

Object-Oriented Parsing and Transformation

Kenneth Baclawski[‡] Scott A. DeLoach[‡] Mieczysław Kokar[§]
Jeffrey Smith[¶]

July 30, 1998

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 a specification or design from one formal or modeling language to another language, usually a less abstract and less formal one. 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, all refinement and transformational systems 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 expressed in object-oriented terms. 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, modeling language, transformational reuse, code generation, context-free grammar.

1 Introduction

In this paper we discuss the problem of transformation of object-oriented representations into formal representations in non-object-oriented languages. We encountered such a problem while attempting to translate UML diagrams [8, 9] into formal specifications expressed in the formal specification language Slang [18]; 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 Refine¹ [16]. Our goal, in addition to the translation, was also

*College of Computer Science, Northeastern University, Boston, Massachusetts 02115

†Research supported by the National Science Foundation Grant CCR 87-16485

‡Air Force Institute of Technology, Wright-Patterson AFB, Ohio 43433

§Department of Electrical and Computer Engineering, Northeastern University, Boston, Massachusetts 02115

¶Sanders, a Lockheed Martin Company, Nashua, New Hampshire

¹Refine is a trademark of Reasoning Systems Inc. Palo Alto California

to establish a formal semantics for the UML and to prove the correctness of the translation.

One possible way of performing such a translation would be to take data models of UML and translate them 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 both for parsing sentences of a language generated by a context-free grammar, and for automatic generation of a parser for a given context-free grammar. In our case, however, this involves the translation of an object-oriented representation to a context-free grammar. There are then two ways to achieve this goal: either to represent UML as a context-free grammar and then perform the translation(s) in the category of context-free grammars, or first perform translation(s) of the data model of UML, moving to a context-free grammar as the last step.

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 transforming, 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 4, 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. Finally, in Section 6 we summarize our conclusions.

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. More generally, any 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. That a grammar defines a data model was first observed by Gonnet and Tompa in [13], 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, more recently, HTML/XML. For a detailed discussion of the limitations of grammars as data models see [4]. The reverse of parsing transforms a parse tree into linear text. This process is *linearization* or “pretty printing.”

In the rest of this section we present an example to compare the modeling power of grammars with object-oriented modeling languages.

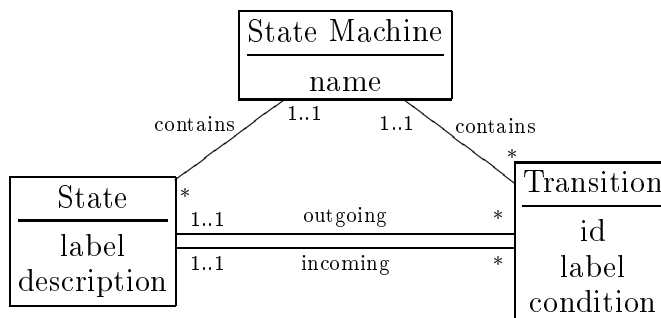


Figure 1: State Machine Data Model

Consider the example of a database of state machines as specified in Figure 1. This figure uses the UML notation. It is loosely based on the state machine concept in UML. Each state and each transition is contained in a state machine, and each transition links one state with another state. State machines, states and transitions have various attributes as shown in the figure. In addition, we impose a few uniqueness constraints. The name of a state machine is unique, and the id of a transition is unique within the state machine that contains it. Note that there is no requirement that a transition join states in the same state machine.²

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. The grammar for this kind of parse tree is represented below. For simplicity, we have omitted some technical details from the grammar. The “syntactic sugar” was omitted. This includes the various keywords and delimiters that are needed to make a grammar unambiguous and context-free. In addition, it is not shown how to parse

²In UML, transitions cannot join states in one state machine with those of another. However, they can join substates in one state with those of another. The example model is designed to capture this idea without introducing the full complexity of UML.

strings, such as names, ids and text descriptions.

```
Root ← State_Machine*
State_Machine ← string State* Transition*
State ← string string
Transition ← string string
```

To complete the representation of the State Machine data model, it remains to represent the relationships between states and transitions. Although each transition links exactly one outgoing state with exactly one incoming state, one cannot simply include two states in each Transition, as in the following grammar:

```
Root ← State_Machine*
State_Machine ← string State* Transition*
State ← string string
Transition ← string string State State
```

The problem with the grammar above is that the state objects contained in a transition object are different objects from the ones contained in the state machine objects and also from those contained in the other transition objects. This is a subtle point that can be easily missed. The nonterminals of a grammar represent nodes in a tree, and the nodes that occur below a Transition nonterminal cannot also 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, as in the following grammar:

```
Root ← State_Machine*
State_Machine ← string State* Transition*
State ← string string Transition* Transition*
Transition ← string string
```

However, this has the same problem as the previous grammar, except that it is now the transition objects which 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 a 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:

```

Root ← State_Machine*
State_Machine ← string State* Transition*
State ← string string transition_ref* transition_ref*
Transition ← string string
transition_ref ← string | string string

```

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. In particular, the strings occurring in each transition reference must occur as state machine names or as transition ids, such that if just one occurs then it represents the transition id of a transition in the same state machine as the state, while 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 requirements 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 a data model need not have any grammatical representation at all. If the state machines names were not unique or if transitions did not have to have ids (both of which are true in practice), then even compound identifiers would not uniquely determine transitions. To represent such a data model it would be necessary to augment the data model with artificial unique identifiers. In addition to modifying the original data model, such identifiers can have an adverse impact on the readability of the language defined by the grammar.

This example 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, if it is possible to express the data model at all. 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 The nu& Approach to Transformations

The purpose of the nu& Project [2, 3, 5] 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 not flexible and are not 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 [16], 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[18], and the O-SLANG object-oriented formal methods language [10]. The O-SLANG language was developed in [10] as a target O-SLANG 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 [11]. The theory-based object model formalized basic object-oriented concepts such as object classes, attributes, operations, state, inheritance, association, aggregation, and inter-object communication. The structure of object classes was captured using multi-sorted algebras while the relationships between classes were captured using mathematical notions from category theory..

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

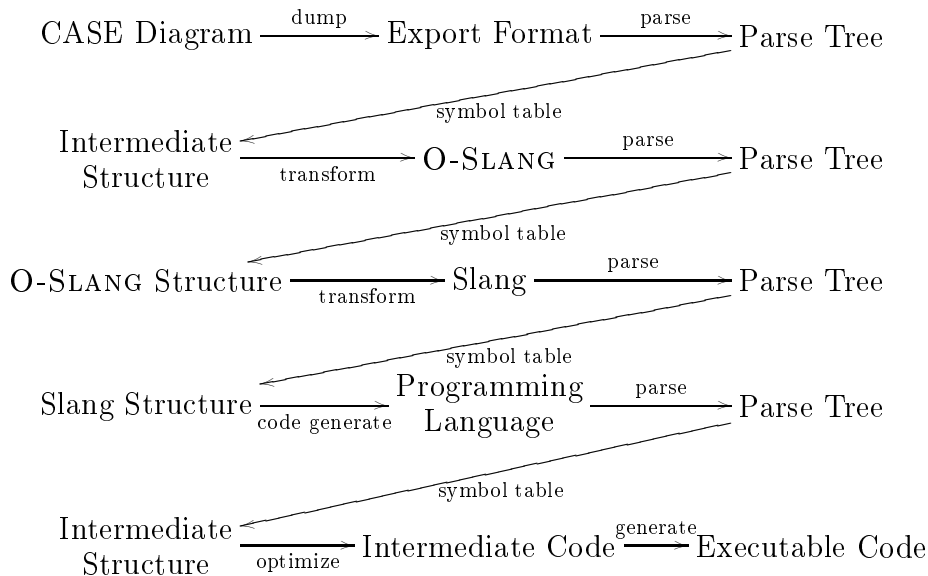


Figure 2: Transformation Pipeline

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 the same as 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 4 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. More surprisingly, reducing the number of steps could also reduce the complexity of individual steps. For example, the two data structures labeled “CASE diagram” and “Intermediate structure”

are, in principle, isomorphic.

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 kind of new structure

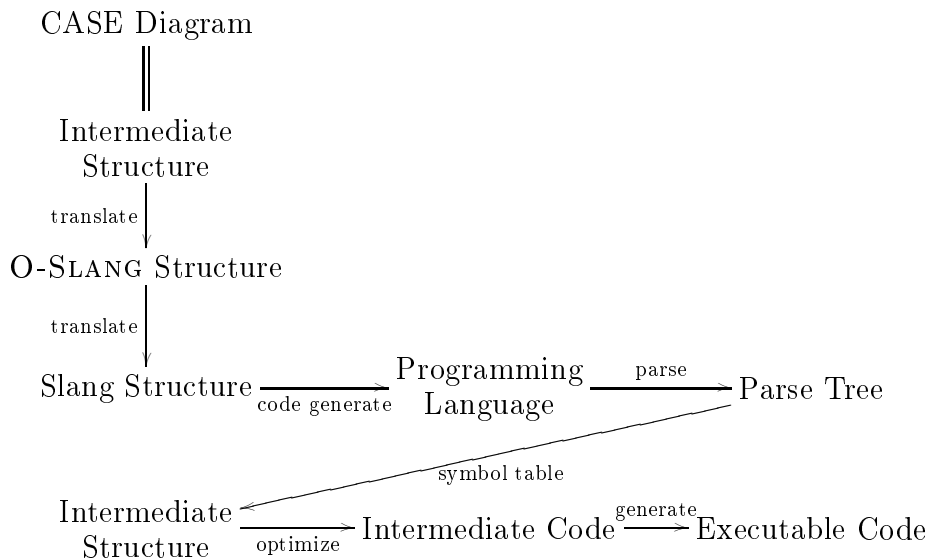


Figure 3: Simplified Transformation Pipeline

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` [16] and `GenVoca` [7, 6]. 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 disambiguation 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 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. This example is derived

from [10]. 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 4.

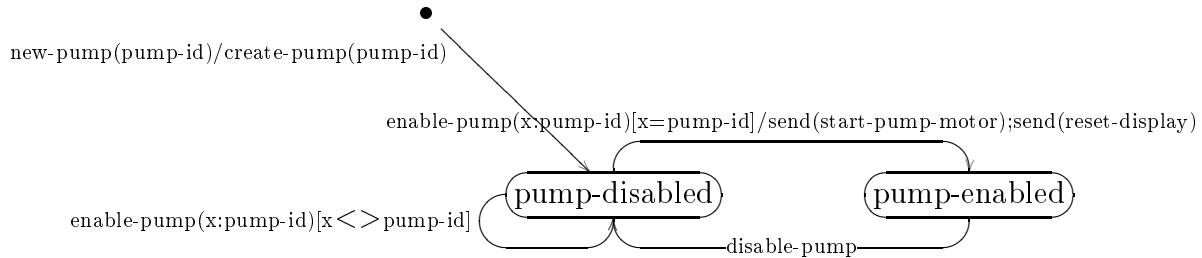


Figure 4: Pump State Diagram

The CASE tool used by DeLoach was a commercially available object-oriented drawing package, ObjectMaker³. 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 transforms 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 transforms 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 transforming 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 5. The Refine tool allows some of the steps

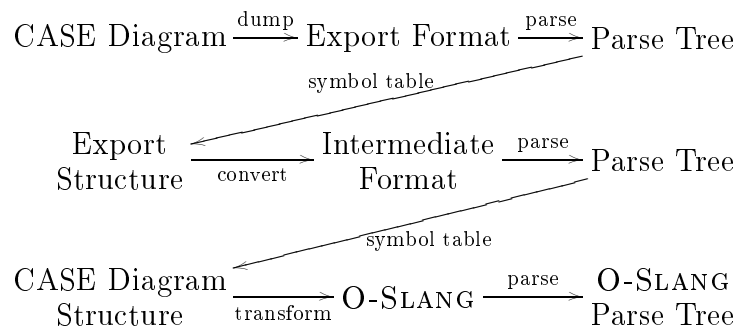


Figure 5: Actual Transformation Pipeline from CASE Diagram to O-SLANG

³ObjectMaker is a registered trademark of Mark V Systems Limited Encino California

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

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

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
  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
    enable-pump : Pump, integer -> Pump
    disable-pump : Pump -> Pump
    new-pump : integer -> Pump
  methods
    create-pump : -> Pump;
    enable-pump : Pump -> Pump;
  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)));
    (pump-state(P) = pump-disabled & (X = pump-id(P)))
      => (pump-state(enable-pump(P, X)) = pump-enabled);
    (pump-state(P) = pump-disabled & (X <> pump-id(P)))
      => (pump-state(enable-pump(P, X)) = pump-disabled);
    (pump-state(P) = pump-disabled)
      => (pump-state(disable-pump(P)) = pump-disabled);
    (pump-state(P) = pump-enabled)

```

```

=> (pump-state(enable-pump(P, X)) = pump-enabled);
end-class

```

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.addProperty (new State (state.name));
    ...
    for every transition in state.outTransitions {
      oclass.addProperty (new Event (transition.name));
      oclass.addProperty (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.

5 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 [1]. Fraser, Kumar, and Vaishnavi proposed an interactive, rule-based transformation system to translate Structured Analysis specifications into VDM specifications [12]. In both cases, the output of the process is a text-based formal specification that would require parsing for further automated refinement.

Specware [17] is a transformational program derivation system based on Slang [18] 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 [14] has a survey of such tools. Some of the most prominent among these tools are Batory's GenVoca [7, 6], Neighbors' Draco [15], and Reasoning Systems' Refine [16]. 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.

6 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. We also described `nu&`, a system that allows for direct transformation of UML into an object-oriented representation. We argued that this translation step is much simpler than the translation to a context-free grammar. As a result such a translation is less error-prone, a very important and desirable feature of a software tool.

References

- [1] 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.
- [2] 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.
- [3] K. Baclawski. Transactions in the nu& system. In *OOPLSA/ECOOP'90 Workshop on Transactions and Objects*, pages 65–72, October 1990.
- [4] K. Baclawski. Panoramas and grammars: a new view of data models. Technical Report NU-CCS-91-2, Northeastern University College of Computer Science, 1991.
- [5] 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.
- [6] 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.
- [7] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, October 1992.
- [8] G. Booch, J. Rumbaugh, and I. Jacobsen. *UML Notation Guide, Version 1.1*, September 1997.
- [9] G. Booch, J. Rumbaugh, and I. Jacobsen. *UML Semantics*, September 1997.
- [10] 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.
- [11] S. DeLoach, P. Baylor, and T. Hartnum. Representing object models as theories. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference (KBSE '95)*, pages 28–35, Boston, MA, November 1995.
- [12] M. Fraser et al. Strategies for incorporating formal specifications. *Communications of the ACM*, 37:74–86, 1994.
- [13] 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.
- [14] C. Krueger. Software reuse. *ACM Computing Surveys*, 24:131–183, June 1992.

- [15] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. Software Engineering*, pages 564–574, Sept. 1984.
- [16] *Refine 3.0 User's Guide*, May 25, 1990.
- [17] *SpecwareTM User Manual: SpecwareTM Version Core4*, October 1994.
- [18] R. Waldinger et al. *SpecwareTM Language Manual: SpecwareTM 2.0.2*, 1998.