Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach[1]

Shih-Hsi Liu[2], Fei Cao[2], Barrett R. Bryant[2], Jeff Gray[2], Rajeev R. Raje[3], Andrew M. Olson[3], and Mikhail Auguston[4]

Abstract

Component-based software engineering offers the opportunity to assemble entire systems from components. When applied to Distributed Real-Time and Embedded (DRE) systems, which components to assemble and how to assemble them are determined not only from functional correctness criteria but also assurance of the system's quality of service (QoS). This paper presents a grammatical QoS-driven approach to optimize component assembly by reducing the search space of assembly alternatives by eliminating infeasible components, with feasible components selected based on reasoning about non-functional requirements. The reasoning is realized by a rule engine with a knowledge base derived from the requirements phase of the software lifecycle. In addition, the grammatical approach introduces well-defined semantics among the components being composed. The semantics assist in precisely and efficiently evaluating the individual component QoS, as well as system-wide QoS in a programmable fashion. The result is to facilitate straightforward and manageable component composition analyses from the perspective of QoS requirements.

1 Introduction

Distributed Real-Time and Embedded (DRE) software systems are becoming increasingly complex. Such complexity can only be managed by Component-Based Software Engineering (CBSE), that is, building such systems from a collection of standardized and customized components. The integration of such components into a software system is the major effort in constructing such systems. Another dimension of such systems is the notion of Quality of Service (QoS), which transcends functional properties to include non-functional properties such as real-time and security issues. When DRE systems are constructed, QoS plays a critical role in determining the quality of the system. Along with functional specifications and models of the components, QoS attributes must also be specified and validated. The vision of the UniFrame project [9] is the development of techniques and tools that will enable software engineers to construct a DRE system by locating software components scattered about an organization or from third parties, evaluating the compatibility of heterogeneous components, generating connectors for the dissimilar pieces and validating a system composed from them.

This paper presents a grammatical QoS-driven approach to solve the challenges of black box component composition based on QoS. This approach expresses the system requirements in terms of QoS parameters and manipulates the QoS requirements using grammar rules which assure the correctness of the composition with respect to QoS and pre-conditions and post-conditions of each composition. This verification assists in eliminating the infeasible alternatives for any pre-condition or post-condition that does not satisfy the corresponding QoS constraints (i.e., facts) stored in the knowledge base. The knowledge base consists of specific composition rules for inferring the applicability of component composition. If all conditions are verified, the composition is assured. The systematic optimal solution of all QoS parameters can be evaluated by defining a specific QoS utility function of various QoS parameters. The specification of QoS requirements using grammar and rules facilitates the straightforward and manageable component composition analyses from the perspective of QoS parameters.

The paper is organized as follows: the next section provides background; section 3 proposes the concepts and an example; section 4 concludes the paper.

2 Background

The evolution of new techniques for software development is driven by the requirements of scalability within the growing complexity and size of modern software. To avoid developing scalable complex systems from scratch, CBSE enables the composition of commercial off-the-shelf (COTS) components, thereby benefitting software develop-

[1]This research is supported in part by U. S. Office of Naval Research award N00014-01-1-0746.

[2]Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, USA, {liush, caof, bryant, gray}@cis.uab.edu

[3]Department of Computer and Information Science, Indiana University-Purdue University-Indianapolis, Indianapolis, IN 46202-5132, USA {rraje, aloson}@cs.iupui.edu

[4]Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943-5193, USA {maugusto}@nps.navy.mil

ment by reusing and replacing components as needed. Software product lines [4] enrich the merits of CBSE by analyzing and constructing a set of software systems that share commonality and variability under specific considerations. The integration of CBSE and software product lines expedites the pace of software development, and proliferates the productivity of software products. The integration poses the following challenges for QoS-sensitive systems:

The Component Perspective Problem
Functional requirements define the functionality that systems should perform, and non-functional requirements specify constraints on system resources. Most systematic requirements analyses are component-driven [8], i.e., the analyses are based on the perspective of components and their functional requirements rather than non-functional requirements. The primary insufficiency of the component-driven analyses for QoS-sensitive system is that non-functional requirements are often tangled with functional ones. As numerous QoS characteristics require evaluation, separation of requirements concerns assists in manageably evaluating functional and non-functional requirements.

The Abundant Alternatives Problem
Hundreds of alternatives are generated based on the requirements of different composition decisions and permutations of selected components. The evaluation and management of abundant alternatives result in intensive workloads in the requirements phase.

The Composition Semantics Problem
Because component-driven analyses concentrate on the component units, the correlative composition semantics are not rich enough to state the composition influences on the QoS parameters. For example, the description of degradation and upgrade of certain QoS parameters is difficult by the component-driven composition semantics. Therefore, the evaluation of QoS parameters may not be performed in isolation, especially for some QoS parameters which mutually influence one another.

3 A Grammatical QoS-Driven Approach

Two-Level Grammar++ (TLG++) [3] is an object-oriented formal specification language, which consists of two Context-Free Grammars (CFGs) defining the set of parameters and the set of function definitions over the parameters, respectively. Originally, TLG++ was used for defining the syntax and semantics of programming languages: the first level consists of the production rules of the syntax and the second level interprets the semantics of these rules. TLG++ has been used for both specification of rules for component assembly [2] and for composing features to describe the characteristics of components [10]. In addition, TLG++ code can be automatically converted into Java using T-Clipse [7], an Integrated Development Environment for TLG++. In our approach, every QoS parameter is represented by a class of TLG++: the first CFG shows the components of alternatives and the necessary parameters used for the function definitions. The second CFG describes the function definitions, which include the reasoning operations and computational operations (i.e., composition semantics) regarding QoS parameters. The reasoning operations are used for analyzing and verifying pre-conditions and post-conditions of each composition. For the pre-conditions, preliminary queries verify that the components own the appropriate functions operating the QoS. Analytic queries then request the QoS information of specific components. For the post-conditions, the conclusive queries send back the composed "pattern" (i.e., the selected components and the QoS dataflow among these components) to avoid any conflict with respect to the constraints; namely, verification of post-conditions. If preliminary, analytic or conclusive queries return false, the alternative is infeasible and discarded.

We use Jess [5] as the underlying rule inference engine for reasoning about alternatives' feasibility regarding QoS requirements. Jess is a forward and backward chaining rule engine for the Java platform, which bridges Java and the rule-based language. Jess includes a Java library for defining rules, facts and queries, and for invoking the rule engine. The knowledge base accumulates the facts and rules regarding the components and QoS parameters. Queries request answers inferred from the facts and rules stored in the knowledge base. The querying results obtained from the rule engine are converted into interpretable Java objects for further processing tasks written in Java.

The primary concepts and motivations of applying TLG++ to a QoS requirements analyses approach in the context of CBSE and software product lines assume the following: (a) the components, having functions computing a QoS parameter, are like the operands of an expression; (b) composition semantics are treated as the operator of two (sets of) components; (c) production rules[5] are the counterparts of composition decisions, which imply the dataflow of the QoS parameters among components. Constructing a system is actually the same as defining a programming language with syntax and semantics. Under such a concept, Extended Backus-Naur Form (EBNF) [1] can represent mandatory, alternative (i.e., one of), optional and "OR" (i.e., more of) features of components involved in a software product line, as in Feature-Oriented Domain Analysis [6]. The syntax trees generated by applying different sets of production rules can be treated as the counterparts of the alternatives of a software product line. TLG++, consisting of two tightly coupled CFGs, is appropriate for the grammatical QoS-driven approach to define customized and comprehensive semantics for component composition.

[5] Production rules may have ambiguity, left recursion and left factoring problems. Analyzers should avoid these grammatical problems.

Figure 1 shows the procedures for analyzing systematic QoS requirements. First, analyzers write all QoS parameter classes in TLG++, which define the involved components and the composition semantics among the components regarding the QoS parameter. T-Clipse transforms TLG++ into Java. Second, the strict QoS parameters are evaluated, because they are the strict feasibility criteria for the alternatives. Third, all orthogonal QoS parameters are individually evaluated, and every set of non-orthogonal QoS is collectively estimated. Orthogonal QoS parameters imply that adaptation will not influence other QoS parameters, yet non-orthogonal QoS parameters substantially influence other QoS parameters. After all sets of non-orthogonal QoS are assured, the cumulative goals, the final selection criteria of alternatives, can be computed by a user-defined algebraic function over all assured QoS parameters. All of the fulfilling patterns of the software product lines will be stored in the knowledge base for the future queries. In the situations that strict, orthogonal or non-orthogonal QoS are not satisfied, a new (set of) component(s) will be selected as a new alternative to be evaluated.

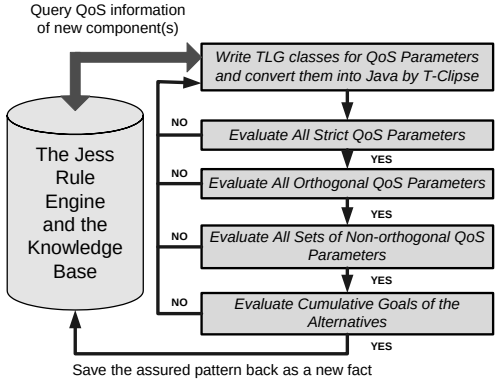

Figure 1. The procedures of the approach.

Figure 2 shows the user-defined grammars for each QoS parameter: the C_i are the terminals that represent components, and D_j, E_k, and F_l are nonterminals that describe the composition decision and the QoS dataflow. The left box, the middle box, and the right box, are the grammars for *Security*, *Signal*, and *a set of non-orthogonal QoS (Time, CPU Usage,* and *Battery Life)*, respectively. Please note that some production rules have left factoring, which may be eliminated as described in [1].

1 Security → C1 C2 D1 2 D1 → C3 D2 | C4 D3 3 D2 → C4 C5 | C5 C6 4 D3 → C5 D4 | C5 C7 5 D4 → C3 C7	1 Signal → C1 C2 E1 2 E1 → C3 E2 | C4 E3 | C5 E4 3 E2 → C6 C7 4 E3 → C3 C5 E5 | C3 C6 5 E4 → C4 C6 C7 6 E5 → C7	1 CPU → C1 F1 | C2 F2 2 F1 → C2 C4 F3 | C3 C4 F4 3 F2 → C5 C6 F5 | C5 | C6 F6 4 F3 → C7 C6 5 F4 → C2 C5 6 F5 → C3 C7 7 F6 → C1 C4

Figure 2. Grammars for QoS parameters

The cascading scenario is introduced to evaluate orthogonal QoS parameters. A set of components is chosen as the starting point of a QoS dataflow. The consequent components are opted by specific decisions such as *AND* and *OR*. *AND* means the dataflow streams into a set of components, and *OR* implies the new alternatives of the software product line are generated. As a QoS dataflow requires a new composition decision, a new TLG++ class is written: the parameters include the new components being selected, and the functions define the composition semantics between its ascendant and itself with respect to the QoS dataflow.

The upper box of Figure 3 represents the TLG++ class for the first production rule in Figure 2, the starting point of the *Security* QoS dataflow. In the upper box, line 2 comprises the first CFG that defines the selected components for the second CFG. Lines 3 to 29 comprise the second CFG that describes the semantics for composition, including computational and reasoning operations. Lines 3 and 4 verify the pre-conditions of Components Comp_1 and Comp_2. In "queryComponent" (lines 12 to 27), the functions of the Java API for the Jess rule engine (e.g., executeCommand) are invoked. Lines 13 to 15 define the query for searching the facts of QoS parameters. Lines 18 to 21 define where the querying results should be stored. Lines 23 to 25 comprise the semantics that define how to fetch the elements of the facts of the QoS parameter. After verifying the pre-conditions, lines 5 to 6 compute the QoS value based on the composition semantics defined in line 28. Finally, line 9 verifies the post-condition of the composition by checking if the composed QoS value is out of range. The lower box of Figure 3 is the Security_2 class for composing Security_1 using the second production rule based on the cascading scenario. In the lower box, the semicolon in line 2 means there are optional components for the software product line (i.e., the counterpart of "|" in the EBNF). Therefore, this box contains two composition semantics for components Comp_3 and Comp_4, respectively. *Signal* is defined in the similar way using its grammar in Figure 2.

For non-orthogonal QoS analyses, it is difficult to find the optimal balance when one non-orthogonal QoS parameter increases and the other one decreases. The coarse-grained scenario extends the cascading scenario for non-orthogonal analyses. All sets of non-orthogonal QoS parameters are written in TLG++ classes using the cascading scenario. A TLG++ class defines a weighted algebra function over each set of non-orthogonal QoS parameters (in this paper, *Time*, *CPU Usage*, and *Battery Life*) to discover the maximum value. Figure 4 shows the decision trees of five QoS parameters, expressing every composition decision as a branch of the tree. If any component in a QoS dataflow violates strict QoS (i.e., gray nodes), the following nodes (i.e., stripe nodes) are eliminated. The cumulative goal is computed by a user-defined algebraic function over all feasible goals of QoS parameters.

```
class Security_1 implements Serializable
2  Product_Line :: Comp_1 Comp_2. // All other parameter declarations ignored
3  Query_1 := semantics of queryComponent with Comp_1;//verify pre-cond. of Comp_1
4  Query_2 := semantics of queryComponent with Comp_2;//verify pre-cond. of Comp_2
5  Query_3 := if Query_1 && Query_2, then semantics of minimum with
6    Comp_1 and Comp_2, else False, end if;
7  //if both Query_1 and Query_2 are true, compute the composition semantics of
8  //Comp_1 and Comp_2. Otherwise, stop analyzing the alternative
9  Query_4 := semantics of queryPattern with QoSValue;//verifies post-cond. check range
10 if Query_4, then MyRete semantics of UpdatePattern, else "False", end if.
11 //if Query_4 is true, the composed pattern is assured. Update the pattern to KB
12 semantics of queryComponent with Component :
13   MyRete semantics of executeCommand with "(defquery QoSSearch (declare
14   (variables ?comp)) (qos (mycomponent ?comp) (myfunc ?func) (qoslow ?low)
15     (qosup ?up)))";
16   //define the Jess query for the QoS parameter, which has the fields of components,
17   //functions, lower bound and upper bound.
18   ValueVector_1 := ValueVector semantics of addAll withValue_1;
19   //Store the fields into ValueVector, an API provided by Jess' Java library
20   MyRete semantics of store with "RESULT" and MyRete semantics of RunQuery
21     with "QoSSearch" and ValueVector_1;
22   //Store the result of the query into the RESULT variable
23   MyRete semantics of executeCommand with "(run-query QoSSearch "+
24     component+")"";  //Run the query component is the variable of the query
25   Iterator_1 := MyRete semantics of fetch with "RESULT";//RESULT saved to Iterator
26   if Iterator_1 != null, then return TRUE, else return FALSE, end if.
27   //if the first field has no component defined, the pre-condition is not verified
28 semantics of minimum with Component1 and Component2 ://…ignored
29 //semantics of queryPattern, and UpdatePattern are ignored.
end class
```

```
class Security_2 implements Serializable. // All other parameter declarations ignored
2  Product_Line :: Comp_3 ; Comp_4.  //Comp_3 OR Comp_4 as alternatives
3  semantics of ProductLine_1 with Component1 : //semantics for Comp_3 OR Comp_4
4    Query_1 := semantics of queryComponent with Component1;//verify pre-cond.
5    //queryComponent has same semantics in Figure 3
6    if Query_1, then semantics of addition with Security_1 and Component1,
7    else False, end if;
8    Query_2 := Rete semantics of queryPattern with QoSValue;
9    if Query_2, then Rete semantics of UpdateFact, Rete semantics of UpdatePattern, else
10   "Composition False", end if. //verify the post-condition
11 semantics of addition with Component1 and Component2 : //…ignored
12 //semantics of queryPattern, UpDateFact and UpdatePattern are ignored here.
end class
```

Figure 3. Security_1 and Security_2 in TLG++

4 Conclusions

The grammatical QoS-driven approach defines the syntax of software product lines, and the semantics of the component composition from the QoS parameter perspective. The approach eases the burden of management and evaluation of QoS that the component-driven approaches suffer from. It also achieves three goals: reducing the infeasible alternatives, assuring the feasible ones, and manageably evaluating orthogonal QoS and mutually-influenced QoS. Finally, a stand-alone inference engine separates the inference concern for component composition.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] B. R. Bryant, M. Auguston, R. R. Raje, C. C. Burt, and A. M. Olson. Formal specification of generative component assembly using Two-Level Grammar. In *Proc. of 14th Intl. Conf. on Software Engineering and Knowledge Engineering*, pages 209–212, 2002.

[3] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an object-oriented requirements specification language. In *Proc. of the 35th Hawaii Intl. Conf. on System Sciences*, 2002. http://www.hicss.hawaii.edu/HICSS_35/HICSS papers/PDFdocuments/STDSL01.pdf.

[4] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[5] E. J. Friedman-Hill. *Jess 7.0, The Rule Engine for the Java Platform*. Sandia National Laboratories, 2005.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[7] B.-S. Lee, X. Wu, F. Cao, S.-H. Liu, W. Zhao, C. Yang, B. R. Bryant, and J. G. Gray. T-Clipse: An integrated development environment for Two-Level Grammar. In *The OOPSLA'03 Eclipse Technology Exchange Workshop*, pages 91–95, 2003.

[8] M. Matinlassi. Comparison of software product line architecture design methods: COPA, FAST, FORM, KorbA and QADA. In *Proc. of the 26th Intl. Conf. Software Engineering*, pages 127–136, 2004.

[9] R. R. Raje, B. R. Bryant, M. Auguston, A. M. Olson, and C. C. Burt. A QoS-based framework for creating distributed and heterogeneous software components. *Concurrency and Computation: Practice and Experience*, 14:1009–1034, 2002.

[10] W. Zhao, B. R. Bryant, F. Cao, R. R. Raje, M. Auguston, C. C. Burt, and A. M. Olson. Grammatically interpreting feature composition. In *Proc. of 16th Intl. Conf. on Software Engineering and Knowledge Engineering*, pages 185–191, 2004.

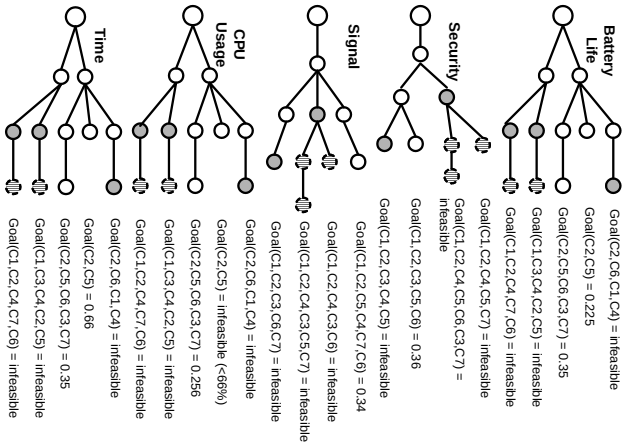

Figure 4. Decision trees of QoS parameters