Automated Support for the Development of Formal Object-Oriented Requirements Specifications

Robert B. Jackson
David W. Embley
Scott N. Woodfield

Department of Computer Science
Brigham Young University
Provo, UT 84602 USA

Abstract. The creation of a requirements specification document for systems development has always been a difficult problem and continues to be a problem in the object-oriented software development paradigm. The problem persists because there is a paucity of formal, object-oriented specification models that are seamlessly integrated into the development cycle and that are supported by automated tools. Here, we present a formal object-oriented specification model (OSS), which is a seamless extension of an object-oriented analysis model (OSA), and which is supported by a tool (IPOST) that automatically generates a prototype from an OSA model instance, lets the user execute the prototype, and permits the user to refine the OSA model instance to generate a requirements specification. This approach leverages the benefits of a formal model, an object-oriented model, a seamless model, a graphical diagrammatic model, incremental development, and CASE tool support.

1. Introduction

Perhaps the most critical element in the development of a software system lies in properly understanding and in properly documenting the requirements for a system to be developed. A precise, formal, easily understood requirements specification is one of the most important, yet one of the most elusive components of the entire software development process. We suggest that the development of a requirements specification does not have to be elusive, and, indeed, can be a natural and integrated component of the software life cycle.

This general problem of more easily creating and integrating high quality requirements specifications into the software development process is exacerbated by the current major shift in development paradigms from structured to object-oriented. Although the object-oriented paradigm is proving to be effective, many of the current methods, models, CASE tools, prototyping languages (4th GLs), and formal specification languages still contain substantial structured technology. The end result is a development approach that tries to marry the structured paradigm with the

object-oriented paradigm. Unfortunately, this marriage does not work well. Software engineering principles and tools that support prototyping and the development of requirements specifications within the object-oriented paradigm are critically needed.

Our approach to more easily developing high quality object-oriented requirements specifications is based on a formal, object-oriented model that serves all phases of the software life cycle. It is our belief that a formal model is a prerequisite for assimilation of engineering principles into software development, and that a seamless development cycle requires a pervasive model around which tools and methods can be built.

The formal, object-oriented model we use in our approach is OSM (the Object-oriented Systems Model), which is based primarily on an analysis model called OSA (Object-oriented Systems Analysis) [8]. The formal definition for OSA is formulated using set theory and first-order predicate calculus (see Appendix A of [8] and [3]). OSA is an integrated model because it can be used to describe object structure, object relationships, object behavior, and object interaction, all within the same formal context. A second component of OSM is OSS (Object-oriented Systems Specification), which is a formal model that extends OSA and is appropriate for specification. Our approach to the development of formal specifications is to generate an OSS model by executing (prototyping) and incrementally extending an OSA model through the use of a CASE tool, called IPOST (Interactive, Prototyping Object-oriented Specification Tool).

Our approach is as follows: Using OSA, a systems analyst builds an object-oriented model instance (generally a partially complete instance) of the problem domain. Next IPOST reads the OSA model instance from the data repository and automatically creates a user interface and working prototype. As the user executes and modifies the prototype, the OSA model instance is enhanced and becomes an OSS requirements specification. The end result of prototype execution and model enhancement is a precise, formal specification in a graphical notation with embedded formal textual descriptions of behavior and interactions. The details of our approach in the remainder of this paper are as follows. In Section 2 we outline some of the difficulties of current approaches to specification development, and we show how our approach builds on some of the best work of others. In Section 3 we describe OSA and the specification language extensions required for OSS. In Section 4, we describe IPOST and our methodology for developing an OSS model instance. In Section 5 we discuss the implementation of current support tools. We conclude in Section 6.

2. The Problems with Specifications

2.1 Informal Specification Techniques

Early attempts at explaining to clients the details of a proposed system were done with informal, natural-language narratives. The narratives were later enhanced to include strict guidelines, organization directives, diagrams and informal models.

Informal models, which are characterized by the lack of an all-encompassing theoretical foundation, are included in techniques such as Modern Structured Analysis [20], Data Structured System Development (DSSD) [19,15], Structured Analysis Design Technique (SADT) [16], and Object Oriented Analysis (OOA) [4]. Inclusion of more stringent directives and informal diagrammatic models have helped informal requirements specifications become more precise and understandable, but they still suffer from several problems, including problems of organization, redundancy, incompleteness and misinterpretation.

To ease the problem of misinterpretation, developers frequently build prototypes to raise the level of understanding between clients and developers. The addition of a requirements prototype to the development cycle has been beneficial. "Operational Prototyping" [5] is one of various new prototyping approaches that are increasing the benefits derived from prototyping. However, prototyping is not without its own problems. The addition of a prototype to an informal method requires two paradigm shifts during this first phase of a project. A paradigm shift occurs between the analysis model and the prototyping language. Then another paradigm shift occurs in writing the specification. Not only are these shifts time consuming, but they also raise the potential for information loss.

2.2 Formal Specification Techniques

To increase precise communication between developer and client, researchers and a small number of practitioners have begun using more formal analysis and specification techniques. These techniques usually contribute to the development process by adding principles of engineering discipline. An additional benefit is that formal languages can frequently be directly executed. These benefits usually come with a cost, however; namely, the cost of formal, mathematical constructs that are difficult to understand.

Examples of formal languages used for specification include PLEASE and SPEC [2, 18], which are algebra based languages. Z and VDM are formal models based on logic, sets, sequences, lists, relations and functions [7, 11, 17]. SXL (State Transition Language) [13] and PAISley ([21], which are more operative based, are oriented towards describing behaviors. All these formal languages provide the precision required for unambiguous interpretations for contractual needs.

There are two major problems with formal languages. First, formal languages are difficult to read and write. Second, there is frequently a major paradigm shift between analysis and specification. Analysis is done using one model or language and specification is done in another.

2.3 CASE Tool Techniques for Specification

Although the primary focus of CASE tools has not been to create specifications, they have nevertheless made a substantial contribution to producing

precise specifications. CASE tools, especially I-CASE tools, are based on a central repository of information that helps integrate analysis and prototyping with specification. Information captured from analysis is stored in a central repository and can be used to generate forms and reports to assist in the development of a working prototype. As the prototype is refined, the repository is updated. This updated repository can thus serve as an information base to generate specifications for client agreement.

There has been some excellent research in the use of CASE tools for prototyping and especially in using more formal models in a CASE environment. MASCOT [9] (Modular Approach to Software Construction, Operation and Test) is a diagrammatic approach for parallel processes. Execution is effected by translating MASCOT diagrams to a formal notation. PROTOB is an object-oriented CASE tool based on high-level Petri Nets and is used to model distributed systems [1]. A method for transforming between formal languages to develop executable prototypes using PROLOG has also been developed [10].

2.4 Formal Model with CASE Tool Technique

Our approach builds on both the formal-model approach and the CASE-tool approach. The fundamental principle is that there must first exist a formal, seamless model that integrates all relevant information and that can serve all phases of the software development cycle. Given that we have a formal, integrated, seamless model, CASE tools can be built around this model to provide a tool-supported, seamless development methodology. Because the model is seamless, there are no paradigm shifts between analysis, specification, design and implementation.

Other benefits also accrue from this approach. Because the model is formal, it can be precisely interpreted for contractual purposes. Also because it is formal, it can be executed as a working prototype. Furthermore, the model is graphical which facilitates understanding. A textual form is also available for situations in which this may be more suitable.

3. OSA and OSS Model Components

3.1 OSA Model

An OSA model is comprised of three submodels: an object-relationship model (ORM), an object-behavior model (OBM), and an object-interaction model (OIM). Figure 1 parts a,b, and c depict the three submodels for a simple library application. In our brief tutorial, we have included only those parts of OSA that are germane to our discussion. For further details see [8].

ORM instances describe object classes, relationship sets, and constraints. Boxes in an ORM diagram represent object classes, such as *Book*, *Loan*, and *Librarian* in Figure 1a. *Book*, for example, represents the set of books in the library.

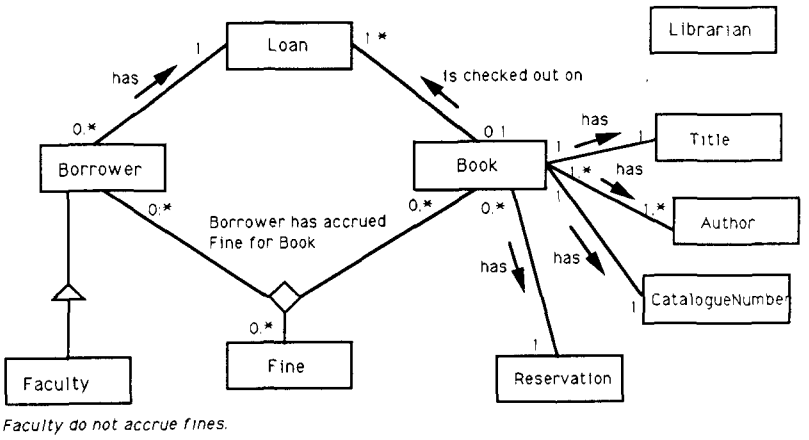

Figure 1a. Object Relationship Model of Library system classes.

We show three different types of relationship sets. *Borrower has Loan* is a binary relationship set. *Borrower has accrued Fine for Book* is a ternary relationship set. *Faculty IsA Borrower* is a Generalization/Specialization relationship set denoted by an open triangle. Participation constraints constrain the possible relationships among objects. The *Book is checked out on Loan* relationship set has a *0:1* participation on the *Book* side, indicating that a book may participate either zero or one times in the relationship set. General constraints, such as *Faculty do not accrue fines*, are shown by italics on an ORM diagram. OSA also permits more abstract constructs such as high-level object classes and relationship sets.

The behavior of objects within an object class is described by a state net. Figure 1b shows a state net for the *Book* object class. This state net serves as a template for the behavior of all book objects in the set. Each transition (represented by a box) defines both a trigger (described in the upper part of the box) and a set of actions (described in the lower part of the box). A set of state nets, one for each object class, make up the OBM, which thus describes the behavior of all objects.

Figure 1b is interpreted as follows: When the AddBook event occurs, the trigger evaluates to true, transition *[1]* fires, and a new book object is created. At the completion of the action in transition *[1]*, the newly created book goes into the *Ready to Loan* state. If a book is in the *Ready to Loan* state, and the *BorrowBook* event occurs, then the action in transition *[2]* is executed and the book goes into the *On Loan* state. The half circle and arrow going to transition *[4]* indicate the spawning of a new, concurrent thread of control. The multiple threads indicate that a book may in both states *On Loan* and *On Reserve* at the same time.

Interactions among objects are described in an OIM instance, which is comprised of various types of interactions and may be organized using various levels of abstraction. Figure 1c shows some possible interactions between a librarian and a book, and are shown as interactions between the *Librarian* class and the *Book* class.

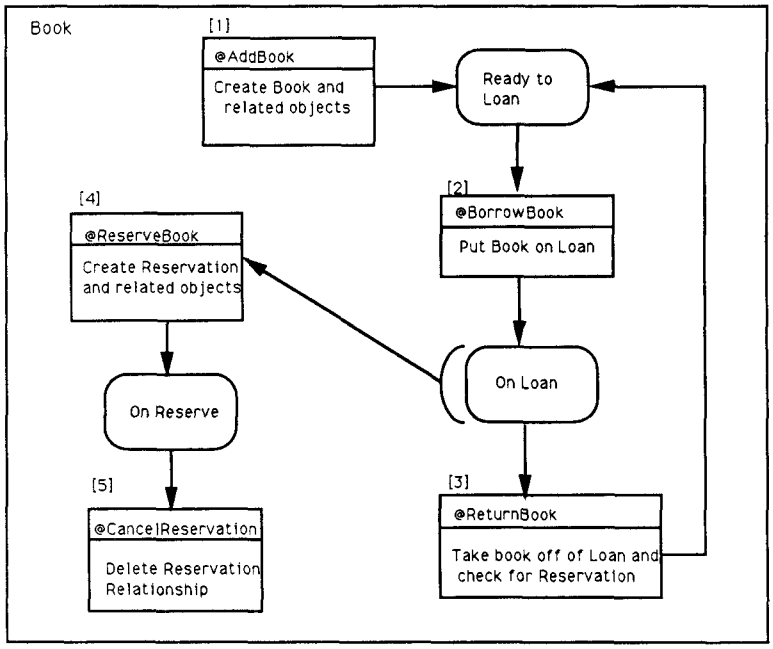

Figure 1b. State Net for Book from Object Behavior Model.

A librarian can add new books as well as check out and return books for patrons.

One feature of OSM modeling is that more detailed information can be provided by views which integrate submodels together. Figure 2 is a combination that contains elements of all three submodels in one view. In this case, the *Librarian*

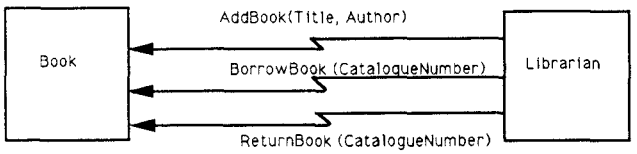

Figure 1c. Interaction with origin Librarian and destination Book.

class is still the origin of the interactions, but the interaction destinations have been integrated with the *Book* state net by denoting destination transitions for each interaction. The *AddBook* interaction has transition *[1]* as its destination, and in fact the event type trigger of transition *[1]* becomes true when the interaction is received.

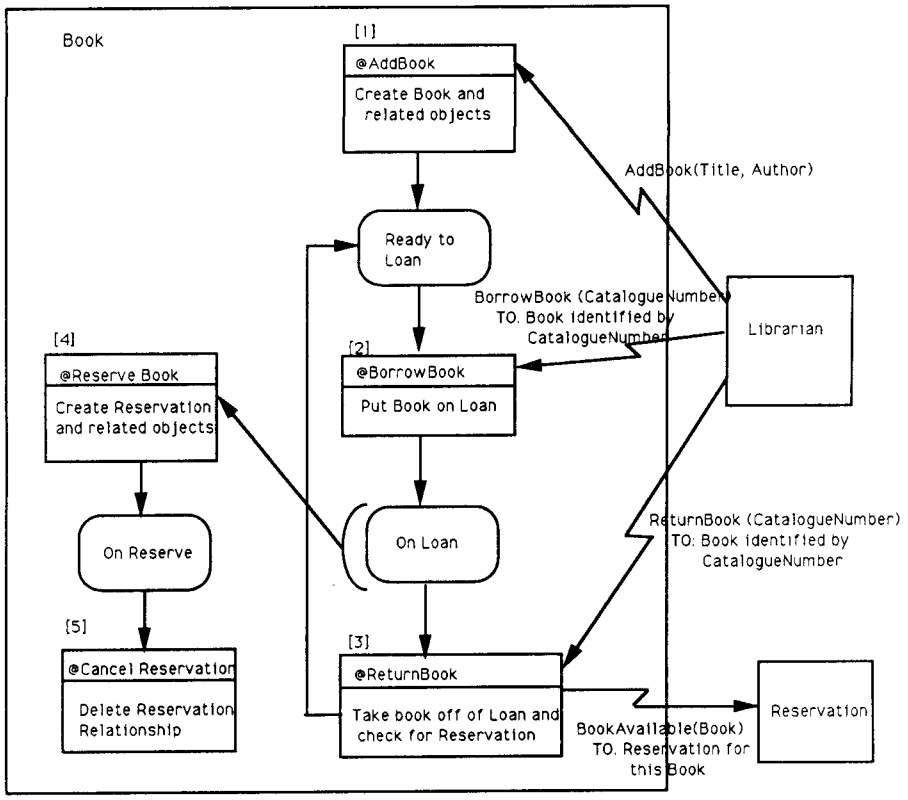

Figure 2. Interaction Diagram combined with Book State Net.

The *TO* clause on the *BorrowBook* interaction indicates that the interaction cannot go to just any book, but must go to a particular book, in this case to the *Book identified by CatalogueNumber*.

3.2 OSS Extensions

Although, both the syntax and semantics of OSA have been formally defined, an OSA model instance does allow for some constructs -- namely, triggers, actions, interaction descriptions, and general constraints -- to be written in natural language. For example, transition *[1]* in Figure 1b has an informal action: *Create Book and related objects*. Components written in an informal language may be subject to misinterpretation. Here, for example, what exactly are the related objects? A formal OSS language has been defined that can formally express these natural language statements.

Several criteria were considered in the development of the OSS language. One

consideration is that this language should be easy to write and understand by non-technical clients. To satisfy this criteria, an English like syntax was chosen. Next, it had to support triggers and general constraints which are logic statements, and actions, which are procedural statements. Finally it had to be precisely defined.

In Figure 1a the general constraint, *Faculty do not accrue fines*, is informal. The OSS language expression for this general constraint is:

Faculty() IsA Borrower(x) IMPLIES NOT Borrower(x) has accrued Fine() for Book().

This statement illustrates a logic statement in the OSS language, and based on the classes and relationship sets in the OSA instance. The terms with parentheses are class names which are embedded within relationship set names from the model instance. During prototype execution, "x" is unified to those borrowers who are members of the *Faculty IsA Borrower* relationship set and then tested against the *Borrower has accrued Fine for Book* relationship set. Unspecified variables (empty parentheses) are "don't care" variables as in PROLOG.

The procedural portion of the language has control statements for decision making and looping as well as expressions for manipulating objects. In Figure 1b, the informal English statement in transition *[3]*, for example, says *Take book off of Loan and check for reservation*. In the OSS language this becomes:

```
DELETE Book(this) is checked out on Loan();
IF Book(this) has Reservation(w) THEN
  SEND_INTERACTION BookAvailable (Book(this)) TO Reservation(w);
ENDIF;
```

Here, "this" represents the book object that received the interaction. Prototype execution is done by unifying "w" based on the value bound to "this" and existing relationships, and then executing the statements. The combination of both procedural and logical constructs in the language make it suitable for specifying a wide range of constraints, triggers and functions.

4.0 Specification Development

Figure 3 illustrates our approach to the incremental development of OSA model instances and OSS specifications, using IPOST. Initially, a systems analyst develops an OSA model instance, for example the one in Figures 1 and 2. A high-level object class which identifies the system boundary is required. IPOST reads an OSA model instance, with system a boundary and creates an executable prototype. Prototype generation, including a user interface, is automatic.

There are three types of refinements that can occur. Two of these are directly supported by IPOST and are illustrated by the feedback paths in Figure 3 labeled *OSA Model Instance Refinements* and *Logical Interface Refinements*. Model-instance refinements include replacing natural-language descriptions for triggers, actions, and

constraints with OSS language statements. Logical interface refinements allow a user to formalize interaction descriptions and refine the interface. When major errors or omissions are observed in the OSA model instance, the user can make appropriate changes to the OSA instance using the drawing tool. A new prototype can be automatically generated from the modified OSA model instance. This process may continue until a fully-formalized OSS model is developed or it may stop at any point deemed satisfactory to both client and developer.

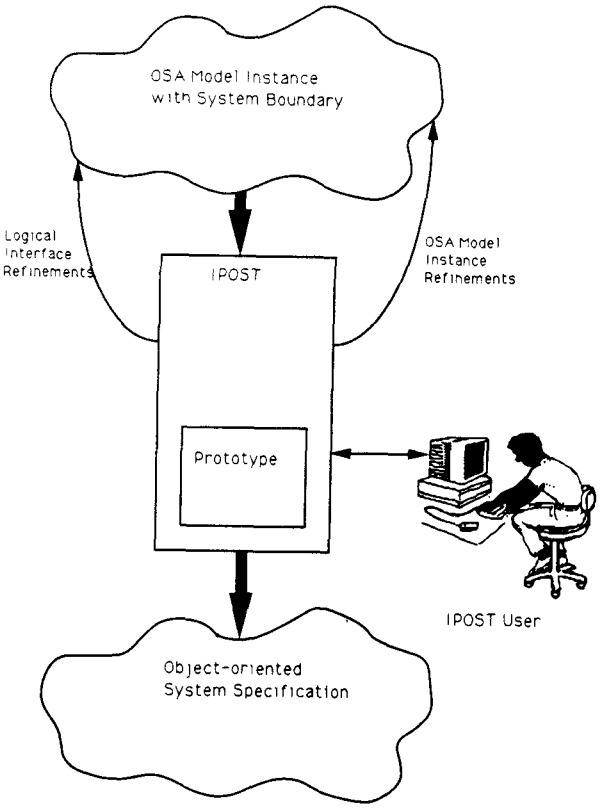

Figure 3. Specification Development using IPOST.

4.1 Automatic Generation of an Executable Prototype

Addition of a System Boundary. The system boundary encloses those components of an OSA model instance that are to be included within the system to be specified and excludes other components. Technically, in OSA, the enclosed components constitute a high-level object class that only has interactions crossing its boundary. All other components of an OSA model instance must move to one side or the other of a system boundary, or must divide into multiple components, each of which are placed on one side or the other of the system boundary. In Figure 4, for example, we define a high-level object class that includes all the ORM components except the *Librarian* object class.

Creating an Interface. The set of interactions that cross a system boundary become the components that define the logical interface. The interface is logical because it focuses on the events and the information flows, rather than on actual physical devices, the look and feel of a user interface, or a particular user interface implementation. IPOST generates a default user interface and does not require an additional user interface specification. The information necessary to develop a logical

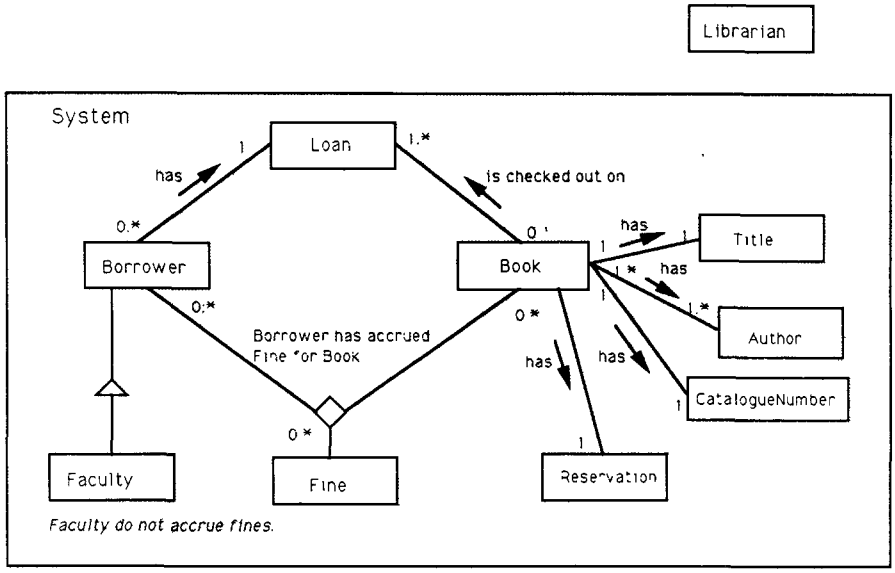

Figure 4. Library ORM with High-Level Object Class "System".

system interface is all contained within an OSA model instance.

Figure 5 shows the interface window for IPOST. The top panel is a menu panel and provides organization for all interactions crossing the system boundary. Origin and destination classes are selected from popup windows. Then the appropriate interaction for the selected origin and destination classes is selected. In Figure 5, the *AddBook* interaction has been selected. The middle panel illustrates how an interaction corresponds to a logical form. Since the *AddBook* interaction was chosen, the parameters listed in the middle panel are *Author* and *Title*, the object parameters sent by the interaction. Space is provided for the IPOST user to enter data for the object parameters and to "execute" the interaction.

4.2 Executing and Refining OSA Model Instances.

Prototype Execution Mode. Prototype execution consists of presenting to the user the set of menus and forms defined by the interface. The user initiates interactions through these forms. As prototype execution proceeds, objects are instantiated and they behave and interact according to the defined OSA model instance. Thus, prototype execution consists of creating objects, deleting objects, initiating interactions, evaluating and firing triggers, evaluating constraints, and performing actions within transitions.

Refer again to Figure 2, which shows the destinations of interactions within the *Book* object class. As these interactions are initiated by the IPOST user, the

trigger on the appropriate transition is evaluated. The bottom panel of Figure 5 shows transition *[1]*, which is the destination transition for the *AddBook* interaction. When the *AddBook* interaction arrives, then the trigger *@AddBook* evaluates to true.

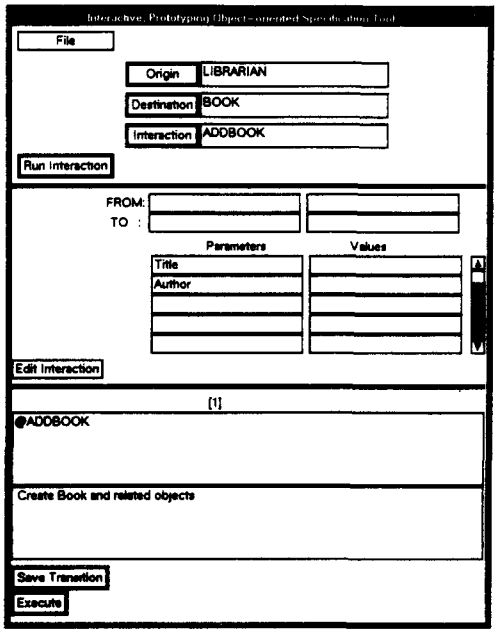

Figure 5. IPOST interface showing Menu Panel, Input Form Panel, and Execution Panel.

After the user enters the *Title* and *Author* data into the form and initiates the *AddBook* interaction, IPOST attempts to execute the action for transition *[1]*: *Create Book and related objects*. However, since the action is informal, IPOST cannot execute it, and provides an execution message to the user, as shown in Figure 6. At this point the user can select one of several possible courses of action. For triggers and general constraints, the user can act as an "oracle" and provide a true or false answer. For actions, the user may skip execution of that component. Of course, just skipping execution may cause the prototype to behave incorrectly. To achieve the objective of specification development, it is preferable that the user enter edit mode and replace the informal statements with formal OSS language statements. At the end of each transition, IPOST verifies the populated model against all constraints any informs the user of any constraint violations.

Refinement (edit) Mode. Several types of changes to the OSA model instance are made through IPOST during prototype execution. Figure 7 illustrates the change of the informal action *Create Book and related objects* into formal OSS language statements. The edit mode also supports changes to the interface forms. Since each logical form is mapped to an interface interaction, changes to a form also modify the underlying interaction. Changes can be made to the TO clause, the FROM clause, the interaction description, and the object list.

For example, upon execution of the action statement for transition *[1]*, IPOST would note that a participation constraint for the *Book has CatalogueNumber* relationship set is violated. The participation constraint (see Figure 4) requires that there be a *CatalogueNumber* for every book. The user could then edit the input form by adding another parameter for catalogue number. This change would be reflected back to the OSA model instance.

5. Implementation of Tools

At BYU we are developing research versions of CASE tool support for the OSM seamless development process. We are developing these tools so that they can be integrated together to demonstrate the feasibility of a completely integrated CASE environment. The following paragraphs briefly describe several support tools that are under development.

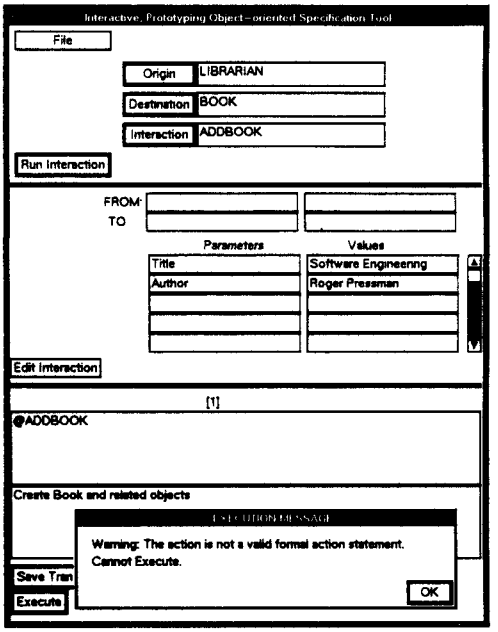

Figure 6. Execution of an *AddBook* Interaction with informal action statement.

A prototype version of the OSA Drawing Tool has been developed and is being used in our research environment. It is written in C++ utilizing X-Windows. It supports the drawing of OSA including object classes, relationship sets, state nets, and interactions. Other more complex OSA constructs, such as high-level views are being developed. It is based on a graphical user interface development tool called ART, which has also been developed at BYU [14].

The OSA Storage Facility serves as the data repository for OSA and OSS model instances. This repository has also been written using C++ under X-Windows. In addition to providing standard database management facilities for storing, modifying, and querying the database, it also provides comprehensive validity checking of OSA model instances. Finally, it also generates database schemas based on OSA model instances stored under the OSA metamodel schema.

An initial prototype version of IPOST is running, and is in systems test. IPOST is written in C++ to run under X-Windows. The user interface is written using ART and the data manipulation uses the OSA Storage Facility. In Edit mode, the user can observe and modify those components of an OSA model instance that contain natural language components. Changes to these components are captured and saved back in the Storage Facility. In Execute mode, IPOST organizes and presents to the user the set of interactions that comprise the user interface for an OSA model instance. As the user "fires" these interactions, the appropriate state net paths are followed to emulate the behavior of objects moving between states via transitions. The appropriate triggers and actions within the transitions are read, parsed, interpreted and executed based on the OSS language. Any applicable constraints are also read, parsed, interpreted and executed at the appropriate time. As execution occurs, the user is able to view the results of the execution by querying the storage

facility for the presence of objects, their states of behavior, and associated relationships.

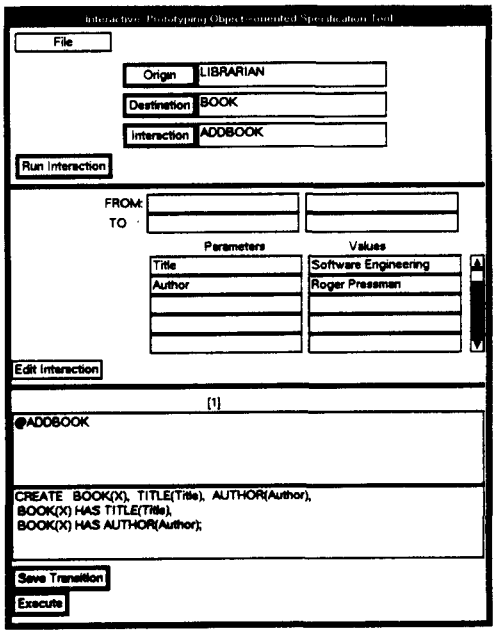

Figure 7. IPOST Window showing the formal action statements to create a Book.

6. Conclusions and Future Research

A requirements specification has always been an elusive part of software development--not only its production, but also its definition. The objective of this research has been twofold: (1) to define a specification model that is formal, is object based, can be used to specify all parts of a proposed software system, and conceptually integrates well with the other models and phases of the software development cycle; and (2) to demonstrate a method to develop a specification that naturally integrates with accepted methods of software development (analysis and prototyping) and can be supported by automation tools.

Specifically this research is of benefit because it demonstrates improvements in software development through the addition of formalism and principles of engineering. These benefits include: (1) direct execution of analysis model instances; (2) generation and refinement of a formal specification through rapid prototyping; (3) a seamless systems development approach that requires no paradigm shifts between analysis, specification, and prototyping; and (4) the foundation of a formal specification that is based on the object-oriented systems development paradigm.

The OSM Research Group at BYU is currently engaged in long term model development and software engineering research. Although the research agenda is broad, its focused purpose is to provide a systems development paradigm that includes analysis, specification, design, implementation, evolution, re-engineering, and management, all based on a single conceptual model. Research in all areas is proceeding in parallel.

Further research in the area of specification, beyond the project described in this paper, has two major thrusts. The first is to expand the capabilities of IPOST to include expert system capabilities, enhanced model modification capabilities and user interface "look and feel". The second thrust is to integrate the OSS CASE tools with the other phases of software development, such as design and code generation.

Acknowledgements: To Steve Clyde and Jeff Pinkston who helped with this research.

References

1. Baldassari, Marco, Giorgio Bruno and Andrea Castella: PROTOB: and Object-oriented CASE Tool for Modelling and Prototyping Distributed Systems. *Software-Practice and Experience.* 21 No 8, 822-844 (August 1991)
2. Berzins, Valdix and Luqi: An Introduction to the Specification Language Spec. *IEEE Software.* SE-11 No 8, 74-84 (Mar 1990)
3. Clyde, Stephen W., David W. Embley, and Scott N. Woodfield: The Complete Formal Definition for the Syntax and Semantics of OSA. *Brigham Young University Technical Document #BYU-CS-92-2.* (February 1992)
4. Coad, Peter, and Edward Yourdon: *Object Oriented Analysis.* Yourdon Press, 1990.
5. Davis, Alan M.: Operational Prototyping: A New Development Approach. *IEEE Software.* 70-78 (September 1992)
6. Davis, Alan M.: *SOFTWARE REQUIREMENTS: Analysis and Specification.* Prentice-Hall, 1993.
7. Duce, D.A. and E.V.C. Fielding: Formal Specification of Two Techniques: *The Computer Journal.* 30 No 4, 316-327 (1987)
8. Embley, David, Barry Kurtz and Scott Woodfield: *Object-oriented Systems Analysis: A Model-Driven Approach.* Prentice-Hall, 1992.
9. Friel, G. and D. Budgen: Design Transformation and abstract design prototyping. *Information and Software Technology.* 33 No 9, 707-719 (November 1991)
10. Habra N.: Computer-aided prototyping: transformational approach. *Information and Software Technology.* 33 No 9, 685-697 (November 1991)
11. Hekmatpour, Sharam, and Darrel Ince: *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, 1988.
12. Lantz, Kenneth E.: *The Prototyping Methodology*, Prentice-Hall, 1989.
13. Lee, Stanley and Suzanne Sluizer: An Executable Language For Modeling Simple Behavior. *IEEE Transactions on Software Engineering.* 17 No 6, 527-543 (June 1991)
14. Olsen, Dan R.: *The ART Users Manual*, Brigham Young University, 1992.
15. Orr, Kenneth T.: *Structured Requirements Definition*, Kenn Orr & Associates, 1981.
16. Ross, Douglas T.: Applications and Extensions of SADT. *Computer.* 18, 25-34 (April 1985)
17. Spivey, J.M.: *The Z Notation: A Reference Manual*, Prentice-Hall, (1992).
18. Terwilliger, Robert B. and Roy H. Campbell: PLEASE: Executable Specifications for Incremental Software Development. *The Journal of Systems and Software.* 10, 97-112 (Oct 1989)
19. Warnier, Jean Dominique: *Logical Design of Systems*, Van Norstrand Reinhold, 1981.
20. Yourdon, Edward: *Modern Structured Analysis*, Prentice-Hall, 1989.
21. Zave, P.: An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering.* SE-8 No 8, 250-269 (1982)