Grammatically Interpreting Feature Compositions

Wei Zhao[1], Barrett R. Bryant[1], Fei Cao[1], Rajeev R. Raje[2], Mikhail Auguston[3], Carol C. Burt[1], and Andrew M. Olson[2]

[1] *Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, USA. {zhaow, bryant, caof, cburt}@cis.uab.edu*

[2] *Computer and Information Science, Indiana University Purdue University Indianapolis, Indianapolis, IN 46202, USA. {rraje, aolson}@cs.iupui.edu*

[3] *Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA auguston@cs.nps.navy.mil*

Abstract. Feature modeling is a popular domain analysis method for describing the commonality and variability among the domain products. The current formalisms of feature modelling do not have enough support for automated domain product configuration and validation. We have developed a theory of feature modeling: a feature model is analogous to a definition of a language; a particular feature composition instance (domain product) is analogous to a program written in that language; and the way the features can be assembled to form a product is analogous to the way various tokens can be assembled to form a program. To apply this theory, we have developed a meta-language Two-Level Grammar++ to specify feature models. The interpreter derived from the feature model specification performs automated product configuration and product quality validation.

1. Introduction

The systematic discovery and exploitation of commonality across related software systems is a fundamental technical requirement for achieving successful software reuse [13]. Domain analysis is one technique that can be applied to meet this requirement. Feature modeling is a popular domain analysis method originated in [11]. The current formalisms [6], [11] of feature modeling do not have enough integrity for supporting automated domain product configuration and validation (some tools were implemented to support limited automation up to the power of the original formalism, e.g. [4]). We have developed a theory of feature modeling so that the existing compiler technologies can be leveraged for automated domain product configuration.

A feature is a distinguishable characteristic of a concept that is relevant to the stakeholder of that domain [6]. We have defined that the anatomy of a feature is a modular encapsulation of three-dimensional views: an abstract view at the domain business level, a constructive view at the architectural pattern level and a concrete view at the implementation technology level. The artifacts in this encapsulation consist of both code and non-code. Examples of the artifacts are business domain models, design models, make files, HTML documents, XML descriptors, etc. We consider features to be concrete and non-cross-cutting concepts of a domain, i.e., a feature can be incarnated as a software component with specific programming and component technologies.

We consider a feature model to be a general specification of a domain: the rules about feature configurability and how to manufacture the valid product instances in that domain. So, a feature model is a definition of feature compositions. By observing that any language (machine, assembly, and high level languages) is a composition of language elements (constructs and tokens) at different abstraction levels, we are motivated to develop a language-based theory of feature modeling: a feature model is analogous to a definition of a language; a particular feature composition is analogous to a program written in that language; the way the features can be assembled to form a product is analogous to the way various tokens can be assembled to form a program; the interdependency relationships among the feature models are analogous to the object relationships that can be defined in object-oriented programming languages. A valid product for a domain can be created by composing a set of features adhering to the composition rules in the feature model. In a feature model, there are atomic features and composite features. An atomic feature is a feature that does not need to be further refined into sub features when there are no variations among different products. A composite feature is a composition of one or more atomic or composite features. Both the atomic and composite features are relative concepts. A composite feature in one feature model can act as an atomic feature in a foreign feature model. This hierarchy is called the

feature organization, and the structure of a product is called the product organization.

To apply successfully the programming language techniques to feature modeling, the first question to be answered is whether there exist concepts in feature models that are the counterparts of syntax and semantics in programming languages. The syntax of the feature model is the business domain feature organizational structure. The static semantics indicates the configuration constraints such as feature attributes, feature relationship cardinalities, interdependencies, and domain-specific business operational rules. The dynamic semantics models the states of system property changes after the steps of feature compositions. That includes pre- and post-conditions for the configurations, temporal concerns, and the Quality of Service (QoS) attributes [14], [17]. An example of a QoS attribute is transaction speed in the banking domain. We draw a clear delineation of semantics of a composition model (feature model) from semantics of a composed system. The semantics of a feature model is the non-functional quality aspect of a composition; the semantics of the composed system is the functional quality aspect of a composition. Feature model syntax defines the semantics of the composed system meaning that as long as the features are composed in a proper hierarchy, the composed system should function correctly assuming correct feature implementations and correct feature model. For example, if we build a money transfer system by composing features withdraw and deposit, the balance calculation is the semantics of the composed system, whereas the transaction speed calculation is the semantics of the composition model.

We have developed a meta-language called Two-Level Grammar++ (TLG++), an object-oriented extension of Two-Level Grammar (TLG) [18], to specify feature models. TLG, a Turing complete grammar, has been used for integrated definition of programming language syntax, static semantics and dynamic semantics, which makes TLG ideal for specifying the feature organization along with static configuration constraints and various dynamic semantic concerns. Because of object-oriented features, TLG++ naturally fits in the conceptual modeling of inter-connected object relationships among the feature organizations. The interpreter derived from the feature model specification performs automated product configuration and predicted product functional and non-functional quality validation.

According to the three-dimensional views of domain features, there are three dimensions of feature compositions: semantic-business composition, syntactic-architecture composition, and lexical-technology composition. For a particular product created by composing a set of features, the semantic-composition dimension defines the entangled business logic among the features and semantics for individual features; the syntactic-composition dimension defines a compositional architecture for this product; the lexical-composition dimension defines how each feature is technologically formed thus contributing to the binary connection, interoperation and deployment between any two feature-implementations. In this paper we only demonstrate the first dimension. However, a complete product quality validation requires all three-dimensional composition validations.

The following section introduces TLG and TLG++. A case study is given in section 3. Section 4 compares our work to related work. The paper concludes in section 5.

2. Two-Level Grammar++

Two-Level Grammar (van Wijngaarden grammar or W-grammar) is an extension of Context-Free Grammar (CFG) and was originally developed to define syntax and semantics of programming languages. It has been shown that TLG defines the family of recursively enumerable sets [15], and suitable restrictions yield context-sensitive languages [1]. It has been used to define the complete syntax and static semantics of Algol 68 [18] and dynamic semantics of programming languages [5]. Recently, it was developed as an object-oriented requirements specification language integrated with VDM[1] tools for UML[2] modeling and Java/C++ code generation [3].

The term “two-level” comes from the fact that a TLG is composed by two finite sets of CFG rules: a set of formal parameters may be defined using a CFG, with the possible generated strings used as arguments in predicate functions defined using another CFG. Originally, the first level CFG rules were called the meta-productions/rules, while the second level parameterized CFG rules were called hyper-productions/rules. After the meta-rules get substituted into the hyper-rules, a third implicit and possibly infinite set of CFG rules, called the production-rules, are derived. It is the production-rules that finally generate the target language that a TLG describes.

TLG++ is the object-oriented TLG for specifying feature models. Moving from TLG to TLG++ poses a paradigm shift. TLG is used to physically define programming language syntax and semantics, while TLG++ is used to abstractly define the concepts of a domain. To specify the feature model, we do not have the concept “terminal symbol in the target language” in TLG++. In fact, the terminals in the feature models are either the atomic features or domain-specific keywords.

[1] VDM – Vienna Development Method – http://www.ifad.dk/vdmtools
[2] UML – Unified Modeling Language – http://www.omg.org/uml

Examples of "domain-specific keyword" might be: a particular domain logic control pattern, domain-specific algorithm, and so on.

The atomic features in a feature model are represented by Universal Resource Identifiers (URIs)[3]. In the process of interpreter generation, the URIs are simply treated as terminals. While interpreting a specific product, there are two cases under consideration: 1) the atomic feature in this feature model is a composite feature in another feature model and there is no direct implementation for this atomic feature, then preprocessing can be adopted to ensure the instance atomic feature used in this particular product is in fact a valid instance defined by the corresponding URI, which might just involve another interpretation process; 2) this atomic feature has a direct implementation, in which case the Unified Meta-component Model (UMM)[4] needs to match the URI to complete the interpretation. From a single feature model perspective, the URI is treated programmatically rather than syntactically or semantically. Reasons for the use of URIs are:

First, the URI for an atomic feature or the URI compositions for a composite feature are the types of the component that implements this feature. Because of the nature of composition, the atomic feature has a single type, and the composite feature has type variations.

for example, `A :: B C; D. B :: E; F.` (5)
in (5), `A` and `B` are composite features, and `C, D, E,` and `F` are atomic features. ";" means "or". Suppose the URI for each atomic feature is the `http://` plus the letter symbol. So, atomic feature `C` has type `http://c`, `D` has type `http://d`, and so forth. The types of a composite feature are the composition of types of atomic features. The composition is a tree structure. The number of types of a composite feature equals the number of strings it can generate. In this case, `A` has 3 types shown in fig. 1. If the component developer chooses to implement a composite feature directly, he/she must identify a specific type of choice. A component is considered plug-compatible for another component that implements the same feature if and only if their types match. One reason we developed TLG++ to specify the feature model is because TLG++ naturally supports this hierarchical type structure since each parameter is defined by a context-free grammar.

Second, the use of URI gives the potential to specify very large and highly distributed domains, as some fairly complex features can be defined separately and linked by a URI. The mechanism of URI complements object-orientation for distribution and encapsulation.

[3]Naming and Addressing, http://www.w3.org/Addressing/

[4] UMM is the meta-model for feature implementation. Detailed explanation of UMM can be found in [13].

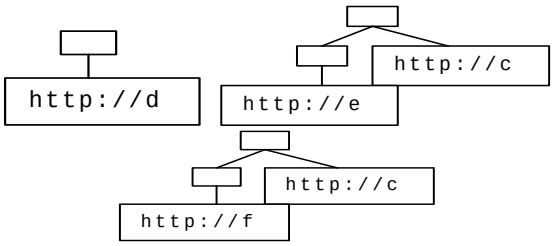

Fig. 1. The feature A has three types.

Third, compositions can be easily reused, e.g., a composite feature represented by a URI in a domain can stand as an atomic feature in another domain.

Lastly, the separation of the feature id from the feature representation allows features to have any physical form that the designer wishes for designing the visualized domain analysis tools, such as an icon, a name, or a simple box.

Object-orientation is proper for modeling real-world relationships. The feature models specified in TLG++ naturally model object relationship graphs in real-world business domains, since both domains of different categories and domains (sub-domains) in different levels of the hierarchy of the same category can have a feature model. Real-world domains are usually hierarchical: one narrower-scoped domain inherits the concepts from many other broader-scoped domains. With the encapsulation, a feature model for a domain is expressed in a main class, from which other classes might be linked. TLG++ has a root class `Notion` that is comparable to the `Object` class in the Java programming language. The class `Notion` groups general notational extensions (e.g., list, sequence and tree definition) [5], built-in data types, and primitive grammatical computations. Those pre-established grammatical concepts are inherited by any other TLG++ classes. Inheritance and encapsulation provide TLG++ much more power than CFG in terms of reusability and modularity. By polymorphism, users do not need to cite the `Notion` class in order to call its rules. A class can override the rules in its parent classes, and any class can override the rules in the `Notion` class.

3. A Case Study

In this section, we give an example on how a feature model is formulated, how a feature model is represented in TLG++, and how a product can be validated based on the feature model.

Fig. 2 shows a fragment of the feature model for a `PersonalAccount` domain. From the viewpoint of stakeholders of this domain, the feature model should capture the distinguished domain concepts (i.e., features: objects and operations) and the business rules on how

those concepts are composed to form a product. For the sake of this example, we assume the `PersonalAccount` domain has an object (`PersonalAccount`), and the operations (`Withdraw`, `Deposit`, and `MoneyTransfer`). `MoneyTransfer` is a composite feature composed of `Withdraw` and `Deposit`.

```
class PersonalAccount extends Banking.
  Account :: Integer.                              1
  Customer :: Name SocialNumber.                   2
  Bank :: Integer.                                 3
  Balance :: Integer.                              4
  Amount :: Integer.                               5
  TurnAround :: Integer.                           6
  Name :: String.                                  7
  SocialNumber :: Integer.                         8
  PersonalAccount::http://omg.org/bankdoma
    in/personalAccount.                            9
  Withdraw ::
    http://omg.org/bankdomain/withdraw.            10
  Deposit ::
    http://omg.org/bankdomain/deposit.             11
  MoneyTransfer :: .                               12
  ……
  turnaround TurnAround1 moneyTransfer
  MoneyTransfer :                                  13
    turnaround TurnAround2 customer
    Customer withdraw amount Amount1 from
    Withdraw,
    personal account customer Customer has
    account Account1 in bank Bank1 with
    balance Balance1 PersonalAccount,
    turnaround TurnAround3 customer
    Customer deposit amount Amount2 to
    Deposit,
    personal account customer Customer has
    account Account2 in bank Bank2 with
    balance Balance2 PersonalAccount,
    where Balance1 != 0,
    where Amount1 = Amount2,
    where Account1 != Account2,
    where TurnAround1 = TurnAround2 +
      TurnAround3.
  ……
end class.
```

Fig. 2. Bank domain feature model in TLG++

In the TLG++ representation, the first thing to be noticed is the separation of meta-rules and hyper-rules. Rules 1 to 12 are meta-rules, and rule 13 is a hyper-rule differentiated by using "**::**" and "**:**". Parameters start with a capitalized letter. The values (generated strings) of parameters defined in the meta-rules are called the Terminal Meta-Production (TMP) of parameters. Plugging the TMPs into the hyper-rules, we get the production rules. This parameterization, called the Uniform Replacement Rule (URR), is the essential theory that distinguishes the TLG from the pure CFG. The rule is that each occurrence of a parameter in a single hyper-rule needs to be replaced by the same TMP of that parameter. A parameter followed by a number is a new distinct parameter with the same definition as the root parameter. In the rule 13 `Acount1` and `Account2` will not necessarily be replaced by a same TMP of `Account`.

The convention we used is: the meta-rules are used to define the hierarchical context-free type structure for parameters; and the hyper rules define the syntax and semantics for feature compositions. `Integer` and `String` are built-in data types defined in the `Notion` class. Rule 12 is an empty definition, which shows that `MoneyTransfer` is a new composite feature to be defined in hyper rules. We assume the domain specifications should be created by domain experts working with standards organizations such as OMG[5].

Rule 13 specifies the syntax and semantics for the `MoneyTransfer` composition. TLG++ rules are natural language based, and are flexible in terms of writing styles. The convention we have adopted is that some words indicating the meaning are followed by the parameter, e.g., `turnaround TurnAround1.` In rule 13, the atomic feature is expanded with its specific parameters, e.g., `turnaround TurnAround2 customer Customer withdraw amount Amount1 from` stands for atomic feature `Withdraw`. Each rule begins with the syntax definition followed by the definition of semantics in `where` clauses. The static semantics here includes: the customers must be identical (ensured by URR); the accounts must be distinct; the amount withdrawn and deposited must be the same; the account to be withdrawn from must have a positive balance. Recall that the dynamic semantics of the feature composition and the dynamic semantics of the composed system are distinct. The balance calculations, after the actions withdraw and deposit, are the dynamic semantics of the composed system, which we will not be able to specify in a feature model. One QoS parameter, `TurnAround` time, is defined as the composition dynamic semantics. This example should convince the reader that features until being implemented as components are static concept entities rather than computation entities.

`PersonalAccount` is a sub-class of `Banking` where some basic features and feature compositions can be inherited. Polymorphism may exist as well. For example, the syntax definition of `MoneyTransfer` could be moved up to the `Banking` feature model. There might be another class `BusinessAccount` sub-classing `Banking`. So, the `BusinessAccount` feature model only needs to specify its specific semantic rules such as the requirement of a special security monitor for the `MoneyTransfer` composition, and `TurnAround <=10`.

Suppose the product we are trying to build is a simple `MoneyTransfer` system that can be created by

[5] OMG - Object Management Group - http://www.omg.org.

composing `Withdraw` and `Deposit`. For the validation of this product, the following are important points:

1. The goal of the validation is to find out if the feature compositions (the business logic or semantics of the product) are correct, and whether the product will have expected QoS using the supplied components.
2. The composition of components in this example occurs dynamically, so the state is an important concept. The state of a running component that implements a particular feature is the business data currently maintained. Please note that this paper presents the composition in the semantic-business dimension, not on the architecture or implementation dimension, so the state is not the state of the machine that runs this component. There are two cases regarding the state: first, the component is already running; second, the component has been produced, but is not yet running. In the second case, the state is the initial state, i.e., the state that is instantly after the component is invoked. In both cases, the component is treated as a black box. Those two cases give the views of dynamic product-line assembly and static product-line assembly respectively. The example in this section is of the first case.
3. Yet the validation for the composition is static. We are not going to run the system in order to test if the system is built by a correct composition. The UMMs of the implementation components provide the feature URIs, the QoS values, and the states information. This information comprises a sentence that stands for the product we are going to build; and this sentence should be interpreted according to the feature model definition. Currently, the UMM is represented in XML and is generated automatically by the tool support from the component developer [14]. We are investigating how to convert the XML based representation to a string of text so that the product can be interpreted, or to extend the ability of the interpreter to interpret the XML strings directly.

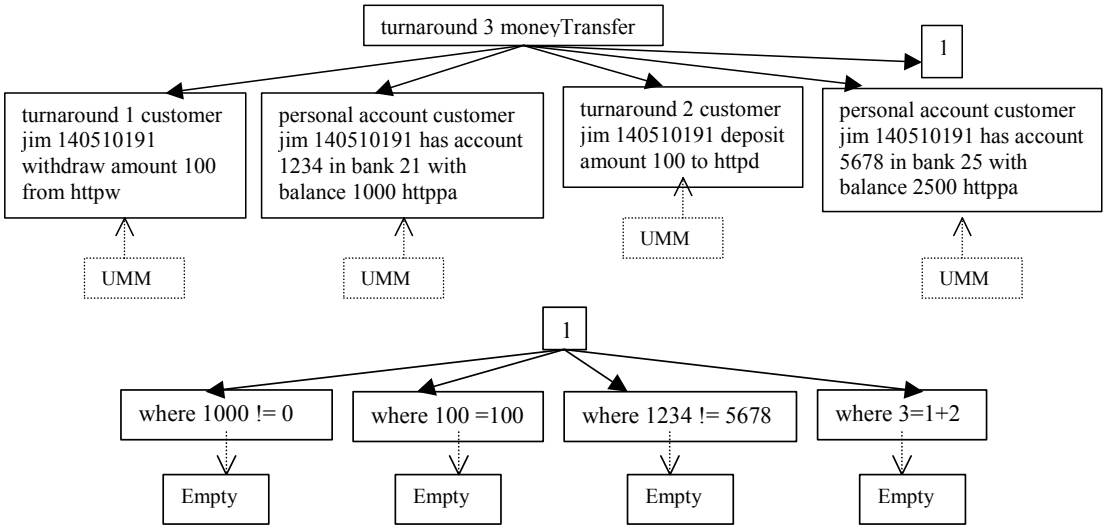

Fig. 3. The parse tree for the product `MoneyTransfer`. `httpw`, `httppa`, `httpd` stand for the URIs of `Withdraw`, `PersonalAccount`, and `Deposit`, respectively.

Fig. 3 provides a parse tree for a sample `MoneyTransfer` product to show how TLG++ can grammatically interpret this particular product to validate both functional and non-functional composition. As this composition is simple, the parse tree is not so deep. The QoS attribute `TurnAround` time for this product is expected to be 3 milliseconds. The state information and the QoS value for the implementation components are randomly selected for the illustration of this example. For an easier understanding, the interpretation process illustrated in fig. 3 is top-down and directly depends on the implicit production-rules, i.e., all the parameters have been non-deterministically substituted into the hyper-rules before the interpretation process begins. In fig. 3, consider when we interpret the product `turnaround 3 moneyTransfer`, we apply rule 13 because the parameter `TurnAround1` has been non-deterministically and implicitly substituted by its value `3` before the interpretation begins.

From this bank domain feature model specification, the bank domain product interpreter can be generated automatically using our tool—the TLG++ interpreter. The TLG++ interpreter uses the CUP[6] parser generator once

[6] CUP – Construction of Useful Parsers - http://www.cs.princeton.edu/~appel/modern/java/CUP

for the meta-rules and once for the hyper-rules to generate two sets of parsers. So, from the implementation point of view, the interpretation process of a product is bottom up and driven by the hyper-rules looking up the generated parser for each parameter whenever it encounters a parameter in the hyper-rules. This look-up process resembles looking-up the value of variables in the symbol table during the interpretation of programming languages. For example, while parsing the money transfer product, in order to apply rule 13, the parser for the parameter `TurnAround` is picked up and parses the string `3` to test if it may be derived by `TurnAround`.

4. Related Work

Feature Diagrams. In the literature, a feature model usually includes [6]: a feature diagram that portrays feature organization; feature semantic definitions; feature composition rules and configuration constraints; rationale for features indicating the reason for choosing or not choosing a given feature. Normally, the feature diagram is represented graphically by a CASE tool, and other semantic aspects of the feature model are annotated using natural language [12], or are linked to other more formal techniques such as object diagrams, interaction diagrams, state-transition diagrams, and synchronization constraints [6]. The separation of the feature diagram and its semantic aspects drastically hinders the automated configuration and validation of domain products. Furthermore, the popular feature diagram computation model is rather primitive in terms of computation power of the mathematical computation models. When the layers of the feature diagram are flattened, the terminal productions (a set of terminal features generated from the feature diagram) can be represented by a regular expression. The tree-shaped feature diagram is even less powerful than the regular expression because the star operation in regular expressions does not have a counterpart in the feature diagram. Compared to the feature diagram, TLG++ is much more powerful in computation and presents better integrity in representing both the syntactic and semantic aspects of feature models.

Domain specific languages. Domain-Specific Languages (DSLs) [7] always have the pre-constructed notations and abstractions offering expressive power for a particular domain. No matter whether a DSL is in a graphic form, or in textual form, it has its own syntax and semantics definition. But in this paper, we define the domain directly as a language. Any compositions in the domain are the relationships presented by the grammar rules and are not physically represented by any non-grammatical symbols (+, *, etc.), or built-in operator notations in the meta-language. We call this an *open* operator definition, which gives much flexibility to the meta-language for the evolution of operators of a domain, i.e., we only need add some new TLG++ rules for the new operators.

Composition Language. Composition Language (CL) [8] has defined composition semantics (QoS) such as latency, safety, and availability on the component model level. It did not address how to formalize QoS in the dimension of business domain semantics.

GenVoca. GenVoca is a software system generator [2]. The composition validation in GenVoca is also based on the claim that the domain defines a grammar whose sentences are software systems. Attribute grammars are used for the design rules validation including pre/post condition and pre/post restrictions. Although the model of validation sounds similar, there are some fundamental differences. In GenVoca, the principle for component composition is parameterization among components, and hence the composition is directly coupled with the component implementation language. In this paper, the composition is defined by the domain feature organizational structure and the associated semantic rules. This give a higher level of view of composition and the features can be potentially implemented in different technologies.

5. Conclusions

We have offered a foundation for the feature composition and an automated way to validate a composed system. We have addressed how both the functional and non-functional aspects of composition can be formally modeled and validated.

The method chosen to specify the feature model is Two-Level Grammar++. We could choose attribute grammar to specify the feature model, and it also provides Turing computability. However, just as the attribute grammar is not proper for specifying programming language dynamic semantics [9], it is not proper for specifying the dynamic semantics of feature composition. Compared to a regular programming language or other formal notations such as Z [16] and feature logic [20], a grammar has better constructs and computation mechanism for specifying languages. Specifically, TLG++ as a meta-language chosen in this paper has the following advantages:

1. The ability of TLG++ to integrate the feature model syntax and semantics into one formal grammar notation gives formal consistency and completeness of the specification, and eliminates the task of building a separate interpreter. Compared to the conventional way that defines the syntax with a grammar and explains the

semantics in natural language, the integration of syntax and semantics poses easier maintenance.

2. TLG has a context-free hierarchical type structure, which supports the type system in feature modeling.
3. Because both the meta and hyper rules of TLG++ are CFGs, the derivation of the product parser/interpreter can be automated and facilitated by existing parser generators, which also makes the implementation of the TLG++ meta-language easier.
4. As can be seen from the examples, the natural language style of TLG++ rules improves the language flexibility and readability.

TLG++ is simple (the only rule is URR) and flexible (natural language based). Flexibility presents a great descriptive potential but also gives the disadvantage to well control the language. The notation is not desirable to be directly used by the domain engineers, so we are investigating embedding the grammatical interpretation engine into a domain specific modeling tool such as GME [10], [19] to complement the graphic modeling notations with the automated semantic interpretation. Furthermore, formally specifying the feature models are magnitudes harder than specifying programming languages because in the programming language domain, common patterns of language constructs are well known and the conventions of writing a correct and complete specification are easy to establish. We have experienced that not only the formulation of a domain abstraction, but also the establishment of TLG++ conventions for the purpose of feature model specification are inventive and challenging tasks.

6. Acknowledgement

This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

[1] J. L. Baker, Some Formal Properties of the Syntax of ALGOL 68, Doctoral Dissertation, University of Washington, 1970.

[2] D. Batory, B. J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", IEEE Trans. Softw. Eng., Vol. 23, No. 2, pp. 67-82, 1997.

[3] B. R. Bryant, B.-S. Lee, "Two–Level Grammar as an Object-Oriented Requirements Specification Language," Proc. 35th Hawaii Int. Conf. System Sciences, Vol. 9, 2002.

[4] F. Cao, Z. Huang, B. Bryant, C. Burt, R. Raje, A. Olson, M. Auguston, "Automating Feature-Oriented Domain Analysis," Proc. of the 2003 International Conference on Software Engineering Research and Practice (SERP'03), CSREA Press, pp. 944-949, 2003.

[5] J. C. Cleaveland, R. C. Uzgalis, Grammars for Programming Languages, Elsevier North-Holland, Inc., 1977.

[6] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[7] A. van Deursen, P. Klint, J. Visser, "Domain-Specific Languages: An Annotated Bibliography", CWI, 2000, http://homepages.cwi.nl/~arie/papers/dslbib/

[8] J. Ivers, N. Sinha, K. Wallnau, "A Basis for Composition Language CL", Technical Note, CMU/SEI-2002-TN-026, 2002.

[9] G. E. Kaiser, "Incremental Dynamic Semantics for Language-based Programming Environments", ACM Trans. Program. Lang. Syst. Vol. 11, pp. 169-193, 1989.

[10] GME User's Manual. The Institute for Software Integrated Systems, Vanderbilt University. http://www.isis.vanderbilt.edu/projects/gme/

[11] K. C. Kang, S, G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, 1990.

[12] K. Lee, K. C. Kang, J. Lee, "Concepts and Guidelines of Feature Modeling", Proc. 7th Int. Conf. Software Reuse, pp. 62-77, 2002.

[13] R. Prieto-Diaz, "Domain Analysis: An Introduction", ACM SIGSOFT Softw. Eng. Notes Vol. 15, pp. 47-54, 1990.

[14] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, "A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components," Concurrency Comput.: Pract. Exp. Vol. 14, pp. 1009-1034, 2002.

[15] M. Sintzoff, "Existence of van Wijingaarden's Syntax for Every Recursively Enumerable Set," Ann. Soc. Sci. Bruxelles, Vol. 2, pp. 115-118, 1967.

[16] J. M. Spivey, The Z Notation: A Reference Manual. Prentice Hall, New York, 1989.

[17] C. Sun, R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt, Z. Huang, "Composition and Decomposition of Quality of Service Parameters," Proc. 5th Int. Conf. Algorithms and Architectures for Parallel Processing, pp. 273-277, 2002.

[18] A. van Wijngaarden, "Revised Report on the Algorithmic Language ALGOL 68." Acta Informatica, Vol. 5, pp. 1-236, 1974.

[19] W. Zhao, B. Bryant, J. Gray, C. Burt, R. Raje, M. Auguston, A. Olson. "A Generative and Model Driven Framework for Automated Software Product Generation". Proc. of the 6th ICSE Workshop of Component Based Software Engineering, pp. 103-108, 2003

[20] A. Zeller, G. Snelting, "Unified Versioning Through Feature Logic", ACM Transactions on Software Engineering and Methodologies, Vol. 6, No. 4, pp. 398-441, 1997.