A Component-Based Approach for Constructing High-Confidence Distributed Real-Time and Embedded Systems

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

[1] University of Alabama at Birmingham
{liush, bryant, gray}@cis.uab.edu
[2] Naval Postgraduate School
auguston@cs.nps.navy.mil
[3] Indiana University Purdue University Indianapolis
{rraje, tuceryan}@cs.iupui.edu

Abstract. In applying Component-Based Software Engineering (CBSE) techniques to the domain of Distributed Real-time and Embedded (DRE) Systems, there are five critical challenges: 1) discovery of relevant components and resources, 2) specification and modeling of components, 3) exploration and elimination of design assembly options, 4) automated generation of heterogeneous component bridges, and 5) validation of context-related embedded systems. To address these challenges, this paper introduces four core techniques to facilitate high-confidence DRE system construction from components: 1) A component and resource discovery technique promotes component searching based on rich and precise descriptions of components and context; 2) A timed colored Petri Net-based modeling toolkit enables design and analysis on DRE systems, as well as reduces unnecessary later work by eliminating infeasible design options; 3) A formal specification language describes all specifications consistently and automatically generates component bridges for seamless system integration; and 4) A grammar-based formalism specifies context behaviors and validates integrated systems using sufficient context-related test cases. The success of these ongoing techniques may not only accelerate the software development pace and reduce unnecessary development cost, but also facilitate high-confidence DRE system construction using different formalisms over the entire software life-cycle.

1 Introduction

As the complexity of Distributed Real-Time Embedded (DRE) software systems continues to increase [11], there is a need to facilitate the construction of such systems from reusable components that can be configured for the particular implementation being constructed. Component-Based Software Engineering (CBSE) [8] addresses this issue, providing the mechanism to leverage existing artifacts and resources rather than handcraft DRE systems from scratch, as is often observed in current practice. CBSE techniques, however, only partially fulfill the objective of software development. For example, to meet both longevity

F. Kordon and J. Sztipanovits (Eds.): Monterey Workshop 2005, LNCS 4322, pp. 225–247, 2007.

and changeability requirements demands continuous optimizations to the configuration of the component interactions and application logic. Furthermore, end users' demands on confidential, high quality, and time-to-market software products have not yet been completely addressed. Endeavoring to redeem the promises to both organizations and end users leads to five core challenges:

- **Discovery of relevant components and resources:** Amid a repository of available components, discovering relevant components is non-trivial. Particularly, DRE systems not only require stringent demands on functional correctness, but also non-functional (i.e., Quality of Service (QoS)) satisfaction. Such QoS demands, however, are not purely influenced by standalone systems composed by selected components - the context of the system under development also has a major influence. For example, there may be several implementations of the same functional component with different run-time features (e.g., battery consumption versus throughput). Additionally, two components may also have functional and/or QoS dependencies between each other that lead to mutual influence. A manual discovery process by embedded system engineers may be time consuming and error prone. An automated and unified resource discovery process based on component specifications, component dependencies, and context specifications may accelerate search speed as well as select the best component for specific DRE system construction.
- **Specification and modeling of components and their relevant properties:** As described in the first challenge, in order to discover an appropriate component, that component must be entered into the repository with an appropriate specification and model that can be detected by the discovery service. The specification indicates the relevant functional and non-functional (i.e., QoS) properties of the component and dependencies between components. The model indicates the domain the component belongs to in order to narrow and expedite the search to the appropriate application domain. A consistent and understandable specification syntax and semantics may reduce possible accidental complexity during DRE software development.
- **Exploration and elimination of design assembly:** Different challenges faced by embedded systems developers require effective design and fine tuning, crosscutting multiple layers of infrastructure and system logic. Such challenges result from diverse configuration possibilities, numerous appropriate component candidates for composition, and highly complex component dependencies in embedded systems. The combination of these challenges results in abundant design alternatives. Embedded systems engineers must be able to examine and deploy various design alternatives quickly and easily amid possible configurations, component candidates, and component dependencies.
- **Automatic generation of correct component bridges:** Some of the available components may be applicable only to specific technology platforms, requiring an approach that operates in a heterogeneous manner. The generation of component wrappers from formally specified behavioral characteristics may offer assistance in verifying the correctness of component

interactions that are more difficult or impossible to perform in handcrafted solutions. Furthermore, the specifications of component properties provides a capability to check if a set of components are assembled in a valid and legal manner. For example, adjustments made at one layer of the infrastructure may lead to unforeseen consequences at some other layer of the infrastructure, or may adversely affect application logic.

– **Validation of context-related embedded systems:** The factors of validation emerge from component specifications, component dependencies, component configurations and system logics, and heterogeneous component bridges. Such factors are, in fact, all context-related, and thus require the knowledge of different contexts and sufficient random test cases to cover all possible states under each given context. For a large number of test cases in different contexts, efficiently managing and reusing them to address the regression test problem are required. Cohesively tieing such test cases to the artifacts of the earlier software life-cycle to cover the quantitative and qualitative validation of context-related embedded systems are also imperative.

Although CBSE techniques lift the abstraction to a higher level and use interface description languages to specify the characteristics of composition units [8], these five accidental complexities still arise. This paper introduces four core techniques to facilitate high-confidence DRE system construction in the vision of the UniFrame project [20]: 1) A component and resource discovery technique promotes component searching based on multi-level descriptions of components and context; 2) A timed colored Petri Net-based modeling toolkit enables design and analysis of DRE systems and eliminates infeasible design options to avoid unnecessary later work; 3) A formal specification language consistently describes all specifications and automatically generates component bridges for seamless system integration; and 4) A grammar-based formalism specifies context behaviors and validates integrated systems using sufficient context-related test cases. The success of these progressive techniques may not only accelerate software development pace and reduce unnecessary development cost, but also enable high-confidence DRE system construction by formalizing the static and dynamic properties of a DRE system, and facilitating validation of the functional and QoS requirements of the system at component, service, and system levels.

The rest of the paper is organized as follows. Section 2 introduces UniFrame, the application domain in this paper, and its case study. In Section 3, four core techniques to address the five challenges are presented. Section 4 discusses the current CBSE techniques in the DRE domain. Section 5 concludes the paper and discusses future work stemming from current limitations.

2 Background

This section offers an overview of the UniFrame process and the domain of mobile augmented reality [22]. A case study, called the Battlefield Training System (BTS), is also described and applied to four techniques in the later sections.

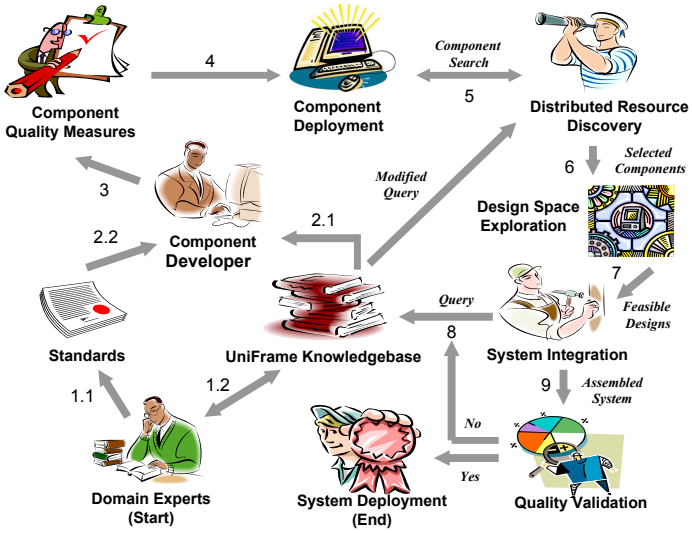

Fig. 1. The Overview of the UniFrame Process

2.1 UniFrame

UniFrame is a knowledge-based framework that offers techniques and tools for composing distributed systems from possibly heterogeneous components [21]. Figure 1 is an overview of the UniFrame process. The process starts from acquiring knowledge from domain experts. As shown in arrows 1.1 and 1.2, UniFrame engineers collaborate with domain experts to obtain sufficient backgrounds and knowledge on the application domain, components, component assemblies, component dependencies, and their functional and non-functional requirements and standards. Such information may be converted into an executable formal specification and stored in the knowledgebase [13]. Component quality measures concentrate on the evaluation of components according to their functional and non-functional requirements. Validated components are deployed to the distributed environment for future acquisition and assembly. Please note that the descriptions of the deployment environment context are stored in the knowledgebase for the searching procedure. The distributed resource discovery procedure searches and locates relevant components using the Unified Meta-component Model (UMM) [20]. QoS-UniFrame [16], as a design space exploration and elimination toolkit, utilizes timed colored Petri Nets [9] to model possible designs of DRE systems and analyzes the feasibility of design artifacts in compliance with their QoS requirements. During the system integration procedure, Two-Level Grammar (TLG) [4] formally and seamlessly bridges heterogenous components. Lastly, Attributed Event Grammar (AEG) [1] specifies possible event traces and provides a uniform method for automatically generating and executing test cases for quality validation purposes. This paper concentrates on the last four procedures (the right half of Figure 1) for high-confidence DRE system construction

during the entire software life-cycle. Such procedures may reduce possible accidental complexity and increase confidence of the software development.

2.2 Mobile Augmented Reality Systems

An augmented reality system [7] enriches the environment by merging real and virtual objects in a real environment. The real-time interactions and the registration (alignment) for real and virtual objects with each other are also required. The integrated concepts of augmented reality, mobile computing, wearable computing and ubiquitous computing systems enable research into Mobile Augmented Reality Systems (MARSs) [22].

Generally, a MARS consists of six subsystems: computation, presentation, tracking and registration, geographical model, interaction, and wireless communication [22]. The computation subsystem performs specific computational tasks for the application. The presentation (rendering) subsystem computes and depicts virtual multimedia objects. The geographical model stores the geometrical and detailed hierarchical 3D information of the environment where a demonstrator works. The interaction subsystem offers a user friendly interface that allows a demonstrator to conveniently input the data for processing as well as see the output generated by the presentation subsystem. Wireless communication provides the mobile communication between the subsystems. The tracking and registration subsystem tracks a user's (or an object's) position and orientation using trackers or sensors and registers virtual objects in compliance with the tracking results. The tracking data can be used both by the rendering and presentation subsystem to generate the 3D graphics properly aligned with the physical world, and also could be utilized by the computing subsystem for various tasks such as spatial queries for location-aware computational tasks.

There are numerous off-the-shelf or custom-built hardware solutions to tracking. These often consist of sensors that provide position (2 or 3 dimensions), orientation (2 or 3 Degrees of Freedom), or a combination. They utilize a variety of technologies (magnetic, ultrasound, vision-based, infra-red, wireless, ultra-wide-band, mechanical) to achieve the tracking and have various QoS properties such as resolution, accuracy, and range. For example, Global Positioning System (GPS) is a 2-dimensional position tracker that has a world-wide range, but has a resolution on the order of 1 meter. The Inertia-Cube^{TM} from Intersense Technologies[1], is a self-contained (sourceless) inertial orientation tracker that outputs three orientation angles and has a 1 degree yaw[2] accuracy and 0.25 degree accuracy in pitch and roll angles. In a mobile augmented reality system that covers a wide area, many such trackers may need to be deployed in various locations.

Many challenges exist in utilizing the trackers in such an environment that contains multiple trackers with different characteristics (heterogeneity) and spread over large spaces, with possible redundancies in their sensing modalities. The first challenge is the discovery by the tracked object of all the sensors available in a

[1] http://www.intersense.com/company/whatismotion.htm

[2] Yaw corresponds to how far the object is pointing away from its direction of travel due to rotation about its vertical axis.

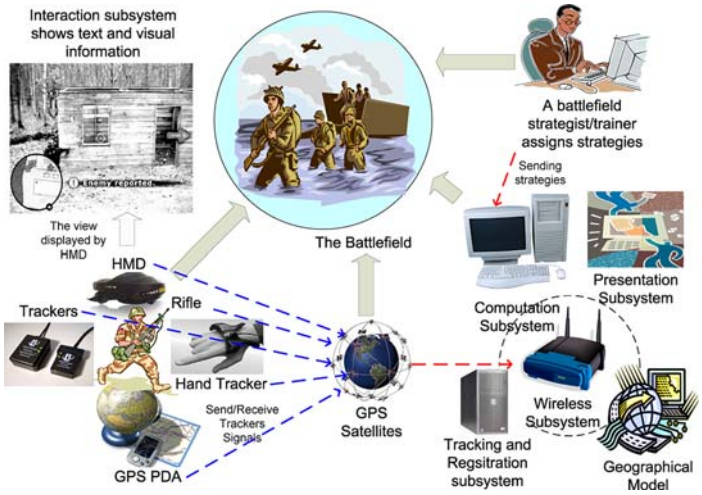

Fig. 2. The Battlefield Training System Example

given location. The next challenge is to select a subset to be utilized. Finally, the last challenge is to utilize the selected sensors to fuse the data and provide a single, high quality measurement of the pose (position and orientation) of the tracked object. UniFrame is used to accomplish the discovery and selection tasks.

In order to demonstrate the advantages of the UniFrame process over high-confidence component-based DRE system construction, a Battlefield Training System (BTS) example is introduced. Figure 2 shows an overview of the BTS example.

The following description is an example scenario for the BTS system. Imagine a soldier who is walking on the street to rescue a virtual hostage hidden in one of the buildings. The position and orientation sensors on his body send back the 6 Degrees of Freedom (6DOF) data to the tracking subsystem every half-second. When the soldier is in a certain position and is looking in a particular direction, the rendering subsystem will display enemy soldiers in certain 3D positions according to a training scenario generated by the computational subsystem. This rendering of enemy soldiers, therefore, is intimately tied to the position and orientation information coming from the tracking subsystem. The soldier has to shoot the enemies using a specialized rifle whose pose is also tracked. By computing the bullet trajectory, the system computes if the enemy is killed and updates the view of the soldier accordingly. The soldier can communicate with the command center via his headphone. The information of each building and the soldier's current position can be displayed on the Head Mounted Display (HMD) by text. Several movement, light, audio and temperature sensors will periodically send the physical conditions of the battlefield back to the computation subsystem. All of this simulation, computation, and rendering depends greatly on accurate tracking of the various objects such as the soldier, the HMD, and the rifle.

Several high-level functional and QoS requirements are required to establish a satisfactory BTS.

- **Functional Requirements**
  - (F1) A soldier should wear both position and orientation sensors on his body to obtain 6 Degrees of Freedom (6DOF) results: 3 for position and 3 for orientation. The soldier should also wear a hand tracker on his hand to sense the 6DOF of the hand.
  - (F2) Each rifle should contain position and orientation sensors to the 6DOF of an objective that the soldier may target.
  - (F3) Audio input and output devices should be provided to the soldier for communicating with his teammates.
  - (F4) An optical see-through Head Mounted Display (HMD) should provide the interaction subsystem that displays both text and visual objects. This could be a one-eye, monocular system that leaves the second eye unobstructed for other tasks.
  - (F5) The computation subsystem should compute the scenarios and strategies for training a soldier.
  - (F6) The geographical model should store all the necessary geographical and geometrical information of the battlefield. Such geographical information should be hierarchical in compliance with the three dimensions of the battlefield.
  - (F7) A GPS PDA (Personal Data Assistant) should provide the up-to-date geographical information of the battlefield obtained from the geographical model.
  - (F8) GPS satellites and relevant wireless communication devices should transfer tracking results and registered virtual objects between tracking and registration, geographical model, and computation subsystems.
  - (F9) A battlefield training system strategist/trainer should assign training strategies and adaptable scenarios to the computation subsystem.
- **Quality of Service Requirements**
  - (Q1) Each visual object should be displayed on the correct coordinates of the HMD. The coordinate inaccuracy should not exceed 5mm.
  - (Q2) Each visual object should be displayed and continuously updated on the HMD. The sampling frequency of each object should be at least 24Hz. The residual visual object that misses the hard deadline should not be displayed to confuse the soldier.
  - (Q3) Each text object should be displayed on the correct coordinates of the HMD. The coordinate inaccuracy should not exceed 5mm.
  - (Q4) Each text object should be displayed on the HMD in real-time. The sampling frequency of such an object should be at least 24Hz. The residual text object that misses the hard deadline should not be displayed to confuse the soldier.
  - (Q5) Each audio signal should be transmitted to the soldier in real-time. The sampling frequency of each signal should be at least 44Hz.

- (Q6) Each position sensor and orientation sensor on the soldier, the rifle, and the hand should send at least 120 6DOF sampling information back to the computation subsystem every second.
- (Q7) The interaction subsystem should display text and visual objects with a reasonable resolution (e.g., resolutions for position and orientation sensors should be respectively at least 0.75mm and 0.05 degrees).
- (Q8) The presentation/rendering subsystem should not provide obscure text and visual objects to the interaction subsystem. For example, 12-point (or more) proportional spaced bitmap fonts should be provided.
- (Q9) The geographical model should provide the geographical information in time upon the request from other subsystems. The query processing time of each geographical information should not exceed 0.01 second.

The listed functional and QoS requirements can be classified into three abstraction levels in UniFrame: component, service, and system. Functional or QoS requirements at the component level mean that a specific component correctly performs a functional task and satisfies *how well* it should perform as specified in the corresponding QoS requirements. F1, F2, F4, F6, Q7, and Q8 are examples of such requirements. To perform a service obeying its functional requirements at the service level, a sequence of components (i.e., a functional path [30]) 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 path quantitatively describes how well the corresponding functional path can be satisfied [30]. For F3, F7, F9, Q1, Q2, Q3, Q4, Q5, and Q6, each of which is achieved by comprising at least two components that interact with each other. F5 and Q9, however, can be mistakenly classified into the component level because of the brief descriptions. From the perspective of the component level, F5 and Q9 are realized by the computation or the geographical model subsystem. Such a classification, in fact, does not consider the entire picture of the BTS example. After obtaining the training strategies, the computation subsystem should compute and then assign specific tasks to other appropriate subsystems. Such tasks may request some collaborations among different subsystems. Additionally, because there may be more than one virtual object displayed on the interaction subsystem, the computation, tracking and registration, presentation, and wireless subsystems frequently interact with the geographical model subsystem to access the geographical results. Therefore, F5 and Q9 are regarded as system level requirements.

3 The UniFrame Approach

To tackle accidental complexities as mentioned in Section 1, UniFrame offers four kernel techniques in the requirements, analysis, design, implementation, and testing workflows.

3.1 Distributed Discovery of Components

As indicated previously, the underlying model for the UniFrame approach is the Unified Meta-component Model (UMM) [20]. The UMM has three parts: a) Components, b) Service and associated guarantees, and c) Infrastructure [19]. Components in UniFrame are units of independent development and deployment and offer specific services with associated guarantees about the QoS. The infrastructure provides the environment for deploying such independently developed components and discovering them via the UniFrame Resource Discovery System (URDS), which is a pro-active and hierarchical discovery service [23].

UniFrame Resource Discovery System (URDS). URDS consists of three levels: a) registration level, b) pro-active search level, and c) user interaction and administration level. The registration level is realized by active registries, which are enhanced versions of the basic publication mechanisms provided by different component deployment environments (e.g., the built-in registry in Java-RMI[3]). The enhancement is in the form of an ability for these basic mechanisms to actively listen and communicate with the head-hunters (described shortly). Component developers are required to use the UniFrame knowledgebase (as indicated in Figure 1) and create, in addition to the implementation of the components, comprehensive specifications called the UMM-Specifications. An example of such a specification is shown in the next section. Once the component and its associated specification is ready, both of these are published with the corresponding local active registry and deployed on the network.

The pro-active search level is implemented by head-hunters. These are specialized components who are entrusted with the task of pro-actively gathering component specifications from various active registries. Head-hunters store these specifications in their local store, called a meta-repository. Head-hunters, in addition to gathering specifications, carry out the task of matching specifications stored in their meta-repositories with incoming queries. It is quite conceivable that any single head-hunter may not contain all the specifications that are deployed over a network, and hence, head-hunters may collaborate with one another to cover a much larger search space. Various techniques for the collaboration between head-hunters have been experimented with. These include random, long-term, short-term, and profile-based. Results of these experiments [24] demonstrate that such a collaboration allows a selective search, as compared to an exhaustive search (which may be costly in a large setup), without substantially sacrificing the quality of the selected components.

The top level of URDS is achieved by the Internet Component Broker, which is made up of the Domain Security Manager, Query Manager, Link Manager, and Adapter Manager. The Internet Component Broker is responsible for authenticating head-hunters and active registries (via the Domain Security Manager), receiving incoming queries and returning results (via the Query Manager) to the system integrator, for linking different Internet Component Brokers (via the

[3] Remote Method Invocation - http://java.sun.com/products/jdk/rmi

Link Manager), and providing adapter components for bridging the technological heterogeneities (via the Adapter Manager).

UMM Specification and Discovery of Components. The UMM specification of components is in accordance with the concept of multi-level specification [2]. The UMM specification of a component, in addition to its name, type and informal description, consists of computational attributes, cooperation attributes, auxiliary attributes, QoS attributes, and deployment attributes.

The computational attributes describe the functional characteristics of a component. These include inherent attributes, which contain the book keeping information (such as the ID and version) of that component and functional attributes. The functional attributes contain the syntactical, semantical, and the synchronization contracts, along with a few additional fields such as technology of implementation and the algorithm (if any) used. The cooperation attribute indicates possible collaborators of a component. The auxiliary attributes provide information about special features that may be incorporated in a component such as security. The QoS attributes, which are critical in the case of DRE systems such as MARS, contain information about the QoS parameters (e.g., latency), their values (or a range), associated costs and the levels of quality that a component provides. The deployment attributes indicate the execution environment needed for that component and the effects of the environment on the QoS characteristics of the component. For example, the partial UMM specification of an *IS-PCTracker* that can be used in the MARS environment for providing the position and orientation information (6DOF) is shown below (an example of a complete UMM specification is found in [19]):

```
Component Name: IS-PCTracker Domain Name: Distributed Tracking
Informal Description: Provides the position and orientation
information.

Computational Attributes
   Inherent Attributes:
       Id: cs.iupui.edu/ISPCTracker;
       ...
       Validity: 12/1/07
       Registration: pegasus.cs.iupui.edu/HH1
       Technology: CORBA

   Functional Attributes:
       Functional Description: Provides the position and
                               orientation of a tracked object.
       Algorithm: Kalman Filter;
       Complexity: O(n^6)

       Syntactical Contract:
             Vector getPosition();
             Vector getOrientation();
```

```
        Semantic Contract:
              Pre-condition: {calibrated (PCTracker)== true}
              Post-condition: {sizeof (posVector) == 3) &&
                              sizeof (orientationVector) == 3}
        Synchronization Contract:
              Policy: Mutual Exclusion
              Implementation Mechanism: semaphore
        ....

Quality of Service Attributes
        QoS Metrics: tracking_volume, resolution_pos,
                     resolution_orientation, accuracy_pos,
                     accuracy_pitch, accuracy_yaw,
                     accuracy_roll, sampling_freq

                     tracking_volume: 2mx2mx3m
                     resolution_pos: 0.75mm
                     resolution_orientation: 0.05 degrees
                     accuracy_pos: 2-3mm
                     accuracy_pitch: 0.25 degrees
                     accuracy_yaw: 0.5 degrees
                     accuracy_roll: 0.25 degrees
                     sampling_freq: 100-130 Hz
         ...
```

The above specification indicates various important factors: a) it is comprehensive and embodies the multi-level specification concepts, b) it places an emphasis on functional as well as non-functional (QoS) features of a component, and c) it is consistent with the concepts of service-oriented approaches for developing DRE systems. Due to its comprehensive nature and multi-levels, the UMM specification of a component (such as an *IS-PCTracker*) allows complicated matching techniques during the discovery process of the URDS for appropriate components. For example, a system integrator may specify a subset of typical attributes (e.g., the type, the syntactical attributes, pre- and post-conditions associated with the interface, and QoS parameters with specific values) for an *IS-PCTracker*. Once this query is received by the Query Manager, it will pass it on to a subset of the head-hunters to search for appropriate components. URDS uses multi-level matching, i.e., depending upon the level, a different technique is used to match the corresponding part of the incoming query with the specifications stored in the local meta-repository. This approach is an enhancement of the one discussed in [32]. For example, matchings such as type and technology use keyword match, syntactical matching uses type relations, semantical matching uses theorem provers, synchronization matching uses keywords and temporal logic, and QoS matching uses numerical relationships. Thus, the multi-level matching is more comprehensive than simple attribute-based matching. Also,

different head-hunters may use different algorithms for discovering components that match the given query from their local meta-repository. Once appropriate components are discovered, they are presented back to the system integrator who can select an appropriate one for his/her current needs.

3.2 Design Space Exploration and Elimination

UniFrame advocates the principles of CBSE [8], design by contract[4], and multi-level contracts [2]. Such principles facilitate URDS to discover relevant components from the repository in compliance with their functional and QoS requirements. The complexity and magnitude of a design space increases exponentially as more appropriate components are found for a distributed embedded system. QoS-UniFrame [16] is a two-level modeling toolkit for designing and analyzing distributed embedded systems. Such a toolkit explores and eliminates the design space of a DRE system and assures its QoS requirements. At the first level, QoS-UniFrame performs design space exploration and elimination using the formalism of timed colored Petri Nets [9]. A Petri Net graph visually achieves design space exploration by depicting all relevant components (places in a Petri Net graph) and design decisions (transitions in a Petri Net graph). Design space elimination is accomplished by a reachability tree construction of the Petri Net graph. Such a reachability tree comprises a number of sequences of states (i.e., markings) that represent selected component status and dynamic behaviors regarding QoS at given points of execution. A QoS-UniFrame interpreter implements the tree construction that obeys the formalisms of timed colored Petri Nets and the static and dynamic properties embedded in the Petri Net graph.

Besides the formalisms, an aspect-oriented programming approach using AspectJ [10] is utilized to insert (i.e., weave) statements into the interpreter for analyzing and/or asserting static or strict QoS requirements regarding components, execution paths, and the system [16]. If the inserted statements are not fulfilled, QoS-UniFrame stops constructing new nodes in the reachability tree whereas all the leaves generated are the design space that satisfies static and strict QoS requirements. Because dynamic QoS information accordingly relates to the deployment environment, a statistical and stochastic approach is exploited at the second level [16]. The previous state and observations of components can be accessed from the knowledgebase for the evaluation of dynamic QoS requirements. QoS-UniFrame utilizes a meta-programmable approach, called PPCEA (Programmable Parameter Control for Evolutionary Algorithms) [14], that prunes off less probable design alternatives by means of statistic and stochastic evolutionary algorithms.

Figure 3 (a) shows a partial high-level design of the BTS example represented by a Petri Net graph using QoS-UniFrame. It describes three execution paths that perform rendering text on the Head Mounted Display (HMD), rendering a three dimensional graph on the HMD, and speech processing. White circles (i.e., places) are the hardware (e.g., *hmd*) or software components (e.g., *renderProcessing*) selected for the design; light colored circles are notations (called stub

[4] http://archive.eiffel.com/doc/manuals/technology/contract

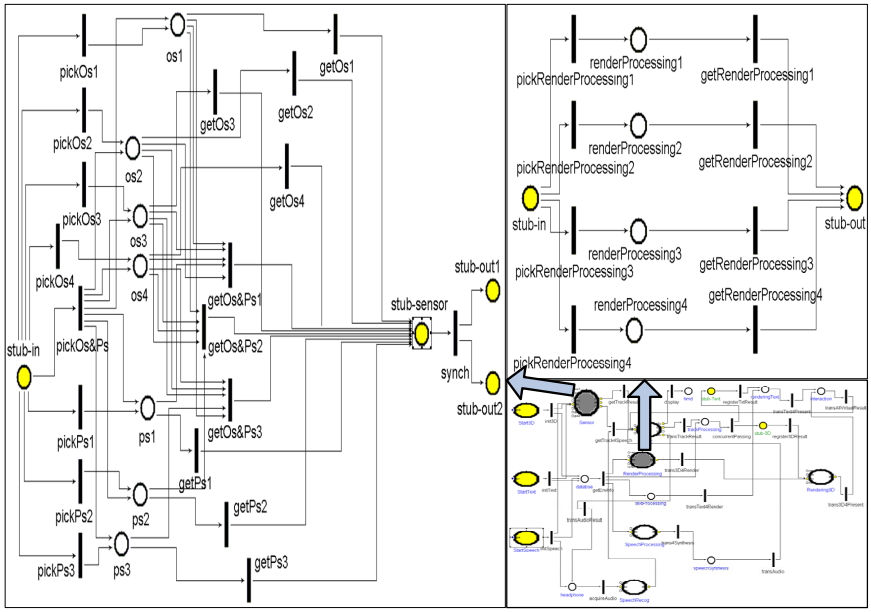

Fig. 3. (a) Timed Colored Petri Net Graph of BTS (at bottom right) (b) Design exploration for *IS-PCTracker* (at left) and (c) for *renderProcessing* (at top right)

places) for decision making complying with the syntax of timed colored Petri Nets; black bars are the functions performed along the execution paths (e.g., *getTrackResult*) or selection actions exploring the design space (e.g., *pickOs4* of Figure 3 (b)); and arrows are the direction of the execution paths (e.g., all arrows in Figure 3 (a)) or design decisions (e.g., all arrows in Figure 3 (b) and (c)).

Figure 3 (a) describes the behavioral view of software architecture of the BTS. The enlarged view of Figure 3 (a) and its details may be found in [17]. Figure 3 (b) is a containment component of Figure 3 (a) that represents all possible design alternatives of the *IS-PCTracker* derived from selecting different combinations of orientation sensors (OS) and position sensors (PS). Because of the non-determinism of timed colored Petri Nets, tokens flowing along the *stub-in* place can be directed to any of seven transitions without preference. Transitions *pickOs1* to *pickOs4* and *pickPs1* to *pickPs3* mean that only one of the sensors is selected. Transition *pickOs&Ps* forces all seven sensors to be possible candidates, and transitions *getOs&Ps1* to *getOs&Ps3* choose two sensors from OS and PS, respectively. There are twelve design alternatives generated due to the non-determinism. Figure 3 (c) is also a containment component in Figure 3 (a) that shows four *renderProcessing* components appropriate for constructing the BTS. To guarantee high-confidence DRE system construction, analysis and assertion statements, treated as pre-conditions and/or post-conditions of component composition, are written in AspectJ following QoS requirements and woven into the source code of the QoS-UniFrame interpreter, as shown in Figure 4.

```
1  pointcut analyzeQoS() : call(public void *.enableTrans(..)) &&
     args(QoSPar qos);
2  after (QoSPar qos) : analyzeQoS(qos){
3     double qosValue=0.0;   double [] compValue, serviceValue, systemValue;
4     boolean flag = false;
5     Object [] obj = thisJoinPoint.getArgs();
6     JBuilderAtom tran = (JBuilderAtom)obj[0];
7     Vector inConn = tran.getInConnections("Place2Trans");
8     for (int i=0;i<inConn.size();i++){
9        JBuilderConnection place2Trans = (JBuilderConnection)inConn.get(i);
10       JBuilderModel place = (JBuilderModel)place2Trans.getSource();
11       Vector myToken = place.getAtoms(qos.getName());
12       JBuilderAtom token = (JBuilderAtom)myToken.get(0);
13       if (qos.getAnalysisLevel(“component”)==true){
14          flag = token.getAttribute(qos.getName(), compValue);
15          if (flag) qosValue = compValue[0];
16          if (qosValue > qos.getStrictComponentRequirements())
17             flag = Global.storeEnableTran.removeElement(tran);
18       }
19       if (qos.getAnalysisLevel(“service”)==true){
20          flag = token.getAttribute(qos.getName(), serviceValue);
21          if (flag) qosValue = serviceValue[0] +
               token.getAttribute("CurrentService",serviceValue);
22          if (qosValue > qos.getStrictServiceRequirements())
23             flag = Global.storeEnableTrans.removeElement(tran);
24          else token.setAttribute("CurrentService",qosValue);
25       }
26       if (qos.getAnalysisLevel("system")==true)
27       {   /*...similar to the analysis at the service level..*/     }
28 }}
```

Fig. 4. An AspectJ example to analyze and assert QoS requirements

Figure 4 asserts the satisfaction of the lower bound of a QoS parameter at the component, service, and system levels. *enableTrans* is a function that verifies if a transition is enabled to facilitate the reachability tree generation. All enabled transitions are stored in a global vector, called *storeEnableTrans*. The loop from lines 8 to 28 examines all the places connected to the transition. Lines 13 to 18 assure a requirement of the QoS parameter at the component level. If the requirement is not met, the enabled transition will be removed from the vector, as shown in line 17. For the service level QoS requirements analysis, line 21 is the QoS formula computing *how well* the corresponding functional task performs. If the requirement is met, the current value of the QoS parameter is updated (line 24). Conversely, line 23 deletes the transition such that the reachability tree will not generate new nodes related to this transition.

QoS-UniFrame performs design space exploration and elimination during a DRE system construction. The design space exploration approach visually

depicts the behavioral view of software architecture at the higher abstraction level. The design space elimination approach analyzes all kinds of QoS requirements by passing various types of QoS parameters (i.e., *QoSPar*) into Figure 4 and by revising the QoS formulae accordingly. Due to the space considerations, please refer to [16] for the stochastic design space elimination using PPCEA .

3.3 System Integration

In UniFrame, application domains described in the knowledgebase are assumed to be formalized using a Generative Domain Model [6]. A key aspect of a GDM is the presence of generative *rules* which formalize its structure. GDM's may be constructed for various domains according to the standards. Furthermore, components developed for that domain will also follow these standards. We use Two-Level Grammar (TLG) [4] to express the GDM since TLG's class hierarchy allows convenient expression of abstract component hierarchies and TLG rules may be used to express the generative rules required by the GDM [5]. TLG may be used to provide attribute evaluation and transformation, syntax and semantics processing of languages, parsing, and code generation. All of these are required to use TLG as a specification language for components and domain-specific generative rules.

An example TLG for a sound sensor GDM is:

```
class SoundSensor is subclass of Sensor.
  SoundLocation :: Location.
  SoundVolume :: Float.
  AlarmThreshold :: Float.
  SafeArea :: {Location}*.
  alarm : SoundVolume > AlarmThreshold,
                        SoundLocation not in SafeArea.
end class Sensor.
```

`SoundSensor` inherits various `Sensor` properties such as the location of the sensor itself and adds additional properties such as the location and volume of the sound detected, the threshold at which an alarm should be sounded, and a safe range to ignore sounds. These type declarations are established by the first level of the TLG and correspond to context-free grammar rules (the `::` corresponds to the `::=` in traditional BNF notation). Note that `SafeArea` is a set of 0 or more locations. The second level of the grammar contains rules (e.g., in the above TLG, `alarm` will be true if the sound volume exceeds the alarm threshold and the sound location is not in the set of safe area locations). Additional rules may establish pre-conditions, post-conditions, and invariants, including QoS constraints.

The component development and deployment process starts with a UMM requirements specification of a component, following the established GDM for a particular domain. The UMM specification is informal and indicates the functional (i.e., computational, cooperative and auxiliary aspects) and non-functional (i.e., QoS constraints) features of the component. This informal specification may

also be formalized using TLG to provide additional semantics such as rules for validating the component and pre and post-conditions. Validated components are deployed on the network for potential discovery by the URDS. If the component does not meet the requirement specifications then the developer refines either the UMM requirements specification or the design.

MDA[5] Platform Independent Models (PIM's) are based upon the domains and associated logic for the given application. TLG allows these relationships to be expressed via inheritance. If a software engineer wants to design a server component to be used in a distributed embedded system, then he/she should write an informal requirements specification in the form of a UMM describing the characteristics of that component. We use the UMM and domain knowledge base to generate platform independent and platform-specific UMM specifications expressed in TLG (which we will refer to as UMM-PI and UMM-PS, respectively). UMM-PI describes the bulk of the information needed to progress to component implementation. UMM-PS merely indicates the technology of choice (e.g., CORBA[6]). These effectively customize the component model by inheriting from the TLG classes representing the domain with new functionality added as desired. In addition to new functionality, we also impose end-to-end Quality-of-Service expectations for our components (e.g., a specification of the minimum frame-rate in a distributed video streaming application). Both the added functionality and QoS requirements are expressed in TLG so there is a unified notation for expressing all the needed information about components. A translation tool [12] may be used to translate UMM-PI into a PIM represented by a combination of UML and TLG. Note that TLG is needed as an augmentation of standard modeling languages such as UML to define domain logic and other rules that may not be convenient to express in UML directly.

A Platform Specific Model (PSM) is an integration of the PIM with technology domain-specific operations (e.g., in CORBA, J2EE[7], or .NET[8]). These technology domain classes also are expressed in TLG. Each domain contains rules that are specific to that technology, including how to construct glue code for components implemented with that technology. Architectural considerations are also specified, such as how to distinguish client code from server code. PSMs may be expressed in TLG as an inheritance from PIM TLG classes and technology domain TLG classes. This means that PSMs will contain not only the application-domain-specific rules, but also the technology-domain-specific rules. The PSM also maintains the QoS characteristics expressed at the PIM level. Because the model is expressed in TLG, it is executable in the sense that it may be translated into executable code in a high-level language. Furthermore, it supports changes at the model level, or even requirements level if the model is not refined following its derivation from the requirements, because the code generation itself is automated.

[5] Model Driven Architecture - http://www.omg.org/mda
[6] Common Object Request Broker Architecture - http://www.omg.org/corba
[7] Java 2 Enterprise Edition - http://java.sun.com/javaee
[8] http://www.microsoft.com/net

An example of high-level rules to generate connector code between client-side and server-side operations is given below:

```
ClientUMM, ServerUMM :: UMM.
ClientOperations, ServerOperations :: {Interface}*.
```

Here it is assumed that UMM specifications exist for both the client and server and that the operations of each are represented as a syntactic interface (although we may wish to include semantic information in practice). The second level of the grammar provides for generating code to map the client operations to the server operations according to a specific component model. Additional rules would specify the details of these mappings. Such rules may use both application-specific and technology-specific domain knowledge.

3.4 Quality Validation

After system integration, a validation procedure demonstrates the functionality correctness and quality satisfaction of a DRE system. The Attributed Event Grammar (AEG) approach [1], as shown in Figure 5, is introduced for creating and running test cases in automated black-box testing of real-time reactive systems (e.g., reactive behaviors of triggering rifles).

The purpose of the attribute event grammar is to provide a vehicle for generating event traces (Step 1 in Figure 5). An *event* is any detectable *action* in the environment that could be relevant to the operation of the System Under Test (SUT). For example, an event may be a time interval or a group of sensors triggered by a soldier that has a beginning, an end, and duration. There are two basic relations defined for events: two events may be ordered or one event may appear inside another event. The behavior of the environment (i.e., *event trace*) can be represented as a set of events with these two basic relations defined for them. Two events can happen concurrently as well. An event may have attributes associated with it. Each event type may have a different attribute set. Event grammar rules can be decorated with attribute evaluation rules. The

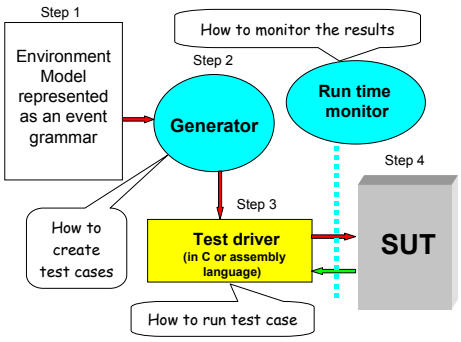

Fig. 5. An Overview of the AEG approach

action is performed immediately after the preceding event is completed. Events usually have timing attributes like begin_time, end_time, and duration. Some of those attributes can be defined in the grammar by appropriate actions, while others may be calculated by appropriate default rules. For example, for a sequence of two events, the begin time of the second event should be generated larger than the end time of the preceding event.

The event traces generated by the generator (Step 2) are not completely random since they fulfill constraints embedded in the environment model. Event attributes provide inputs to the SUT, and the event trace structure facilitates the necessary timing constraints. The test driver (e.g., a C program) can be derived from the given event trace (Step 3). Generated test drivers may interact with the system and adjust the evolving event trace based on the results of that interaction. The environment model can contain descriptions of hazardous states in which SUT could arrive. Thus, it becomes possible to conduct experiments with the SUT in the simulated environment and gather statistical data about the behavior of SUT in order to estimate operational effectiveness, safety and other dependability properties of the SUT (Step 4). By changing the values of parameters of the environment model (e.g., adjusting frequencies of some events in the model and running experiments with the adjusted model), the dependencies between environment parameters and the behavior of the system can be identified. This approach integrates the SUT into the environment model, and uses the model for both testing of the SUT in the simulated environment and assessing risks posed by the SUT. Such an approach may also be applied to a wide range of reactive systems, where environment models can be defined to specify typical scenarios and functional profiles.

The following (oversimplified) example of a missile defense scenario of the BTS demonstrates how to incorporate an interaction with the SUT into AEG. We assume the SUT tracks the launched missile by receiving specific geographical data from the orientation and position sensors of *IS-PCTracker* on the soldier (*send_sensor_signal*() action in the model simulates sensor inputs to the SUT), and at a certain moment makes a decision to fire an anti-missile (i.e., *interceptor*) by generating an output to a corresponding actuator (*SUT_launch_interceptor*()). The catch construct represents an external event generated at runtime by the SUT. The external event listener is active during the execution of a test driver obtained from the generated event trace. This particular external event is broadcast to all corresponding event listeners. The following event grammar specifies a particular set of scenarios for testing purposes.

```
Attack::= { Missile_launch } *
```

The Attack event contains several parallel Missile_launch events.

```
Missile_launch::= Boost_stage / Middle_stage.completed := True/
                  Middle_stage  WHEN (Middle_stage.completed) Boom
```

The Boom event (which happens if the interception attempts have failed) represents an environment event, which the SUT in this case should try to avoid.

```
Middle_stage::= ((CATCH  SUT_launch_interception(hit_coordinates)
                  WHEN(hit_coordinates == Middle_stage.coordinates)
                    [ p(0.1) interception
                      / Middle_stage.completed := False;
                      send_hit_input(Middle_stage .coordinates);
                      BREAK; /   ] END_CATCH   |  move )
                ) *
```

The sequence of move events within Middle_stage may be interrupted by receiving an external event *SUT_launch_counterattack* (*hit_coordinates*) from the SUT. This will suspend the move event sequence and will either continue with event counterattack (with probability 0.1), which simulates the enemy-counterattack event triggered by the SUT, followed by the BREAK command, which terminates the event iteration, or will resume the move sequence. This model allows several counterattack attempts through the same missile launch event. For simplicity it is assumed that there is no delay between receiving the external event and the possible counterattack event.

```
move ::= /adjust( ENCLOSING Middle_stage .coordinates) ;
          send_sensor_signal(ENCLOSING Middle_stage.coordinates);
          move.duration:= 1 sec /
```

This rule provides attribute calculations and sends an input to the SUT. In general, external events (i.e., events generated by the SUT) may be broadcast to several event listeners in the AEG, or may be declared as exclusive and will be consumed by just one of the listeners. If there is not a listener available when an external event arrives, there may be an error in the environment model, which can be detected and reported at the test execution time. To alleviate this problem, AEG may contain a mechanism similar to an exception handler for processing external events which have missed regular event listeners.

The environment model defined by AEG can be used to generate (pseudo) random event traces, where events will have attribute values attached, including time attributes. The events can be sorted according to the timing attributes and the trace may be converted into a test driver, which feeds the SUT with inputs and captures SUT outputs. The functionality of this generated test driver is limited to feeding the SUT inputs and receiving outputs and may be implemented as an efficient C or even assembly language program that meets strict real-time requirements. Only send and catch actions obtained from the event trace are needed to construct the test driver; the rest of the events in the event trace are used as "scaffolds" to obtain the ordering, timing and other attributes of these actions. The generator takes as input the AEG model and outputs random event traces. Necessary actions are then extracted from the trace and assembled into a test driver.

The main advantages of the approach are as follows: 1) The environment model provides for automated generation of a large number of random test drivers; 2) It addresses the regression testing problem: generated test drivers can be saved and reused; 3) The generated test driver contains only a sequence

of calls to the SUT, external event listeners for receiving the outputs from SUT, and time delays where needed to fulfill timing constraints, hence it is quite efficient and could be used for real-time test cases; 4) Different environment models for different purposes can be designed; 5) Experiments with the environment model running with the SUT provide a constructive method for quantitative and qualitative software risk assessment [28]; and 6) Environment models can be designed in early stages, before the system design is complete and can be used as an environment simulation tool for tuning the requirements and prototyping efforts. The generated event traces can be considered as use cases that may be used for requirements specification on early stages of system design.

4 Related Work

In recent years, there have been multiple research theories and industrial standards proposed for DRE systems (e.g., TAO [25]). Because various kinds of complexities are omni-present in DRE systems, there are many possible solutions to such complexities that have been introduced at different abstraction levels. Among many tools presented by different institutes or vendors, the following are relevant to UniFrame.

- **RAPIDware:** RAPIDware [18] is a project for component-based development of adaptable and dependable middleware. It uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments throughout the entire software lifecycle. RAPIDware consists of three major techniques to fulfill its objectives: in terms of the design workflow, adaptable design techniques are utilized to design components that comprise crosscutting concerns (e.g., QoS and security); a programming paradigm is introduced to specify QoS requirements, evaluate the system accommodation in terms of different configurations and contexts, and validate functional and non-functional properties via automated checking; and a middleware development toolkit that assists software engineers in implementing and selecting components and composing the entire system.
- **APEX:** Advanced Programming Environment for Embedded Computing Systems (APEX) [29] is a promising infrastructure for software development in the domain of embedded systems, especially for digital signal processing. Similar to UniFrame, APEX consists of five core techniques that cover the entire software life-cycle: the Online Repository for Embedded Software is a web-based repository systems to facilitate component management and retrieval; the COTS Aware Requirement Engineering methodology adapts and analyzes product requirements for any possible artifact reuse during the software development; the Design for Independent Composition and Evaluation techniques decomposes an embedded system into a set of independent subsystems in the design workflow for better modularization; the Automated Modification and Integration of Components utilities compose and customize components by generating glue code using existing design patterns and class

templates, respectively; and the Environment for Automated Simulation and Quality Analysis toolkit simulates the embedded systems and performs the coverage and performance analysis.

There are three key differences between APEX, RAPIDware, and UniFrame. First, UniFrame and RAPIDware are promising in seamlessly integrating a system from homogeneous and heterogeneous components by respectively using automated glue/wrapper code generation and middleware techniques. APEX has not explicitly discussed this issue [29]. Second, in order to reuse components effectively and efficiently, UniFrame introduces a QoS-driven Product Line (QoSPL) [17] framework to assist in constructing a set of DRE systems that share common features in the design and analysis workflows. RAPIDware introduces a middleware development toolkit for selecting and integrating components. APEX mainly concentrates on the reusability analysis at the requirements workflow and exploits the analysis results to the following workflows. Finally, to our best knowledge, the formalisms (e.g., stochastic Petri Nets) that APEX applies mostly concentrate on performance analysis and validation. The usage of formalisms is relatively less mentioned in other workflows. Conversely, both UniFrame and RAPIDware use formalisms throughout the software development.

5 Conclusion and Future Work

Rapid advances in hardware, software, and networking technologies are enabling an unprecedented growth in the capabilities of complex DRE systems. However, the traditional development pressures continue to force the introduction of creative ways to develop systems more rapidly and with less cost. For years, many such creative ways have been derived from the concepts of essential and accidental complexities [3]. UniFrame addresses such complexity by utilizing a unified component and resource discovery technique, a timed colored Petri Nets modeling toolkit, an automatic code generation paradigm, and an event trace approach. Additionally, the last formal method technique enhances the confidence of DRE system construction by specifying event traces, generating and executing test cases, and validating quality issues.

Currently, various prototypes of URDS have been constructed and experimented with. These prototypes contain the features of pro-active discovery, multi-level matching (matching restricted to only a few levels), and customization based on reinforcement learning principles. The results of these experimentations are promising and hence, efforts are underway to customize the URDS to the domain of MARS. The scope of design space exploration and elimination that QoS-UniFrame covers is mostly on software and hardware issues. Design and analysis paradigms to address network latencies are under the situation of local area network communication such that the latencies can be ignored. Enriching the notations of timed colored Petri Nets to comprise various communication approaches and heterogeneous protocols over network is our current plan. In addition, as a part of the prototype of the product line engineering framework, QoS-UniFrame is planned to cohesively collaborate with QoS-driven TLG [15]

for DRE product line construction. As for AEG, the first prototype of the test driver generator has been implemented at Naval Postgraduate School and used for several case studies. In the area of AR, extensive work has been done on the registration and calibration aspects that relate the coordinate systems, including those of trackers [26][27][31].

References

1. Auguston, M., Michael, J. B., Shing, M.-T.: Environment Behavior Models for Scenario Generation and Testing Automation. Proc. ICSE Workshop Advances in Model-Based Software Testing (2005)
2. Beugnard, A., et al.:Making Components Contract Aware. IEEE Computer (1999) **32**(7) 38–45
3. Brooks, F. P.: No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer (1987) **20** 10–19
4. Bryant, B. R., Lee, B.-S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. Proc. 35th Hawaii Intl. Conf. System Sciences (2002), `http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf`
5. Bryant, B. R., et al.: Formal Specification of Generative Component Assembly Using Two-Level Grammar. Proc. 14^{th} Intl. Conf. Software Engineering and Knowledge Engineering (2002) 209–212
6. Czarnecki, K., Eisenecker, U. W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
7. Klinker, G. J., et al.: Confluence of Computer Vision and Interactive Graphics for Augmented Reality. Presence: Teleoperators and Virtual Environments (1997) **6**(4) 433–451
8. Heineman, G., Councill, W. T.: Component-Based Software Engineering. Addison-Wesley (2001)
9. Jensen, K.: Coloured Petri Nets V Basic Concepts, Analysis Methods and Practical Use, Volume 1, Basic Concepts. Monographs in Theoretical Computer Science. Springer-Verlag (1997)
10. Kiczales, G., et al.: Getting Started with AspectJ. Communication of the ACM (2001) **44**(10) 59–65
11. Kordon, F., Lemoine, M., eds.: Formal Methods for Embedded Distributed Systems: How to Master the Complexity. Springer-Verlag (2004)
12. Lee, B.-S., Bryant, B. R.: Automation of Software System Development Using Natural Language Processing and Two-Level Grammar. Proc. 2002 Monterey Workshop on Radical Innovations of Software and Systems Engeering in the Future. Springer-Verlag Lecture Notes in Computer Sciences (2004) **2941** 219–223
13. Lee, B.-S., Bryant, B. R.: Applying XML Technology for Implementation of Natural Language Specifications. Intl. Journal of Computer Systems, Science and Engineering (2003) **5** 3–24
14. Liu, S.-H., Mernik, M., Bryant, B. R.: Parameter Control in Evolutionary Algorithms by Domain-Specific Scripting Language PPCEA. Proc. Intl. Conf. Bioinspired Optimization Methods and their Applications (2004) 41–50
15. Liu, S.-H., et al.: Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach. Proc. 17^{th} Intl. Conf. Software Engineering and Knowledge Engineering (2005) 731–734

16. Liu, S.-H., et al.: QoS-UniFrame: A Petri Net-Based Modeling Approach to Assure QoS Requirements of Distributed Real-time and Embedded Systems. Proc. 12^{th} IEEE Intl. Conf. and Workshop Engineering of Computer Based Systems (2005) 202–209
17. Liu, S.-H., et al.: QoSPL: A QoS-Driven Software Product Line Engineering Framework for Distributed Real-time and Embedded Systems. Proc. 18^{th} Intl. Conf. Software Engineering and Knowledge Engineering (2006) 724–729
18. Michigan State University. RAPIDWare: Component-Based Development of Adaptable and Dependable Middleware. (2006) `http://www.cse.msu.edu/rapidware`
19. Olson, A. M., et al.: UniFrame: A Unified Framework for Developing Service-Oriented, Component-Based Distributed Software Systems. Service-Oriented Software System Engineering: Challenges and Practices, eds. Stojanovic Z., Dahanayake, A., Idea Group Inc. (2005) 68–87
20. Raje, R. R., et al.: A Unified Approach for the Integration of Distributed Heterogeneous Software Components. Proc. 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration (2001) 109–119
21. Raje, R., et al.: A QoS-based Framework for Creating Distributed and Heterogeneous Software Components. Concurrency and Computation: Practice and Experience (2002) **14** 1009–1034
22. Reicher, T.: A Framework for Dynamically Adaptable Augmented Reality Systems. Doctoral Dissertation, Institutfür Informatik. Technische Universität München (2004)
23. Siram, N. N., et al.: An Architecture for the UniFrame Resource Discovery Service. Proc. 3^{rd} Intl. Workshop Software Engineering and Middleware (2002) 20–35
24. Siram, N. N.: An Architecture for the UniFrame Resource Discovery Service. Master Thesis, Dept. of CIS. Indiana Univ.-Purdue Univ. Indianapolis (2002)
25. TAO (The ACE ORB). Distributed Object Computing (DOC) Group for Distributed Real-time and Embedded Systems. (2006) `http://www.cs.wustl.edu/~schmidt/TAO.html`
26. Tuceryan, M., Genc, Y., Navab, N.: Single Point Active Alignment Method (SPAAM) for Optical See-through HMD Calibration for Augmented Reality. Presence: Teleoperators and Virtual Environments (2002) **11**(3) 259–276
27. Tuceryan, M., et al.: Calibration Requirements and Procedures for a Monitor-based Augmented Reality System. IEEE Trans. on Visualization and Computer Graphics (1995) **1**(3) 255–273
28. Tummala, H. et al.: Implementation and Analysis of Environment Behavior Models as a Tool for Testing Real-Time, Reactive System. Proc. 2006 IEEE Intl. Conf. on System of Systems Engineering (2006) 260–265
29. University of Texas at Dallas. APEX: Advanced Programming Environment for Embedded Computing Systems. (2006) `http://www.utdallas.edu/research/esc`
30. Wang, N., et al.: QoS-enabled Middleware. Middleware for Communications. Wiley and Sons (2003)
31. Whitaker, R., et al.: Object Calibration for Augmented Reality. Proc. Eurographics '95 (1995) 15–27
32. Zaremski, A. Wing, J.: Specification Matching of Software Components. ACM Trans. on Software Engineering (1995) **6**(4) 333–369