Two-Level Grammar as an Object-Oriented Requirements Specification Language *

Barrett R. Bryant   Beum-Seuk Lee
Department of Computer and Information Sciences
The University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, U. S. A.
{bryant, leebs}@cis.uab.edu

Abstract

Two-Level Grammar (TLG) is proposed as an object-oriented requirements specification language with a natural language (NL) style but sufficiently formal to allow automatic transformation of the TLG specification into formal specifications in VDM++, an object-oriented version of the Vienna Development Method. The VDM++ specification may be further transformed into Java^{TM} code or integrated with the Unified Modeling Language (UML) using the IFAD VDM Toolbox^{TM}. The translation into an executable programming language facilitates rapid prototyping of TLG specifications and the integration with UML allows TLG specification to be used in conjunction with software systems being constructed using UML. This software specification approach is supported by a specification development environment (SDE) for constructing TLG specifications and a natural language processing system to assist in translating an NL requirements specification into TLG. The system described is a useful and constructive tool for automating the production of software systems from NL specifications.

1. Introduction

Despite a wide variety of formal specification languages [1] and modeling languages such as the Unified Modeling Language (UML) [11], natural language (NL) remains the method of choice for describing software system requirements. Informal specifications in NL must be turned into more formal designs on the way to a complete implementation. These formal requirements are necessary not only for the rapid prototyping of the evolving software systems but also to provide a standard reference model upon which all successive implementations should be constructed. Since object-oriented modeling using UML and associated tools is now a standard for software system design, there is a need for a requirements specification language which may be both conveniently used to express the original NL specification but also mapped into an object-oriented design. Since objects are already concepts in the domain of an NL vocabulary, an object-oriented design has the potential for most closely matching a requirements specification in the user's vocabulary. In fact, one technique of object-oriented analysis is to determine the objects of the problem domain using nouns in the requirements specification and determine the interactions between objects and their associated operations using verbs and their direct objects [2]. While objects may be more natural to describe in a requirements specification, some additional tools are needed to facilitate the mapping between the user's description of requirements and the actual design. Toward this end, we have developed a requirements specification language based upon Two-Level Grammar (TLG) [13] with the following advantages:

1. The NL nature of a TLG specification makes it very understandable and useful as a communication medium between users, designers, and implementors of the software system.

2. Despite an apparent NL quality, the TLG notation is sufficiently formal to allow formal specifications to be constructed using the notation.

*This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350 and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

3. TLG specifications are wide-spectrum, meaning that the specification may be very detailed for implementation as well as very general for design.

4. We have developed implementation techniques to rapidly prototype the TLG specifications, when a sufficient level of detail is specified, by means of translation into efficient executable code in object-oriented programming languages.

This paper describes the details of the TLG specification language and its implementation, including type system, object-orientation, and natural language base, and shows how TLG is mapped into VDM++.

2. Two-Level Grammar

Two-level Grammar (TLG, also called W-grammar) was originally developed as a specification language for programming language syntax and semantics, and later used as an executable specification language [4], and as the basis for conversion from requirements expressed in natural language into a formal specification [3].

2.1. Language Description

The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars defining the set of type domains and the set of function definitions operating on those domains, respectively. Note that while we use the term "domain" in a type-theoretic context, the notion can be scaled up to a much larger context as in domain of "objects." These grammars may be defined in the context of a class in which case the type domains define the instance variables of the class and the function definitions define the methods of the class.

2.1.1. Types. The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. In traditional TLG literature, these declarations are referred to as *meta-rules.* The function domains of TLG may be formally structured as linear data structures such as lists, sets, bags, or singleton data objects, or be configured as tree-structured data objects. The standard structured data types of product domain and sum domain may be treated as special cases of these.

Domain declarations have the following form:

```
Identifier-1, Identifier-2, ..., Identifier-m ::
  data-object-1; data-object-2; ...; data-object-n.
```

where each `data-object-i` is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together as a union form the type of `Identifier-1`, `Identifier-2`, ..., `Identifier-m`. If n=1, then the domain is a true singleton data object, whereas if n>1, then the domain is a set of the n objects. Syntactically, domain identifiers are capitalized, with underscores or additional capitalizations of successive words for readability (e.g., `IntegerList`, `Symbol_Table`, etc.), and singleton data objects are lists of NL words written entirely in lower case letters (e.g., `sorted list`). A list, set or bag structure is denoted by a regular expression or by following a domain identifier with the suffix `List`, `Set`, or `Bag`, respectively. Following conventional regular set notation, * implies a list of zero or more elements while + denotes a list of one or more elements. Furthermore, there exists a predefined environment of primitive types, such as `Integer`, `Boolean`, `Character`, `String`, etc. To clarify these, consider the following examples.

```
Person :: first name String middle initial Character
          last name String.
Persons :: PersonList.
People :: {first name String middle initial Character
          last name String}*.
Symbol_Table :: {id Identifier type Type value Integer}+.
```

`Person` denotes a product of `String`, `Character`, and `String` types, each tagged with an appropriate identifier to establish context. The types `Persons` and `People` are equivalent, as is the type `{Person}*`. `Symbol_Table` denotes a compiler symbol table configured as a list of records, each with three fields: `id`, `type` and `value`, with corresponding types `Identifier`, `Type`, and `Integer` (the first two of these are not standard TLG types and so should be explicitly declared). Each type name which appears on the right side of a declaration rule represents a value of that type, i. e., type names may be used as variables, making type declarations unnecessary although they enhance readability.

These examples have illustrated list structured types which essentially correspond to regular sets in formal language theory. Type checking then corresponds to simple pattern matching between regular sets. Determining the equivalence between two types is always decidable and checking the type of a value is equivalent to executing a deterministic finite automaton ($O(n)$).

The main difference between list structures and tree structured domains in terms of their declaration is whether the defining domain identifier declaration is recursive or not. Recursive domains are more powerful in that they allow "context-free" data types to be defined, such as expression strings with balanced parentheses as in the following example:

```
Expression :: ( Expression ).
```

The context-free grammars defining such data types may not be left recursive and must be unambiguous, so as to allow proper parsing. Left recursion is not needed since regular expression notation may be used in it's place. For example, instead of expressing:

```
Expression :: Expression + Term | Term.
```

we may express:

```
Expression :: Term {+ Term}*.
```

Type checking on tree structures corresponds to pattern matching over context-free grammars, i.e., parsing. Since we have imposed the restrictions of no left recursion and no ambiguity, we can guarantee that type checking a value may be done in $O(n)$ time using conventional context-free parsing techniques (e.g. LL (k) parsing). However, we can not determine the equivalence of two tree structured types as equivalence of context-free grammars is undecidable.

2.1.2. Functions. Function definitions comprise the operational part of a TLG specification. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. In traditional TLG literature, these are referred to as *hyper-rules.* Function definitions take the forms:

```
function signature.
function signature :
  FunctionCall-1, FunctionCall-2, ..., FunctionCall-n.
```

where n≥1. Function signatures are a combination of NL words and domain identifiers, corresponding to variables in a logic program. Some of these variables will typically be input variables and some will be output variables, whose values are instantiated at the conclusion of the function call. Therefore, functions usually return values through the output variables rather than directly, in which case the direct return value is considered as a Boolean `true` or `false`. `true` means that control may pass to the next function call, while `false` means the rule has failed and an alternative rule should be tried if possible. Alternative rules have the same format as that given above. If multiple function rules have the same signature, then the multiple left hand sides may be combined with a ; separator, as in:

```
function signature :
  FunctionCall-11, FunctionCall-12, ..., FunctionCall-1j;
  FunctionCall-21, FunctionCall-22, ..., FunctionCall-2k;
  ...
  FunctionCall-n1, FunctionCall-n2, ..., FunctionCall-nm.
```

where there are n alternatives, each having a varying number of function calls. Besides Boolean values, functions may return regular values, usually the result of arithmetic calculations. In this case, only the last function call in a series should return such a value.

An important aspect about TLG is that the functions may be written at a very high level of abstraction (e.g. `compute the total mass and total cost`) or embedded into a domain definition as in traditional object-oriented programs (e.g. `compute the TotalMass and TotalCost of This Part by computing the TotalMass and TotalCost of its Subparts`, which might be embedded as a method in a `Part` class). The use of NL in the function may be regarded as a form of infix notation for functions, in contrast with the customary prefix forms of most other programming languages. It is similar to multi-argument message selectors in Smalltalk but provides even greater flexibility, including the presence of logical variables, denoted by the use of domain names (capitalized). This notation provides a highly readable way of writing what is to be done and is wide-spectrum in the sense that "what is to be done" may be expressed at multiple levels. The functions typically return a Boolean value as the main operation is to instantiate the logical variables, but simple function values such as arithmetic expressions may also be computed. These function definitions form the basis for the initial design. In an implementation, they may be represented by functions in traditional object-oriented programming languages, such as Java.

A function may be defined as a *rule.* For example, we could define an expensive part using the syntax `Expensive part : part with an imported base part and cost more than $100.` or alternatively we could write in more natural form `Expensive parts are parts with an imported base part and cost more than $100.` An implementation would transform the second form into the first, and even that form into the more formal rule for `Part` objects: `expensive : BasePart imported, Cost > 100.`

To explain the operational semantics of TLG function rules, note that each function call on the right hand side of a function definition should correspond to a function signature defined within the scope of the TLG program or be a special operation such as a Boolean comparison, assignment statement, or if-then-else statement. Every domain identifier with the same name is instantiated to the same value within a function invocation. This is called *consistent substitution.* If variables have the same root name but are numbered, then the numbers are used to distinguish between variables. A numbered variable `V1` will then be different from a variable `V2` and the two can have different values. However, they will be of the same type, namely type `V`. Once a variable has been assigned a value, it may not be reassigned, unless it is an instance variable of a class, and even in this case, it would not be usual to do so in the same function. Each function definition may therefore be thought of as a set of logical rules. The function calls are executed in the order given in the function definition. Functions may be recursive with the expected operational behavior.

Besides defined functions, TLG supports the usual arithmetic and Boolean operations, as well as list comprehensions and iterators over lists. The syntax of a list comprehension is `list all Element from ElementList1 such that Element condition giving ElementList2`. This returns a list, `ElementList2`, of all `Element` values in `ElementList` satisfying the given condition. The syntax of an iterator is `select Element from ElementList with Element condition`. This returns the first `Element` from `ElementList` which satisfies the condition.

To explain the language further, consider the following examples.

Example 1. Palindrome.

```
Character is a palindrome.
Character String Character is a palindrome :
  String is a palindrome.
```

This TLG specification has no explicit type declarations since the function rules use the type names directly as variables. The two function rules are mutually exclusive, the first handling single characters and the second handling strings of two or more characters. The second rule matches if and only if the first and last characters of the string argument are the same.

Example 2. Quick Sort.

```
Pivot :: Integer.
IntegersLess, IntegersGreater, SortedIntegersLess,
  SortedIntegersGreater :: IntegerList.

quick sort Empty into Empty.

quick sort Pivot IntegerList into SortedIntegersLess
    Pivot SortedIntegersGreater :
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot,
  quick sort IntegersLess into SortedIntegersLess,
  quick sort IntegersGreater into SortedIntegersGreater.

split Empty into lists Empty and Empty using Pivot.

split Integer IntegerList into lists Integer IntegersLess
    and IntegersGreater using Pivot :
  Integer <= Pivot,
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot.

split Integer IntegerList into lists IntegersLess and
    Integer IntegersGreater using Pivot:
  Integer > Pivot,
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot.
```

The two `quick sort` rules are mutually exclusive, but the second and third `split` rules may both match nonempty lists. Each of these two `split` rules serves to distribute the `Integer` at the beginning of the list to the `IntegersLess` list or `IntegersGreater` list, depending on its relationship to `Pivot`. The first function call in each case serves as a guard to distinguish the two rules. This could have been written using an if-then-else construction, avoiding the need for the guard.

```
split Integer IntegerList into lists IntegerList1 and
    IntegerList2 using Pivot :
  split IntegerList into lists IntegersLess and
    IntegersGreater using Pivot,
  if Integer <= Pivot then begin
    IntegerList1 := Integer IntegersLess,
    IntegerList2 := IntegersGreater,
  end
  else begin
    IntegerList1 := IntegersLess,
    IntegerList2 := Integer IntegersGreater,
  end.
```

This imperative style of writing TLG's includes the begin-end grouping block and assignment statements.

The `split` rule may be eliminated completely by using list comprehensions to determine the `IntegersLess` and `IntegersGreater`, as shown below.

```
quick sort Pivot IntegerList into SortedIntegersLess
    Pivot SortedIntegersGreater :
  list all Integer from IntegerList such that
    Integer <= Pivot giving IntegersLess,
  quick sort IntegersLess into SortedIntegersLess,
  list all Integer from IntegerList such that
    Integer > Pivot giving IntegersGreater,
  quick sort IntegersGreater into SortedIntegersGreater.
```

Note that the variable `Integer` appearing in the `list all` function is not actually instantiated and so may be used in both `list all` functions without confusion.

2.1.3. Classes.

TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance. The syntax of TLG class definitions is:

```
class Identifier-1
  [extends Identifier-2, ..., Identifier-n].
  {instance variable and method declarations}
end class [Identifier-1].
```

`Identifier-1` is declared to be a class which inherits from classes `Identifier-2`, ..., `Identifier-n`. In the above syntax, square brackets are used to indicate the `extends` clause is optional so a class need not inherit from any other class. The instance variables comprising the class definition are declared using the domain declarations described earlier. In general, the scope of these domain declarations is limited to the class in which they are defined, while the methods, corresponding to TLG function definitions, have scope anywhere an object of the given class is referred to. These notions of scoping correspond to *private* and *public* access respectively in object-oriented languages such as Java, and either scope may be declared explicitly or the scope may be made *protected.* Methods are called by writing a sentence or phrase containing the object. The result of the method call is to instantiate the logical variables occurring in the method definition.

For every class, there are predefined methods beginning with `This` which serve only to select the instance variables of a class (e.g., `This InstanceVariable` returns the value of `InstanceVariable`). `This` serves as a special variable used within the method body to denote the object to which the method is being applied. Likewise, for every instance variable of simple type there are `get` and `set` methods to access or modify that variable. For every instance variable of list type, there are `add` and `remove` methods. These are assumed and do not need to be explicitly defined.

TLG class declarations serve to encapsulate the TLG domain declarations and function definitions. The class hierarchy which is resident in TLG is a small forest of built-in classes, such as integers, lists, etc. The "main" program is nothing more than a set of object declarations using the existing class identifiers as domain names and a "query" of the appropriate methods.

3. Implementation

To effectively use TLG in the requirements specification process, we have developed a Specification Development Environment (SDE) which facilitates the construction of TLG specifications from requirements documents expressed in natural language, and then translates TLG specifications into executable code. NL requirements are translated into TLG through Contextual Natural Language Processing (CNLP) [10] which constructs a knowledge representation of the requirements which may be expressed using TLG. The TLG is then translated into VDM++ [5], the object-oriented extension of the Vienna Development Method (VDM) Specification Language (VDM-SL) [9]. The IFAD VDM Toolbox [8] is then used to generate code in an object-oriented programming language such as Java. The complete system structure is shown in Figure 1.

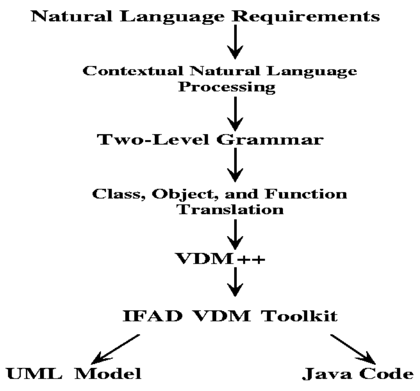
Natural Language Requirements
Contextual Natural Language Processing
Two-Level Grammar
Class, Object, and Function Translation
VDM++
IFAD VDM Toolkit
UML Model
Java Code

Figure 1. Structure of Specification Development Environment

These components are explained in the following sections in terms of an example, the Automatic Teller Machine (ATM) requirements specification below.

```
The bank keeps the list of accounts.
Each account has three integer data fields; ID, PIN, and
balance. The ATM machine has 3 service types; withdraw,
deposit, and balance check. For each service first it
verifies ID and PIN from the bank.

Withdraw service withdraws an amount from the account of
ID with PIN in the bank in the following sequence:
First it gets the balance of the account of ID from
the bank, if the amount is less than or equal to the
balance then it decreases the balance by Amount,
updates the balance of the account of ID in the bank,
and then outputs the new balance.

Deposit service deposits an amount to the account of ID
with PIN in the bank in the following sequence:
First it gets the balance of the account of ID from the
bank, it increases the balance by Amount, updates the
balance of the account of ID in the bank, and then
outputs the new balance.

Balance check service checks the balance of the account
of ID with PIN in Bank in the following order:
It gets the balance of the account of ID from the bank,
and then outputs the balance.

Transfer service withdraws an amount from the account of
ID1 with PIN in the bank and deposits the amount to the
account of ID2.
```

3.1. Processing NL Requirements Specifications

The SDE has NL parsing capabilities as well as a lexicon to aid in classification of words into nouns (objects) and verbs (operations on objects) and their relationship. Since all domain knowledge is specified by the domain definitions of the specification, the requirements written by the user can be parsed to determine the object being acted upon and the operation needed to be performed. This initial analysis of the requirements document provides the basis for further refinement according to the syntax of Two-Level Grammar function and domain definitions. The SDE analyzes each function definition and attempts to classify from the NL text which domains were involved, including the primary domain, perhaps a class, the function belongs to. A sufficient degree of interaction with the user ensures a correct interpretation. Any aspect of the specification which cannot be understood by the system can be resolved through further querying of the user. This may include the specification of additional domains and/or functions which make the specification more detailed. Once the system has "understood" the requirements that the user has specified, it can proceed with the transformation into the design and the underlying design tool can further refine this into a prototype implementation for the user to review. This process may be repeated iteratively until the requirements have been sufficiently developed to satisfy both the user and designer. By "user" we refer to either the end-user who has commissioned the system or requirements specification engineer working with the end-user. The designer can then finalize the mapping of the requirements specification into the final design. Applying this NL processing front end to the ATM requirements specification gives the following TLG.

```
class Account.
  Id, Pin, Balance, Amount :: Integer;

  withdraw Amount giving Balance1 :
    Amount <= Balance,
    Balance1 := Balance - Amount,
    set balance to Balance1.

  deposit Amount giving Balance1 :
    Balance1 := Balance + Amount,
    set balance to Balance1.
end class.

class Bank.
  Accounts :: AccountList.
  Id, Pin :: Integer.

  get account using Id giving Account :
    select Account from Accounts
      with id of Account = Id.

  get account using Id and Pin giving Account :
    select Account from Accounts with
    id of Account = Id and pin of Account = Pin.
end class.

class ATM.
  Id, Pin, Balance, Amount :: Integer.

  withdraw Amount from account of Id with Pin in Bank
      giving Balance :
    get account from Bank using Id and Pin
      giving Account,
    withdraw Amount from Account giving Balance.

  deposit Amount account of Id with Pin in Bank
      giving Balance :
    get account of Bank using Id and Pin
      giving Account,
    deposit Amount to Account giving Balance.

  check balance of Id with Pin in Bank giving Balance :
    get account of Bank using Id and Pin giving Account,
    get balance of Account giving Balance.

  transfer Amount from account of Id1 with Pin1 to
      account of Id2 in Bank :
    withdraw Amount from account of Id1
      with Pin1 in Bank giving Balance1,
    get account of Bank using Id2 giving Account2,
    deposit Amount to Account2 giving Balance.
end class.
```

It can be seen that the TLG is a structured form of the original NL specification. The exact same vocabulary is used as it is extracted by the NL processing front end. Additional information is added as needed to provide object data member access, e.g., `get` functions to access component objects.

Previous work in the area of NL specification of requirements includes a software reuse system which uses NL descriptions of library components to facilitate their selection for incorporation into an implementation [7], and "controlled natural language" [6], which is NL of a specific syntax with all vocabulary coming from a fixed domain. The latter system is able to translate the controlled NL specifications into Prolog so that they may be executed. We believe that our object-oriented approach to this problem offers a number of advantages with respect to both formal specification and object-oriented modeling.

3.2. Translation of TLG into VDM++

VDM++ has been selected as the target specification language for TLG because VDM++ has many similarities in structure to TLG and also has tool support for analysis and code generation. Although TLG and VDM++ are both formal specification languages, the translation from TLG into VDM++ is not simply a direct mapping between them. We will first give an overview of VDM++ and then explain how TLG is translated into VDM++.

3.2.1. VDM++. The structure of a VDM++ specification is organized as a collection of classes which take the following general form:

```
class identifier
```

```
  [is subclass of identifier-1, ..., identifier-n]
  value definitions
  type definitions
  instance variable definitions
  operation definitions
end identifier
```

Value and type definitions define constants and types that may be used in the class, respectively. VDM++ types include the basic data types as well as compound types in the form of sets, sequences, and maps. Instance variable definitions are the state variables of the class. Operation definitions correspond to methods. Operations have a signature and a body which may be an expression in the style of functional programming languages or a collection of imperative statements with `return` statements to return the function values. VDM++ also includes the option of defining state invariants, and pre-conditions and post-conditions for operations. Synchronization of concurrent operations and multi-threading are also provided for. At present we do not use these features in our translation schemes.

3.2.2. Translation Schemes. The translation of class definitions, including with inheritance, and compound type declarations, may be described through the tables shown in Figures 2 and 3. The translation of basic types is straightforward and so is not shown here. Type declarations in TLG specifications occur in class definitions for two purposes: 1) to define an instance variable of the class, and 2) to define variables which may be used in function definitions, either as function arguments or to calculate intermediate values. These are not difficult to distinguish as instance variables are related only to the state of the object and so must be used in function definitions other than as function arguments, typically a `get` or `set` operation. It is also straightforward to determine a variable used only for intermediate value calculation as such a variable will always be written before it is read - instance variables must have some function which reads them only.

A TLG function is translated into a VDM++ operation. TLG variables local to that function will be translated into VDM++ function local variables. Figure 4 indicates the general scheme for function definitions, which essentially consist of a function signature and a series of function calls. In these translations schemes, `Arg-1`, `Arg-2`, etc., are the arguments to the function, `Return-1`, `Return-2`, etc., are the results of the function, and `Arg-Type-i` and `Return-Type-i` are their respective types. The declaration of a result variable occurs only if the variable is not an instance variable of the class. This would not normally be the case unless the function was a `get` method associated with that instance variable. Since TLG functions may return many result values whereas VDM++ operations only return a single value, these multiple result values should be constructed into a product for the purpose of returning them as a single value. The `mk_` operation accomplishes this. `mk_` is not needed if only one return value is required. Figure 4 also shows the translation schemes for function calls. The declaration of a return variable occurs only if the variable has not been declared previously either as a return variable of the function definition in which the function call appears, or as an instance variable of the class. Since function `g` may return multiple values, the VDM++ operation returns a product of those values which may then be extracted into the individual values.

In addition to returning the values of result variables, TLG functions will either succeed or fail, as in logic programming predicates. Failure implies that no result variables are instantiated. This situation must be detected by VDM++ operations corresponding to those functions. In our generated VDM++ code, a special Boolean variable is introduced into the state of every object to indicate whether an operation performed on that object succeeded or failed. If the operation O fails, then so does the operation O' that invoked O, the operation that invoked O', etc. That is, this failure may be propagated to each previous operation until it causes the entire operation to fail or an alternative operation is possible. An alternative operation is one in which multiple rules are given for the same function signature. For function definitions defined by several rules, TLG uses pattern matching to determine which rule is appropriate. This pattern matching is implemented in VDM++ by either comparisons in cases where the pattern is a simple data type or by VDM++ pattern matching for compound data types. The examples in Figures 5 and 6 illustrate each case. Note that the `factorial` function is not defined over all integers as the TLG rules will succeed only for natural numbers. Therefore, the VDM++ operation may fail on a negative number argument, rendering the return value invalid. Functions calling `factorial` must also check for this failure. This does not include the recursive call since it can be detected that `factorial (n - 1)` will never fail since `n` $>$ `1`.

3.2.3. Example. The VDM++ translation of our running example, according to the rules given in the previous section, is shown below. As with the generated TLG, this code has been distilled for readability.

```
class Account
  instance variables
    id, pin, balance : int;

  operations
```

Simple Class

TLG	VDM++
`class C.`	`class C`
	`instance variables`
domain declarations	variable declarations
	`operations`
function definitions	operation definitions
`end class.`	`end C`

Class With Inheritance

TLG	VDM++
`class SC`	`class SC`
`extends C.`	`is subclass of C`
. . .	. . .
`end class.`	`end C`

Figure 2. Translation Schemes for Classes

TLG	VDM++	Type
`DataObj :: DataTypeSet.`	`DataObj = set of DataType`	Set
`DataObj :: DataTypeList.`	`DataObj = seq of DataType`	Sequence
`DataObj :: {DataType}*.`	`DataObj = seq of DataType`	Sequence
`DataObj :: {DataType}+.`	`DataObj = seq1 of DataType`	Sequence
`DataObj :: DataType1 DataType2.`	`DataObj = DataType1 * DataType2`	Product
`DataObj :: {DataName1 DataType1 DataName2 DataType2}.`	`DataObj = DataName1 : DataType1 DataName2 : DataType2`	Composite
`DataObj :: DataType1; DataType2.`	`DataObj = DataType1 | DataType2`	Union

Figure 3. Translation Schemes for Compound Data Types

Function Definitions

TLG
```
f of Arg-1 and ...  and Arg-n
      giving Return-1 and ...  and Return-m :
   function calls
```

VDM++
```
f :  ArgType-1 * ...  * ArgType-n ==>
   ReturnType-1 * ...  * ReturnType-m
f (arg-1, ..., arg-n) ==
   (dcl Return-1 :  ReturnType-1;
    ...
    dcl Return-m :  ReturnType-m;
    function calls
    return mk_ (Return-1, ..., Return-m)
   )
```

Function Calls

TLG
```
g of Arg-1 and ...  and Arg-n
   giving Return-1 and ...  and Return-m
```

VDM++
```
dcl Return-1 :  ReturnType-1;
...
dcl Return-m :  ReturnType-m;
dcl Returns :
    ReturnType-1 * ...  ReturnType-m;
Returns := g (Arg-1, ..., Arg-n);
Return-1 := Returns .  #1;
...
Return-m := Returns .  #m;
```

Figure 4. Translation Scheme for Functions

TLG	VDM++
`factorial of 0 :  1.`	`factorial :  int ==> int`
`factorial of Integer :`	`factorial (n) ==`
`Integer > 1,`	`if n = 0 then return 1`
`Integer * factorial of (Integer - 1).`	`elseif n > 1 then return n * factorial (n - 1)`
	`else (fail := true; return 0)`

Figure 5. Simple Data Type Pattern Matching

TLG

```
quick sort Empty into Empty.
quick sort Pivot IntegerList into SortedIntegersLess Pivot SortedIntegersGreater :
    split IntegerList into lists IntegersLess and IntegersGreater using Pivot,
    quick sort IntegersLess into SortedIntegersLess,
    quick sort IntegersGreater into SortedIntegersGreater.
```

VDM++

```
quicksort :  seq of int ==> seq of int
quicksort (pivotIntegerList) ==
    cases pivotIntegerList :
       [] -> return [];
       [pivot] ^ integerList ->
          (dcl splitReturns, integersLess, integersGreater :  seq of int;
           dcl sortedIntegersLess, sortedIntegersGreater :  seq of int;
           splitReturns := split (integerList, pivot);
           integersLess := splitReturns .  #1; integersGreater := splitReturns .  #2;
           sortedIntegersLess := quicksort (integersLess);
           sortedIntegersGreater := quicksort (integersGreater);
           return sortedIntegersLess ^ [pivot] ^ sortedIntegersGreater
          )
    end
```

Figure 6. Compound Data Type Pattern Matching

```
    ... getId, setId, getPin, setPin, etc. ...

    withdraw : int ==> int
    withdraw (amount) ==
      (dcl amount : int;
       if amount <= balance then
         (dcl balance1 : int;
          balance1 := balance - amount;
          setBalance (balance1)
         );
       return balance
      );

    deposit : int ==> int
    deposit (amount) ==
      (dcl amount, balance1 : int;
       balance1 := balance + amount;
       setBalance (balance1);
       return balance
      );
end Account

class Bank
  instance variables
    accounts : seq of Account;

  operations
    .. addAccount and removeAccount ...

    getAccountById : int ==> Account
    getAccountById (id) == ...

    getAccountByIdPin : int * int ==> Account
    getAccountByIdPin (id, pin) == ...
end Bank
```

```
class ATM
  instance variables
    bank : Bank;

  operations
    ... getBank and setBank ...

    withdraw : int * int * int ==> int
    withdraw (amount, id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . withdraw (amount);
       return balance
      );

    deposit : int * int * int ==> int
    deposit (amount, id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . deposit (amount);
       return balance
      );

    checkBalance : int * int ==> int
    checkBalance (id, pin) ==
      (dcl account : Account;
       dcl balance : int;
       account := bank . getAccountByIdPin (id, pin);
       balance := account . getBalance ();
       return balance
      );
    transfer : int * int * int * int ==> ()
    transfer (amount, id1, pin1, id2) ==
      (dcl account2 : Account;
       dcl balance, balance1 : int;
       balance1 := withdraw (amount, id1, pin1);
       account2 := bank . getAccountById (id2);
       balance := account2 . deposit (amount)
       return;
      );
end ATM
```

4. Summary and Conclusions

Two-Level Grammar has been presented as an object-oriented requirements specification language which is natural language-like in style but sufficiently formal to allow automatic transformation of the TLG specification into a VDM++ object-oriented formal specification. The IFAD VDM Toolbox provides for an integration of VDM++ with the Unified Modeling Language (UML) [11] through a link between the Rational Rose 2000^{TM} [12] implementation of UML and VDM++. This tool translates between UML and VDM++ and so supports round-trip engineering which may be iterative. Presently we use this in a single direction, from TLG to VDM++ to UML. This effectively allows for UML modeling of the TLG specification and so is useful for integration with existing UML models. Rational Rose does provide an "Add-In" mechanism with which we hope to have a direct integration with TLG in the future. The translation into an executable programming language using the IFAD VDM++ to Java code generator facilitates rapid prototyping of TLG specifications. Our approach to software specification is supported by a specification development environment (SDE) for constructing TLG specifications and a natural language processing system to assist in translating an NL requirements specification into the TLG. The system is a useful and constructive tool for automating the production of software systems from NL specifications.

At present the SDE exists only in prototype form but is able to handle simple NL specifications, as our example illustrated. We are extending this system so that more complex NL specifications may be handled. We would also like to automate the interaction between our SDE and tools like Rational Rose directly, in addition to going through VDM++. This will give us a complete visual modeling tool not only for object-oriented design but also for specification as well.

Acknowledgements. The authors would like to thank IFAD for providing an academic license to the IFAD VDM Toolbox in order to conduct this research.

References

[1] V. S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer-Verlag, 1998.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[3] B. R. Bryant. Object-Oriented Natural Language Requirements Specification. *Proc. ACSC 2000, 23rd Australasian Computer Science Conf.*, pages 24–30, 2000.

[4] B. R. Bryant and A. Pan. Formal Specification of Software Systems Using Two-Level Grammar. *Proc. COMPSAC '91, 15th Ann. Intl. Computer Software and Applications Conf.*, pages 155–160, 1991.

[5] E. H. Dürr and J. van Katwijk. VDM++ - A Formal Specification Language for Object-Oriented Designs. *Proc. TOOLS USA '92, 1992 Technology of Object-Oriented Languages and Systems USA Conf.*, pages 63–278, 1992.

[6] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *Proc. CLAW '96, First Intl. Workshop Controlled Language Applications*, 1996.

[7] M. Girardi and B. Ibrahim. A Software Reuse System Based on Natural Language Specifications. *Proc. ICCI '93, 5th Intl. Conf. Computing and Information*, pages 507–511, 1993.

[8] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD (`http://www.ifad.dk`), 2000.

[9] P. G. Larsen, et al. Vienna Development Method - Specification Language - Part I: Base Language. Report, International Standard ISO/IEC 13817-1, December 1996.

[10] J. McCarthy. Notes on Formalizing Context. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1993.

[11] Object Management Group. OMG Unified Modeling Language Specification, Version 1.3. Technical report, Object Management Group, June 1999.

[12] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML.* Addison-Wesley, 2000.

[13] A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.