Formal Specification of Non-Functional Aspects in Two-Level Grammar [*]

Chunmin Yang Beum-Seuk Lee Barrett R. Bryant Carol C. Burt

Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170, U. S. A.
{yangc, leebs, bryant, cburt}@cis.uab.edu

Rajeev R. Raje Andrew M. Olson

Department of Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, U. S. A.
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, U. S. A.
mikau@cs.nmsu.edu

Abstract

In the UniFrame project, non-functional aspects of distributed software systems are described informally in natural language based on a quality of service (QoS) parameter catalog. Then the descriptions are automatically translated into specifications in a formal specification language, Two-Level Grammar (TLG). The result is a formal QoS specification for rapid prototyping of non-functional aspects of a system as well as their efficient distribution.

Keywords: Formal Specification, Non-functional properties, Quality of Service, Two-Level Grammar (TLG), UniFrame, Vienna Development Method (VDM)

1 Introduction

With the rapid development and increased demand for software systems implemented on computer networks, distributed computing has become the focus of research interest. Even though many techniques have been developed for this purpose most of them focus mainly on the functional aspects of the system neglecting the non-functional aspects. It has been more and more realized that non-functional properties are as important as the functional ones for a successful software product.

The non-functional aspects of software systems are not so much emphasized as the functional aspects due to several reasons:

[*]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.

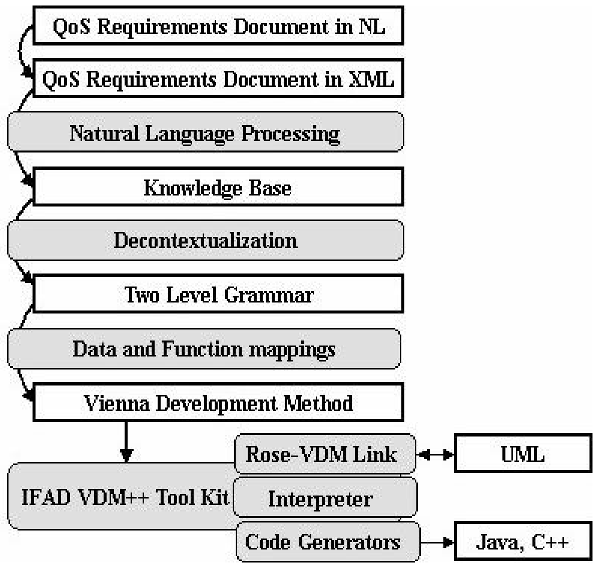
QoS Requirements Document in NL
QoS Requirements Document in XML
Natural Language Processing
Knowledge Base
Decontextualization
Two Level Grammar
Data and Function mappings
Vienna Development Method
Rose-VDM Link
UML
IFAD VDM++ Tool Kit
Interpreter
Code Generators
Java, C++

Figure 1: System Structure.

1. The developers are more concerned with the functionality of the software product than its quality. Their main goal is first to make sure the software is able to provide the functionality as specified by the users. With the move toward component-oriented development, functionality by itself is not enough to meet the users' expectations. To develop a software with high quality, both the functional and the non-functional aspects of the software have to be considered with care.

2. Unlike functional aspects of the specifications, non-functional aspects of the specifications are usually described in an abstract and non-quantified way, thus making it more difficult to describe formally.

3. Non-functional aspects of the specification are complex. Some of the non-functional properties may interact with other non-functional properties. Therefore the effect of non-functional properties of the system does not remain the same all the time, but rather change dynamically according to other non-functional properties.

4. Unlike functional properties, it is difficult to formally specify non-functional properties, although there have been several research projects with this goal, e.g. Aster [1], Qedo [13], QuO [12], to name a few.

Our goal is to enable non-functional requirements to be described informally in natural language and then automatically translated into a formal specification for use in validating component-based software system quality. In our project, first Quality of Service (QoS) requirements in natural language (NL) are represented using eXtensible Markup Language (XML) [3] element and attribute notations which specify the types of non-functional properties and attributes (meta information). This XML specification is translated into a Knowledge Base using Natural Language Processing. Knowledge Base contains the linguistic information as well as meta information of the QoS description. This Knowledge Base is then converted into Two-Level Grammar (TLG) [4] using the collected information in the Knowledge Base. TLG is a formal specification language that is flexible in its natural-language like syntax without losing its formalism. The non-functional specifications in TLG in turn can be translated into VDM++ [7] (an object-oriented extension of the Vienna Development Method [2]) using data and function mappings. VDM++ is chosen as the target specification language because VDM++ has many similarities in structure to TLG and also has a good collection of tools for analysis and code generation. Once the VDM++ representation of the specification is acquired we can do prototyping of the specification using the VDM++ interpreter. Also we can convert this into a high level language such as Java$^{\mathrm{TM}}$ or C++ or into a model in the Unified Modeling Language (UML) [14] using the VDM++ Toolkit [8]. The entire system structure is shown in Figure 1. In this paper, we mainly focus on the mechanism to formally specify non-functional aspects of a system in TLG followed by brief illustration of the conversion process from the QoS descriptions in NL into TLG.

2 Quality of Service (QoS)

Quality of Service is a concept originated in the networking area and now it has been extended to software development, in which it is also referred to as "non-functional properties."

To describe the properties of a software product, we need to consider both the functional and non-functional aspects. The former is very straightforward and describes what the software is expected to do. The latter describes how the functions are exhibited. Functional aspects, in practice, earn more attention than the non-functional aspects. From the users' point of view, whether the software can provide the functions as expected is the main issue. More over it is usually easy to prototype or to verify the functionality of the system. On the other hand, it is not so easy to measure the non-functional properties. Functional properties typically have localized effects in the sense that they affect only the part of software functionality whereas non-functional properties specify global constraints that must be satisfied by the software such as performance, fault-tolerance, availability and security.

Along with the development of software engineering techniques, the non-functional properties of a software product become more and more important criteria in classifying a good software product from a poor one since most of the software would successfully provide the required functionality. Therefore the product with non-functional properties will dominate the ones without them. At the same time, there is an increasing demand for fault-tolerance, multimedia, real-time, and other high quality applications, thus the requirement for non-functional properties will become an essential part of software development.

To describe and analyze the non-functional properties, we divide them into three aspects: non-functional attributes, non-functional actions, and non-functional properties. Non-functional attributes are the features or characteristics to be described. A significant characteristic of a non-functional attribute is its decomposability, i.e., a non-functional attribute could be decomposed into multiple more detailed non-functional attributes. Non-functional actions are the input from the outside world which has effect on the attributes. Non-functional properties are the constraints of non-functional actions over the non-functional properties.

This work, to formally specify the quality of components and component complexes (results of compositions of components), is a part of the UniFrame project [16] in which the aspects of a meta-model will be specified and verified in the context of combining heterogeneous components, and provides a QoS management to the interactions between clients and services for distributed object systems by supporting frameworks for multiple QoS categories.

In the project, three steps are taken to assure the QoS of a Distributed Computing System (DCS): first, creation of a catalog for the QoS parameters, then provision of a formal specification of these parameters, and construction of a mechanism for ensuring these parameters, both at the individual component level and at the entire system level.

A catalog of Quality of Service parameters is proposed in [15] which contains the parameters such as throughput, capacity, end-to-end delay, parallelism constraints, availability, ordering constraints, error rate, security, transmission, adaptivity, evolvability, reliability, stability, result, achievability, priority, compatibility, and presentation. The format of this catalog is based on the format of the design patterns catalog. Each parameter is described according to the following features: name, intent, description, influencing factors, measure, known usages, aliases, related parameters, consequences, levels, technologies, applications, exceptions, and example scenario.

There are some reasons that non-functional properties are not explicitly described. First of all, non-functional parameters are more difficult than functional parameters to be dealt with in the sense that they are far more abstract and more complex than the functional parameters. For example, the requirement description may have a phrase like "the system should have very high level of security". But what level of security is considered to be "high?" How can we verify if this system meets this requirement? Obviously, this ambiguous and very inexact description is not descriptive enough to be used as the specification on which the software is developed. In addition, non-functional aspects of the software specification are rarely supported by computer languages, methodologies, or tools [12]. They are usually specified in an informal way and in most cases, they are not quantified thus are more difficult to manipulate. Moreover, it is especially hard to formulate the non-functional aspects of software at early stages of software development. It is not easy to prototype if the system meets the non-functional requirements until the software development phase, thus it is even harder to validate the non-functional properties of a software product. Lastly, the non-functional attributes may conflict or interact with each other. This is called correlation among attributes. When a non-functional action is performed on a system adjusting one non-functional attribute, it may have effects on other non-functional attributes as well. Even though the effect may be unexpected it has to be foreseen and controlled by the software developers.

Although QoS and its guarantees have been widely used in networking, not many attempts have been made to incorporate QoS into component-based software systems [6]. As described above, the informal and ambiguous natural language is not enough for this purpose, and on the other hand, by nature of specification, a programming language is not appropriate either as it has too much detail involved. Formal specification can overcome the problem of natural language being too ambiguous and programming language being too detailed, also formal specification languages have a friendly interface with component based software development techniques, thus our goal is to describe the non-functional properties with such a formal language so as to standardize the software development of systems meeting QoS properties.

3 Specification of QoS in TLG

In UniFrame, Two-Level Grammar (TLG) is used to specify the non-functional properties. TLG is a formal specification language, originally developed as a specification language for programming language syntax and semantics, and later used as an executable specification language and as the basis for conversion from requirements expressed in natural language into formal specifications [4]. It is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The combination of natural language and formalization is unique to TLG and also fits the Unified Meta-component Model (UMM) for component description [16] used in UniFrame well.

The name "two-level" in TLG comes from the fact that TLG consists of two Context Free Languages defining the set of type domains and the set of function definitions operating on those domains, respectively. These grammars may be defined in the context of a class in which case type domains define instance variables of the class and function definitions define methods of the class, and they interact with each other to achieve the power of a Turing Machine.

The syntax of TLG class declarations is:

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

From this definition, we can see that TLG supports multiple inheritance. The instance variables (also called as meta-rules) comprising the class definition are declared using domain declarations of the following form:

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

The function signature (referred to as a hyper-rule as well) is defined as follows.

```
function signature : function-call-1, ..., function-call-n.
```

where $n \geq 1$. 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.

TLG is a suitable specification language to represent non-functional properties for the following reasons. First of all, TLG has a class hierarchy which corresponds to the way we describe non-functional properties. This class consists of instance variables and functions, just like the non-functional attributes and non-functional actions encapsulated together. Thus meta-rules of TLG can be used to represent the non-functional attributes while hyper-rules of TLG can be used to represent the non-functional actions.

The classes in TLG may inherit from other classes and this hierarchical structure may be used to represent the decomposability of the non-functional properties as mentioned above so as to take advantage of software reuse, an important idea in component-based software development. Furthermore, TLG is natural language like, and thus it is easier to translate from natural language specification to TLG than to other formal specification languages. TLG is also appropriate for the basis of converting from requirement specifications into other formal specification languages.

Lastly the specification with TLG has a high level of abstraction and its representation is flexible - not all the members (variables or functions) have to be quantifiable. For example, to represent the effect of non-functional actions over the non-functional attributes, especially in the case of correlation, we do not have to quantify all the attributes or properties. In most cases, we only need to know if an action has effect on an attribute or not, and how it affects the attribute if it does have effect. So we only need some variables to describe the relationship above: "no effect," or effects in favor of, or against, respectively. These are just variables, and do not indicate how much the action affects the attribute.

A simple ATM (Automated Teller Machine) example is used to illustrate our approach of using Two-Level Grammar to represent non-functional properties. Here is a brief description of the non-functional requirements of ATM:

```
ATM's security property is as follows. The length of the encryption byte should be bigger than 3 and
the allowed attempts has to be smaller than the maximum allowed attempts. If the encryption byte
length is 6 and the maximum allowed attempts is less than 5 then the system is 80% secure. If the account type is
a savings account or the maximum allowed connections of the bank is less than 50 or the delay level is less
than 50 then the maximum allowed attempts is limited to 4.
If the user timeout is between 10000 and 120000 milliseconds we have a good delay level. If the response
time is longer than 30000 milliseconds, the delay level drops down to 40%.
```

To implement the above requirements specification, four classes are declared : `Property`, `Bank_Capacity`, `ATM_Security`, and `ATM_Delay`. In this simple example, only several non-functional properties are indicated. For each class, there are non-functional attribute definitions, and non-functional action declarations, especially the correlated attributes are defined. In general, not all the non-functional attributes need to be defined exhaustively.

```
class Property.
  Level :: int.
end class.

class Bank_Capacity extends Property.
  Maximum_Connections :: Integer.
end class.

class ATM_Security extends Property.

  Maximum_Allowed_Attempts :: Integer.
  Encryption_Byte_Length :: Integer.
  Allowed_Attempts :: Integer.
  Account_Type :: String.

  check satisfaction :
    Encryption_Byte_Length > 3, Allowed_Attempts < Maximum_Allowed_Attempts.

  update level :
    Encryption_Byte_Length = 6,
    Allowed_Attempts < 5,
    Level := 80.
  update attributes :
    Account_Type = "savings", Maximum_Allowed_Attempts := 4;
    Bank_Capacity Maximum_Connections < 50, Maximum_Allowed_Attempts := 4;
    ATM_Delay Level < 50, Maximum_Allowed_Attempts := 4.

end class.

class ATM_Delay extends Property.

  Response_Time :: Integer.
  User_Timeout :: Integer.

  check satisfaction :
    User_Timeout > 10000, User_Timeout < 120000.

  update level :
    Response_Time > 30000, Level := 40.

end class.
```

Each property is defined as a TLG class whereas the non-functional attributes are defined as TLG instance variables such as Level, Maximum_Connections, Maximum_Allowed_Attempts, Encryption_Byte_Length, Allowed_Attempts, Account_Type, Response_Time, and User_Timeout. In our example, ATM has Security and Delay properties and Bank has Capacity property which is used in update attribute operation of ATM_Security. As the above TLG specification illustrates, all the property classes extend the class Property which has the instance variable Level. This variable is a representative value for the property, with which the decomposability of QoS is implemented. For example ATM_Security property has several attributes such as Encryption_Byte_Length and Allowed_Attempts. The value of Level for ATM_Security represents the overall security level after evaluating all the attributes.

Non-functional actions are represented as methods in the classes. In this example, there is a method that checks the level of property satisfaction (check satisfaction), that updates the overall level of the non-functional properties (update level), or that updates the individual attribute according to dynamic changes of other attributes (update attribute).

Attributes may be updated in a method when some conditions hold. These conditions may include not only the attributes in the same property of the same class, but also the attributes of other property or even in other classes. This is how the correlation of non-functional actions are implemented in TLG. For example, in the ATM_Security class above, if any of the following 3 conditions holds, the maximum number of allowed attempts (Maximum_Allowed_Attempts) is set to be 4: the account type (Account_Type) is a savings account, or the maximum number of connection allowed by the bank at one time (Maximum_Connections) (which is an attribute of Bank_Capacity class) is less than 50 connections, or the Level of ATM_Delay is less than 50.

Usually when a non-functional action is performed on a non-functional attribute, the non-functional attributes may change which, in turn, may trigger other actions to take place. In the ATM example, if some non-functional actions change Account_Type (which is an attribute of ATM_Security), Maximum_Connections (which is an attribute of Bank_Capacity), or the Level of ATM_Delay not only they themselves will be updated, but the value of Maximum_Allowed_Attempts will be updated as well according to the specification in the update attributes method in ATM_Security class.

In summary, as illustrated using a simple ATM example, TLG is proven to be a powerful specification language to formally specify non-functional aspects of a system with a mechanism to abstract the decomposability and to express dynamic correlations among properties and attributes.

4 Conversion from Natural Language Description of QoS into TLG

First the natural language description of QoS of the system is represented in XML to specify which role each sentence plays as a non-functional aspect or attribute. This process is carried out by a natural language parser as a preprocessing of the actual translation into TLG. A sample XML representation of ATM example is shown as follows.

```
<document>
<c title = "ATM">
<c title = "Security">
<p meta = "satisfaction check">
<s>The length of the encryption byte should be bigger than 3 and the allowed attempts has to be smaller
    than the maximum allowed attempts</s>
</p>
<p meta = "level update">
<s>If the encryption byte length is 6 and the allowed attempts is less than 5 then the system is 80%
    secure</s>
</p>
<p meta = "attribute update">
<s>If the account type is a savings account or the maximum allowed connections of the bank is less than 50
    or the delay level is less than 50 then the maximum allowed attempts is limited to 4</s>
</p>
</c>
<c title = "Delay">
<p meta = "satisfaction check">
<s>If the user timeout is between 10000 and 120000 milliseconds we have a good delay level</s>
</p>
<p meta = "level update">
<s>If the response time is longer than 30000 milliseconds the delay level drops down to 40%</s>
</p>
</c>
</c>
</document>
```

Titles such as `Security` and `Delay` indicates the property types whereas the meta information such as `satisfaction check`, `level update`, and `attribute update` indicates the non-functional actions within the property.

Given this XML representation of QoS, each sentence of the specification is tokenized and then by using computational linguistic parsing techniques the system constructs its correct parsing tree. This parsing tree contains the linguistic information about the sentence such as the part of speech (e.g. noun, verb, adjective) and the part of sentence (e.g. subject and object) of each word in the sentence. Obtaining this type of linguistic information is important in the later conversion into TLG because usually the subject of the sentence is identified as the component name. The verb normally indicates what kind of action this component takes to affect a specific QoS. Also anaphoric references (pronouns), elliptical compound phrases, comparative phrases, compound nouns, and relative phrases are handled to allow the input natural language description to be as less controlled as possible. The same technique has been used to automatically translate functional requirements documents into a formal specification language as well [10].

Using this linguistic information and the meta information from XML tags, a Knowledge Base is constructed. The Knowledge Base is an explicit and declarative representation that is used to represent, maintain, and manipulate knowledge about QoS of the system. In addition, the knowledge base has to reflect the structure of TLG into which the Knowledge Base is translated later. The Knowledge Base of the ATM example is shown in Figure 2. In the figure, the blank oval indicates `OR` where as the black ovals indicate `AND` relation. The sentences that are grayed out are the conditional statements compared with normal statements.

This Knowledge Base is converted into TLG by identifying the classes, data types, and operations. Once TLG specifications are obtained, the specifications are translated into VDM++ (we refer the readers to [4] for details). Using the VDM++ tool kit [9] the specifications can be in turn translated into a high level language such as Java or C++ or into a model in UML (Figure 3).

In summary, the QoS description in NL is represented in XML to specify the meta information and the Knowledge Base with a systematic structure can be used to capture this meta information as well as the linguistic information to be used to convert the description into TLG.

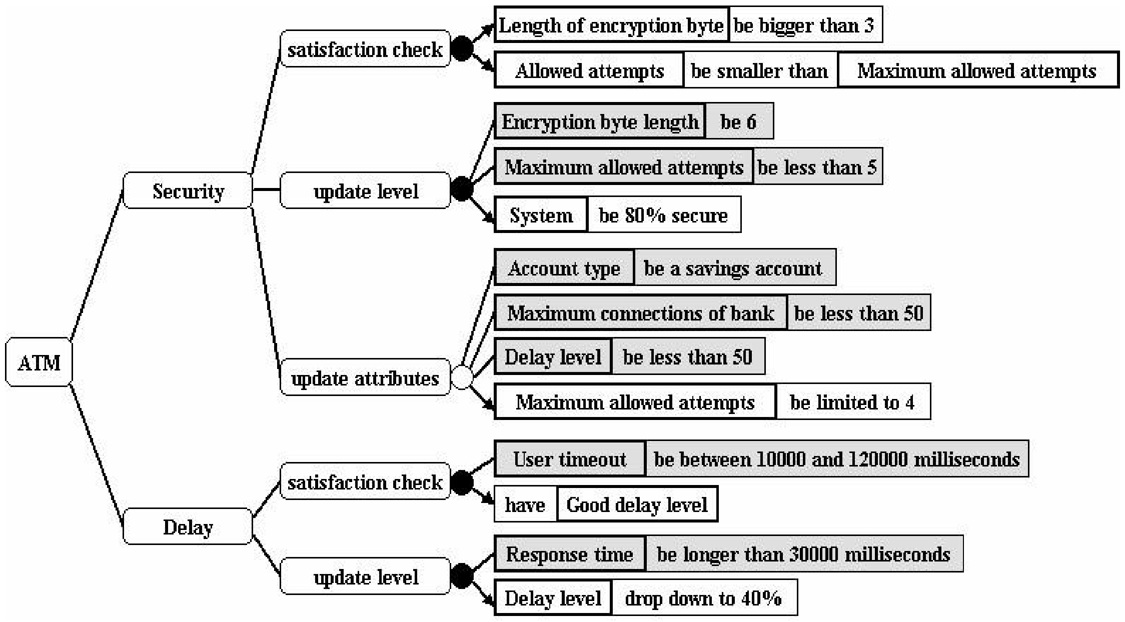

Figure 2: Knowledge Base for ATM example.

5 Conclusion

Non-functional aspects of the software specification are as important as functional aspects in software development. Formal representation of non-functional aspects is of great contribution to software engineering especially in distributed component-based systems. The specification has to be expressive enough to cover all the useful non-functional specifications while being able to describe complex decomposability and dynamic correlations among the non-functional properties.

In our research first the non-functional specifications are described informally in natural language according to a QoS parameter catalog. Then this specification in natural language is translated into TLG, a natural language like formal specification language. TLG is used to formally represent non-functional aspects of requirements for rapid prototyping and optimal distribution of components. We are performing evaluations of the system for various requirements documents. It is expected that the technology we are developing will be applicable to these requirements documents. If successful, this will provide a very useful tool to assist software engineers in moving from the requirements document to the formal specification.

OMG's Model Driven Architecture (MDA) [11] includes standards that enable the use of generative techniques for construction of interoperability bridges between platform technologies. It will be a promising and useful approach to combine Model Driven Architecture and formal methods in representing the non-functional aspects of software specifications. QoS issues in MDA have been explored in [5]. Our future work is to express the constraints in Object Constraint Language (OCL), and to automatically generate the OCL representation from the TLG representation of the non-functional aspects of software specification, and implement the representation within MDA. At the same time, we will continue developing the system to improve system usability and robustness with respect to its coverage of requirements documents.

References

[1] ASTER. Software Architectures for Distributed Systems (ASTER). Technical report, (`http://www-rocq.inria.fr/solidor/work/aster.html`), 2000.

[2] D. Bjørner and C. B. Jones. *The Vienna Development Method: The Meta-Language.* Springer-Verlag, 1978.

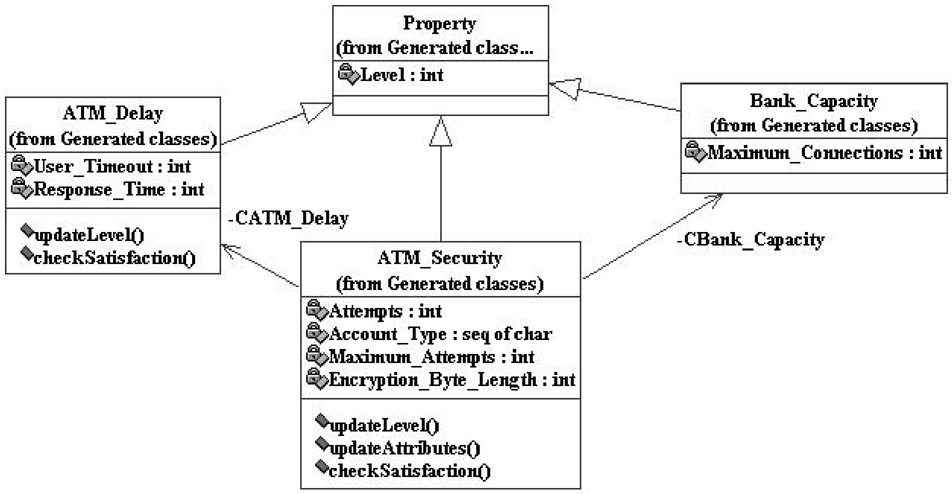

Figure 3: UML for ATM.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, W3C (`http://www.w3c.org/xml`), 2000.

[4] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Hawaii Int. Conf. System Sciences*, Jan. 2002.

[5] C. C. Burt, B. R. Bryant, R. R. Raje, A. Olson, and M. Auguston. Quality of Service Issues Related to Transforming Platform Indepent Models to Platform Specific Models. *Proc. EDOC 2002, 6th IEEE Int. Enterprise Distributed Object Computing Conf. (to appear)*, 2002.

[6] L. A. Campbell and B. H. C. Cheng. Integrating informal and formal approaches to requirements modeling and analysis. *Proc. IEEE International Symposium on Requirements Engineering (RE01)*, pages 294–295, 2001.

[7] 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.

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

[9] IFAD. VDMTools - Java/C++ Code Generator. Technical report, IFAD, 2000.

[10] B.-S. Lee and B. R. Bryant. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proc. ACM 2002 Symposium on Applied Computing*, pages 932–936, 2002.

[11] OMG. Model Driven Architecture (MDA). Technical report, (`http://www.omg.org/mda/`), 2000.

[12] P. Pal, J. Loyall, and R. Schantz et al. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. *Proc. 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, 2000.

[13] Qedo. QoS Enabled Distributed Objects. Technical report, (`http://qedo.berlios.de`).

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

[15] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. Burt. A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components. *to appear in Concurrency and Computation: Practice and Experience*, 2002.

[16] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, and C. C. Burt. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proc. 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 109–119, 2001.