QoSPL: A QoS-Driven Software Product Line Engineering Framework for Distributed Real-time and Embedded Systems

Shih-Hsi Liu[1], Barrett R. Bryant[1], Jeff Gray[1], Rajeev Raje[2], Mihran Tuceryan[2], Andrew Olson[2] and Mikhail Auguston[3]

Abstract

The current synergy of Component-Based Software Engineering (CBSE) and Software Product Line Engineering (SPLE) requires evolution to facilitate Distributed Real-time and Embedded (DRE) system construction. Such evolution is driven by inherent Quality of Service (QoS) characteristics in DRE systems. This paper introduces a QoS-driven SPLE framework (QoSPL) as an analysis and design paradigm for constructing a set of DRE systems as a product line. Leveraging separation of concerns, DRE systems are analyzed and designed by a collection of QoS systemic paths, each of which individually determines how well the service performs along the path and as a whole represents a behavioral view of software architecture. The paradigm reduces construction workload from the problems of tangled functional and QoS requirements and abundant infeasible design alternatives, and offers a less subjective QoS evaluation. The adopted formalisms also facilitate high-confidence DRE product line construction.

1. INTRODUCTION

Component-Based Software Engineering (CBSE) offers a possible solution to foster high-confidence system construction stipulating design by contract [3]. Software Product Line Engineering (SPLE) [10] further enriches CBSE by analyzing and reusing common features. Distributed Real-time and Embedded (DRE) systems are mission-critical and highly complex [14], requiring satisfaction not only of time-critical issues, but also numerous functional and Quality of Service (QoS) requirements specific to the mission. To handle such complexity, the synergetic technologies of the current CBSE and SPLE concepts are adapted to DRE system construction. Such technologies, however, are not a panacea because of the inherent QoS characteristics of DRE systems [13]. The three main obstacles to building such systems by the current CBSE and SPLE techniques are:

Challenge 1 - The QoS Sensitive Problem: The performance fulfillment of mission-critical component-based DRE systems pertains to the availability of system resources, which directly affect the stringent QoS demands of the system [14]. Insufficient or unmanageable QoS provisioning [16] over system resources results in inferior performance or failures on missions.

Challenge 2 - The Tangled Requirements Problem: In the validation of requirements after component composition by CBSE, QoS tuning is problematic in that obtaining an optimal solution from hundreds of requirements is arduous and tedious. Composition may require effort on many infeasible designs to satisfy all requirements. For dynamic evaluation, such wasted effort is eliminated by evaluating functional and QoS requirements interchangeably. Such tangled requirements, however, complicate the evaluation and suppresses the development pace, because CBSE treats components as composition units and primarily concentrates on functional requirements.

Challenge 3 - The Abundant Alternatives Problem: One of the common objectives of CBSE and SPLE is to increase the diversity among software products. Despite abundant variable alternatives generated, many of them are inadequate regarding satisfying their requirements. Infeasible design alternatives should be avoided.

The three challenges are derived from the fact that functional and QoS requirements are equivalently important for DRE system construction. To address these obstacles, this paper introduces a separation of concerns analysis and design paradigm in the vision of the UniFrame project [11], a standardized framework for knowledge-based component composition, as part of a QoS-driven product line framework (QoSPL). QoSPL expresses a DRE system as a collection of QoS systemic paths [16]. To perform a service obeying its functional requirements (e.g., draw a virtual object on a monitor) at the service level, a sequence of components (i.e., a functional path [16]) collaborates with each other in a specific order. Each component carries out a specific task (e.g., rendering) and the combination of these tasks fulfills the overall requirements. Regarding QoS requirements, a QoS systemic path quantitatively describes *how well* the corresponding functional path can be satisfied. Such a path applies QoS formulae (from the specification document) for analyzing the resulting QoS behavior. Given specific component dependencies and composition rules, QoS systemic paths for each QoS property are constructed and analyzed using a specification language approach in a way that the synergy of commonality and QoS

1 Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294, USA, {liush, bryant, gray}@cis.uab.edu

2 Department of Computer and Information Science, Indiana University Purdue University Indianapolis, Indianapolis, IN 46202, USA, {rraje, tuceryan, aolson}@cs.iupui.edu

3 Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA, maugusto@nps.navy.mil

satisfaction analyses for constructing a product line is fulfilled. Suitable QoS systemic paths are accumulated, rendered and assured by a modeling approach as a behavioral view of the DRE software architecture. The entire product line can be constructed in the same way by selecting different combinations of QoS systemic paths. Because this paradigm only pertains to quantitative QoS (e.g., deadlines) described by the QoS formulae and in the representations of QoS systemic paths, non-quantitative QoS (e.g., security) is not considered in this paper.

The contributions of this paper are twofold: (a) The paper presents an approach that solves the above stated obstacles by utilizing QoS formulae indicated in the specification documentation for more precise and less subjective QoS measurement (Challenge 1), by reducing the amount of workload using a separation of concerns evaluation (Challenge 2) and by eliminating infeasible design alternatives in compliance with QoS constraints specified in QoS requirements (Challenge 3), and (b) It describes a specification language and modeling approaches to analyze commonality, variability, and reusability at the component and service levels, which facilitate finer-grained analytical results.

The paper is organized as follows: the next section introduces the formalisms applied in the paper; Section 3 presents QoSPL and a case study of Mobile Augmented Reality Systems (MARS) [12]; Section 4 summarizes the related work; and Section 5 concludes the paper.

2. BACKGROUND

Two-Level Grammar++ (TLG++) [2] is an object-oriented formal specification language that consists of two Context-Free Grammars (CFGs) defining syntax and semantics, respectively. TLG++ has been used for design space exploration and QoS requirements analyses in UniFrame [8]. In this paper, TLG++ is used for syntactically and semantically defining QoS properties with respect to corresponding QoS systemic paths and for analyzing commonality and variability of a product line.

Timed Colored Petri Nets (TCPNs) [4] are formalisms beneficial in modeling concurrent and asynchronous distributed systems with ancillary notations for time and user-defined data types and values. A TCPN is graphically represented by a Petri Net graph, which statically expresses the interrelationships between states of a modeling system by the abstractions of tokens, places, arcs, types, and transitions. The dynamic and executable properties of the TCPN are described by variables and binding, enabling, occurrence, and occurrence sequences and steps on the Petri Net graph. The reachability tree [4] is the execution result derived from the static and dynamic properties of the Petri Net graph. The execution orders of the reachability tree can be expressed by sequences of markings, which are distributions of tokens over the places of a TCPN.

The QoS systemic paths expressed in the first CFG of a TLG++ class are manually depicted in a Petri Net graph. The QoS semantics described in the second CFG of the TLG++ class are manually mapped to the dynamic properties of the Petri Net graph. Besides the synergetic advantage of the commonality and QoS satisfaction analyses, TLG++, applied in the UniFrame project reduces the possible accidental complexity. TCPN enriches TLG++ by introducing asynchronous, concurrent, and time properties to design and analyze a DRE product line. A MARS example is presented in the next section to describe how to apply these two formalisms and why they are suitable and beneficial to construct a DRE product line.

3. QoSPL

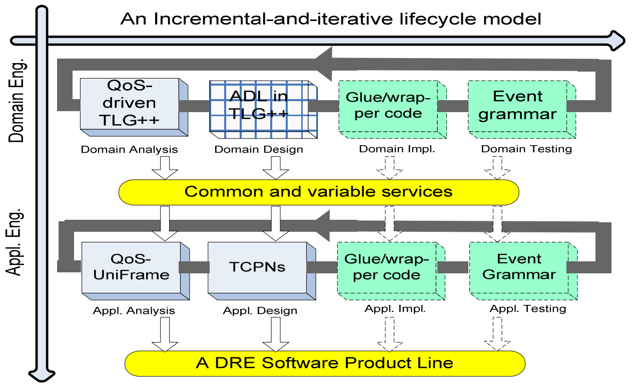

Figure 1. The overview of QoSPL.

QoSPL for DRE systems complies with the purposes and concepts of SPLE and software architectures [15] and introduces a QoS perspective to analyze and design a DRE product line. Figure 1 shows the overview of QoSPL based on the incremental-and-iterative lifecycle model. The objectives in the domain engineering process are to analyze common and variable requirements, to design prescribed reference architecture [10] for its product line, to implement a set of reusable core assets and to verify and validate the core assets. TLG++ is employed in the analysis workflow and is used as an Architecture Description Language (ADL) [15] in the design workflow (the grid box). The application engineering process aims at reusing the core assets of each workflow in the domain engineering process to exploit the artifacts of variable features of each workflow on individual applications. QoS-UniFrame [7], a TCPN-based modeling tool, is applied in the analysis and design workflows in this process. The implementation and testing workflows of both processes (dashed boxes) are out of the scope of this paper.

This section presents QoSPL using a case study of a Battlefield Training System (BTS) from the domain of mobile augmented reality. A Mobile Augmented Reality System (MARS) is a DRE system concentrated on enriching the user environment by merging real and virtual objects. Generally, a MARS consists of six subsystems: computation, presentation, tracking and registration, environmental model, interaction, and wireless communication [12] (as shown in Figure 2). The BTS is an outdoor MARS for training soldiers to conduct military operations. The

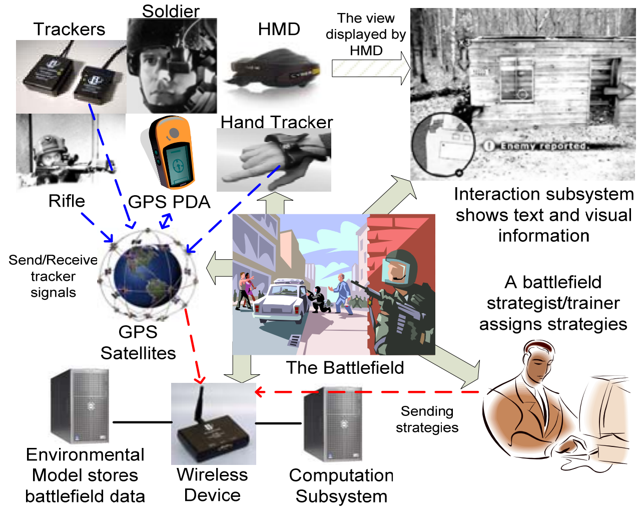
Trackers
Soldier
HMD
The view displayed by HMD
Enemy reported
Rifle
GPS PDA
Hand Tracker
Interaction subsystem shows text and visual information
Send/Receive tracker signals
GPS Satellites
A battlefield strategist/trainer assigns strategies
The Battlefield
Sending strategies
Environmental Model stores battlefield data
Wireless Device
Computation Subsystem

Figure 2. The overview of the BTS example.

hardware requirements of the BTS are a video see-through Head-Mounted Display (HMD), a headphone, position sensors (PS), orientation sensors (OS), mobile devices, and a specialized rifle comprising both types of sensors. Uni-Frame, in its knowledge base, stores functional and QoS requirements of components for rendering processing, speech processing, speech recognition, speech synthesis, text processing (TxP), tracking processing (TkP), presentation (Present), wireless communication (WC), and environmental model (EM) that stores the geometrical and hierarchical 3D information. The following paragraph describes a scenario for a BTS example.

A soldier is to rescue a virtual hostage in a battlefield. The position and orientation sensors on his body send back the 3 Degrees Of Freedom (3DOF) data to the tracking subsystem every half second via wireless communication. As the soldier is standing on specific positions with specific orientations in some buildings derived from a predefined tactical scenario stored in the computation subsystem, his HMD displays the enemies registered by the tracking system, computed by the presentation subsystem and rendered by the interaction subsystem. The soldier communicates with the command center via his headphone. The information of the soldier's current position is displayed on the HMD by text.

The scenario is applied to the domain engineering and application engineering processes respectively using TLG++ and QoS-UniFrame in the next two subsections.

3.1 Domain Engineering

This section describes the commonality and QoS satisfaction analyses for the quantitative services of the BTS example using TLG++ in the analysis workflow.

Figure 3 defines the `TextDeadlineFromClient` class (i.e., service) in TLG++ for commonality and QoS satisfaction analyses. Lines 2 to 9 constitute the first CFG that describes all possible QoS systemic paths of `TextDeadlineFromClient`. Line 2 shows the types of

```
1  class TextDeadlineFromClient.
2    Syntax :: Sensor WC TkP.
3    Sensor :: OS PS ; PS ; OS.
4    PS :: ps1 ; ps2 ; ps3.
5    OS :: os1 ; os2 ; os3 ; os4.
6    WC :: wc1 ; wc2.
7    TkP :: trackProcessing.
8    PreCondition, PostCondition :: Boolean.
9    Sum :: Double.
10   semantics of sendPositionFromClient :
11     PreCondition := semantics of queryComponent with OS PS
           WC and TkP, //please refer to [8] for the semantics
12     if PreCondition then Sum := semantics of sumOfMTAT
           with OS PS WC and TkP,
13     else ErrorMessage, end if,
14     Double semantics of sumOfMTAT with OS PS WC and
          TkP :
15         return OS semantics of getMTAT + …….
16     PostCondition := semantics of queryPattern with Sum.  …
17  end class. //please refer to [8] for the semantics
18  class OS extends Id.
19    token : "{letter}({letter}|{digit})*".
20    semantics of getMTAT : ……//please refer to [8] for how to
          //access its value specified in TLG++
21  end class.
22  class TextDeadlineFromServer.
23   Syntax :: GetEnvInfo TxP TkP Present Interact WC Display.
24   GetEnvInfo :: EM TkP.
25   FontResult :: Font.
26   Character :: Integer.
27   PreCondition, PostCondition :: Boolean.
28   Sum :: Double.
29   semantics of sendTextToClient :
30       Sum := EM semantics of getMTAT with getEnvInfo +
31       TkP semantics of getMTAT with getTrackingResult +
32       TxP semantics of getMTAT with decideTextContents +
33       TkP semantics of getMTAT with registerTextResult +
34       Present semantics of getMTAT with glutBitmapCharacter
            with FontResult and Charater +
35       Interact semantics of getMTAT with
            manageDisplayPosition +
36       WC semantics of getMTAT with sendText +
37       Display semantics of getMTAT with display.    ……
38 end class.
```

Figure 3. The commonality and QoS analyses in TLG++.

components involved in the text rendering task. The path starts from obtaining tracking results from the position and orientation sensors (Sensor) of a soldier. The wireless communication (WC) transmits the results to the `trackProcessing` (TkP) component. Three combinations in line 3, divided by semicolons as the counterpart of the meta-symbol "|" in Extended Backus-Naur Form, show that the `MTAT` (Mean Turn Around Time) values for tracking position and orientation are affected by the combinations. Because there may be more than one appropriate component for functionality-determined tasks, lines 4 to 6 show the suitable different components for position sensors (PS), orientation sensors (OS), and wireless communication (WC). After parsing the first CFG, 38 (4*3*2*1 + 3*2*1 + 4*2*1) parse trees, as shown in Figure 4, will be generated following the gray arrows (i.e., six horizontal and a vertical arrows). Infeasible paths of the composed service can be eliminated in compliance with the pre- and post-conditions for composition (e.g., QoS constraints), as shown in lines 11 and 16. The semantics of pre- and post-conditions can be obtained in [8]. The `OS` class describes legitimate instance identities in line 19 and the `getMTAT` method, is invoked in `TextDeadlineFromClient` and returns a value defined by MARS experts and stored in the knowledge base. The `TextDeadlineFromServer` class accumulates `MTAT` values by assigning specific functionalities (e.g., `getEnvInfo`) in the `getMTAT` method of each component in line 23. In the case study, all components are deployed on a Local Area Network (LAN), and the communication time between components is negligible. The QoS evaluation for MTAT at the service level is therefore the sum of all individual components along the `TextDeadlineFromServer` QoS systemic path. Lines 30 to 31 access the environmental and tracking information. The text contents are computed by the text processing component (`textProcessing`) in line 32. The registration for the new text result is updated in line 33. Consequently, the `renderingText` component invokes its function `glutBitmapCharacter` (an OpenGL utility function), to depict the text. The text outcome is managed by the interaction subsystem, transmitted by the wireless communication subsystem and displayed on the HMD.

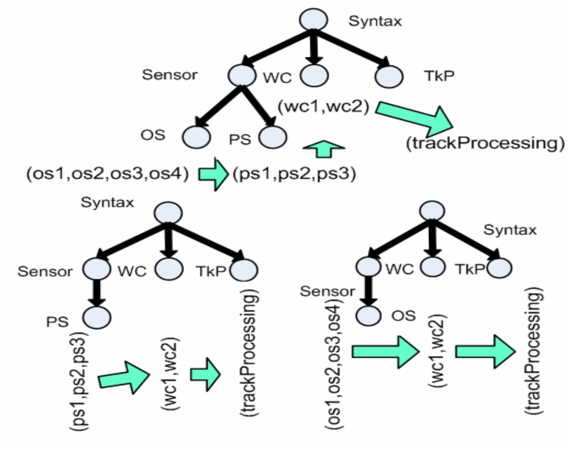

Figure 4. The parse tree of `TextDeadlineFromClient`.

The commonality, variability and reusability analyses in this workflow can be observed in Figure 4. `WC` and `TkP` are mandatory/reusable component types (shown as intermediate nodes in a parse tree); because these types exist in all three parse trees and have no further children. `Sensor` is mandatory and an "OR" (i.e., more-of) component type, because it is present in all parse trees and each tree has three different intermediate child types, which relates to line 3 in Figure 3. `trackProcessing` is a mandatory component (shown as a leaf in a parse tree). The components of the `OS`, `PS`, and `WS` component types are alternative atomic features. Because each component appears once in each parse tree in Figure 4, there is no optional (i.e., one or none) component in the BTS example. The commonality and reusability rate of each component and each QoS system path within the family will be decided based on the following intuition: because more satisfactory QoS systemic paths have higher probabilities to be selected for the product line construction, common and reusable components are most likely found in the paths with higher analysis results computed from their evaluation formulae.

Table 1. The symbol table of `TextDeadlineFromClient`.

Identity	os1	os2	os3	os4	ps1
Type	OS	OS	OS	OS	PS
MTAT (ms)	0.2	0.3	0.45	0.2	0.25
Identity	ps2	ps3	wc1	wc2	trackProcessing
Type	PS	PS	WC	WC	TkP
MTAT (ms)	0.15	0.2	0.5	0.4	0.9

Table 1 is a partial symbol table generated after parsing `TextDeadlineFromClient` where the values are defined by BTS experts and accessed from the knowledge base. Derived from Table 1, `os1`, `os4`, `ps2`, `wc2`, and `trackProcessing` are the most commonly used components at the component level. At the service level, [`ps2`, `wc2`, `trackProcessing`] is the most appropriate QoS systemic path within the family, because its cumulative `MTAT` value is the shortest among all paths (1.45 *ms*). Without the orientation information of a soldier, however, it is impossible to depict a virtual object on the HMD correctly. Consequently, [`os1`, `ps2`, `wc2`, `trackProcessing`] and [`os4`, `ps2`, `wc2`, `trackProcessing`] are the most promising QoS systemic paths (1.65 *ms*) for `TextDeadlineFromClient`. The TLG++ classes of `ThreeDimDeadlineFromClient`, `ThreeDimDeadlineFromServer`, and `SpeechDeadline` are omitted due to space considerations.

3.2 Application Engineering

After the commonality and QoS satisfaction analyses, QoSPL obtains quantitative (i.e., *how well*) results of each service as shown in the middle round box in Figure 1. In application engineering, QoSPL utilizes QoS-UniFrame [7] to construct a set of DRE systems. QoS-UniFrame is a TCPN-based modeling toolkit implemented in the GME[4] (Generic Modeling Environment), a meta-configurable modeling environment. Initially, QoS systemic paths (i.e., the outcomes of the domain analysis workflow), the reference architecture, and sorted QoS requirements are acquired from the knowledge base. The behavioral view of a DRE system that comprises a collection of satisfactory and necessary QoS systemic paths is depicted by the Petri Net graph of a TCPN in QoS-UniFrame. QoS-UniFrame also generates the reachability tree of the Petri Net Graph. Such a tree introduces a sequence of markings, which individually represents a state and as a whole describes a valid

4. http://www.isis.vanderbilt.edu/Projects/gme/

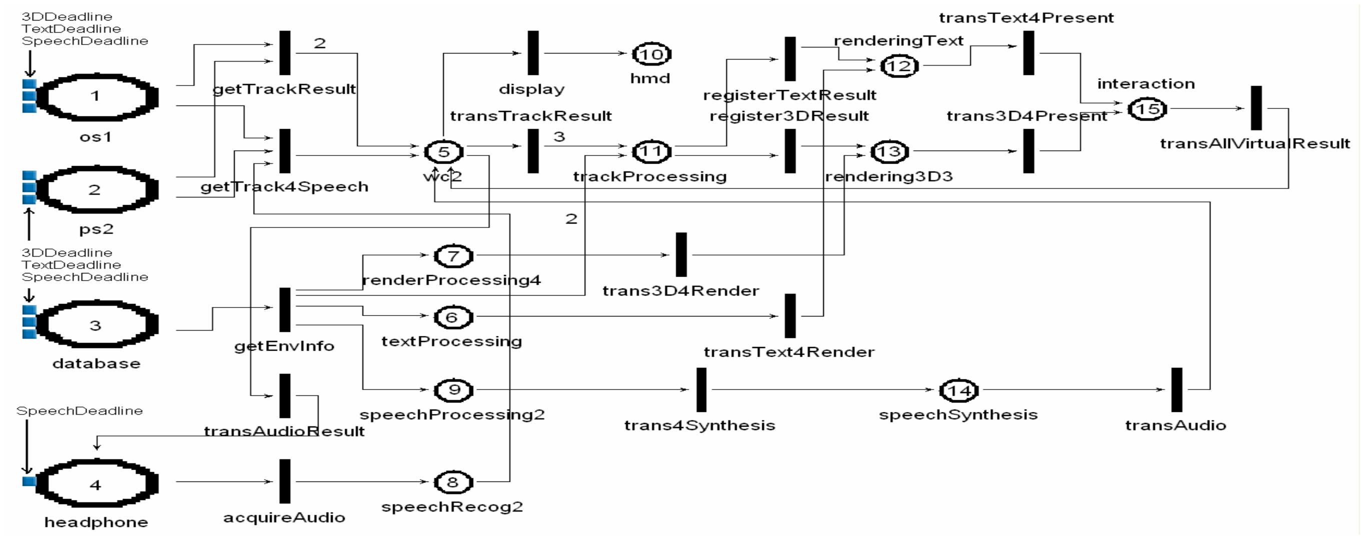

Figure 5. The behavioral view of the BTS example.

execution order under given static and dynamic properties (i.e., enabling and firing rules in transitions and/or arcs [4]) and constraints. As system-level QoS predicates are inserted in specific transitions, and the reachability tree is generated, the modeling results validate the QoS requirements at component, service, and system levels in such a behavioral view of software architecture.

Figure 5 shows the behavioral view of the BTS example using a Petri Net graph. `TextDeadline` (i.e., `TextDeadlineFromClient` and `TextDeadline-FromServer`), `ThreeDimDeadline` (i.e., `ThreeDimDeadlineFromClient` and `ThreeDim-DeadlineFromServer`), and `SpeechDeadline` are three required QoS systemic paths described in the BTS scenario. In Figure 5, circles and ellipses, representing places in TCPNs, are the necessary hardware and software components; black bars, as transitions in TCPNs, are functionalities provided in the components; the small boxes within `os1`, `ps2`, `database`, and `headphone` are the tokens for representing the initial marking of the Petri Net graph; and the number 2 below `trackProcessing` is the weight assigned to the arc from `getEnvInfo` to `trackProcessing` for the purpose of virtual object registration. Because TCPNs are used for modeling DRE systems including synchronous and asynchronous characteristics, a timer and predicates are embedded in the transitions of Figure 5 to control the token flows.

Figure 6 shows a sequence of markings resulting from QoS-UniFrame that expresses a valid execution order considering three deadlines. Row Marking contains the place identities numbered in Figure 5. For simplicity, rows M0 to M8 only express the number of tokens (instead of the identities) in each place after specific transition(s). M0 is the initial marking as depicted in Figure 5. Timers and predicates within transitions decide the following markings to be generated along with arcs and weights. For example, the predicate in `transTrackResult` specifies that if (a)

Marking	(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
M0	(3,3,3,1,0,0,0,0,0,0,0,0,0,0,0)
M1	(1,1,3,0,2,0,0,1,0,0,0,0,0,0,0)
M2	(0,0,3,0,3,0,0,0,0,0,0,0,0,0,0)
M3	(0,0,3,0,0,0,0,0,0,0,3,0,0,0,0)
M4	(0,0,0,0,0,1,1,0,1,0,2,0,0,0,0)
M5	(0,0,0,0,0,0,0,0,0,0,0,2,1,1,0)
M6	(0,0,0,0,1,0,0,0,0,0,0,0,0,0,2)
M7	(0,0,0,0,3,0,0,0,0,0,0,0,0,0,0)
M8	(0,0,0,1,0,0,0,0,0,2,0,0,0,0,0)

Figure 6. The markings of the BTS example.

the tokens of `TextDeadline`, `ThreeDimDeadline`, and `SpeechDeadline` are residing in `wc2`, and (b) each currently evaluated deadline is fewer than its QoS constraint, `transTrackResult` fires a token to `track-Processing` at the beginning of the next periodic firing. To evaluate system-level QoS requirements (e.g., all deadlines should not exceed 10 seconds), the corresponding QoS evaluation formulae can be embedded into either each transition (i.e., dynamic evaluation) or into the transitions right before the interaction subsystem (i.e., evaluation after composition). If all QoS requirements are satisfied, the Petri Net graph (Figure 5) is the behavioral view and a member of the product line of the example.

To construct other members of the product line of the BTS example, different combinations of QoS systemic paths can be instantiated in the TCPN model. For example, [`os4`, `ps2`, `wc2`, `trackProcessing`] can be selected as an alternative of `TextDeadlineFromClient` and composed with the other QoS systemic paths depicted in Figure 5. As long as all QoS requirements are fulfilled at the component, service, and system levels, the second member of the product line is designed. Please note that the QoS evaluation formulae and predicates need no revision as long as all requirements remain the same. Other product line members can be constructed using the same approach as shown in the bottom round box in Figure 1.

A BTS product line is constructed by QoSPL that shares common components and services with their unique features and satisfies both functional and QoS requirements at component, service, and system levels. Following the incremental-and-iterative lifecycle, the common components and/or services of the BTS product line may reduce the development cost and time to market.

4. RELATED WORK

Feature-Oriented Reuse Method (FORM) is derived from Feature-Oriented Domain Analysis (FODA) [6]. Non-functional features are used for decomposing functional features. Such a framework is not sufficient to analyze and manage numerous DRE QoS requirements (Challenge 1). *KorbA* [1] separates abstraction, specificity (SPLE) and composition (CBSE) concerns strictly. The specification and realization of each sub-component are activated interchangeably (Challenge 2). It provides quality assurance and quality assessment techniques by integrating inspections (Challenge 1) and quantitative analysis of UML models. *Quality-Driven Architecture Design and Analysis (QADA)* [9] processes quality analysis and architecture design phases interchangeably (Challenge 2). Scenario-based (i.e., consensus or questionnaire (Challenge 1)) analysis is applied to evaluate the conceptual architecture alternatives. The architecture design phase iteratively consults the feedback from the quality analysis and then designs the architecture accordingly. *Mini-Middleware* [5] treats the DRE middleware as the assemblage of common white box components that provides commonly used features. To construct a middleware product line, the DRE middleware is specialized and optimized by shrinking its specific functionalities based on stringent QoS properties.

QoSPL not only comprises the FODA, SPLE, CBSE, and quality-driven characteristics, but also solves the three obstacles existing in FORM, KorbA, and QADA. The core difference between QoSPL and Mini-Middleware is QoSPL utilizes two formalisms to facilitate high-confidence DRE system construction.

5. CONCLUSION

This paper presents a novel design and analysis paradigm for developing a QoS-oriented product line in the domain of DRE systems. For domain engineering, TLG++ analyzes the common and variable features as well as QoS requirements at a finer-grained abstraction level. For application engineering, QoS-UniFrame concurrently analyzes and designs DRE systems by collecting satisfactory QoS systemic paths out of each family. After assurance by the TCPN reachability tree, the output of QoS-UniFrame is the behavioral view of the DRE software architecture. QoSPL solves the QoS-sensitive, component composition, and abundant alternatives problems that plague many CBSE and SPLE. QoS analyses for shared resources have been tackled by a system level statistical and stochastic approach [7]. QoS-UniFrame promises to analyze finer-grained QoS at the component and service levels.

REFERENCES

[1] C. Atkinson, et al. *Component-based Product Line Engineering with the UML*. Addison-Wesley, 2001.

[2] B. R. Bryant, B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification. *Proc. 35th Hawaii Intl. Conf. System Sciences*, pp 19-26, 2002.

[3] G. Heineman, W. T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[4] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Volume 1, Basic Concepts*. Springer-Verlag, 1997.

[5] A. Krishna, et al. Model-driven Middleware Specialization Techniques for Software Product Line Architecture in Distributed Real-time and Embedded Systems. *Proc. MoDELS 2005 Workshop MDD for Software Product-Lines: Fact or Fiction?*, 2005.

[6] K. Kang, et al. FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. *Annuals of Software Engineering*, 5: 143-168, 1998.

[7] S.-H. Liu, et al. QoS-UniFrame: A Petri Net-based Modeling Approach to Assure QoS Requirements of Distributed Real-time and Embedded Systems. *Proc. 12th Intl. Conf. Eng. of Computer Based Systems*, pp 202-209, 2005.

[8] S.-H. Liu, et al. Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach. *Proc. 17th Intl. Conf. Software Eng. and Knowledge Eng.*, pp 731-734, 2005.

[9] M. Matinlassi, et al. Quality-Driven Architecture Design and Quality Analysis Method: a Revolutionary Initiation Approach to Product Line Architecture. Tech. Report, *VTT Pub.: 456*, VTT Electronics, 2002.

[10] K. Pohl, G. Böckle, F. van der Linden. *Software Product Line Engineering: Foundations, Pinciples, and Techniques*. Springer-Verlag, 2005.

[11] R. R. Raje, et al. A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components. *Concurrency and Computation: Practice and Experience*, 14(12): 1009-1034, 2002.

[12] T. Reicher. *A Framework for Dynamically Adaptable Augmented Reality Systems*. Doctoral Dissertation, Technical University of Munich, 2004.

[13] D. C. Schmidt. Model Driven Development for Distributed Real-time and Embedded Systems (Keynote presentation), *8th Intl. Conf. Model Driven Eng. Languages and System*, 2005.

[14] D. C. Schmidt. R&D Advances in Middleware for Distributed Real-time and Embedded Systems. *Communications of the ACM*, 45(12):43-48, 2002.

[15] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[16] N. Wang, et al. QoS-enabled Middleware, *Middleware for Communications*. ed. Q. H. Mahmoud. John Wiley and Sons, 2003.