Chapter 1

OBJECT-ORIENTED TRANSFORMATION

Kenneth Baclawski

Northeastern University
Boston, Massachusetts 02115
kenb@ccs.neu.edu

Scott A. Deloach

Air Force Institute of Technology
Wright-Patterson AFB, Ohio 43433
Scott.Deloach@afit.af.mil

Mieczyslaw M. Kokar

Northeastern University
Boston, Massachusetts 02115
kokar@coe.neu.edu

Jeffrey Smith

Sanders, a Lockheed Martin Company
Nashua, New Hampshire
jeffrey.e.smith@lmco.com

Abstract Modern CASE tools and formal methods systems are more than just repositories of specification and design information. They can also be used for refinement and code generation. Refinement is the process of transforming one specification into a more detailed specification. Specifications and their refinements typically do not use the same specification language. Code generation is also a transformation, where the target language is a programming language. Although object-oriented (OO) programming languages and tools have been available for a long time, refinement and transformation are still based on grammars and parse trees. The purpose of this paper is to compare grammar-based transformation with object-oriented transformation and to introduce a toolkit that automates the generation of parsers and transformers. A more specific objective is to apply these techniques to the problem of translating a CASE repository into logical theories of a formal methods system.

Keywords: CASE tool, formal methods, specification, modeling language, transformational reuse, code generation, context-free grammar

1. INTRODUCTION

In this chapter, we discuss the problem of transformation of object-oriented representations into formal representations. We encountered such a problem while attempting to translate UML diagrams [BRJ97a, BRJ97b] into formal specifications expressed in the formal specification language Slang [W$^+$98]; this step was part of the process of formalization of the UML. In order to simplify this rather complex task, we wanted to take advantage of existing translation tools, like Refine[1] [Ref90]. Our goal, in addition to the translation, was also to establish a formal semantics for the UML and to prove the correctness of the translation.

It is well known that UML diagrams, by themselves, are insufficient for representing the semantics of a software system. Additional conditions (such as pre- and post-conditions) are required. Establishing a formal semantics for the UML would clarify the meaning and limitations of the diagrams as well as eliminate ambiguities and conflicts between different diagrams.

One possible way of performing such a translation would be to translate data models of UML directly into expressions in the Slang grammar – a one-big-leap-transformation approach. Even if we establish a clear representation for the UML data models and use the Slang grammar, the process of such a direct translation would be quite complex. The complexity of this step can be reduced by decomposing it into a number of smaller simpler steps. Another reason for such a multi-step approach is that there is no single tool that could be used in this process. On the other hand, a number of excellent tools exist that could be used for smaller steps.

Existing tools can be used to generate a parser for a given context-free grammar. However, as we discuss in Section 2, context-free grammars only specify syntax, not semantics. In our case, using such a tool involves translating an object-oriented representation, with all its rich semantics, to a context-free grammar which has no semantics at all. Accordingly, there are then two ways to achieve our goal: either represent UML as a context-free grammar and then perform the translation(s) in the category of context-free grammars, or perform translation(s) of UML using only object-oriented representations, representing the result using a context-free grammar as the last step if necessary.

In this paper, we argue for the latter solution. In Section 2, we show an example of an object-oriented diagram and discuss the difficulties with representing this kind of diagram using context-free grammars. Then in Section 3 we describe a system, called **nu&**, developed at Northeastern University by K. Baclawski. The **nu&** toolkit is the basis for our object-oriented approach to parsing and transformation. We use this approach specifically for translating UML to Slang. The translation is decomposed into a number of smaller stages, each of which involves translation, parsing and symbol table manipulation. The two processing paths mentioned above – translation of data models and translation of context-free grammars – are discussed in detail. In Section 5, a specific example is used to illustrate the steps in the transformation pipeline of Section 3. The intent here is to show that the process of transformation of object diagrams is much simpler if it is carried out directly on the object level than by continually constructing linear textual representations which must be parsed before the next stage of the transformation may be performed.

Simplifying the transformation pipeline is one of the main themes of this paper. Certainly simplification has many obvious benefits. Simplification makes it easier to construct the transformation and to prove that it is correct Simplification also makes it easier to comprehend the transformation and to compare alternatives. This is especially important for a formalization of the UML because the UML is only a semi-formal modeling language. For example, the concept of inheritance varies from one programming language to another, and there can be several alternatives within a single language such as virtual and nonvirtual derivation in C++.

2. COMPARISON OF GRAMMARS WITH OBJECT-ORIENTED DATA MODELING LANGUAGES

Context-free grammars (also known as abstract syntax trees or ASTs) are the basic formalism for expressing modern programming languages. The first step in the compilation of a program is to parse the program as a sentence in the language defined by the grammar. The results of the parsing step are passed to the later phases of the compilation process. Translation from one language to another begins with parsing, when the source language is defined by a context-free grammar. The grammar is said to define the *syntax* of the language, while the subsequent phases of compilation are said to represent the *semantics* of the language. Excellent tools are available that automate the task of generating a parser from a grammar. Such tools are often called "compiler-compilers" even though they only automate the generation of the parser. To specify the semantics of the language with a compiler-compiler, one must specify the action associated with each grammar rule.

The result of parsing is often referred to as the *parse tree.* A parse tree is a hierarchical representation of information that conforms to a data model defined by the grammar. The fact that a grammar defines a data model was first observed by Gonnet and Tompa [GT87], whose p-string data model has powerful query operations for grammatical data models. Since then, there has been much work on elaborate grammatical data modeling languages, such as SGML and HTML/XML. For a detailed discussion of the limitations of grammars as data models see [Bac91]. The reverse of parsing represents a parse tree as linear text. This process is *linearization* or "pretty printing."

The rest of this section presents an example to compare the modeling power of grammars with object-oriented modeling languages.

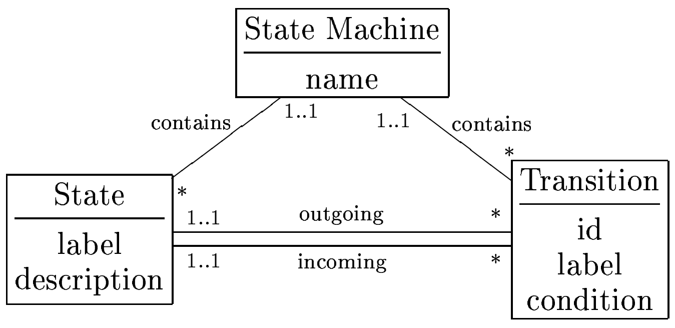

Figure 1 State Machine Data Model

Consider the example of a database of state machines as specified in Figure 1. This figure uses a simplified form of the UML notation to define a data model. The data model in the figure is similar to the state machine concept, but it is not the same, for the sake of simplicity. Each state and each transition is contained in a state machine, and each transition links exactly two states. State machines, states and transitions have various attributes. The name of a state machine is unique. The identifier of a transition is unique within the state machine that contains it. There is no requirement that a transition join states in the same state machine. If a transition joins states in the same state machine, then the transition is contained in the same state machine as the states. If a transition joins states in different state machines, then the transition can be contained in the state machine of either state.

One can represent an instance of the state machine data model as a parse tree in a variety of ways. One could represent it as a list of state machines, each of which contains a list of states and transitions. In addition, each transition is related to exactly one incoming state and exactly one outgoing state. This suggests that the following grammar represents the state machine data model:

$$\begin{aligned}
\text{Root} &\leftarrow \text{State\_Machine}^* \\
\text{State\_Machine} &\leftarrow \text{string State}^* \text{ Transition}^* \\
\text{State} &\leftarrow \text{string string} \\
\text{Transition} &\leftarrow \text{string string string State State}
\end{aligned}$$

A subtle problem with the above grammar is that the state objects contained in a transition object are different objects from the ones contained in the state machine objects and those contained in the other transition objects. The nonterminals of a grammar represent nodes in a tree, and the nodes that occur below a Transition nonterminal cannot occur below a State_Machine nonterminal or below another Transition node. Such an arrangement would violate the requirement that the parse tree be a tree. One could, in theory, add the constraint that each state linked by a transition must have the same information as one of the states contained in a state machine. Aside from the huge amount of redundancy that is caused by this design, it is also ambiguous because there could be states that have exactly the same attributes, since there is no uniqueness condition imposed on the states.

Alternatively, one might try to represent the relationships between states and transitions by including lists of incoming and outgoing transitions in each state, but now it is the transition objects that are being redundantly represented. Yet another possibility is to represent the two relationships as two independent entities. This design is even worse than the others, for now one is representing both the state objects and the transition objects redundantly.

In order to represent the incoming and outgoing relationships of the state machine data model, it is necessary to introduce some kind of reference mechanism. For example, instead of having two state objects within each transition object, one might specify that each transition object contain two state identifiers. This would work if states had unique identifiers, but there is no uniqueness condition on the state attributes. In the grammar above transition objects are uniquely identified within each state machine, so a compound identifier consisting of a state machine name and a transition id will uniquely identify each transition, because state machine names are unique. Assuming that most transitions will be contained in the same state machine as the states being linked, one should also allow transition references to consist of just a transition id which can be disambiguated by the context. The following is the grammar in this case:

$$\text{Root} \leftarrow \text{State\_Machine}^*$$

$$\begin{array}{rcl}
\text{State\_Machine} & \leftarrow & \text{string State}^{*} \text{ Transition}^{*} \\
\text{State} & \leftarrow & \text{string string transition\_ref}^{*} \text{ transition\_ref}^{*} \\
\text{Transition} & \leftarrow & \text{string string string} \\
\text{transition\_ref} & \leftarrow & \text{string} \mid \text{string string}
\end{array}$$

It appears that one has, at last, fully represented the original data model of Figure 1 as a grammar. However, a number of important considerations are not included in the grammar specification. The strings occurring in each transition reference must occur as state machine names or as transition ids with the following additional constraints:

1. If just one string occurs, then it represents the transition id of a transition in the same state machine as the state.
2. If two strings occur, then the first must be a state machine name and the second is the transition id of a transition in that state machine.

These constraints must be enforced by actions triggered by the grammar rules.

If this example seems a little contrived, exactly the same issues arise in programming languages for which identifiers are used for variables and methods within classes and the same identifier may be used in different classes. In programming languages the disambiguation of identifiers is a very complex problem.

This example also shows that expressing an object-oriented data model in terms of a grammar typically results in a grammar that is much more complex and awkward than the data model. However, tree representations of data do have some advantages. There are easily available tools for automatically generating parsers from a grammar, and there are several tools for transforming trees in one grammar to trees in another grammar.

3. A NEW APPROACH TO TRANSFORMATIONS

The purpose of the `nu&` Project [Bac90a, Bac90b, BMNR89] is to provide automated support for transformations from one language to another with emphasis on object-oriented modeling languages. This project combined the advantages of automated parser generation with the modeling power of object-oriented data models. Like grammar-based compiler-compilers, the `nu&` tools automatically generate parsers. However, the `nu&` toolkit uses the more powerful object-oriented data models rather than grammars, and the `nu&` toolkit transforms linear text directly into an object-oriented data structure. The toolkit can also be used to linearize an object-oriented database. Parsing and linearization of object-oriented data structures are similar to the marshaling and unmarshaling of data structures in remote procedure call mechanisms. The main distinction between RPC and the `nu&` toolkit is that `nu&` allows one to specify details about the grammar that is produced so that the resulting linear representation is readable. RPC linear representations, by contrast, are neither flexible nor intended to be read by people.

While the automated generation of parsers and linearizers is a useful feature, the main function of the `nu&` toolkit is to support transformations from one modeling language to another. In this respect, the `nu&` toolkit is similar to transformational reuse systems, such as Refine [Ref90], except that `nu&` supports a large variety of data modeling languages, including object-oriented data models while existing transformational reuse systems are grammar-based.

One of the problems with traditional approaches to transformations is the insistence on communicating using linear text. This is fine for simple transformations and has proved to be very effective in environments, such as the Unix shell, where "pipelines" join together relatively simple transformations to form more complex transformations. For example, `sort file | uniq -c | sort -nr | head -20` will compute the 20 most commonly occurring lines in a file. However, this technique becomes increasingly unwieldy as the complexity of the textual representation increases. For more complex languages, one requires a parser to produce a parse tree from the text, after which the identifiers in the parse tree must be disambiguated using a *symbol table*, and finally an internal (sometimes called an *intermediate* representation) is constructed. The intermediate representation is then processed to produce linear text to be used in the next stage of the pipeline.

Consider the problem of transforming a CASE tool diagram to a formal methods language. The traditional approach requires a series of transformational stages, each consisting of a series of steps. Each step involves processing output of the previous step. The whole process forms a pipeline of steps. To simplify the transformation, the diagram is first transformed to an object-oriented formal methods language, which is then transformed to a more traditional formal methods language. The formal specification can then be used to generate code in a programming language.

To illustrate the traditional transformational pipeline, we will use the example of the Slang formal methods language[W^{+}98], and the O-SLANG object-oriented formal methods language [DeL96]. The O-SLANG language was developed in [DeL96] as a target structure that could be later transformed into Slang. O-SLANG is based on the formalization of object-oriented concepts defined via a theory-based object model [DH99].

The full pipeline looks like that depicted in Figure 2. The middle column in this

$$\begin{tikzcd}
\text{CASE Diagram} \arrow[r, "\text{dump}"] & \text{Export Format} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"] \\
\text{Intermediate Structure} \arrow[r, "\text{translate}"'] & \text{O-Slang} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"] \\
\text{O-Slang Structure} \arrow[r, "\text{translate}"'] & \text{Slang} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"] \\
\text{Slang Structure} \arrow[r, "\text{code generate}"'] & \text{Programming Language} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"] \\
\text{Intermediate Structure} \arrow[r, "\text{optimize}"'] & \text{Intermediate Code} \arrow[r, "\text{generate}"] & \text{Executable Code}
\end{tikzcd}$$

Figure 2 Transformation Pipeline

figure consists of the various linear representations that act as the communication language between the processing modules in the pipeline. The original diagram is dumped to a standard format of some kind. This standard format is parsed, and the identifiers placed in a symbol table, so that when one is encountered, it can be replaced with a reference to the object being referenced. The result is an intermediate structure which is essentially equivalent to the original diagram. This structure is then translated to the O-SLANG object-oriented formal methods language and given to the O-SLANG compiler. The same kind of parsing and symbol table manipulation is then performed so that O-SLANG can be translated to the Slang formal methods language, which is then used to generate code in a programming language. Finally, the programming language is compiled. A specific example of the transformation pipeline in Figure 2 is given in Section 5 below.

While many of the steps in the pipeline of Figure 2 are important, many of them represent duplication of effort. None of the steps in the traditional transformational pipeline are easy for nontrivial languages, and any one of the steps is a source of error. Proving the correctness of the entire pipeline is a difficult task. Reducing the number of steps is certainly desirable in itself, and this is one of the primary motivations for the `nu&` approach.

Using the `nu&` toolkit, one can make significant simplifications to the transformational pipeline of Figure 2. In Figure 3, the CASE diagram is isomorphic to a CASE tool's intermediate object structure. This structure is typically translatable to any

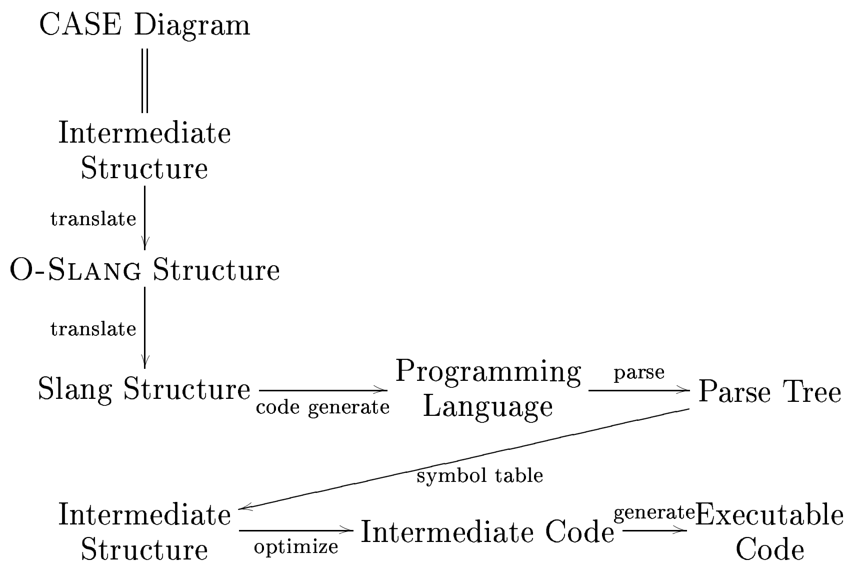

Figure 3 Simplified Transformation Pipeline

kind of new structure by a vendor-provided scripting language. Rather than translate the CASE diagram to text in any form (as suggested by Figure 2), the `nu&` approach is to translate directly to the O-SLANG structure using object-oriented techniques and to continue to translate entirely at the level of data structures (i.e., the left column of Figure 3). Unfortunately, it is difficult to streamline the entire transformation pipeline because one rarely has access to all of the internal data structures. For example, it is not currently possible to circumvent the parser of a compiler and present it with its intermediate representation directly.

Another possibility for simplifying Figure 2 would be to transform at the level of the parse tree (i.e., the right column in Figure 2). This is the approach taken by traditional transformational code generation systems such as Refine [Ref90] and GenVoca [BO92, BG97]. While this approach is certainly simpler than the original pipeline, it has the disadvantage that the translation code must deal with the table of identifiers, so that identifier lookup and resolution must be handled at the same time as the transformation. Another disadvantage is that the parse tree structures (right column in Figure 2) are generally more complex and unwieldy than the internal data structures (left column in Figure 2).

4. THEORY-BASED OBJECT MODEL

In object-oriented systems, the object *class* defines the structure of an object and its response to external stimuli based on its current state. In our theory-based object model, we capture the structure of a class as a theory presentation, or algebraic specification, in O-SLANG, an object-oriented algebraic specification language. Figure 4 shows some other correspondences between UML notions and theory-based object model notions.

O-SLANG Component	Meaning
sort	collection of values
class type	structure of an object and its response to stimuli
class sort	all possible value representations of objects of the class
abstract class	class with no direct instances
concrete class	blueprint for instances
attribute	function that returns data values or objects – an observable class characteristic
axiom	class attribute value invariant or specification of a function's semantics
state sort	all possible states of an object
state attribute	function mapping from class sort to state sort
state invariant	constraint on class attributes in a given state

Figure 4 Some Components of the Theory-Based Object Model

UML, O-SLANG and Slang must be expressed at the metalevel for any transformation to be possible. In the traditional transformation pipeline, the metalevel is expressed using a grammar and implicit constraints. The `nu&` approach uses an object-oriented definition of the metalevel. The UML metalevel is defined by the UML Semantics Guide [BRJ97a, BRJ97b]. The O-SLANG metalevel is discussed in [DeL96]. Excerpts from the grammar defining the O-SLANG metalevel are given in the Section 5 below.

For example, the meta-class for the O-SLANG class concept is defined as follows.

```
class OSlangClass {
  String name;
  Sort classSort;
  Set<Operationdecl> operations;
  Set<OSlangClass> imports;
  Set<Sort> sorts;
  ...
}
```

Compare this definition with the grammar for O-Slang defined in the next section. The grammar can be generated by the `nu&` toolkit from the O-Slang metalevel class definitions.

We are building a collection of UML to Slang translation rules for the Core Package of UML. Because UML is only a semi-formal modeling language many modeling constructs have alternative choices for their semantics. Inheritance, for example, can be formalized in many ways, and we have developed a framework that includes most of the variations that have been used in object-oriented programming languages as well as many others. The variations include both structural variations such as virtual versus nonvirtual derivation in C++ and behavioral variations such as the many method dispatch mechanisms. These results will be appearing in subsequent reports and papers.

5. STATE MACHINE EXAMPLE

In this section, we will give an example of the traditional transformational pipeline outlined in the previous section. We then compare it to the `nu&` approach.

Traditional Pipeline Approach

This example is derived from [DeL96]. In UML, a state diagram is one technique for describing the behavior of a class. The objective in this example is to convert a state machine diagram to its corresponding O-Slang specification. In this example, we will use the class *pump* whose state diagram is given in Figure 5.

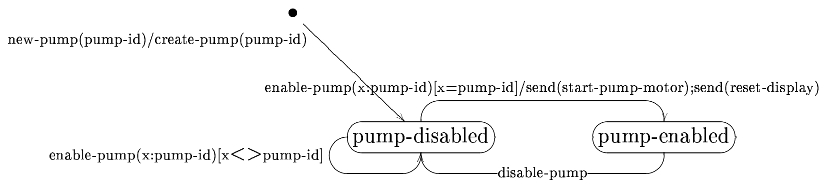
new-pump(pump-id)/create-pump(pump-id)
enable-pump(x:pump-id)[x=pump-id]/send(start-pump-motor);send(reset-display)
enable-pump(x:pump-id)[x<>pump-id]
pump-disabled
pump-enabled
disable-pump

Figure 5 Pump State Diagram

The CASE tool used by DeLoach was a commercially available object-oriented drawing package, ObjectMaker[2]. The textual output from ObjectMaker is parsed into a Refine parse tree using a Refine-based parser. Once in Refine, a rule-based conversion program translates the ObjectMaker parse tree into a Generic parse tree which is isomorphic to the original CASE diagram.

Once in the Generic parse tree, a rule-based transformation program implementing the transformation rules translates the Generic parse tree into an O-Slang parse tree within the Refine environment. Once in a valid O-Slang parse tree, the Dialect pretty printer is used to produce a textual representation of the O-Slang parse tree. The actual transformation is performed by creating the root node of the O-Slang parse tree and then automatically translating each class and association, one at a time, from the Generic parse tree to the O-Slang parse tree.

The actual Refine transformation code is more complex than even Figure 2 suggests. The Export Format of ObjectMaker has a structure that is complex enough to require an additional transformation stage. The actual transformation from CASE Diagram to O-Slang consists of the pipeline shown in Figure 6. The Refine tool

$$\begin{tikzcd}
\text{CASE Diagram} \arrow[r, "\text{dump}"] & \text{Export Format} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"'] \\
\begin{array}{c}\text{Export}\\\text{Structure}\end{array} \arrow[r, "\text{convert}"'] & \begin{array}{c}\text{Intermediate}\\\text{Format}\end{array} \arrow[r, "\text{parse}"] & \text{Parse Tree} \arrow[dll, "\text{symbol table}"] \\
\begin{array}{c}\text{CASE Diagram}\\\text{Structure}\end{array} \arrow[r, "\text{translate}"'] & \text{O-SLANG} \arrow[r, "\text{parse}"] & \begin{array}{c}\text{O-SLANG}\\\text{Parse Tree}\end{array}
\end{tikzcd}$$

Figure 6 Actual Transformation Pipeline from CASE Diagram to O-Slang

allows some of the steps in the pipeline to be combined, but it is still necessary to write (and debug) five separate Refine specifications to achieve the entire transformation from CASE Diagram to O-Slang. Several hundred lines of code are needed for specifying the rules for transforming a state machine diagram. We now show some excerpts from this code.

The grammar for the dynamic model portion of a Generic class is the following:

```
Generic-Class = <name, {Superclass}, [Connection], {Attribute}, {State},
                {Transition}, {Axiom}, {Operation}, {Function}>
State = <name, {State}, {Axiom}>
Transition = <name, [Parameter], Axiom, {Action}, FromState, ToState>
FromState = name
ToState = name
Action = <name, [Parameter], {Action}>
Parameter = <name, datatype>
```

A simplified version of the O-Slang grammar is shown below. Notice that both StateAttr and State are defined as functions. StateAttr is a function that takes an object as its domain and returns a state value as its range. States are defined as nullary functions that return specific values of the state attribute.

```
Class = <name, ClassSort, {Operation}, {Import}, {Sort}, {Attribute},
        {Method}, {StateAttr}, {Event}, {State}, {Axiom}>
StateAttr = Operationdecl
State = Operationdecl
Operationdecl = <name, [Domain-Ident], [Range-Ident]>
Axiom = complex definition of 1st order predicate logic
```

There are three distinct steps to transforming the dynamic model from the Generic parse tree to the O-SLANG parse tree: (1) creation of state attributes, (2) creation of state values, and (3) creation of axioms that implement the transitions. For simplicity, we will just consider the axioms for transitions. Translation of the Generic Transitions into O-SLANG axioms is performed by breaking down each Generic Transition object and processing it in five parts: the current state, transition guard, new state, method invocation, and the sending of any new events.

```
function create-oslang-transition-axiom (x: Transition) :
  Axiom-Def = let (s:object=undefined)
    s <- Make-OslangAxiom(
           concat(create-oslang-current-state-string(x),
           create-oslang-guard-string(x),
           create-oslang-new-state-string(x),
           create-oslang-method-invocation-string(x),
           create-oslang-send-event-string(x),
           ")"))
```

The five parts are concatenated into a string which is parsed into an O-SLANG axiom parse tree by the Make-OslangAxiom function of the form

$$\textit{old-state} \wedge \textit{guard-condition} \Rightarrow \textit{new-state} \wedge \textit{method-invocations} \wedge \textit{event-sends}$$

The final result of the pipeline is an O-SLANG parse tree which can be linearized into the following textual form:

```
class Pump is
  class-sort Pump
  import Pump-Id, Reset-Display, Start-Pump-Motor, Pump-Aggregate
  sort Pump-State
  attributes
    pump-id : Pump -> integer
    pump-state : Pump -> Pump-State
  operations
    attr-equal : Pump, Pump -> Boolean
  states
    pump-disabled : -> Pump-State
    pump-enabled : -> Pump-State
  events
    ...
  methods
    ...
  axioms
    pump-disabled <> pump-enabled;
    attr-equal(P1, P2) <=> (pump-id(P1) = pump-id(P2));
    (pump-state(P) = pump-enabled) =>
      (pump-state(disable-pump(P)) = pump-disabled);
    (pump-state(new-pump(P, A)) = pump-disabled
      & attr-equal(new-pump(P, A), create-pump(A)));
    ...
end-class
```

Object-Oriented Transformation

By contrast the transformation code using the nu& toolkit simply constructs each of the components occurring in the O-SLANG data structure as objects. One can use either rules or a series of nested loops to express the transformation. The following are some fragments of the code that illustrate the nu& nested loop approach:

```
for every c in allClasses {
  OSlangClass oclass = new OSlangClass (c.name);
  ...
  for every state in oclass.states {
    oclass.states.add (new State (state.name));
    ...
    for every transition in state.outTransitions {
      oclass.events.add (new Event (transition.name));
      oclass.axioms.add
        (new Axiom (transition.currentState && transition.guard,
                    transition.newState && transition.methodInvocation
                      && transition.sendEvent));
      ...
    }
  }
}
```

In addition to requiring fewer steps, the nu& approach involves much simpler code that focuses on the fundamental issues rather than myriad syntactic and symbol table issues.

6. RELATED WORK

Several authors have proposed techniques for transforming informal system requirements and specifications into formal specifications. Babin, Lustman, and Shoval proposed a method based on an extension of Structured System Analysis. The method uses a ruled-based transformation system to help transform the semi-formal specification into a formal specification [B$^+$91]. Fraser, Kumar, and Vaishnavi proposed an interactive, rule-based transformation system to translate Structured Analysis specifications into VDM specifications [F$^+$94]. In both cases, the output of the process is a text-based formal specification that would require parsing for further automated refinement.

Specware [Spe94] is a transformational program derivation system based on Slang [W$^+$98] which is the end target for this work. Specware provides the automated tool support for developing and transforming specifications using the Slang formal specification language. Once defined in Slang, all transformations – including algorithm design and optimization, data type refinement, integration of reactive system components, and code generation – are performed on an internal AST-based representation of Slang. However, Specware does not provide the front end as described in our research: an object-oriented, graphically-based semi-formal, community accepted representation.

Although not specifically concerned with formalization, there have been many research efforts and commercial products that support transformations from one language to another. Such tools are called transformational code generators or generative reuse tools. Krueger [Kru92] has a survey of such tools. Some of the most prominent among these tools are Batory's GenVoca [BO92, BG97], Neighbors' Draco [Nei84], and Reasoning Systems' Refine [Ref90]. While the output of these transformational systems can be object-oriented (e.g., by using components from and generating code in an object-oriented programming language), all of these systems use a specification language that is grammar-based. The `nu&` toolkit, by contrast, not only can generate object-oriented data structures, but also supports object-oriented specifications. As noted in Section 3, transforming object-oriented data structures is simpler, more powerful and less error-prone than transforming parse trees.

7. CONCLUSIONS

While object-oriented languages have become very popular in both programming and software specification, the formalism for representing their structure is still that of a context-free grammar, even though this formalism was developed mainly for a different kind of language. In this paper, we argued that for object-oriented representations data models are better suited than such context-free grammars. We showed with an example the difficulties involved in representing an object-oriented diagram using a context-free grammatical representation. We analyzed two possibilities for transforming object-oriented representations (UML diagrams) into formal non-object-oriented representations (Slang specifications):

1. Transform the data model of UML into a context-free grammar and then perform consecutive transformations in the realm of context-free grammars using CASE tools available for such translations, and
2. Translate the UML data model into an intermediate object-oriented representation and perform consecutive translations in the object-oriented domain, while translating into the context-free target language as the last step.

We argued for the latter approach. We showed that this approach is simpler in the sense that it consists of fewer transformational steps, and thus is less error-prone.

Notes

1. Refine is a trademark of Reasoning Systems Inc. Palo Alto California
2. ObjectMaker is a registered trademark of Mark V Systems Limited Encino California

References

[B+91] G. Babin et al. Specification and design of transactions in information systems: A formal approach. *IEEE Transactions on Software Engineering*, 17:814–829, August 1991.

[Bac90a] K. Baclawski. The `nu&` object-oriented semantic data modeling tool: intermediate report. Technical Report NU-CCS-90-18, Northeastern University, College of Computer Science, 1990.

[Bac90b] K. Baclawski. Transactions in the `nu&` system. In *OOPLSA/ECOOP'90 Workshop on Transactions and Objects*, pages 65–72, October 1990.

[Bac91] K. Baclawski. Panoramas and grammars: a new view of data models. Technical Report NU-CCS-91-2, Northeastern University College of Computer Science, 1991.

[BG97] D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23:67–82, 1997. DARPA and WL supported project under contract F33615-91C-1788.

[BMNR89] K. Baclawski, T. Mark, R. Newby, and R. Ramachandran. The `nu&` object-oriented semantic data modeling tool: preliminary report. Technical Report NU-CCS-90-17, Northeastern University, College of Computer Science, 1989.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, October 1992.

[BRJ97a] G. Booch, J. Rumbaugh, and I. Jacobsen. *UML Notation Guide, Version 1.1*, September 1997.

[BRJ97b] G. Booch, J. Rumbaugh, and I. Jacobsen. *UML Semantics*, September 1997.

[DeL96] S. DeLoach. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications.* PhD thesis, Air Force Institute of Technology, WL AFB, OH, June 1996. Ph.D. Dissertation.

[DH99] S. DeLoach and T. Hartnum. A theory-based representation for object-oriented domain models. In *IEEE Trans. Software Engineering*, 1999. to appear.

[F^+94] M. Fraser et al. Strategies for incorporating formal specifications. *Communications of the ACM*, 37:74–86, 1994.

[GT87] G. Gonnet and F. Tompa. Mind your grammar: a new approach to modelling text. In *Proc. 13th VLDB Conf.*, pages 339–346, Brighton, UK, 1987.

[Kru92] C. Krueger. Software reuse. *ACM Computing Surveys*, 24:131–183, June 1992.

[Nei84] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. Software Engineering*, pages 564–574, Sept. 1984.

[Ref90] *Refine 3.0 User's Guide*, May 25, 1990.

[Spe94] *Specware^{TM} User Manual: Specware^{TM} Version Core4*, October 1994.

[W^+98] R. Waldinger et al. *Specware^{TM} Language Manual: Specware^{TM} 2.0.2*, 1998.

About the Authors

Kenneth Baclawski is an Associate Professor of Computer Science at Northeastern University. His research interests include Formal Methods in Software Engineering, High Performance Knowledge Management, and Database Management. He has participated in and directed many large research projects funded by government agencies including the NSF, DARPA and NIH. He is a co-founder and Vice President for Research and Development of Jarg Corporation which builds Internet-based, high performance knowledge management engines.

Scott A. DeLoach is currently an Assistant Professor of Computer Science and Engineering and Director of the Agent Lab at the Air Force Institute of Technology (AFIT). His research interests include design and synthesis of multiagent systems, knowledge-based software engineering, and formal specification acquisition. Prior to coming to AFIT, Dr. DeLoach was the Technical Director of Information Fusion Technology at the Air Force Research Laboratory. Dr. DeLoach received his BS in Computer Engineering from Iowa State University in 1982 and his MS and PhD in Computer Engineering from the Air Force Institute of Technology in 1987 and 1996.

Mieczyslaw M. Kokar is an Associate Professor of Electrical and Computer Engineering at Northeastern University. His technical research interests include formal methods in software engineering, intelligent control, and information fusion. Dr. Kokar teaches graduate courses in software engineering, formal methods, artificial intelligence, and software engineering project. Dr. Kokar's research has been supported by DARPA, NSF, AFOSR and other agencies. He has an M.S. and a Ph.D. in computer systems engineering from Technical University of Wroclaw, Poland. He is a member of the IEEE and of the ACM.

Jeffrey Smith is a Senior Principle Engineer at Sanders and PhD candidate in Computer Systems Engineering at Northeastern University. His research interests include Formal Methods in Software Engineering, Operating Systems and High Performance Computing Frameworks. He has applied his research in defense based applications, as both a practitioner and manager for more than twenty years.