Real-Time Embedded Software Design for Mobile and Ubiquitous Systems

Pao-Ann Hsiung[†], Shang-Wei Lin, Chin-Chieh Hung, Jih-Ming Fu, Chao-Sheng Lin, Cheng-Chi Chiang, Kuo-Cheng Chiang, Chun-Hsien Lu, and Pin-Hsien Lu

Department of Computer Science and Information Engineering
National Chung-Cheng University, Chiayi, Taiwan–621, ROC.
[†]URL: http://www.cs.ccu.edu.tw/∼pahsiung/, hpa@computer.org

Abstract. Currently available application frameworks that target at the automatic design of real-time embedded software are poor in integrating functional and non-functional requirements for mobile and ubiquitous systems. In this work, we present the internal architecture and design flow of a newly proposed framework called *Verifiable Embedded Real-Time Application Framework* (VERTAF), which integrates three techniques namely software component-based reuse, formal synthesis, and formal verification. The proposed architecture for VERTAF is component-based which allows plug-and-play for the scheduler and the verifier. The architecture is also easily extensible because reusable hardware and software design components can be added. Application examples developed using VERTAF demonstrate significantly reduced relative design effort, which shows how high-level reuse of software components combined with automatic synthesis and verification increases design productivity.

Keywords: application framework, code generation, real-time embedded software, formal synthesis, formal verification, scheduling, software components, UML modeling

1 Introduction

With the proliferation of embedded mobile and ubiquitous systems in all aspects of human life, we are making greater demands on these systems, including more complex functionalities such as pervasive computing, mobile computing, and real-time embedded computing. Currently, the design of real-time embedded software is supported partially by modelers, code generators, analyzers, schedulers, and frameworks [1–8]. Nevertheless, the technology for a completely integrated design and verification environment is still immature. Furthermore, the methodologies for design and for verification are also poorly integrated relying mainly on the experiences of embedded software engineers. Motivated by the status-quo, this work demonstrates how the integration of software engineering techniques such as software component reuse, formal software synthesis techniques such as scheduling and code generation, and formal verification technique such as model checking can be realized in the form of an integrated design environment targeted at the acceleration of real-time embedded software construction.

Mobile and ubiquitous systems involve the dynamic reconfiguration of applications in response to changes in their environments. Middlewares such as network layer mobility support, transport layer mobility support, traditional distributed systems applied to mobility, middleware for wireless sensor networks, context awareness based middleware, and publish-subscribe middleware are required for efficient development of mobile and ubiquitous applications. A user can develop an application using such middlewares, however it can sometimes be too tedious and complex to consider all the different possible environments and application features. Examples of environments include office and domestic spaces, educational and healthcare institutions and in general urban and rural environments. Examples of applications include domestic and industrial security applications, education and learning applications, healthcare applications, traffic management, commercial advertising, games and arts, rescue operations, and military. Given such complex combinations of environments and applications, one would desire a higher level of reuse than that allowed by object-oriented design and middlewares. We are thus proposing an integrated design framework that allows such higher level of reuse.

As described below, several issues are encountered in the development of an integrated design framework.

1. To allow software component reuse, how do we define the syntax and semantics of a reusable component?
2. What is the control-data flow of the automatic design and verification process?
3. What kinds of model can be used for scheduling and verification?
4. What methods are to be used for scheduling and for verification?
5. How do we generate portable code that not only crosses real-time operating systems (RTOS) but also hardware platforms. What is the structure of the generated code?

Briefly, our solutions to the above issues can be summarized as follows.

1. Software Component Reuse and Integration: A subset of the Unified Modeling Language (UML) [9] is used with restrictions for automatic design and analysis.
2. Control Flow: A specific control flow is embedded within the framework, where scheduling is first performed and then verification because the complexity of verification can be greatly reduced after scheduling [3].
3. System Models: For scheduling, we use variants of Petri Nets (PN) [5] and for verification, we use Extended Timed Automata (ETA) [10], both of which are automatically generated from UML models that follow restrictions and guidelines.
4. Design Automation: For synthesis, we employ quasi-static and quasi-dynamic scheduling methods [5] that generate program schedules for a single processor. For verification, we employ symbolic model checking [11] that generates a counterexample in the original user-specified UML models whenever verification fails for a system under design. For handling complexity, we applied model-based, architecture-based, and function-based abstractions during verification.
5. Portable Efficient Multi-Layered Code: For portability, a multi-layered approach is adopted in code generation. To account for performance degradation due to multiple layers, system-specific optimization and flattening are then applied to the portable code. System dependent and independent parts of the code are distinctly segregated for this purpose.

In summary, this work illustrates how an application framework may integrate all the above proposed design and verification solutions. Our implementation has resulted in a Verifiable Embedded Real-Time Application Framework (VERTAF) whose features include formal modeling of real-time embedded systems through well-defined UML semantics, formal synthesis that guarantees satisfaction of temporal and spatial constraints, formal verification that checks if a system satisfies all properties, and code generation that produces efficient portable code.

The article is organized as follows. Section 2 described previous related work. Section 3 describes the design and verification flow in VERTAF along with an illustration example. Section 4 presents the experimental results of an application example. Section 5 gives the conclusions with some future work.

2 Previous Work

The software in mobile and ubiquitous systems has both traditional features of real-time embedded systems and also contemporary features such as adaptive resource management, proactive service discovery, context-aware coordination, multi-agents, and models for heterogeneous platforms [12]. This is mainly due to the unique requirements of such systems including interoperability, heterogeneity, mobility, survivability, security, adaptability, ability of self-organization, augmented reality, and scalable content. Though there are numerous work on the software in mobile and ubiquitous systems, besides VERTAF, there is practically no design environment that can encompass the whole design and verification flow of such systems. In the following, we briefly survey two main areas of research in this domain, including middleware and frameworks.

Middleware design is important for ubiquitous systems because it is through this software that an application connects to the network and exchanges data with other applications. Typical examples include the OSA+ middleware architecture [13], the Reconfigurable Context-Sensitive Middleware (RCSM) [14], and the T-Engine architecture [15]. The OSA+ middleware facilitates the development of distributed real-time applications in a heterogeneous environment. Some essential features of OSA+ include quality of service information requirement for each service, explicit support for asynchronous communication, real-time memory services, and small memory footprint. OSA+ has been applied to e-health management services including patient identification, location monitoring, remote checking, and continuous accurate monitoring of patient's vital signs. The RCSM architecture facilitates the development of real-time context-aware software in ubiquitous computing environments. This architecture mainly combines CORBA and FPGA such that CORBA allows mobility and FPGA allows dynamic reconfiguration (ubiquity). RCSM has been applied to sensor networks such that object interactions are context-triggered. The T-Engine architecture is an open, real-time embedded systems platform aimed at improving software productivity. The T-Engine consortium includes computer hardware and software vendors, telecommunication carriers, and computer-using companies. T-Engine adopts a layered architecture including application, middleware, kernel, monitor, and hardware layers.

There are several either fixed architectures or variable frameworks that have been proposed in the literature for mobile and ubiquitous systems. Typical examples include

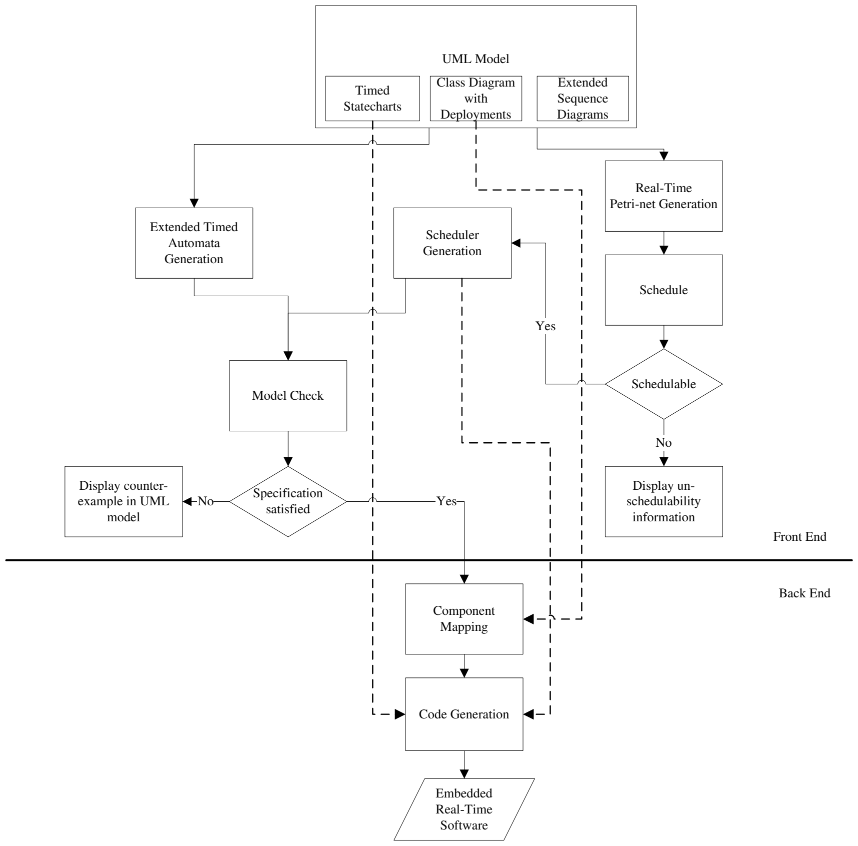
UML Model
Timed Statecharts
Class Diagram with Deployments
Extended Sequence Diagrams
Real-Time Petri-net Generation
Extended Timed Automata Generation
Scheduler Generation
Schedule
Yes
Schedulable
Model Check
No
Display counter-example in UML model
No
Specification satisfied
Yes
Display un-schedulability information
Front End
Back End
Component Mapping
Code Generation
Embedded Real-Time Software

Fig. 1. Design and Verification Flow of VERTAF

the Connected Multimedia Services (CMS) framework [16], the Earl Gray JVM-based Component (EGC) framework [17], and the Static Composition Framework (SCF) of service-based real-time applications [18]. The CMS framework is based on SIP and X.10 protocols and allows multimedia sessions to be preserved when a user moves from one computing environment to another. The EGC framework analyzes component dependencies using a component-based JVM called Earl Gray. The SCF framework allows to announce services, to discover services, and to select services for an application.

3 Design and Verification Flow in VERTAF

As shown in Figure 1, VERTAF provides solutions to the various issues introduced in Section 1. The control and data flows of VERTAF are represented by solid and dotted arrows, respectively. Software synthesis is defined as a two-phase process: a machine-independent software construction phase and a machine-dependent software implementation phase. This separation helps us to plug-in different target languages, middleware, real-time operating systems, and hardware device configurations. We call the two phases as front-end and back-end phases. The front-end phase is further divided into three sub-phases, namely UML modeling phase, real-time embedded software scheduling phase, and formal verification phase. There are two sub-phases in the back-end phase, namely component mapping phase and code generation phase. We will now present the details of each phase illustrated by a running example called Entrance Guard System with Mobile and Ubiquitous Control (EGSMUC). EGSMUC is a real-time embedded system that controls any entrance with a programmable electronic lock installed. Two ways of control accesses are allowed: (a) registered users can be authenticated locally at the entrance itself, and (b) guest users may obtain a remote authentication through master acknowledgment. Here, a master could be the owner of the building to which the entrance system is protecting and he or she can have mobile and ubiquitous control access to EGSMUC. The master can grant entry access to the guest user irrespective of how he or she is connected to EGSMUC (mobile access) and also irrespective of where he or she is located (ubiquitous access). We will model EGSMUC and VERTAF will automatically synthesize and verify the code for the system.

3.1 UML Modeling

Three UML [9] diagrams are extended for real-time embedded software specification as follows.

- *Class Diagrams with Deployment*: A deployment relation is used for specifying a hardware object on which a software object is deployed. Two types of methods: event-triggered and time-triggered are used for modeling real-time behavior.
- *Timed Statecharts*: UML statecharts are extended with real-time clocks that can be reset and values checked as state transition triggers.
- *Extended Sequence Diagrams*: UML sequence diagrams are extended with control structures such as concurrency, conflict, and composition, which aid in formalizing their semantics and in mapping them to Petri net models for scheduling.

For our running EGSMUC example, the system class diagram with deployment is shown in Figure 2. Other diagrams are omitted due to page limit.

3.2 Real-Time Embedded Software Scheduling

There are two issues in real-time embedded software scheduling, namely how are memory constraints satisfied and how are temporal specifications such as deadlines satisfied. Based on whether the system under design has an RTOS specified or not, two different scheduling algorithms are applied to solve the above two issues.

- *Without RTOS*: *Quasi-dynamic scheduling* (QDS) [5] is applied, which requires *Real-Time Petri Nets* (RTPN) as system specification models. QDS prepares the system to be generated as a single real-time executive kernel with a scheduler.
- *With RTOS*: *Extended quasi-static scheduling* (EQSS) [19] with real-time scheduling [20] is applied, which requires *Complex Choice Petri Nets* (CCPN) and set of independent real-time tasks as system specification models, respectively. EQSS

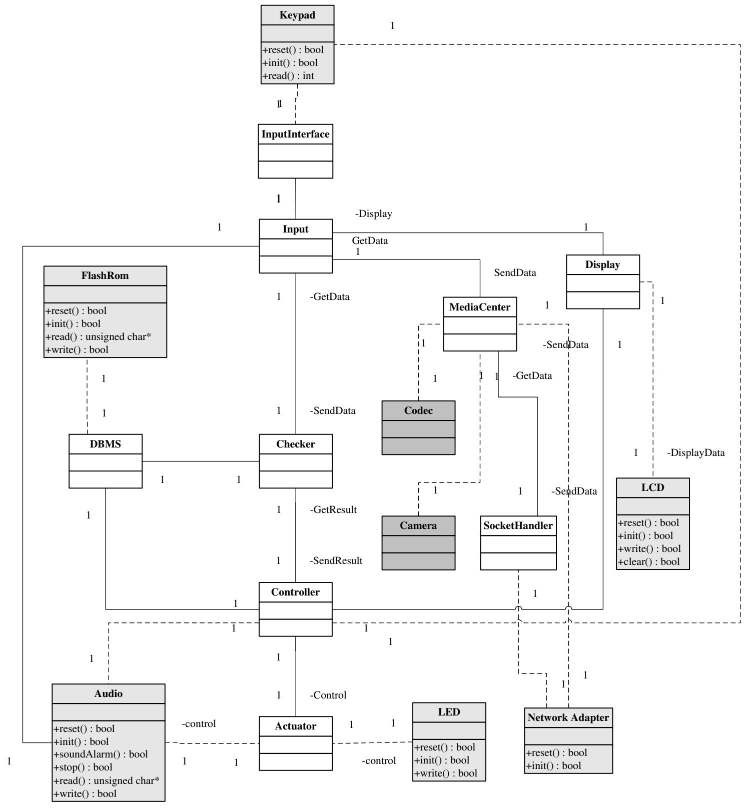

Fig. 2. Class Diagram with Deployment for Entrance Guard System with Mobile and Ubiquitous Control

prepares the system to be generated as a set of multiple threads that can be scheduled and dispatched by a supported RTOS such as MicroC/OS II or ARM Linux.

To apply the above scheduling algorithms, we need to map the user-specified UML models into Petri nets, RTPN or CCPN, which are generated automatically from user-specified UML sequence diagrams, through a case-by-case construction. It is out-of-scope here. The set of RTPN or CCPN is then input to QDS or EQSS, respectively, for scheduling. Details on the scheduling procedures can be found in [5], and [19].

For systems without RTOS, we need to automatically generate a scheduler that controls the system according to the set of transition sequences generated by QDS. In VERTAF, a scheduler is constructed as a separate class that observes and controls the status of each object in the system. Temporal constraints are monitored by the scheduler class using a global clock.

For our running EGSMUC example, a single Petri net is generated from the user-specified set of statecharts, which is then scheduled using QDS. In this example, scheduling is required only for the timers associated with the actuator, the controller, and the input object. After QDS, we found that EGSMUC is schedulable.

3.3 Formal Verification

VERTAF employs the popular model checking paradigm for formal verification of real-time embedded software. In VERTAF, formal ETA models are generated automatically from user-specified UML models by a flattening scheme that transforms each statechart into a set of one or more ETA, which are merged, along with the scheduler ETA generated in the scheduling phase, into a state-graph. The verification kernel used in VERTAF is adapted from *State Graph Manipulators* (SGM) [8], which is a high-level model checker for real-time systems that operate on state-graph representations of system behavior through manipulators, including a state-graph merger, several state-space reduction techniques, a dead state checker, and a TCTL model checker. There are two classes of system properties that can be verified in VERTAF: (1) system-defined properties including dead states, deadlocks, livelocks, and syntactical errors, and (2) user-defined properties specified in the *Object Constraint Language* (OCL) as defined by OMG in its UML specifications. All of these properties are automatically translated into TCTL specifications for verification by SGM.

For our running EGSMUC example, the ETA for each statechart were generated and then merged with the scheduler ETA. There are seven other ETA in this system example. All ETA were input to SGM and AGR was applied. Reduction techniques were then applied to each state-graph obtained from AGR. OCL constraints were then translated into TCTL and verified by the SGM model checker kernel.

3.4 Component Mapping

This is the first phase in the back-end design of VERTAF and starts to be more hardware dependent. All hardware classes specified in the deployments of the class diagram are those supported by VERTAF and thus belong to some existing class libraries. The component mapping phase then becomes simply the configuration of the hardware system and operating system through the automatic generation of configuration files, make files, header files, and dependency files. The corresponding hardware class API will be linked in during compilation.

An issue in this phase is the possible conflicts among hardware devices specified in a class diagram such as interrupts, memory address ranges, I/O ports, and bus-related characteristics such as device priorities. Users are warned in this case.

3.5 Code Generation

There are basically three issues in this phase including hardware portability, software portability, and temporal correctness. We adopt a multi-tier approach for code generation: an operating system layer, a middleware layer, and an application with scheduler layer, which solves the above three issues, respectively. Currently supported underlying hardware platforms include dual core ARM-DSP based, single core ARM, StrongARM, or 8051 based, and Lego RCX-based Mindstorm systems. For hardware abstraction, VERTAF supports MicroHAL and the embedded version of POSIX. For operating systems, VERTAF supports MontaVista Linux, MicroC/OS, Embedded Linux, and eCOS. For middleware, VERTAF is currently based on the Quantum Framework [7]. For

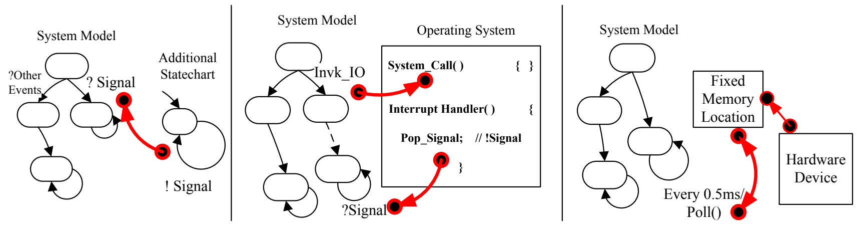

Fig. 3. I/O Delegation, Invocation, and Polling

scheduler, VERTAF creates a custom ActiveObject according to the Quantum API. Included in the scheduler is a temporal monitor that checks if any temporal constraints are violated.

Each ETA that is generated either from UML statecharts or from the scheduled Petri nets (sequence diagrams) is implemented as an ActiveObject in the Quantum Framework. The user-defined classes along with data and methods are incorporated into the corresponding ActiveObject. The final program is a set of concurrent threads, one of which is a scheduler that can control the other objects by sending messages to them after observing their states. For systems without an OS, the scheduler acts as a real-time executive kernel.

During code generation and the validation of automatically generated code, we discovered a peculiar problem as described in the following. UML statecharts have *run-to-completion* (RTC) semantics, that is, all actions within a state will complete execution, even if a new event or signal is received, before transiting to another state. However, in real-time embedded systems, I/O actions are usually infinite loops that poll hardware devices for data. If such I/O related high-latency low-priority events are modeled into a statechart with user-defined low-latency high-priority events, then due to RTC semantics a class object will deadlock during execution if there is no data from an I/O device that is polled. High-priority events cannot be handled. We observed this problem after code was automatically generated for our running EGSMUC example. As solutions, three methods are proposed for synthesizing the I/O interface between a user class and an I/O device. The methods are illustrated in Figure 3 and described as follows.

1. **I/O Delegation**: The deadlock can be removed from a user-defined statechart by introducing an additional statechart, as shown in the leftmost part of Figure 3, which has only one state and one self-loop transition. I/O devices are polled in that state and whenever data is available, an event or signal is broadcast. This statechart never receives any events or signals from other statecharts. The original statechart only waits for events from this statechart. However, while waiting, it can also handle high-priority low-latency events. Thus, there is no deadlock and the RTC semantics is also not violated.
2. **I/O Invocation**: If the operating system supports asynchronous I/O operations, the infinite polling loops can be replaced by invoking asynchronous I/O operations through system calls, as shown in the middle part of Figure 3. After invoking an asynchronous I/O operation, the statechart can continue with other operations.

Table 1. Mapping Devices and I/O Handling Mechanisms

I/O Device			OS support		
Type	**Interrupt**	**Buffer**	**AIO**	**BIO**	**NBIO**
WI	Yes	Yes	D, I	D	D, P
WB	No	Yes	N/A	D	D, P
NB	No	No	N/A	D	D, P

AIO: Asynchronous I/O, BIO: Blocking I/O, NBIO: Non-Blocking I/O,

WI: With Interrupt, WB: With Buffering, NB: No Buffering, D: Delegation, I: Invocation, P: Polling

When an I/O device has finished an invoked I/O operation, it interrupts the processor. The corresponding interrrupt handler then broadcasts an event, which is received by the original system statechart.

3. **I/O Polling**: This approach assumes that the I/O data will be stored in a fixed memory location such as the buffers in a hardware controller or OS. We can use a timer to poll the I/O device periodically instead of polling it in infinite loops, as shown in the rightmost part of Figure 3. Thus, the statechart will not be blocked in an infinite loop and can handle other events or signals between two timer periods.

One of the above proposed methods can be selected for automatic interface synthesis during code generation by identifying the type of I/O device. In general, I/O devices can be classified into three types: (1) WI: with buffering and interrupt support, (2) WB: with buffering but no interrupt support, and (3) NB: with neither buffering nor interrupt support. Examples include hard disk drives with interrupt support, infra-red remote controller with only buffering and no interrupt, and touch or light sensors with neither buffering nor interrupt support. The I/O delegation, invocation, and polling methods that are applicable for the three types of devices are given in Table 1 under different conditions of OS support. Normally, an OS might support asynchronous I/O (AIO), blocking I/O (BIO), and non-blocking I/O (NBIO). Table 1 can be read as follows. For example, for an I/O device of the WI type, if the OS supports only BIO, then only the I/O delegation method can be used to synthesize the interface between that device and a user-defined statechart.

As observed from Table 1, the I/O delegation method is a universally applicable method, except for cases where no interface is possible such as AIO with WB and with NB devices. For our running EGSMUC example, the interfaces for infra-red remote controller, for the network adaptor, and for the keypad were all synthesized using the I/O delegation method. We also implemented the I/O invocation and polling methods for the network adaptor, which is of the WI type. All three implementations for the network adaptor were functionally equivalent, except for performance differences.

For our running example, the final application code consisted of 9 activeobjects derived from the statecharts and 1 activeobject representing the scheduler. Makefiles were generated for linking in the API of the 8 hardware classes and configuration files were generated for the ARM-DSP dual microprocessor platform called DaVinci from Texas Instruments with MontaVista Linux as its operating system on the ARM processor and DSP/BIOS real-time kernel as the operating system on the DSP TMS6646DSP processor. There were totally 2,340 lines of C code for the full EGSMUC system, out of which the system designers had to write only around 263 lines of C code, which is only 11.2% of the full system code.

4 Analysis and Evaluation

For the running example EGSMUC, we now analyze why VERTAF is capable of generating a significant part of the system implementation code, thus alleviating the designer from the tedious and error-prone task of manual coding. Due to its application framework architecture, VERTAF supports software components that are commonly found in mobile, ubiquitous, real-time, and embedded application domains. We classify the components supported by VERTAF into the following.

- Storage and I/O Devices: This class includes all the storage and I/O devices that are supported by VERTAF and required for implementing a real-time embedded system. Examples from the EGSMUC system include FlashRom, Keypad, LCD, Audio, LED, and Camera.
- Communication Interfaces: This class includes all the interface components that allow connection with the external world, for example, wired and wireless network connection, Bluetooth, and GSM/GPRS. Network adapter is an example from EGSMUC system.
- Multimedia Processing: This class includes all the components providing API for multimedia encoding and decoding through codecs specific to hardware platforms such as the codecs provided by TI for DaVinci multimedia platform. The DSP class in the EGSMUC system is an example.
- Control and Management Interfaces: This class includes all the components for controlling and managing system components, such as the socket handler in the EGSMUC example.

To implement mobile and ubiquitous control access in a real-time embedded system, a user normally, without VERTAF, would have to install a web server, write multimedia processing code, write network code, and integrate everything together, along with application-specific context awareness or publish-subscribe middlewares. With VERTAF, most of these tedious work are not required as long as the user configures the correct components from the framework for use in his or her application.

For illustration purposes, we show how the Media Center class in the EGSMUC example was implemented using VERTAF. The Media Center class is responsible for getting acknowledgment from a mobile master ubiquitously, which means whenever a guest wants to enter the building that the EGSMUC system is guarding, the Media Center notifies the DSP class to use the Camera to capture an image of the guest and then send the guest image to a master (the owner of the building or house). The master can send an acknowledgment through the web after which the guest can enter the building. A password is setup by a guest so that the guest can enter the building within the span of time set by the master beforehand.

The architecture of the code generated by VERTAF consists of three parts, namely a web server, a QF activeobject, and an image processing interface. The web server allows a master to connect to EGSMUC using a web browser that can run Java applets.

The applet opens a socket connection between the media center and the client machine of the master. The image of the guest requesting entrance is captured and processed through the image processing interface. When a master acknowledges, the guest is notified through the input class. The control and data flows of the media center are automatically generated by VERTAF and the user has to merely specify the sequence diagrams and deploy the related classes to hardware or software components in the class diagram as shown in Figure 2. Hence, VERTAF can save a lot of coding and design efforts.

There were totally 18 objects in the final application generated by VERTAF, out of which the user or designer had to only model 7 classes. The remaining 11 classes included components from all the four categories as described at the start of Section 4. Empirical results obtained from comparing two different implementations of the EGSMUC system, one using VERTAF, and one without using VERTAF, showed that not only the user written code reduced to 11.2% and the number of objects reduced to 41%, but the total time required to develop the application also reduced by more than 60%. The average learning time for each designer using VERTAF was approximately 0.1 day. The experimental and empirical results all show that VERTAF is beneficial to designers of real-time embedded software with mobile and ubiquitous control access.

By employing various construction guidelines for design and several reduction techniques for verification as described in Section 3.3, VERTAF is scalable to large and complex applications. Since VERTAF was constructed in a component-oriented way, one can also easily extend its features by plug-and-play of new components. The flow of VERTAF can also be easily modified to incorporate the changes.

5 Conclusions and Future Work

An object-oriented component-based application framework, called VERTAF, was proposed for the development of real-time embedded system applications with mobile and ubiquitous control access. It was a result of the integration of three different technologies: software component reuse, formal synthesis, and formal verification. Starting from user-specified UML models, automation was provided in model transformations, scheduling, verification, and code generation. VERTAF can be easily extended by integrating new specification languages and scheduling algorithms.

Future extensions will include support for share-driven scheduling algorithms. VERTAF will be enhanced by considering more advanced features of real-time applications, such as: network delay, network protocols, and on-line task scheduling. Performance related features such as context switch time and rate, external events handling, I/O timing, mode changes, transient overloading, and setup time will also be incorporated.

References

1. Amnell, T., Fersman, E., Mokrushin, L., Petterson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems. (September 2003)
2. Douglass, B.: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison Wesley, USA (November 1999)

3. Hsiung, P.: Embedded software verification in hardware-software codesign. Journal of Systems Architecture - the Euromicro Journal **46**(15) (November 2000) 1435–1450
4. Hsiung, P., Cheng, S.: Automating formal modular verification of asynchronous real-time embedded systems. In: Proceedings of the 16th International Conference on VLSI Design, (VLSI'2003), IEEE CS Press (January 2003) 249–254
5. Hsiung, P., Lin, C.: Synthesis of real-time embedded software with local and global deadlines. In: Proceedings of the 1st ACM/IEEE/IFIP International Conference on Hardware-Software Codesign and System Synthesis, ACM Press (October 2003) 114–119
6. de Niz, D., Rajkumar, R.: Time Weaver: A software-through-models framework for embedded real-time systems. In: Proceedings of the International Workshop on Languages, Compilers, and Tools for Embedded Systems. (June 2003) 133–143
7. Samek, M.: Practical Statecharts in C/C++ Quantum Programming for Embedded Systems. CMP Books (2002)
8. Wang, F., Hsiung, P.: Efficient and user-friendly verification. IEEE Transactions on Computers **51**(1) (January 2002) 61–83
9. Rumbaugh, J., Booch, G., Jacobson, I.: The UML Reference Guide. Addison Wesley Longman (1999)
10. Alur, R., Dill, D.: Automata for modeling real-time systems. Theoretical Computer Science **126**(2) (April 1994) 183–236
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
12. Niemelä, E., Latvakoski, J.: Survey of requirements and solutions for ubiquitous software. In: Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia, ACM Press (October 2004) 71–78
13. Brinkschulte, U., Bechina, A., Keith, B., Picioroaga, F., Schneider, E.: A middleware architecture for ubiquitous computing systems with real-time needs. In: Proceedings of the IAR Workshop, Institute for Automation and Robotic Research, France (November 2002)
14. Yau, S.S., Karim, F.: Context-sensitive middleware for real-time software in ubiquitous computing environments. In: Proceedings of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE CS Press (May 2001) 163–170
15. Sakamura, K., Koshizuka, N.: T-Engine: the open, real-time embedded-systems platform. IEEE Micro **22**(6) (December 2002) 48–57
16. Kwak, J.Y., Sul, D.M., Ahn, S.H., Kim, D.H.: An embedded software architecture for connected multimedia services in ubiquitous network environment. In: Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE CS Press (May 2003) 61–64
17. Ishikawa, H., Ogata, Y., Adachi, K., Nakajima, T.: Requirements for a component framework of future ubiquitous computing. In: Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE CS Press (May 2003) 9–12
18. Estevez-Ayres, I., Garcia-Vails, M., Basanta-Val, P.: Static composition of service-based real-time applications. In: Proceedings of the 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, IEEE CS Press (May 2005) 11–15
19. Su, F., Hsiung, P.: Extended quasi-static scheduling for formal synthesis and code generation of embedded software. In: Proceedings of the 10th IEEE/ACM International Symposium on Hardware/Software Codesign (CODES'02), ACM Press (May 2002) 211–216
20. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard-real time environment. Journal of the Association for Computing Machinery **20** (January 1973) 46–61