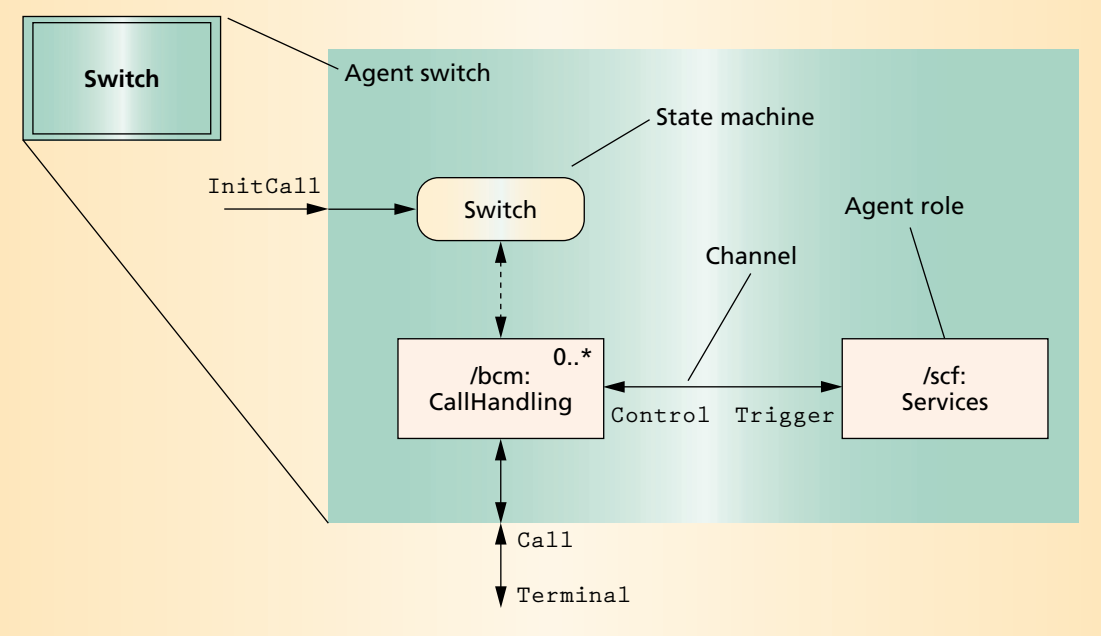

Figure 3. An agent's internal structure. The state machine interacts with and controls agents in the substructure. The bidirectional interfaces on the gates establish connection points between communication paths (channels). The dashed line is a dependency that indicates instance creation, where client instances can create supplier instances. Gates and their owning agents play different roles depending on the context. An agent either uses object references based on interfaces to establish communication or relies on the connection paths to transmit messages.

spond with a sequence of statements, but in UML, sequential execution is not considered a certainty. A procedure always has a context. For example, the procedure can represent the body of a class operation or a state transition. The semantics then specify what happens when these actions execute. The action semantics specifically (and intentionally) do not include the proper notation that SDL provides.

UML state machines do not give transitions the attention they deserve, which is reflected in the lack of a good notation for their actions. Because UML statecharts are, by and large, state-centric, they tend to overlook what happens during transitions. In SDL, the situation is almost the reverse—SDL has a transition-centric view that focuses on actions. These two views are remarkably complementary; UML's state-centric view provides an excellent overview, and SDL's transition-centric view delves into the details while it preserves the context.

Here, we focus on procedures as part of transitions, but the principles are identical for procedures that are part of operations. Figure 4 maps SDL's actions on top of UML's action semantics. A transition consists of several statements enclosed by different kinds of symbols. These symbols carry information about the actions and have semantic meaning to help us visualize what happens during a transition. Most symbols depict a single statement, such as inputs, outputs, loops, decisions, and operation calls. A generalized task symbol that has its own scope can encompass any number of actions, but it usually represents assignments.

Some symbols are recognizable from UML, which uses them in activity diagrams. The input symbol denotes a signal (or operation) that triggers a transition; it can also include a list of the parameters the signal carries. The task symbol denotes an assignment, whereas the output symbol indicates a message that an agent is sending to the specified object. The save symbol specifies signals that the state machine should defer, that is, the state machine should handle the listed signals in subsequent states. The asterisk in the save symbol represents a catchall for signals and operations the state does not explicitly trigger. The symbols interact with actions to increase the awareness and readability of the actions that are part of the transitions.

Other considerations

Further topics for consideration in combining SDL and UML include specialization of transitions, graphical representation of standard and user-defined exceptions, and the use of templates for data and structures. Another area for consideration is the data model, including the data types available to the programmer and how the model resolves expressions.

The mapping to the action semantics handles some of these considerations. Mapping the languages on

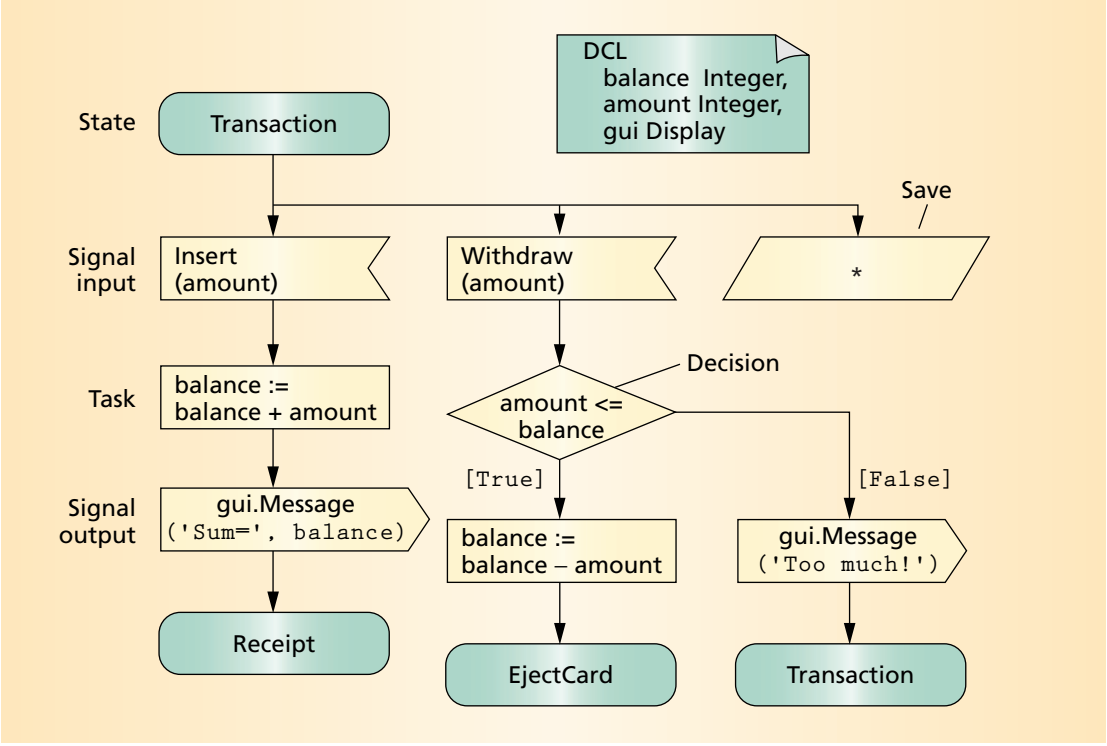

Figure 4. Transitions with actions. A transition consists of statements enclosed by symbols bearing information about actions and semantics meaning. The input symbol denotes a signal or operation to trigger a transition and can include a parameter list; the task symbol usually denotes an assignment; the output symbol indicates a message being sent to the specified object; the save symbol specifies signals that should be deferred; the asterisk in the save symbol represents a catchall for signals and operations not explicitly triggered in the state, and indicates that these should be handled in subsequent states. The note symbol contains declarations of attributes (variables) and is prepended with the keyword DCL to distinguish it from ordinary comments.

top of the action semantics addresses other aspects of these considerations. The border between a modeling language and a programming language is not that obvious. But when we add a sufficient amount of detail to a modeling language, it begins to look and act like a programming language.

CONSEQUENCES OF GRAPHICAL PROGRAMMING

Programming's ultimate goal is to create applications. Accepting that we can use a modeling language as a programming language may present a mental hurdle, and mistrusting the code that this produces is not an uncommon reaction, however unfounded such a reaction might be. But the fact remains that this concept offers several advantages.

Modeling applications

Three approaches currently dominate the way programmers use UML to model and create applications:

- They model the system in UML, then they use their favorite programming language to manually implement it. In this case, the programmers usually discard the model once the actual code exists.
- They model the system in UML, then they use their favorite programming language to generate stub code. Tools usually provide mechanisms for synchronizing the model with the code if changes are made—called "round-trip engineering." If they don't use synchronization, programmers sometimes store the model for documentation purposes, but they may still discard the model after they generate the stub code.
- They model the system in UML and use specialized abstractions such as signal sending to provide the implementation, usually in C++, directly. This approach is similar to stub-code generation, except that the programmer either hides the added code behind transitions, which leads to fragmented programming, or displays the code alongside transitions, which leads to cluttered diagrams. Because there are intimate relationships between the code and the model, the model is useful even after the application is completed.

Application creation

Using the SDL approach in which we program directly in the modeling language goes one step further than the *target-language-dependent* approach, in which programmers model the system in UML and then use specialized abstractions to provide the implementation. With this approach we can interpret or compile the program just as with other programming languages, making the system *target-language-independent*.

We could compile the program directly after ensuring that the system is syntactically and semantically

correct; however, a more flexible approach uses ordinary programming languages like C++ and Java as *intermediate formats*. Rather than making changes in the generated code, we can integrate external code or libraries with the system or use a specific cross-compiler. We can use the appropriate code generators to transform the same model into any target language without making any changes. We perform *debugging* at the modeling-language level.

We can also access the debugging information at the modeling language level if we use an intermediate format, but this is usually only necessary if we suspect that the generated code is broken. This is in contrast with the target-language-dependent approach, where we debug at the target-language level rather than at the modeling level.

Although round-trip engineering is no longer an issue, programmers have to perform *reverse engineering* when they access data structures such as predefined C++ libraries. When we program in C++ or Java, if we don't create proper abstractions in the programming language, we must be explicit about whether we are using CORBA or COM to handle functions such as distribution. However, by raising the abstraction to the model level, we can disregard these considerations—which are best left for the compiler or code generator to resolve—and focus instead on the application's behavior. For example, we can simply change the execution engine to use TCP/IP without making any changes to the program's core functionality.

Developmental process influences

In the paradigm shift we describe, analysts, designers, and programmers all speak the same language. This ability to communicate reduces the differences between their roles in the development process and helps to remove the natural barriers that moving from the model domain to the code domain inadvertently introduces. This improved communication makes relying on quick iterations to test ideas more attractive.

Code is generally considered king, and the model is regarded as its poor country cousin. The concept we propose offers two advantages because the model *is* the code, and it also requires fewer artifacts. An additional advantage is that it isn't necessary to synchronize the model and the code because they are essentially one and the same.

SDL has been thriving as a programming language for nearly ten years, providing irrefutable proof that we can program in a modeling language. UML is slowly but surely heading in a similar direction. We demonstrate how merging the two languages can provide a superior modeling paradigm for visual software engineering. ✷

Acknowledgments

Our results are based on standardization work performed within the Object Management Group and the International Telecommunication Union.

References

1. ITU-T, ITU Recommendation Z.100: "The Specification and Description Language (SDL)," ITU, Geneva, 2000.
2. OMG, "OMG Unified Modeling Language Specification," Version 1.3, OMG, ad/99-06-08, June 1999.
3. ITU-T, ITU Recommendation Z.109: "SDL Combined with UML," ITU, Geneva, 2000; http://www.itu.int/itudoc/itu-t/rec/z/index.html.
4. OMG, "Action Semantics for the UML, Request for Proposal," OMG, ad/99-11-01, 1998; http://www.omg. org.
5. OMG, "UML 2.0 Superstructure, Request for Proposal," ad/2000-09-02, 2000; http://www.omg.org.

Morgan Björkander *has an MSc in computer science from the Lund Institute of Technology and is working as a methods specialist at Telelogic AB. He is currently involved in the development of UML within the Object Management Group. Contact him at mbj@telelogic.com.*

REACH HIGHER

Advancing in the IEEE Computer Society can elevate your standing in the profession.

Application to Senior-grade membership recognizes

✔ **ten years or more of professional expertise**

Nomination to Fellow-grade membership recognizes

✔ **exemplary accomplishments in computer engineering**

GIVE YOUR CAREER A BOOST

UPGRADE YOUR MEMBERSHIP

computer.org/join/grades.htm