From Graphical Representations to Formal Specifications and Return: Translation Algorithms in the Harmony Environment

Sergiu M. Dascalu*, Peter Hitchcock, Narayan C. Debnath***, Andrew Klempau***

* *Dept. of Computer Science*
University of Nevada, Reno
Reno, NV 89557, USA
dascalus@cs.unr.edu

** *Faculty of Computer Science*
Dalhousie University
Halifax, NS, Canada
peter.hitchcock@dal.ca

*** *Dept. of Computer Science*
Winona State University
Winona, MN 55987, USA
ndebnath@winona.edu

Abstract

This paper introduces the translation algorithms that support the combination of semi-formal graphical representations with formal notations in the Harmony environment for software specification. Background information is presented about the Harmony approach and details of both formalization and deformalization processes are provided. Even though the algorithms described are focused on UML to Z++ translations, their underlying principles and rules can be adapted and reused for other combinations of modeling notations, in particular for combinations that involve UML for the graphical representation of software models. Several directions of enhancing the existing translation algorithms are also discussed in the paper.

1. Introduction[1]

In software engineering, programs (software products) should typically be built in phases: requirements definition (determining *what* the product is supposed to do), design (indicating *how* the product will be built, from what components), implementation (writing the code), integration (putting together separate pieces of code), and testing (verifying that the software satisfies its requirements). After the above development phases, a software product enters the evolution phase of its life-cycle (or software process) [1, 2].

For defining requirements and developing designs the software engineer needs to create software models, which can be graphical, textual, or a combination of both. Because laying the right foundation is highly beneficial for any project, we have focused on the early phases of the software process: requirements definition and design. The scientific literature clearly shows that errors discovered in later phases (e.g., implementation or maintenance) are much more expensive to correct than in earlier phases of the software life-cycle [1, 2, 3]. Starting from this pragmatic consideration, we have looked at various ways of improving the specification of a software product and thus its overall quality.

As a matter of terminology, in this paper the word *specification* is used in the sense defined by Alan Davis, that of a document containing a description. According to this definition, one can use terms such as requirements specification, design specification, or test specification [3]. Since the focus of our research is on the early stages of the software process, *software specification* refers in this paper to software requirements and software design specifications. Also as a matter of terminology, this paper follows [4] to make distinction between formal, semi-formal, and informal specification techniques. According to [4], informal techniques "do not have complete set of rules to constrain the models that can be created," semi-formal techniques have well-defined syntax and their "typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities," and formal techniques have precise syntax and semantics and "there is an underlying model against which a description expressed in mathematical notation can be verified". Furthermore, to be precise, graphical notations can be formal, however, in this paper references are solely to the larger group of semi-formal and informal graphical notations.

Our initial approach for dealing with software specification, described in detail in [5], is based on the idea of combining informality or semi-formality (quick, easy, but imprecise descriptions) with formality (heavy, mathematics-based, but rigorous representations) in describing what a system is supposed to do (or not to do). This approach, investigated further in recent work [6, 7, 8, 9], has led to several preliminary results that suggest that

[1] This work was supported in part by a grant from the University of Nevada, Reno Junior Faculty Research Grant Fund. This support does not necessarily imply endorsement by the University of research conclusions.

0-7803-8819-4/04/$20.00 ©2004 IEEE.

there is significant potential in combining modeling notations in software specification. To support combinations of graphical notations such as UML [10] with formal notations such as Z++ [11, 12] translation algorithms between the two types of notations are necessary. This paper presents details of rules, principles, and implementation solutions for such algorithms. It is worth noting that although the translation algorithms described in this paper are designed for the UML/Z++ combination, their underlying principles and rules are sufficiently 'notation-independent' to be reused in other similar combinations, in particular in combinations that involve UML as semi-formal graphical notation.

The remainder of this paper is structured as follows: section 2 briefly overviews the Harmony environment, section 3 summarizes the specification approach used in Harmony, sections 4 and 5 discuss the translation algorithms between Harmony's semi-formal and formal spaces (formalization and deformalization processes), section 6 identifies directions of enhancing the proposed algorithms, and section 7 finalizes the paper with several conclusions.

2. The Harmony Environment

Harmony, whose main interface is shown in Fig. 1, is a window-based integrated software specification environment that presents to the user a main window with three panes. On the left-hand side of the environment's main window the *project pane* shows the structure of the entire project. Next, in the center of the main window, the *UML space* allows the development of various UML model elements, including use cases, scenarios, sequence diagrams, class diagrams, classes, and statecharts. Lastly, on the right-hand side of the environment's main window, the *formal space* is used for editing the formal specifications that complement the graphical UML models of the software system being developed. Even though Fig. 1 shows a Z++ formal specification, other formal notations such as TLA+ (Temporal Logic of Actions Plus) [13] or Object-Z [14, 15, 16] could be considered as well. Harmony's functionality includes the capability of bi-directional (albeit partial) translation between the UML space and the formal space.

The Harmony project [6] has been pursued recently not only in terms of developing a new version of the environment but also in terms of enhancing the translation algorithms between the graphical and formal spaces. A major aspect of this ongoing project consists in adapting Harmony for multiple-notation software specification. More precisely, while in the graphical space the only notation used currently is UML, in the formal space provisions have been made to allow plugging-in diverse formal specification notations.

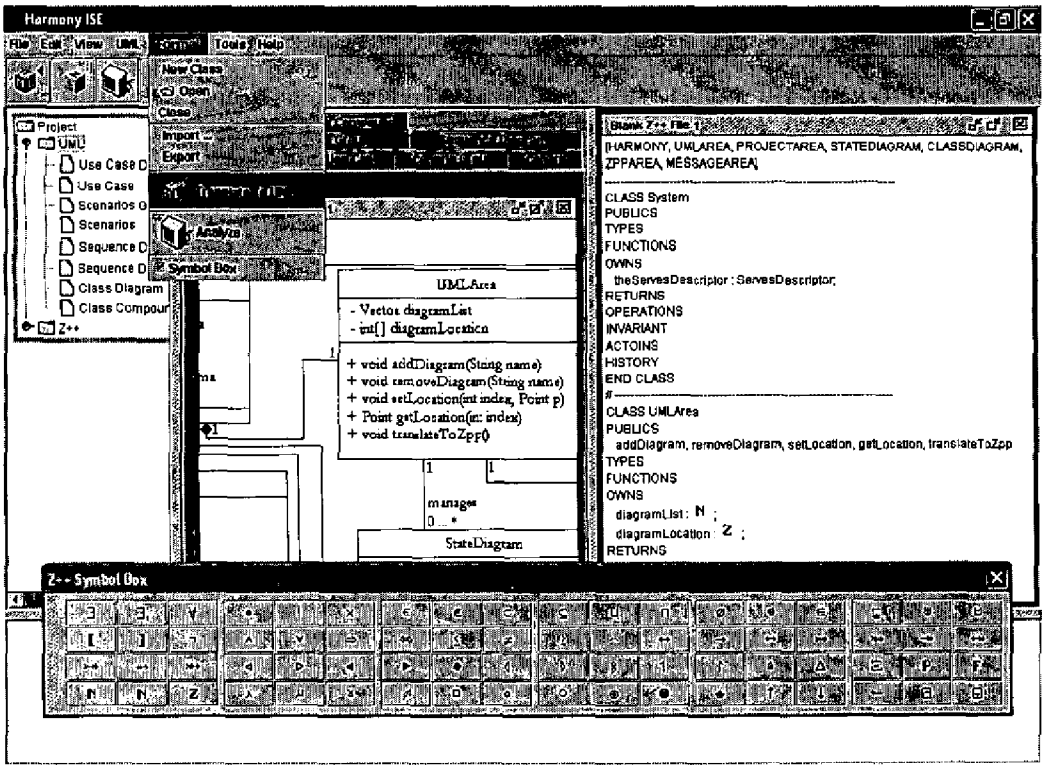

Figure 1. Harmony's user interface

For example, a notation currently considered as formal counterpart for UML in Harmony's workspaces is Object-Z. In the same line of work on combining semi-formal representations with formal notations in software specification, Carol Freinkel, a recent Master in Computer Science graduate from UNR, has worked on an solution for combining UML and TLA+. Details of this solution, illustrated on a cruise control application, can be found in [17].

3. The Specification Approach in Harmony

The modeling approach that we propose, emphasizing the combined use of semi-formal and formal notations in software specification, relies on two main components: a set of translation algorithms between UML and Z++ (algorithms for *formalization* and *deformalization*), and a set of specification steps (modeling activities). The latter define a procedural frame in which the software is specified, while the former are used during particular steps of this frame.

In this section, the artifacts that are produced during the combined semi-formal and formal specification of the system as well as the activities of the specification process are briefly described. More details on artifacts and steps, including justifications for selecting a specific subset of UML constructs and for focusing on a specific subset of modeling activities can be found in [5, 7, 9].

3.1. Artifacts

During the specification of the software system, a series of diagrams are drawn, modeling constructs are completed, including both UML and Z++ specification of classes, and the formalization and deformalization processes are performed.

Based on a set of requirements that describe the intended properties of the system, the following basic artifacts are created as components of the integrated semi-formal/formal model of the system: uses cases, scenarios, sequence diagrams, class compounds, and Z++ classes. While use cases, scenarios and sequence diagrams are typical UML model elements and Z++ classes are the fundamental constructs of Z++ specifications, *class compounds* are new model elements included in our approach to provide better support for the translation of UML models into Z++ specifications. In essence, the class compound is a simple yet practical extension of a regular UML class, obtained by attaching the associated state diagram to the class. As indicated in [8], the pairing of the two UML model elements, class and its associated state diagram, although conceptually simple, is nevertheless powerful in that it extends the fundamental notion of encapsulation, *data + operations*, to a more general description *data + operations + allowable sequences of operations*. Moreover, it corresponds to the class structure of formal specification languages such as Z++ or Object-Z, thus providing enhanced support for formalization.

In our approach, we have considered mechanisms for grouping logically related artifacts in addition to those provided by the UML. Specifically, with respect to the model elements employed in our approach, UML provides use case diagrams for gathering related use cases and class diagrams for grouping connected classes. In order to provide better support for formalization, we have introduced supplementary structuring mechanisms in the form of *groups* (e.g., scenario groups) and *collections* (e.g., use case collection, which is the set of all use case diagrams included in the system model).

A summary of the artifacts used in the Harmony approach, together with their abbreviations (included for easier referencing) is provided in Table I. Note that in this table the term *class description* (CLS) stands for the regular UML class and *class state diagram* (CLSTD) stands for the regular UML state diagram associated with the class. Together, they make the *class compound*, i.e., COMP = CLS + CLSTD. Class compounds are included in class diagrams and one or more class diagrams make up the *class diagram collection* CDC of the system model.

3.2. Modeling Activities

The steps of the specification approach that we propose can be described briefly as follows (this is a simplified representation of the suggested procedural frame). Details are available in [5, 7], where in addition to the regular flow of modeling activities summarized below examples of "irregular" sequencing of activities are given. Of course, as it is always the case in software development, iterations of steps and refinements of artifacts are necessary. Note that in the following basic ("regular") flow of modeling activities the deformalization step is not included because it is less frequently used. Also, note that the integrated system model (SM) obtained can be described concisely as:

$$SM = UML_{SM} + ZPP_{SM}$$

where the UML part of the system model is:

$$UML_{SM} = UCC + SCC + SQC + CDC$$

and the formal (Z++) part of the system model is:

$$ZPP_{SM} = ZSPEC.$$

Table I. Artifacts and their abbreviations

Element	Abbreviation
Use case	UC
Use case diagram	UCD
Use case collection	UCC
Scenario	SC
Scenario group	SCG
Scenario collection	SCC
Sequence diagram	SQD
Seq. diagram collection	SQC
Class description	CLS
Class state diagram	CLSTD
Class compound	COMP
Class diagram	CD
Class diagram collection	CDC
Z++ class	ZPPC
Z++ specification	ZSPEC

Table II. Modeling steps

Step 1	Definition of use cases, leading to the creation of the UCC.
Step 2	Elaboration of scenarios ("instances of use cases"), grouped in SCGs and making up the SCC part of the integrated model of the system.
Step 3	Construction of class diagrams, in which a rough sketch of the system's class structure is obtained.
Step 4	Specification of the sequence diagrams, in which components of the SQC part of the model are developed.
Step 5	Elaboration of class compounds, in which class descriptions and class state diagrams are detailed (the system structure sketched in Step 3 evolves into the detailed CDC part of the model).
Step 6	Formalization of UML class diagrams, involving the generation of the system's Z++ specification.
Step 7	Enhancement of the Z++ specification, performed by developers with expertise in formal methods. The ZSPEC part of the system model is completed.

4. Formalization: Translation Algorithms from UML to Z++

Regarding the translation processes between UML models and their corresponding Z++ specifications described in [5], we should note that emphasis is placed on the "direct", UML to Z++ translation, whose purpose is to increase the rigor of the system's description and help both developers and their customers gain a better insight of the system under construction. However, in order to make formal specifications easier to understand, during the integrated modeling of the system the "reverse translation," from Z++ to UML, is also considered. The first type of translation, *formalization*, applies both to UML class diagrams, which capture structural aspects of the system, and to UML state diagrams, which describe the system's dynamics. The second translation, *deformalization*, produces UML classes from the information contained in Z++ specifications.

For both types of translation process the focus has been on those parts of formalization and deformalization that can be performed automatically, a detailed set of translation principles and a translation algorithm based on these principles being proposed for each process. Following are additional details on the two translation processes included in Harmony:

- **Formalization**. This includes formalization of UML *class diagrams* and formalization of UML *state diagrams*. In order to formalize UML class diagrams in Z++ a set of rules for developing well-formed class diagrams (including rules for classes and rules for relationships) and a set of translation principles (e.g., for translation of types, attributes, operations, and classes) have been proposed. These rules and principles have guided the design of an algorithm for formalizing class diagrams, **AFCD**, described in [5] in terms of formal input, formal output, and pseudo-code. AFCD was also implemented in Java. The formalization of UML state diagrams is based on a number of constraints on the contents of the state diagrams and a number of translation principles for states and transitions. An algorithm for formalizing state diagrams, **AFSD**, described also in terms of formal input, formal output, and pseudo-code has also been designed. Examples of translations rules between UML and Z++ used in AFCD are shown in Fig. 2. Pseudocode descriptions of AFCD implementations solutions are illustrated in Fig. 3.
- **Deformalization.** For the reverse translation process, Z++ to UML, several principles for assigning types and generating UML attributes, operations, classes, relationships, and state diagrams from a Z++ specification, as well as an outline of the **ADF** algorithm for deformalization have also been presented in [5].

Principles and guidelines for formalizing object-oriented semi-formal models were proposed by Lano

The name of each regular class is unique within the class diagram.	(6.7)
The name of each parameterized class is the same as the name of its binding classes but is distinct from the names of all other classes that belong to the class diagram.	(6.8)
The name of each binding class is the same as the name of the parameterized class it binds and the name of other binding classes that instantiate this parameterized class, but is distinct from the names of all other classes that belong to the class diagram.	(6.9)
Each parameterized class and each binding class has at least one class parameter.	(6.10)
Each formal class parameter and each actual class parameter is given only as a name.	(6.11)
Each instantiating class has the same number of parameters as the parameterized class it binds.	(6.12)
Each attribute has a name and, optionally, a type, a visibility, an initial value, and a property.	(6.13)
The name of each attribute of a class is distinct from the names of all attributes and operations that belong to the same class.	(6.14)
The visibility of an attribute is one of the following: public, protected, or private.	(6.15)
The property of an attribute is either changeable or frozen.	(6.16)

Figure 2. Examples of rules used in the UML to Z++ translation algorithm AFCD

and Haughton [11, 12]. They represent the starting point for the semi-formal to formal translation process we have developed, but it should be noted that Lano and Haughton's work was concerned with the formalization in Z++ of OMT models, so we have adapted and extended their approach to UML models. Also, we have attempted to provide a new description of formalization, through detailed sets of principles and algorithms, and have additionally tackled the reverse translation, which was not considered by Lano and Haughton.

Additional reference for the formalization processes we suggested has been provided by the work of Kim and Carrington on formalizing UML models in Object-Z [14, 15]. In particular, their formal Z description of UML class diagram constructs, preliminary to the translation procedure from UML to Object-Z, has helped us define and organize the rules for well-formed UML class diagrams presented.

The formalization of UML models applies only to the core elements of the language (class diagrams, classes, relationships, and state diagrams) but, as shown in studies published by authors who have worked on similar approaches, these constructs provide good insight into the system and allow formal reasoning about its properties [14, 18].

```
-- UML to Z++ translation of a class diagram
   procedure CDTranslate(CD:ClassDiagram;
                         ZPPS:ZPPSpec)
     begin
       -- process classes
          TranslateClasses(CD;ZPPS);
       -- process relationships
          TranslateRelationships(CD;ZPPS)
       -- apply hiding operations on Z++ classes
          ResolveVisibility(;ZPPS)
     end CDTranslate;

-- Translation of classes
   procedure TranslateClasses(CD:ClassDiagram;
                              ZPPS:ZPPSpec)
     begin
       -- inspect all classes in the class diagram
          for i = 0 to N-1 do
          -- translate regular and parameterized
             if (CD.C[i].ctype /= bind) then
             -- classes only (ignore binding classes)
               TranslateClass(CD,CD.C[i];ZPPS)
             endif;
          end for;
     end TranslateClasses;

-- Translation of an individual class
   procedure TranslateClass(CD:ClassDiagram,
                C:UMLClass; ZPPS:ZPPSpec)
       -- Z++ class to be created
          ZC:ZPPClass;
       begin
          -- create corresponding Z++ class
             AppendClass(C.name; ZPPC, ZC);
          -- if UML class is generic transfer formal
             if (C.ctype==para) then
             -- class parameters to Z++ class
                TransferCParams(C; ZC)
             endif;
          -- process parents and fill EXTENDS clause
             ProcessParents(CD,C; ZC);
          -- formalize attributes
             TranslateAttributes(CD,C; ZPPS,ZC);
          -- formalize operations
             TranslateOperations(CD,C; ZPPS,ZC);
          -- fill FUNCTIONS, OWNS, and ACTIONS
             PlaceAttributes(;ZC);
          -- fill FUNCTIONS, OWNS, and ACTIONS
             PlaceOperations(;ZC);
       end TranslateClass; -- work done on this class
```

Figure 3. Examples of pseudo-code descriptions for the translation algorithm AFCD

5. Deformalization: Translation Algorithms from Z++ to UML

In what regards the reverse translation, from Z++ specifications to UML constructs, it should be noted that it has a secondary role in the modeling process, its purpose being to make easier the interpretation of the integrated model by developers and users not trained in formal methods. This feature may or may not be used within a particular modeling context, but its inclusion in

the proposed approach allows a form of "reverse engineering," from formal specifications to semi-formal graphical descriptions. In practice, it is thus possible to have certain Z++ specifications developed first and then their class structure propagated into the UML space. This allows an improved communication between developers skilled in formal methods and developers and users that favor the graphical representation of the system. The deformalization option is not a regular feature in integrated approaches and its practical utility is smaller than that of formalization. Several principles for translating Z++ classes into corresponding UML classes are shown in Fig. 4.

Each class C in Z++ that is not a descriptor of an association will have a corresponding class C in UML. If the Z++ class C has an associated hiding class H_C in Z++, the list of hidden features used in the hiding operation that defines H_C will be employed to assign the visibility private to the corresponding features (attributes and operations) of the UML class C.
Each generic class G in Z++ will be translated to a generic class G in UML, the names of the formal class parameters of the Z++ class G being used as names for the formal class parameters of the UML class G.
A binding UML class G[actual_params] will be created whenever a type G[actual_params] is encountered in the Z++ specification, with G matching the name of an existing generic class G in Z++ and the number of actual parameters actual_params equal to the number of the formal parameters of the Z++ class G (however, the names of the actual_params should not be the same with the names formal_params of the generic class).
The attributes and the operations of each regular or parameterized UML class will be obtained as indicated in [5], based on the inspection of the corresponding Z++ class.

Figure 4. Examples of principles used in the Z++ to UML translation algorithm ADF

6. Analysis

Several notes regarding the application of the proposed AFCD, AFSD, and ADF algorithms for formalization and deformalization are necessary.

First of all, while the focus in this paper was on those aspects of translations between UML and Z++ that can be automated, it is necessary to mention that the proposed algorithms are intended only to serve as aids during the modelling process, and not to substitute the human developer. In fact, we can hardly stress enough the importance of the human factor in the process of formalization (and, generally, in the development process), the quality of the software product depending essentially on the skills and motivation of its developers. Also, while we assign a prominent role in the modeling process to the activities of formalization and deformalization, the emphasis is not that much on automated translations between UML and Z++, but on the combined, efficient use of the two notations.

In practical terms, the current algorithms need be further refined in several aspects. In particular, in conjunction with the new Harmony environment, whose design incorporates the mechanics of translation presented in this paper, the following issues need to be addressed.

First, while the AFCD applies to class diagrams, for practical purposes it is necessary to allow the formalization of a single class or of a selected group of classes. The solution for this is to allow the AFCD to continue to operate within the context of the class diagram and to visually mark in the generated Z++ specification all the references made from within the group of formalized classes to classes outside this group (e.g., by including a comment listing the names of referenced but not formalized classes). This would allow the developer to decide if additional classes need to be formalized.

Second, also regarding the AFCD, its application to two or more related class diagrams need to be considered. This is not so much an issue of the algorithm itself as it is an issue of combining and representing the related class diagrams in the environment that uses the AFCD. The problem resides in classes included in one diagram that are in relationships with classes from another class diagram. The suggested solution is to attach a descriptor to the class indicating the relationships in which the class is involved, irrespective of the class diagram.

Third, the combined use of the AFCD and of the AFSD can also be improved. At this point in time, AFCD is applied first, followed by the AFSD, the latter algorithm only appending information in a Z++ class created by the former. The AFSD can be extended without difficulty to create itself the target Z++ class and, more generally, the operations of both algorithms can be integrated in a single formalization algorithm. Since the same translation principles apply and the data structures used by the algorithms is already in place this integration should be straightforward.

Fourth, regarding the AFSD, its extension to composite and concurrent states is a topic that deserves further investigation. The first thing in such extension is to create an enumerated type for each composite state in the state diagram, with an attribute of this type describing the current local state. Then, more complex descriptions of transitions are necessary. Parallel executions can be expressed via the || operator available in RTL [19], used in the HISTORY clause of Z++ class.

Finally, the combined use of the three algorithms, AFCD, AFSD, and ADF, is to be considered in an integrated environment. The main issue is the "update problem," which arises when a model is switched back

and forth between the two spaces, graphical and formal. The solution, similar to the one used in version control systems, is to let the developer decide on committing the changes. To help his or her decision, things to be added can be marked in a specific way (e.g., with an indicator such as >>> meaning *in* or new information) and things to be removed in an another way (e.g., with <<< meaning *out* or information to be discarded).

7. Conclusions

This paper has introduced the main principles, rules, and implementation solutions for translation algorithms between the graphical models developed using UML in the semi-formal space of the Harmony environment and the corresponding Z++ specifications in the formal space of this environment. Examples of applying the algorithms for software modeling are available in [5, 7] and more applications are currently being developed by the authors.

It is useful to note that even though the algorithms have been focused on UML to Z++ translations, they can be easily adapted and reused in other combinations of notations involving semi-formal representations (in particular, UML) and formal object-oriented languages (for example, Object-Z). As such, they could provide support for newer CASE tools such as the one described in this paper.

The benefits of developing and enhancing the algorithms presented stem primarily from their increased support for automated translation between the two components of a combined graphical (semi-formal) and formal integrated software model. As such, adjusting the levels of informality and, respectively, formality in a software specification becomes easier. This could have significant advantages in practice because different software development applications may have different priorities (for example, time to market, product reliability, product maintainability), which can be better accommodated by a software specification approach based on combination of modeling notations.

8. References

[1] Sommerville, I., *Software Engineering*, 7th edition, Addison-Wesley, 2004.

[2] Pressman, R., *Software Engineering: A Practitioner's Approach*, 6th edition, McGraw-Hill, 2004.

[3] Davis, A., *Software Requirements: Objects, Functions & States*, Prentice Hall, 1993.

[4] Fraser, M.D., Kumar, K., and Vaishnavi, V.K., "Strategies for Incorporating Formal Specifications in Software Development," *Communications of the ACM*, 37 (10), October 1994, pp. 74-85.

[5] Dascalu, S.M., *Combining Semi-Formal and Formal Notations in Software Specification: An Approach to Modelling Time-Constrained Systems*, PhD thesis, Dalhousie University, Halifax, NS, Canada, 2001.

[6] Dascalu, S.M., and Hitchcock, P., "Harmony: An Environment for the Combined Use of UML and Z++ in Software Specification," in Parsons, J., and Sheng, O. (editors), *Procs. of the 11th Workshop on Information Technologies and Systems* (WITS 2001), New Orleans, LA, Dec. 2001, pp. 103-108.

[7] Dascalu, S.M., and Hitchcock, P., "An Approach to Integrating Semi-formal and Formal Notations in Software Specification," *Procs. of SAC 2002, the ACM Symposium on Applied Computing*, Madrid, Spain, March 2002, pp. 1014-1020.

[8] Dascalu, S.M., and Hitchcock, P., "Towards Enhanced Description of Objects' Behavior: The Class Compound Model Element," *Procs. of the 2002 Intl. Conference on Software Engineering Research and Practice*, Las Vegas, NV, June 2002, pp. 240-245.

[9] Dascalu, S.M., Hitchcock, P., and Vert, G., "Combining Graphical Representations and Formal Notations in Software Specification: A Case Study," *Procs. of the 2003 Intl. Conf. on Software Engineering Research and Practice*, Las Vegas, vol. II, pp. 483-489.

[10] OMG's *UML Resource Page*, Object Management Group web site, www.omg.org/uml (September 2004).

[11] Lano, K., and Haughton, H., "Object-Oriented Specification Languages in the Software Life Cycles," in Lano, K., and Haughton, H. (eds.), *Object-Oriented Specification Case Studies*, Prentice Hall, 1994, pp.55-79.

[12] Lano, K., *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[13] Lamport, L., *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2003.

[14] Kim, S.-K., and Carrington, D., "A Formal Mapping between UML Models and Object-Z Specifications," in Bowen, J.P., Dunne, S., Galloway, A., and King, S. (editors), *Intl. Conf. of B and Z Users ZB2000, LNCS-1878*, Springer-Verlag, Berlin, Feb. 2000, pp. 2-21.

[15] Kim, S.-K., and Carrington, D., "An Integrated Framework with UML and Object-Z for Developing A Precise and Understandable Specification: The Light Control Case Study," *Procs. of the 7th Asia-Pacific Software Eng. Conf. ASPEC 2000*, pp. 240-248.

[16] Mahony, B., and Dong, J.S., "Timed Communicating Object Z," *IEEE Transactions on Software Engineering* 26 (2), Feb. 2000, pp. 150-176.

[17] Freinkel, C., *An Approach to Combining UML and TLA+ in Software Specification*, MSc thesis, University of Nevada, Reno, USA, Dec. 2003. Available as of September 2004 at www.cs.unr.edu/ paw2.0/techreports/ TRCS-2003-0001.pdf

[18] Howerton, W., and Hinchey, M.G., "Using the Right Tool for the Job," *Procs. of the 6th IEEE Intl. Conf. on the Engineering of Complex Computer Systems*, Sep. 2000, pp. 105-115.

[19] Jahanian, F. and Mok, A.K., "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering* 12(9), Sep. 1986, pp. 890-904.