

Object-Oriented Natural Language Requirements Specification

Barrett R. Bryant
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Blvd.
Birmingham, Alabama 35294-1170, U. S. A.
bryant@cis.uab.edu

Abstract

A methodology is proposed for the formal development of software systems from a user's requirements specification in natural language into a complete implementation, proceeding through the steps of formal specification, and detailed design and implementation in an automated manner. Our methodology is based upon the theories of Two-Level Grammar (TLG) and object-oriented design and takes advantage of the existence of several existing software design tools. We develop an iterative transformation process from the natural language specification into the final implementation, with a specification development environment to assist the user, as well as the designer in accomplishing this transformation. Our underlying formal specification methodology may also be used in the final development of the implementation. This methodology is a major advance in object-oriented software development and the software engineering process in general.

1. Introduction

Recent advances in software technology such as the development of the Unified Modeling Language (UML) [10] for object-oriented design have not reduced the need for better requirements specification. Natural language remains the method of choice for producing such documents. It is recognized that these informal specifications 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.

Software development tools which support UML, such as Rational Rose as described by [15], provide a means of incorporating the natural language specifications into the

design as a comment or annotation but there is no direct connection between these comments and the design being produced. We have developed a methodology for requirements specification which: 1) integrates the natural language requirements specification with the rest of the software by establishing a natural language-like notation for writing the requirements specification, 2) provides technology to automate the process of moving from requirements to design and implementation, and 3) establishes the requirements specification as a formal yet understandable document that is the cornerstone upon which the software system will be based. Our notation is based upon Two-Level Grammar (TLG) [17] for the following reasons:

1. The natural language-like 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 natural language-like quality, the TLG notation is sufficiently formal to allow formal specifications to be constructed using the notation.
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 interpretation techniques to rapidly prototype the TLG specifications, when a sufficient level of detail is specified.
5. TLG specifications can also be translated into efficient executable code in procedural programming languages.

In this paper, we extend these results to show how the TLG notation may be used as a formal notation for requirements specification which may be automatically translated into an object-oriented design and an object-oriented implementation, thus integrating the type of informal natural language

requirements specifications now in use into the formal specification, design and implementation cycle. This integration is accomplished using a Specification Development Environment (SDE) which facilitates the construction of natural language specifications in TLG and their corresponding transformation into an object-oriented design.

This paper is organized as follows. In Section 2, we briefly review software requirements specification and in section 3, our Two-Level Grammar requirements specification language is introduced. Section 4 explains the Specification Development Environment and this is followed by an example of our approach in Section 5. Finally we conclude in Section 6.

2. Software Requirements Specification

Users typically state their requirements to the software system designers using some form of informal document written in natural language. The primary goal of the requirements document is to be a reference for the software designers, facilitating improved software design through detection of incompleteness, inconsistency and ambiguity. The designer needs to be able to express the user requirements correctly, as he/she understands them. This is one of the major places where discrepancies are introduced into the system. Natural language tends to be ambiguous and not always recognizably so. This ambiguity may cause a designer to misinterpret some requirements which may not be detected until very much later in the system development stage. To avoid this problem, rapid prototyping of requirements specifications allows the user to evaluate the system in experimental form before a massive implementation effort has been undertaken.

With the adoption of the Unified Modeling Language as a standard for object-oriented modeling, and the development of associated design tools, such as Rational Rose, there is a need for a requirements specification language which may be conveniently mapped into an object-oriented design. Since objects are already concepts in the domain of a natural language vocabulary, it can be argued that 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 [1]. While we believe that objects are more natural to describe in a requirements specification, we believe that some additional tools are needed to facilitate the mapping between the user's description of requirements and the actual design.

Previous work in the area of natural language specification of requirements includes a software reuse system which uses natural language descriptions of library components to

facilitate their selection for incorporation into an implementation [8], and Attempto ([6], [7]) which introduces "controlled natural language," which is natural language of a specific syntax with all vocabulary coming from a fixed domain. The Attempto system is able to translate the controlled natural language specifications into Prolog so that they may be executed.

There has also been much work on using formal specifications, as opposed to informal specifications, as the basis for developing software systems [12]. While formal specifications have many advantages in constructing the software system, the main disadvantages are that it is very difficult to reconcile a formal specification with the user's requirements and formal specification notations are not easily comprehensible to the non-computer scientist. Swatman and Swatman [16] have developed a model for making formal methods more accessible to the users of the system being specified. Our approach is to apply techniques from formal methods to the development of a natural language-like specification methodology using Two-Level Grammar. Since natural language is the basis of the specification, necessary constraints are imposed to achieve a sufficient degree of formality and the advantages of using a formal specification. These constraints are formalized by Two-Level Grammar syntax and semantics.

3. Two-Level Grammar Requirements Specification

Two-level Grammar (TLG, also called W-grammar) was originally developed as a specification language for programming language syntax and semantics and was used to completely specify ALGOL 68 [18]. We showed that TLG could be used as an executable specification language [2], not only for programming language design and implementation, but also for more general software systems such as natural language interfaces and database and knowledge-base systems [3]. In this section we briefly describe the language details and elaborate on how the language may be used in the type of formal specification we desire.

The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing machine. Our previous work refined this notion into a set of domain definitions and the set of function definitions operating on those domains. 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." The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. On the other hand, function definitions may be given without precisely defined domains for a more

flexible specification approach. This framework is the basis for the specification development environment which allows the user to iteratively define domains and the operations on those domains starting with a text document containing references.

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. Function definitions take the forms:

function-predicate.

function-predicate :

sub-function-1, sub-function-2, ..., sub-function-n.

where $n \geq 1$. Function predicates are a combination of natural language words and domain identifiers, which correspond to variables in a conventional logic program. The 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 natural language in the function predicate 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. 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 C++ and Java.

As seen from the above syntax, a function may be defined as a *rule* as in Prolog. 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.** The SDE 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 Two-Level Grammar rules, note that each sub-function on the right hand side of a function definition should correspond to a function predicate defined within the scope of the TLG program. The most important aspect of function definitions is that every domain identifier with the same name is instantiated to the same value, as in Prolog. 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**. Each function definition may therefore be thought of as a set of logical rules.

At the most abstract level of specification, domains need not be specified at all. Since a Two-Level Grammar does not have any specific domain knowledge, such an abstract specification will ultimately have to be refined to include this information. 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 form the type of **Identifier-1, Identifier-2, ..., Identifier-m**. Note that 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 finite lists of English 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. For example, the following type declarations respectively define a list of persons and a compiler symbol table configured as a list of records, each with three fields: id, type and value.

People ::

{first name String middle initial Character last name String}*.

Symbol_Table :: {id Identifier type Type value Integer}+.

Following conventional regular set notation, * implies a list of zero or more elements while + denotes a list of one or more elements. As in any programming language, readability is promoted through the use of appropriate names for identifiers. Furthermore, there exists a predefined environment of primitive types, defining such domains as **Integer, Boolean, Character, String**, etc., in the obvious ways. The main difference between list structures and tree structured domains is whether the defining domain identifier declaration is recursive or not. Recursive domains are slightly 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 must be unambiguous.

The notations for domain declarations allow us to define the principal domains of type theory, namely product domains (tuples), sum domains (discriminated unions), and function domains. Our types are modeled after the ML functional programming language [13]. Because of the potential complexity of domain declarations, the details of these are typically internal to the formal specification being developed and the specification development environment queries the user to get these details as needed.

In order to support object-orientation, 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-3, ... Identifier-n].
  {instance variable and method declarations}
end class [Identifier-1].
```

In the above syntax, square brackets are used to indicate the construct is optional. **Identifier-1** is declared to be a class which inherits from classes **Identifier-2**, **Identifier-3**, ..., **Identifier-n**. Note that 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 C++ and 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.

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. As with the other portions of a TLG specification, the details about class definitions are more internal to the SDE but are constructed from the user’s original natural language specification.

4. The Specification Development Environment (SDE)

While the underlying TLG formal specification may be transformed into an executable prototype, an alternative approach is to glean enough information from the specification to map it into an equivalent design. The original Two-

Level Grammar model may be thought of as a logic programming model where the name of the predicate is written in infix notation and distributed among the variables to read like a sentence in natural language. Each function was required to be completely specified. For use as a requirements specification language, we have relaxed the requirement that functions be completely specified. The SDE has natural language 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 natural language 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.

Our Two-Level Grammar Specification is designed to interact with an object-oriented design tool such as Rational Rose. The process will go through several iteration stages as follows:

1. Analyze the TLG requirements document and identify the domains and operations, displaying these for the user as well as producing an internal organization in more formal TLG class definitions, querying the user if necessary to resolve some ambiguity. The steps in this refinement process will be recorded as a documentation of the user’s requirements specification.
2. The above process is iterative. When the user is confident that he has stated the requirements in a manner understandable to the system, there will be an underlying TLG formal specification, complete in terms of

class structure corresponding to what the user has described in his requirements document. This TLG formal specification will then be translated into the vocabulary of the underlying software design tool. In the case of a UML-based tool, this means that the classes of the TLG will be mapped into UML classes with all their associated properties. Because UML is for design and TLG for specification, it will be expected that the UML design will require more details to be supplied by the designer rather than the user. Usually this is done using a design development environment rather than manually. For consistency with our specification system, these design refinements will be translated back into the formal TLG. At the present time, we are not able to automate this process without support from the underlying tool but the principles for doing so have been fully developed. Alternatively, these details can be directly added into the underlying TLG and then mapped back to the design. In the future, we hope the SDE can be integrated with the design tool to facilitate this process.

3. Once the necessary design has been completed, the software design tool itself is used to produce the code for the system from the design. If additional code is needed to complete the implementation, this may also be added to the underlying TLG formal specification and then automatically transformed into code using the techniques we described in [2], making adjustments for the object-orientation as needed. We may use an object-oriented programming language such as C++ or Java as a target language. Presently we have not added knowledge of the various libraries available with these languages.
4. Finally the prototype has been completed and the user can review the results of the implementation. Based on his or her analyses and others, such as use-case modeling ([4], [11]) or task analysis [9], the original requirements document can be adjusted as needed with the underlying TLG formal specification and the above process repeated. As with the earlier changes in requirements, detailed information about this round of changes will also be recorded.

It can be seen that the overall process is similar to the notion of program transformation [14] except that the original specification will start in a natural language like notation and gradually be transformed into a formal specification, formal design, and executable implementation of the system.

5. Example

As an example of the Two-Level Grammar specification methodology and the Specification Development Environment, we use the library support system described in [5]. This system has requirements:

- The library lends books and magazines to borrowers.
- All books, magazines, and borrowers are registered.
- New titles may be purchased, sometimes as multiple copies. Old titles may be removed.
- The librarian interacts with the borrowers.
- Borrowers can reserve books/magazines not currently available. This reservation system will notify the borrower upon availability and the reservation is subject to cancellation.
- For all entities of the system, information may be created, updated and deleted.

From this set of sentences, we produce the following Two-Level Grammar.

```

class Library.
    lend BookMagazine to Borrower.
end class Library.
class Entity.
    create.
    update.
    delete.
end class Entity.
class BookMagazine extends Entity.
    ThisAvailable :: Boolean.
    isAvailable : ThisAvailable.
    setAvailable Boolean : ThisAvailable < – Boolean.
    register.
end class BookMagazine.
class Book extends BookMagazine.
end class Book.
class Magazine extends BookMagazine.
end class Magazine.
class Borrower extends Entity.
    register.
    reserve BookMagazine.
end class Borrower.
class Title extends Entity.
    ThisCopyNumber :: Integer.
    getCopyNumber Integer : ThisCopyNumber = Integer.
    setCopyNumber Integer : ThisCopyNumber < – Integer.
end class Title.
class NewTitle extends Title.
    bePurchased.
end class NewTitle.

```

```

class OldTitle extends Title.
    beRemoved.
end class OldTitle.
class Librarian extends Entity.
    interact with Borrower.
end class Librarian.
class ReservationSystem extends Entity.
    notify Borrower of BookMagazine availability :
        BookMagazine isAvailable.
end class ReservationSystem.
class Reservation extends Entity.
    cancel.
end class Reservation.

```

All of the sentences except for the last are straightforwardly handled by the SDE natural language text analyzer. Since the last sentence of the requirements refers to two vague nouns, “entities” and “information,” and an unclear reference “the system,” the user is queried as to what these mean. This is done by requesting clarification of “the system,” at which time the user indicates that it is this software system. Since every class is a potential entity, the SDE displays the classes created from the previous sentences and asks the user to identify those which are “entities.” The above TLG results from a selection of all classes except **Library**. The user declines to clarify the concept of “information,” so that is not included. The queries to the user are in the form of a graphical user interface representing a menu of possible choices which are appropriate for the current context.

It can be seen that the TLG resulting from this first analysis of the informal requirements is an outline of a possible system but much detail is missing. The next step is to display all of the classes created in turn to allow the user to enter more information. The user can add this information or defer it until design. The result of the SDE will be input to an object-oriented design tool from which the design details can be further refined. Ideally, the further refinements made in the design can be translated back into TLG for inclusion into the specification, or possible modification of the specification if an inconsistency is detected at that point. We intend to support evolutionary software so it is possible that some aspects of the design may require modification of the specification. Neither of these aspects is currently supported in our prototype system.

Let’s assume that the SDE user would like to formalize a use-case for lending an item. This case is given in [5] as:

1. If the borrower has a reservation: a) identify borrower, b) identify title, c) identify available item, d) library lends item, e) register new loan, f) remove reservation.
2. If the borrower has no reservation: a) identify title, b) identify available item, c) identify borrower, d) library lends item, e) register new loan.

This would initially be translated into the following rules associated with the function **lend BookMagazine to Borrower**

in **Library**, as specified by the user.

```

lend BookMagazine to Borrower : Borrower hasReservation,
    identify Borrower, identify Title, identify AvailableItem,
    lend Item, register new Loan, remove Reservation.
lend BookMagazine to Borrower : Borrower not hasReservation,
    identify Title, identify AvailableItem, identify Borrower,
    lend Item, register new Loan.

```

The SDE will also create three new classes, **Item**, **AvailableItem**, a subclass of **Item**, and **Loan**, methods in **Library** to identify borrowers, title, available items, register a new loan, and remove a reservation, and a Boolean method **hasReservation** for **Borrower**. The user will be asked whether there is a relationship between **Item** and **BookMagazine** since both have an “available” attribute. The user will also be asked about the context of the **Title**, **AvailableItem**, **Item**, **Loan**, and **Reservation** objects on the right side of the constructed rule since these appear as neither arguments to the rule or member variables of **Library**. The user will be given a choice of choosing the context from these two options, in which case the object will be added to the appropriate place, or choosing to get the context from another object. In the case of **Title**, the user might indicate that this is an attribute of **BookMagazine**, in which case that attribute would be added along with an appropriate **getTitle** method to the **BookMagazine** class. In the case of **AvailableItem**, the user might indicate that this is an **Item** with an “availability” attribute instead of the subclass, such that the attribute is true in this case. The user might also indicate that the **Item** class is “owned by” **Title** in the sense that a title may have a set of items and a method for getting an available item. The user might also clarify that the process of lending an item belongs to the **Item** class, i.e. to make the item unavailable. For registering a new loan, the user may indicate that the **Library** has a set of loans and this operation creates one such loan and adds it to the set. Finally, in the event that the borrower had a reservation, the reservation must be removed. The user will need to indicate that the **Library** maintains a set of reservations and this one needs to be removed. Based on this set of enhancements, we now have the following TLG specification for the **Library** class.

```

class Library.
    ThisLoanSet :: LoanSet.
    ThisReservations :: ReservationSet.
    get and set methods for attributes
    lend BookMagazine to Borrower :
        Borrower hasReservation,
        identify Borrower,
        BookMagazine getTitle BookMagazineTitle,
        BookMagazineTitle getAvailableItem Item,
        Item lend,
        register new Loan of Item to Borrower,
        ThisReservationSet remove Reservation.
    lend BookMagazine to Borrower :
        Borrower not hasReservation, ...

```

```
register new Loan of Item to Borrower :
  ThisLoanSet add new Loan of Item to Borrower.
end class Library.
```

To explain some of the added rules, **BookMagazine getTitle BookMagazineTitle** invokes the **getTitle** method of the **BookMagazine** object and the title is returned in the variable **BookMagazineTitle**, which is a **Title** object. We then invoke the method **getAvailableItem** on this object to instantiate the variable **Item**. The above TLG may be further refined by the user indicating the parameters to the rule are not actually **BookMagazine** and **Borrower** objects, respectively, but some identifications of these, e.g. their names. We can see that the system gives the user a great deal of flexibility in deciding what should be part of the specification and what may be delayed until design. If the specification is sufficiently refined, its implementation may be automatically prototyped by implementing the underlying TLG.

6. Summary and Conclusions

In this paper we have defined a method for object-oriented natural language requirements specification using Two-Level Grammar. The TLG notation developed for specification in our previous work is extended to encompass object-orientation and the natural language style is relaxed by adding a Specification Development Environment which allows user interaction to refine very generally defined concepts. The user's refinement may be either rapidly prototyped using existing TLG implementation methods or may be translated into an object-oriented design suitable for use in an appropriate automated design tool.

At present the SDE exists only in prototype form but is able to handle simple natural language specifications, as our example illustrated. In future work, we plan to extend this system so that more complex natural language specifications may be handled. We would also like to automate the interaction between our SDE and existing design tools like Rational Rose. This will give us a complete visual modeling tool not only for object-oriented design but also for specification as well.

Acknowledgements. The author would like to thank the Army Research Laboratory, Software Technology Branch, in Atlanta, Georgia, U. S. A., for providing initial inspiration for this work. The author is also grateful to the referees for their comments in improving the paper.

References

- [1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.
- [2] B. R. Bryant and A. Pan. Formal specification of software systems using Two-Level Grammar. *Proc. COMPSAC '91, Fifteenth Annual Intl. Computer Software and Applications Conf.*, pages 155–160, 1991.
- [3] B. R. Bryant and A. Pan. Two-Level Grammar: A functional/logic query language for database and knowledge-base systems. *Proc. LPAR '92, 1992 Intl. Conf. Logic Programming and Automated Reasoning*, pages 78–83, 1992.
- [4] A. Cockburn. Structuring use cases with goals. *J. Object-Oriented Programming*, 1997.
- [5] H. Eriksson and M. Penker. *UML Distilled: Applying the Standard Object Modeling Language*. John Wiley, 1998.
- [6] N. E. Fuchs and R. Schwitter. Attempto: Controlled natural language for requirements specifications. *Proc. Seventh Intl. Logic Programming Symp. Workshop Logic Programming Environments*, 1995.
- [7] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). *Proc. CLAW '96, First Intl. Workshop Controlled Language Applications*, 1996.
- [8] 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.
- [9] I. Graham. *Requirements Engineering and Rapid Development*. Addison-Wesley, 1998.
- [10] Object Management Group. OMG Unified Modeling Language specification, version 1.3. Technical report, Object Management Group, June 1999.
- [11] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [12] Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January 1997.
- [13] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [14] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag, 1990.
- [15] T. Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.
- [16] P. A. Swatman and P. M. C. Swatman. Formal specification: An analytic tool for (management) information systems. *J. Information Systems*, 2(2):121–160, 1992.
- [17] A. van Wijngaarden. Orthogonal design and description of a formal language. Technical report, Mathematisch Centrum, Amsterdam, 1965.
- [18] A. van Wijngaarden. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5:1–236, 1974.