Overview of the SpecL
Formal Modeling and Specification Language

1. Introduction

This document summarizes a formal modeling and specification language (SpecL). The material introduced here is covered in additional detail in the SpecL Reference Manual and the Formal Specification Primer.

SpecL is a hybrid of features found in other modeling and specification languages. The basic features for specifying objects and operations are found in a long historical line of modeling languages. These include SADT (Structured Analysis and Design Technique) [Ross 77], PSL (Problem Statement Language) [Teichroew 77], RMF (Requirements Modeling Framework) [Greenspan 82], OMT (Object Modeling Technique) [Rumbaugh 91], and most recently UML (Unified Modeling Language) [Rumbaugh 99].

The purpose of SpecL is to describe precisely the external structure of the objects and operations in a software system. While languages such as UML have useful features for structuring a specification, they are not fully formal. SpecL includes features for formal specification, as found in such languages as Larch [Guttag 85], OBJ [Goguen 88], and OCL (Object Constraint Language). An SpecL specification is typically developed in sequences of refinements, where definitions of objects and operations are gradually made more formal.

In SpecL, a specification consists of two parts: (1) an *object-oriented* model, and (2) a *operation-oriented* model. Dividing a specification into these two parts promotes the concept of multiple views of the same system. One view focuses on the system from the perspective of the objects (i.e., data), the other from the perspective of the operations (i.e., functions). Neither view is *the* correct one -- both convey aspects of the same system in different ways. Depending on the natural orientation of the system being specified, one form of view may be the more natural.

Whichever is the more natural view, a requirements specification should always contain both. In this way, the two views provide a form of cross checking on specification consistency and completeness. Something that may have been overlooked in the object-oriented view may show up naturally in the operation-oriented view, and vice versa. When complete, the pair of views provide separate parts of a mutually consistent specification of a system.

2. Underlying Principles

SpecL, and similar languages such as UML, share some common underlying principles. The principles are:

1. *Object/Operation Model* -- a specification is comprised fundamentally of two forms of entity: *objects* and *operations*; these entities have well-defined relations to one another.
2. *Hierarchy* -- the primary relation between entities of the same type is *hierarchy*; that is, an entity is composed hierarchically of components, which may in turn be further hierarchically decomposed.
3. *Input/Output Relationships* -- the primary relation between objects and operations is *input/output*; specifically, operations take objects as inputs and produce objects as outputs.
4. *Attribute/Value Pairs* -- in addition to relations to other entities, an entity may have other general *attributes* that further describe its properties and characteristics.
5. *Composition Primitives* -- when an entity is decomposed into components, specific forms of composition are used; the forms in SpecL are:
   a. *and* composition: an entity is composed as a *heterogeneous collection* of components
   b. *or* composition: an entity is composed as a *selected one of* a heterogeneous collection of components
   c. *repetitive* composition: an entity is composed as a *homogeneous collection* of zero or more components
   d. *functional composition*: an object is composed as a *function from input objects to output objects*
   e. *recursive* -- an entity may contain itself as a component.
6. *Class/Subclass/Instance Composition* -- a secondary form of hierarchical relation is that of *class/instance*;

an entity *class* defines a generic entity template; an entity *subclass* or *instance* specializes the generic class by adding additional attributes.

7. *Object/Operation Duality* -- the composition and relational forms apply equally to both objects and operations; that is, both objects and operations can be decomposed hierarchically, with general attribute/value pairs, and defined as classes.
8. *Strong Typing* -- all objects in a specification define a formal *type*; SpecL formulas and expressions are type checked to confirm that object names are used in type-correct contexts.
9. *Functional Operations* -- all operations are *fully functional* and *side effect free*; an operation may only access explicit input objects, and may effect change only through explicit output objects.
10. *Declarative Specification* -- a specification declares structure and function without specifying operational details.

3. Specifying Objects

An object is specified in a fixed format showing its components and other attributes. The general form is as follows:

```
object name [=]
    components: composition expression defining subobjects;
    [operations: list of applicable operations;]
    [equations: formal equational specification;]
    [description: comment or entity name;]
    [user-defined attributes: comment or entity name;]
end [name];
```

where boldface terms are keywords, italic terms are variables, and optional terms are enclosed in square brackets. For example,

These examples illustrate the use of the **components** and **description** attributes. Examples of the other object attributes (**operations**, **equations**, user-defined attributes) are given in the SpecL Reference Manual.

4. Specifying Operations

An operation specification is much like an object specification. The general form is as follows:

```
operation name
    [components: composition expression defining suboperations ;]
    inputs: list of objects;
    outputs: list of objects;
    [precondition: formal predicate on inputs;]
    [postcondition: formal predicate on inputs and outputs;]
    [description: comment or entity name;]
    [user-defined attributes: comment or entity name;]
end [name];
```

where terms enclosed in square brackets are optional. For example:

This example illustrates the use of the **inputs**, **outputs**, and **description** attributes. Examples of the other operation attributes (**components**, **precondition**, **postcondition**, and user-defined attributes) are given in the SpecL Reference Manual and Formal Specification Primer.

5. Advanced Features

5.1. Inheritance

An object or operation specifies a class from which other objects or may *inherit*. This form of inheritance is conceptually the same as in object-oriented programming languages. For example,

This example specifies that the general class of ScheduledItem has a title, start date and end date. Objects that

inherit from (i.e., specialize) ScheduledItem have the three components of a ScheduledItem, plus additional specialized components of their own. Inheritance can be multiple levels deep. For example, more specialized forms of meeting could be defined by inheriting from the Meeting object, as in

where these definitions would add appropriate new components to a meeting.

5.2. Names and Types

In the examples so far, the components of an entity have been shown as simple names. E.g., ScheduledItem components are Title and StartDate and EndDate. A more detailed way to specify components is to use a name:type pair. For example,

```
object ScheduledItem
    components: t:Title and sd:StartDate and ed:EndDate;
    ...
```

Here, the name component of the name/type pair is a local name by which the component is known. The names are *t*, *sd*, and *ed* in this example. The type component is the name of a defined object, as in the original definition of object ScheduledItem above. Types in this example are `Title`, `StartDate`, and `EndDate`.

Component names are used to refer to object components in formal preconditions and postconditions. Examples of these uses appear in the Reference Manual and Primer.

5.3. Modules

For large specifications, it can be convenient to organize object and operation definitions into *modules*. The basic format of an SpecL module is the following:

```
module name;
    [imports]
    [exports]
    [attribute definitions]
    [entity definitions|
    [formal definitions]
end [name];
```

where terms enclosed in square brackets are optional. For example:

This example illustrates the use of the module imports, exports, and the placement of object and operation definitions. Examples of the other module declarations (user-defined attributes and formal definitions) are given in the SpecL Reference Manual.

6. Tabular and Graphical Notation

Models specified in SpecL can be depicted in tabular and graphical forms. The tabular form is called a "data dictionary", that lists the major attributes of objects and operations. Data dictionaries are a very well established format that have been used in software engineering for many years.

The graphical notation uses concepts from the Unified Modeling Language (UML) and some other commonly used graphical forms. UML is the most recent in a long line of graphical modeling notations that date back as far as the mid-70s with SADT [Ross 77] and PSL/PSA [Teichroew 77]. The development of UML was a collaborative effort of primarily commercial organizations, led by the Rationale Corporation. The goal for UML has been to develop a standard graphical modeling notation. UML is a consolidation of concepts used in earlier notations, in particular the Object Modeling Technique (OMT) [Rumbaugh 91].

UML has a number of features that are not used in the notation described here. In addition, UML is missing certain features that are convenient for diagramming SpecL models, in particular dataflow diagrams. Where a feature is available in UML, the same feature is retained here. Where UML is missing a useful feature, a previously existing standard is used in such a way as not to conflict with UML features. Also of note are some terminology differences

between the notation presented here and UML. In particular, the terms "object", "component", and "composition" are used here as they relate to SpecL. These terms are used differently in standard UML.

6.1. Data Dictionaries

Data dictionaries are common notation for describing objects. A data dictionary shows objects, their components, and descriptions. Figure 1 shows the data dictionary format for the object definitions given as examples in Sections 2 through 5 above. There is no semantic difference between objects defined in a data dictionary versus the textual notation.

As commonly used, data dictionaries are typically a less formal notation than a complete language such as SpecL. Therefore the details of how data dictionaries are used in practice can vary among different authors. In some uses, there is no separate components column, so that everything about the object is described in the prose description column. Also, the exact notation used in components definitions may vary. For example, some authors use "+" instead of "and" for defining tuples.

Object Name	Components	Description
Appointment	*inherits from* ScheduledItem *adds* StartTime and Duration and Location and AppointmentSecurity and Priority and RemindInfo and Details	An Appointment adds a number of components to a generic ScheduledItem. The StartTime and Duration indicate when the appointment starts and how long it lasts. The Location is where it is held. The AppointmentSecurity indicates who can see that the appointment is scheduled. Priority is how important the appointment is. RemindInfo indicates if and how the user is reminded of the appointment. Details are free form text describing any specific appointment details.
Calendar	ScheduledItem*	A calendar is composed of zero or more scheduled items.
ScheduledItem	Title and StartDate and EndDate	A ScheduledItem is the generic definition for the types of items stored in a calendar. A ScheduledItem has a title, start date, and end date. Specializations of ScheduledItem are Appointments, Meetings, Events, and Tasks, q.q.v.
Meeting	*inherits from* ScheduledItem *adds* StartTime and Duration and Location and MeetingSecurity and Priority and RemindInfo and Attendees and Details and Minutes	Like an Appointment, a Meeting adds components to a generic ScheduledItem. A Meeting differs from an Appointment as follows: (1) Security for an appointment includes a private option that is not available for meetings. (2) Meetings have Attendees and Minutes components, Appointments do not.
StaffMeeting	*inherits from Meeting* *adds* ...	
UserMeeting	*inherits from Meeting* *adds* ...	

Figure 1: Example Data Dictionary.

There is a tool for SpecL specifications that automatically generates data dictionaries in HTML form. The SpecL-to-dictionary generator is called "`specldoc`". It is described in the documentation for the SpecL translation tools.

6.2. Class Diagrams

A class diagram depicts the composition and inheritance relationships among SpecL objects. It can also show the operations that are associated with an object, where "associated with" means those operations listed in an object's `operations` attribute. The elements of a class diagram are the following:

1. three-part boxes, labeled at the top with an object name
   a. component names appear immediately below the object name in the three-part box
   b. operations names follow the component names in the three-part box
2. one-part boxes, labeled inside with the object name only
3. ovals (or circles) labeled inside with the name of an operation
4. connecting edges between object boxes, with three forms of augmentation:
   a. a hollow triangle, depicting an inheritance relation
   b. a hollow diamond, depicting a composition relation
   c. a '*' or an integer next to a hollow diamond, indicating multiplicity of composition

Figure 2 shows the general structure. Two equivalent graphical forms are shown on the left, with the corresponding textual form on the right.

The diagram on the top left of the figure shows components and operations using the three-part box notation. The diagram on the bottom left, labeled "Alternative Equivalent", shows the equivalent relations using the hollow diamond notation. To illustrate how modeling concepts are common in a variety of languages and notations, textual forms are shown in Java and C++, as well as in SpecL. All five forms in the figure (two diagram forms, three textual forms) have the same abstract meaning.

Figure 3 shows the basic form of multiplicity annotations, as well as two-way containment. The diagram shows that `Obj1` has exactly 2 `Obj2`'s and that `Obj2` has zero or more `Obj1`'s.

The three-part notation for composition versus the hollow diamond notation are two equivalent forms that have the same meaning. Composition can be shown in either way separately, or with the two forms combined in a single diagram. When the two forms are used together, components should not be repeated. That is, each object component should be shown in the second part of a three-part box or shown as a hollow diamond attachment, but not shown in both ways in the same diagram. A UML convention is to show atomic components in the second part of a three-part box and non-atomic components as hollow-diamond attachments. In SpecL, atomic components are integer, real, string, boolean, and opaque objects. An *opaque object* has no components at all, just a name. It is conceptually comparable to an enumeration literal in common programming languages.

Figure 4 shows an example class diagram depicting the object definitions given as SpecL examples in Sections 2 through 5 above.

Figure 2: General Structure of a Class Diagram.

Figure 3: Two-Way Containment in a Class Diagram.

Figure 4: Example class diagram.

6.3. Dataflow Diagrams

A dataflow diagram depicts the flow of objects between operations in a specification. Elements are:

1. circular or oval graph nodes, depicting operations
2. directed interconnecting edges, depicting operation inputs and outputs
3. graph levels, depicting operation hierarchy

Figure 5 shows the general structure. The leveling in the diagram depicts the level of operation components. The interconnection lines show flow of data between operations, based on input/output types. In order for the digram to be legal, the dataflow must match the input/output declarations in the specification. For example, the connection between `Op1a` and `Op1c` is legal because `Op1a` has an output of type `Data1` and `Op1c` has an input of this type.

A dataflow diagram is only a *possible* depiction of flow since the interconnections actually show more information than is in the abstract SpecL specification. That is, the specification defines the input and output types of each operation, but does not require that they be interconnected as shown in any particular dataflow diagram.

6.4. Package Diagrams

A package diagram is for large-grain modeling, showing the relationship between modules. Elements of a package diagram are the following:

1. folder-shaped rectangles, labeled inside with the name of a module
2. directed interconnection lines, abstractly depicting communication between modules
   a. line labels in regular font are object names, depicting data communication
   b. line labels in italic font are operation names, depicting functional communication

Figure 6 shows the general structure. The diagram shows `Module1` sending data of type `Obj1` to `Module2`. `Module2` invokes `Op1` in `Module1`. `Module1` and `Module3` send data of type `Obj2` to each other.

The interconnection *abstractly* depicts communication in that the diagram does not show the details of the communication. For data communication, the package diagram indicates that one module sends data to another, without indicating the operation that produces the data. With functional communication, only the name of the operation is given without indicating the details of input and output.

7. Conclusion

SpecL is a formal and mechanically checkable language for requirements modeling and formal specification. It is based on principles found in many requirements specification notations, both graphical and textual. Independent of syntactic details, virtually all notations share a common set of underlying principles, most notably object modeling, operation modeling, and hierarchical model decomposition.

References

[Goguen 88] J. A. Goguen and T. N. Winkler, "Introducing OBJ3", SRI International Technical Report, Palo Alto, CA, August 1988.

[Greenspan 82] S.J. Greenspan, J. Mylopoulos, and A Borgida, "Capturing More World Knowledge in the

Figure 5: General Structure of a Dataflow Diagram.

Figure 6: General Structure of a Package Diagram.

Requirements Specification", *Proceedings of the Sixth International Conference on Software Engineering*, 1982.

[Guttag 85] J. Guttag, J. J. Horning and J. M. Wing, "The Larch Family of Specification Languages", *IEEE Software*, May 1985.

[Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.

[Rumbaugh 99] Rumbaugh J., I. Jacobsen, G. Booch, "UML Reference Manual", Addison-Wesley.

[Ross 77] D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering*, January 1977.

[Teichroew 77] D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, January 1977.