Software-platform-independent, Precise Action Specifications for UML

Stephen J. Mellor

Project Technology, Inc.
steve@projtech.com

Steve Tockey

Rockwell Collins, Inc.
srtockey@collins.rockwell.com

Rodolphe Arthaud, Philippe Leblanc

Verilog, SA
arthaud@verilog.fr
leblanc@verilog.fr

Abstract

This paper explores the idea of complementing the UML with a compatible mechanism for precisely specifying action semantics in a software-platform-independent manner. We believe that this would bring significant benefits to the community, such as:

Early Verification. Precise action specifications can be used to perform specification-based simulation and formal proofs of correctness early in the software lifecycle. Problems detected early can be removed with much less rework, leading to a reduction in both project cost and time-to-market.

Domain Level Reuse. With appropriate tooling, software-platform-independent system specifications can be mapped into multiple different implementation technologies at significantly reduced cost.

This paper will explore the nature of such precise action specifications and the rationale for these benefits.

Keywords

UML, Action Specification, execution

The Problem

The UML [UML1, UML2] is a rich and powerful language that can be used for problem conceptualization, software system specification, as well as implementation. The UML also covers a wide range of issues from use cases and scenarios, to state behavior and operation declaration.
However, the UML currently uses uninterpreted strings to capture the description of the behavior of actions and operations. To provide for sharing of semantics of action and operation behavior between UML modelers and UML tools, there needs to be a way to define this behavior precisely, and in a standardized language. Further, action specifications should not be overly constrained by programming-language specific details.

Why Software-platform-independent, Precise Action Specifications?

Model precision and level of implementation detail are two separate things. Software-platform-independent, precise action specifications, in conjunction with the UML, can completely specify a computing problem without actually programming it.

Such precise action specifications, in conjunction with the UML, can be used to build complete and precise models that specify problems at a higher level of abstraction than a programming language or a «graphical programming» system.

Such precise action specifications can support formal proofs of correctness of a problem specification.

Such precise action specifications, in conjunction with the UML, make possible high-fidelity model-based simulation and verification.

Such precise action specifications enable reuse of domain models, because each domain is completely specified, though not in a form embedded in code.

Such precise action specifications, in conjunction with the UML, provide a stronger basis for model design and eventual coding.

Such precise action specifications, in conjunction with UML, could support code generation to multiple software platforms.

Why Not Use an Existing Programming Language?

To be useful, precise action specifications should be abstract, so that the modeler can state the behavior minimally without duplicating implementation information. Because the precise action specification should take a view focused on the high level policies in the domain, it should employ only a restricted conceptual subset of the UML. This issue is explored below in the context of existing programming languages.

To be useful, precise action specifications should allow for smooth incorporation of executable code, so that tools can map actions and operations into code efficiently. This issue is explored below in the context of declarative languages.

To be useful, precise action specifications should take a perspective that is precise and detailed enough to specify the policy and the high level algorithms of a system unambiguously, but without requiring the user to make any decisions about the structure of the software. This issue is explored below as Software-Platform Independence.

Why Not Use An Existing Language?

Existing programming languages already provide a way to precisely specify actions and operations at the implementation level. Why not use one of them and avoid inventing what is effectively "another language"?

Concepts such as Class, Package, and Exception exist in both the UML and in many object-oriented languages, but existing programmin languages provide much, much more than is really needed for precise specification of actions. Consequently, any such action specification language would have to be a subset of an existing programming language.

At the same time, existing programming languages limit implementation options. For example, Java has only one way to represent associations, namely object references in one, the other, or both of the associated classes. An appropriately abstract action specification must be based on the meaning of the association rather than its implementation.

Similarly, existing programming languages do not support directly many UML concepts such as Association or State. A precise action specification language should support directly UML concepts that are appropriate at the level required for a system specification.

Existing programming languages also have serial, sequential execution models, while a precise action specification language should define minimally constrained execution. Note that this is conceptually equivalent to allowing for concurrent execution within an action, though in practice, it merely provides the designer with the information required to determine all possible orders of execution.

Finally, since we are concerned with a "standard" way of precisely specifying actions so that complete models can be exchanged between modelers and between tools, which existing programming language should be chosen? Surely there would be little agreement on which programming language should become the UML standard for precise action specifications.

Why Not Use Declarative Specifications?

Declarative specifications, such as the Object Constraint Language [OCL] already included in UML, can be adequately precise without overly constraining implementation. UML actions and operations can be specified in terms of precise pre-conditions and post-conditions. Declarative specifications allow the ultimate freedom in implementation independence.

Declarative action specifications are entirely sufficient for some modeling needs and we strongly recommend their use where appropriate. But there is often a need to include some level of algorithmic specification to ensure efficient execution. Consider, for example, a pre-condition of a collection of values and a post-condition that the collection of values be sorted in ascending order. It is possible, of course, to implement this by building a list of all possible permutations of the collection and select the permutation that is properly sorted, but this solution is hardly practical for real systems.

Consequently, it is desirable to be able to supplement a specification stated abstractly using pre- and post-conditions with the description of an algorithm that implements it efficiently, without affecting the remainder of the system specification.

Software-Platform Independence

Software-platform independence is analogous to hardware-platform independence. A hardware-platform independent specification must be executable on a variety of hardware platforms with no change. Similarly, a software-platform independent specification must be executable on a variety of software platforms, or designs, with no change. For example, a software-platform independent specification should be able to be mapped into a multi-processor/multi-tasking CORBA environment, or a client-server relational database environment with no change in the specification.

When the concepts Customer and Account exist in the problem domain under analysis, they can be modeled in UML on a Class Diagram. The vocabulary of UML, including the name of class diagram, suggests that the software solution should be expressed in terms of classes named Customer and Account. But there are many possible software designs that can meet these requirements, many of which are not even object-oriented.

This goal of software-platform independence suggests several general implications.

The precise action specifications must enable the generation of a system with a different structure from the model. The organization of the data and processing implied by a conceptual model may not be the same as the organization of the data and processing implied by the model in implementation. For example, between concept and implementation an attribute may become a reference; a class may be divided into sets of object instances according to some sorting criteria; classes may be merged; or split; state charts may be flattened, merged or separated, and so on. The action specifications should be at a level of abstraction that enables such model reorganization.

The precise action specifications must enable reorganization of fundamental problem-oriented computation. Because the conceptual model of a problem may be implemented in a variety of ways, the precise action specifications must not allow the user to specify computation in a manner that depends on assumptions about data access and organization. Consider, for example, a problem in which we compute the monthly interest for all accounts belonging to a subset of customers. One way to define this is to build a double loop that iterates over customers, searching for ones that qualify, then iterates over those accounts, adding some percentage to the balance. This approach assumes a certain data organization, and it would be dreadfully inefficient using some other data organization. The fundamental, problem-oriented computation is simply the interest computation applied to the relevant subset of accounts, all else is concerned with managing the data organization.

One approach to enabling reorganization of fundamental problem-oriented computation is to separate such computation from data access, and vice versa. This approach does not embed the specification of data access or control structure within any computation, but instead places data access and control structure outside the computation.

This approach also suggests that computation specifications should be context-free. A context-free computation is one that has no side-effects and no (internal) state memory. It is a function in the mathematical sense of the word.

The approach also suggests that collections of object instances or collections of data values should be treated as a unit. In a specification, certain collections may be identified as fundamental to the problem; these necessary collections often become the basis for optimization in the implementation.

In short, the precise action specifications should avoid structures that inhibit mapping the problem specification into implementations with different organizations.

Examples

It is well beyond the scope of this paper to define a standard for software-platform-independent, precise action specifications. Indeed, it is the intent of this paper to describe how such a standard would be a valuable addition to the UML, thus setting the stage for such a standard to be developed. However, some examples of precise action specifications can be useful in understanding the issues.

Precise action specifications based on classical third generation languages (such as C and C++), leave much to be desired. The fundamental problems are (1) that they are too low-level and (2) provide too much power and choice. On one hand, they require the analyst to over-specify some aspects of an action (for example, statements in these languages are generally executed sequentially); on the other, constructs are provided for for loops, if statements, switch statements that embed computational code within control structures, inhibiting reorganization of the model.

Consider, for example, the following third-generation style action specification:

```
// In the context of a DogOwner named myDogOwnerID, find all the owned dogs
Select many dog from instances of object Dog 'owned by' myDogOwnerID;
For each dogID in dog
    // Generate a UML signal to the StateChart for each Dog
    Signal D1: ComeToDinner( ) to dogID;
    dogId( Weight ) := ProportionalIncrement( dogID( Weight) );
End for;
If myDogOwnerId.FoodLeft < SafeAmount then
    Signal DO4: BuyMoreFood to myDogOwnerID;
End if;
```

Some of the problems here are:

Because of the sequential nature of the specification, control has been over-specified. There is no apparent reason why the if clause should be after (or before) the for loop.
The for loop is a general structure into which we could place any number of statements. In the example, two unrelated analysis ideas have been related by being placed in the for loop. This obscures the fact that such statements may, in fact, constitute reusable processes. The if statement has the same problem: unrelated analysis thoughts can be placed in the body of the statement.

By examining other specification fragments one can find additional problems. Perhaps the best summary conclusion is this: Action specifications based on third-generation languages tend to produce only a thinly disguised form of the implementation, and do not provide the level of abstraction needed for clear, complete analysis.

One alternative approach is to base software-platform-independent, precise action specifications on data and object flow. This approach has the advantage that data access and computation are completely separated, and therefore that computational code is not embedded in control structures.
This approach matches, to some extent, both the Activity Diagram defined as a part of the UML, and the Shlaer-Mellor [SM] approach of constructing a data flow diagram for each action.

Here are some general properties of this kind of precise action specification:

Each chain of computations, connected by object flow, is a statement. Each statement would be much like Unix pipes and filters.

Within each statement, data is viewed as being active and flowing. No distinction is made between collections and single data values.

The details of the computations are defined separately from the body of the action. We do not provide here any examples of this. It could
be in pre- and post-conditions, an imperative specification, or an existing programming language constrained to prohibit data access.

Execution proceeds in parallel1 for all statements, except where constrained by data or control flow. Consequently, if two statements write
the same variable and their order of execution has not been constrained, it is indeterminate which value will be used in subsequent
processing.

Guards can be set and then used to initiate (or not initiate) the execution of a chain of computations.

Execution of the action terminates when the last statement that can execute has completed.

[1] The statements proceed - in parallel - from the analysts perspective. The architecture can serialize the statements as desired for optimization or other purposes.

We can now examine the dog-feeding example expressed in an object flow and data flow fashion:

```
//    In the context of a DogOwner named myDogOwnerID, find all the owned dogs,
//    pipe that collection of Dogs to a process that generates a signal to each
      myDogOwner.OwnedBy | Signal D1: ComeToDinner;
//    In the context of a DogOwner, get the weight of each, pipe that to a
//    process that computes the new weight for each member of the collection
//    and write it back using the > operator. The Dog( ) refers
//    to the set of Dogs found by the expression myDogOwner.OwnedBy.
// That expression is then dereferenced by .Weight.
// The computation ProportionalIncrement is defined separately.
      myDogOwner.OwnedBy.Weight | ProportionalIncrement >Dog( ).Weight;
//    Get the FoodLeft attribute of the owner, and the variable SafeAmount,
//    pipe the pair to a test process, TestLess, which has two guard conditions
      (myDogOwner.FoodLeft, ~SafeAmount ) | TestLess? !Less, !GreaterEqual;
//    If the Less guard is set, pipe the DogOwner object instance
//    to a process that generates a signal
      !Less: myDogOwner | Signal DO4: BuyMoreFood;
```

A proof of concept

We will illustrate the potential benefits of software-platform-independent, precise action specifications through the example of SDL and MSC [Z100, Z120], a language defined by the ITU. SDL is widely used in the telecommunications industry and is also used for the development of embedded systems. The approach described here has been used successfully in ObjectGEODE [VE] for the design, validation and fully automatic generation of code running in cellular phones, satellites and smart cards.

SDL is similar to the UML in that it describes a system in terms of interacting processes owning state machines and exchanging messages. The parallel could be pushed much further, but the aim of this section is simply to show what has been made possible by the use of software-platform-independent precise action specifications.

100% Code Generation

Though SDL became a formal language in 1988 only, code generators were developed before that, often by users themselves. Of course, there are code generators for the UML, too. How do SDL and UML code generators differ?

The first major difference is that you can generate 100% of the code from your standard SDL (though you do not have to) [VE]. All SDL code generators allow you to combine SDL and user code, but this is mainly used as a way to connect the generated code with legacy code, libraries, or to use code that cannot be generated from SDL (a man-machine interface, access to a database, etc.) since SDL is dedicated to the real time aspects of systems only. In other words, users do not write code simply to fill the holes left by SDL here and there; instead, they isolate whole, meaningful subsystems to be entirely described in SDL. This is not only a productive way to work, but also a safe one, since the generated code will behave consistently with what you have simulated before, as described further in this paper.

The second major difference is that you do not need to rewrite all actions (in transitions or in the body of operations) simply because you have changed tool! You can even change programming language without changing the design. On the contrary, since the UML does not even support assignments, all manipulations of a simple association have to be mapped in a proprietary, tool-dependent fashion (perhaps manually) to a programming language and/or library. As a consequence, you must rewrite all actions if you ever change from one UML tool to another. It is not the case with SDL.

Early Simulation

SDL allows the representation of informal actions, decisions and operators. This means that you can write in an abstract way what can happen, using semi-natural language, without loosing the ability to simulate or to generate a prototype. The actual behavior of informal parts during simulation can be driven in various manners interactively at run time, through lists of predefined values, randomly, with probabilities, etc. The key points are that you can simulate or generate code without having to go into the details of implementation, and that this possibility is an integral part of the SDL, not a vendor-specific extension.

Simulation is not only about executing actions, but also about simulating performance, failure, etc. For example, ObjectGEODE uses SDL formal comments (similar to UML’s tagged values) to tell how much time an action takes, and can use these values to compute the throughput of the system, or the average interval between two events. A precise action specification is necessary, because there must be something to tag with a duration or a MTBF: an action.

Automatic Test Generation, Exhaustive Verification, and more

Since SDL has formal and executable semantics, vendors have been able to develop incredibly powerful services around it, unparalleled for the UML today [PL]. Some examples:

verifying that a given sequence of events can or cannot happen; that a given scenario can still be "played" by the system (non regression testing, compliance with requirements);

verifying that a given property, possibly distributed over several objects and nodes, is always true; if not, generating automatically a sequence that "breaks the law" (exhaustive, aggressive testing);

causing failures and alterations automatically (fault tolerance).

This cannot be done with today's UML because many UML concepts have to be implemented before you can see something running, and because it is not possible to develop such services on top of programming languages for theoretical reasons.

Combining Tools From Several vendors

Of course, a vendor could extend the UML to make all the features previously described available. But the user would then be trapped, because these extensions are not part of the

UML standard. Any UML actions written with tool A would be impossible to use with tool B (even when the Stream-based Model Interchange Format is available) because there is no assurance that the format of action specifications in these tools would be even remotely similar.

SDL users can choose a certain vendor tool for modeling and simulating, and another for code generation, without rewriting a single line. This is often the case since many SDL customers have developed their own code generator for specific targets. Still, they can use a commercial simulator, because there is one language and one interpretation of it.

We are convinced that extending UML would bring a lot to the UML community, and is a key to making better systems. It is interesting to note that SDL's language is not a complex one (you cannot write complex algorithms in SDL for example), but that, under the pressure of major users, the ITU is currently making extensions to make it much more powerful.

Conclusion

We believe that complementing the UML with a compatible, software-platform-independent, precise action specification language that enables mapping into efficient code brings significant benefits to the community. This paper has explored the nature of such precise action specifications and the rationale for the benefits.

The next step is to gather support for an effort to define a standardizable software-platform-independent language for precise action specifications. Since the UML is an adopted technology of the Object Management Group, the precise action specification language is best promoted by that same group. We encourage and solicit support for this effort. This can be done in several ways: by giving us your ideas on precise action specification languages and their requirements; by participating in the OMG Request For Proposal (RFP) process as submitters or reviewers; by pestering your favorite tool vendor to participate in the RFP; and by pestering your favorite tool vendor to offer a standardized, software-platform-independent, precise action specification language in their tool set.

Acknowledgements

Johannes Ernst and Sally Shlaer provided extremely useful review and comments and their help is deeply appreciated by the authors.

References

[OCL]
Object Constraint Language Specification version 1.1
OMG Document ad/97-08-08
[PL]
OMT and SDL based Techniques and Tools for Design,
Simulation and Test Production of Distributed Systems,
Philippe Leblanc,
STTT, International Journal on Software Tools for Technology Transfer, Springer, Dec. 1997
[SM]
Object Lifecycles: Modeling the World in States,
Sally Shlaer and Steve Mellor,
Yourdon Press, 1992
[VE]

How to use modeling to implement verifiable,
scaleable and efficient real-time application programs,
Vincent Encontre,
Real-Time Engineering Journal, Vol. 4, No. 3, Fall 1997

[UML1]
UML Semantics version 1.1
OMG Document ad/97-08-04

[UML2]
UML Notation Guide version 1.1
OMG Document ad/97-08-05

[Z100]
CCITT, Recommendation Z.100
Specification and Description Language (SDL)
1993

[Z120]
ITU-T, Recommendation Z.120
Message Sequence Chart (MSC)
1993