Formal Design and Verification of Real-Time Embedded Software

Pao-Ann Hsiung[†] and Shang-Wei Lin

Department of Computer Science and Information Engineering,
National Chung-Cheng University, Chiayi, Taiwan–621, ROC
[†]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. In this work, we reveal 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. Component reuse is based on a formal UML real-time embedded object model. Formal synthesis employs quasi-static and quasi-dynamic scheduling with multi-layer portable efficient code generation, which can output either RTOS-specific application code or automatically-generated real-time executive with application code. Formal verification integrates a model checker kernel from SGM, by adapting it for embedded software. 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 as compared to design without VERTAF, which also shows how high-level reuse of software components combined with automatic synthesis and verification increase 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 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, embedded computing, and real-time computing. Currently, the design of real-time embedded software is supported partially by modelers, code generators, analyzers, schedulers, and frameworks [2], [5], [8]-[12], [13]-[16], [19], [22]-[27], [29], [30]. Nevertheless, the technology for a completely integrated design and verification environment is still

W.-N. Chin (Ed.): APLAS 2004, LNCS 3302, pp. 382–397, 2004.

relatively 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 above 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.

Several issues are encountered in the development of an integrated design environment. First and foremost, we need to decide upon an architecture for the environment. Since our goal is to integrate reuse, synthesis, and verification, we need to have greater control on how the final generated application will be structured, thus we have chosen to implement the environment as an object-oriented application framework [6], which is a "semi-complete" application, where users fill in application specific objects and functionalities. A major feature is "inversion of control", that is the framework decides on the control flow of the generated application, rather than the designer. Other issues encountered in architecting an application framework for real-time embedded software are as follows.

1. To allow software component reuse, how do we define the syntax and semantics of a reusable component? How can a designer uniformly and guidedly specify the requirements of a system to be designed? How can the existing reusable components with the user-specified components be integrated into a feasible working system?
2. What is the control-data flow of the automatic design and verification process? When do we verify and when do we schedule?
3. What kinds of model can be used for each design phase, such as scheduling and verification?
4. What methods are to be used for scheduling and for verification? How do we automate the process? What kinds of abstraction are to be employed when system complexity is beyond our handling capabilities?
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) [21] is used with minimal restrictions for automatic design and analysis. Precise syntax and formal semantics are associated with each kind of UML diagram. Guidelines are provided so that requirement specifications are more error-free and synthesizable.
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 [9].

3. System Models: For scheduling, we use variants of Petri Nets (PN) [11], [12] and for verification, we use Extended Timed Automata (ETA) [1], [12], both of which are automatically generated from user-specified UML models that follow our restrictions and guidelines.
4. Design Automation: For synthesis, we employ quasi-static and quasi-dynamic scheduling methods [11], [12] that generate program schedules for a single processor. For verification, we employ symbolic model checking [3], [4], [20] that generates a counterexample in the original user-specified UML models whenever verification fails for a system under design. The whole design process is automated through the automatic generation of respective input models, invocation of appropriate scheduling and verification kernels, and generating reports or useful diagnostics. For handling complexity, abstraction is inevitable, thus we apply 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 as well as spatial constraints, formal verification that checks if a system satisfies user-given properties or system-defined generic properties, and code generation that produces efficient portable code.

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

2 Design and Verification Flow in VERTAF

Before going into the component-based architecture of VERTAF, we first introduce the design and verification flow. As shown in Figure 1, VERTAF provides solutions to the various issues introduced in Section 1.

In Figure 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

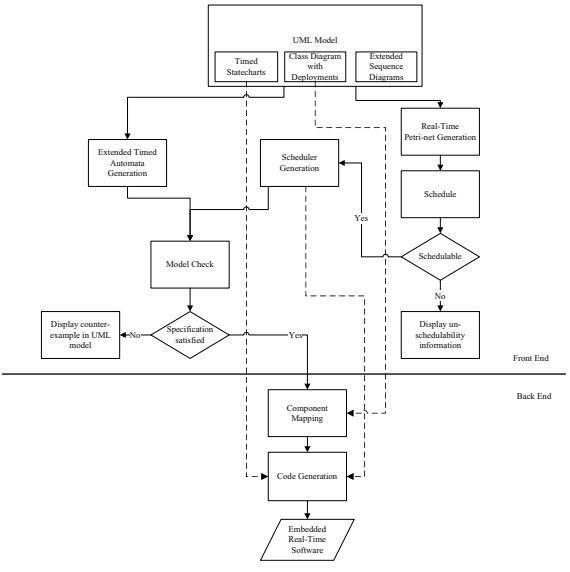

Fig. 1. Design and Verification Flow of VERTAF

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 in the rest of this section illustrated by a running example called Entrance Guard System (EGS). EGS is an embedded system that controls the entrance to a building by identifying valid users through a voice recognition IC and control software that runs on a StrongARM 1100 microprocessor.

2.1 UML Modeling

UML [21] is one of the most popular modeling and design languages in the industry. It standardizes the diagrams and symbols used to build a system model. After scrutiny of all diagrams in UML, we have chosen three diagrams for a user to input as system specification models, namely class diagram, sequence diagram, and statechart. These diagrams were chosen such that information redundancy in user specifications is minimized and at the same time adequate expressiveness in user specifications is preserved. UML is a generic language and its specializations are always required for targeting at any specific application domain. In VERTAF, the three UML diagrams are both restricted as well as enhanced along with guidelines for designers to follow in specifying synthesizable and verifiable system models (just as synthesizable HDL code for hardware designs).

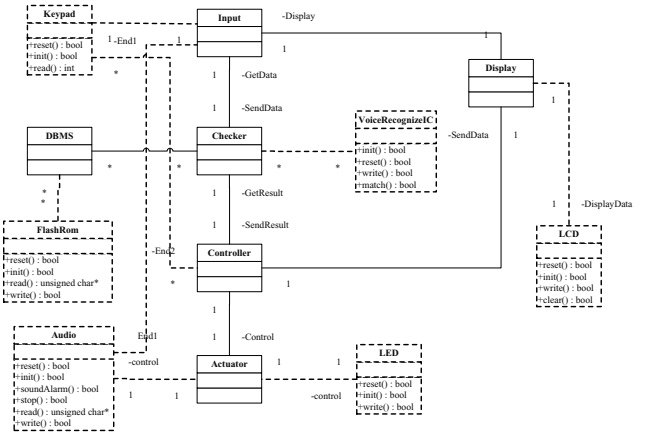

Fig. 2. Class Diagram with Deployment for Entrance Guard System

The three UML diagrams extended for real-time embedded software specification are as follows.

- *Class Diagrams with Deployment*: A deployment relation is used for specifying a hardware object on which a software object is deployed. There are two types of methods, namely event-triggered and time-triggered that are used to model 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 formal Petri net models that are used for scheduling.

For our running EGS example, some of the above diagrams are shown in Figures 2, 3, and 4, respectively.

UML is well-known for its informal and general-purpose semantics. The enhancements described above are an effort at formalizing semantics preciseness such that there is little ambiguity in user-specified models that are input to VERTAF. Furthermore, design guidelines are provided to a user such that the goal of correct-by-construction can be achieved. Typical guidelines are given here.

- Hardware deployments are desirable as they reflect the system architecture in which the generated real-time embedded software will execute and thus generated code will adhere to designer intent more precisely.

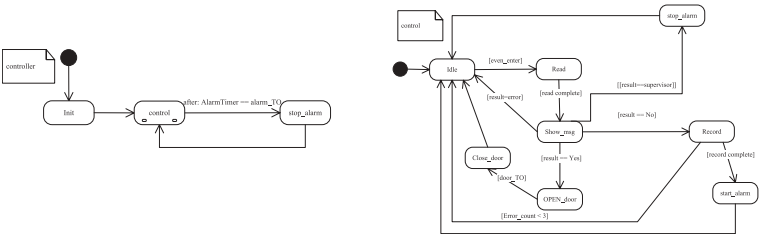

Fig. 3. Timed Statecharts for Controller in Entrance Guard System

- If the behavior of an object cannot be represented by a simple statechart that has no more than four levels of hierarchy, then decompose the object.
- To maximize flexibility, a sequence diagram can represent one or more use-case scenarios. Overlapping behavior among scenarios often results in significant redundancy in sequence diagrams, hence either control structures may be used in a sequence diagram or a set of non-overlapping sequence diagrams may be inter-related with precedence constraints.
- Ensure the logical correctness of the relationships between class diagram and statecharts and between statecharts and sequence diagrams. The former relationship is represented by actions and events in statecharts that correspond to object methods in class diagram. The latter relationship is represented by state-markers in sequence diagrams that correspond to statechart states.

The set of UML diagrams input by a user, including a class diagram with deployments, a timed statechart corresponding to each class, and a set of extended sequence diagrams, constitutes the requirements for the real-time embedded software to be designed and verified by VERTAF. The formal definition of a system model is as follows.

Definition 1. Real-Time Embedded Software System Model
Given a class diagram $D_{class} = \langle C, \delta \rangle$*, a statechart* $D_{schart}(c) = \langle Q, q_0, \tau \rangle$ *for each class* c *in* C*, and a set of sequence diagrams* $\{D_{seq} | D_{seq} = \langle C', M \rangle, C' \subseteq C\}$*, where* C *is a set of classes,* δ *is the mapping for inter-class relationships and deployments,* Q *is a set of states,* q_0 *is an initial state,* τ *is a transition relation between states, and* M *is a set of messages, a real-time embedded software system* S *is defined as a set of objects as specified in* D_{class}*, the behavior of which is represented by the individual statecharts* $D_{schart}(c)$*, and which interact with each other by sending/receiving messages* $m \in M$ *as specified in the set of sequence diagrams* $\{D_{seq}\}$*. A formal behavior model of the system* S *is defined as the parallel composition of the set of statecharts along with the behavior represented by the sequence diagrams. Notationally,* $D_{schart}(c_0) \times \ldots \times D_{schart}(c_{|C|}) \times B(D^1_{seq}, \ldots, D^k_{seq})$ *denotes the system behavior semantics, where* B *is the scheduler ETA as formalized in Section 2.2.*

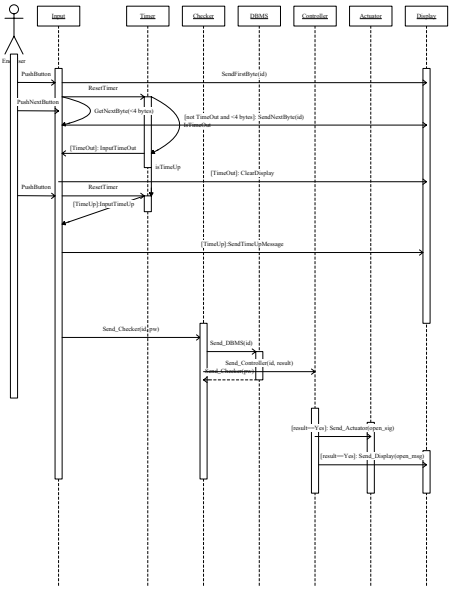

Fig. 4. An Extended Sequence Diagram for Entrance Guard System

2.2 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) [11], [12] 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) [28] with real-time scheduling [18] is applied, which requires *Complex Choice Petri Nets* (CCPN) and set of independent real-time tasks as system specification models, respectively. EQSS 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.

In order to apply the above scheduling algorithms, we need to map the user-specified UML models into Petri nets, RTPN or CCPN. RTPN enhances the standard Petri net with code execution characteristics associated with transitions. Given a standard Petri net $N = \langle P, T, \phi \rangle$, where P is a set of places,

T is a set of transitions, and ϕ is a weighted flow relation between places and transitions, $N_R = \langle N, \chi, \pi \rangle$ is an RTPN, where χ maps a transition t to its worst-case execution time α_t and deadline β_t and π is the period for N_R. CCPN allows non-free choices at transitions [28], but does not allow the computations from a branch place to synchronize at some later place. Further, CCPN only allows a loop that has at least a single token in some place along the loop. These restrictions imposed by CCPN also apply to RTPN and are set mainly for synthesizability. Here, we briefly describe how RTPN and CCPN models are generated automatically from user-specified UML sequence diagrams, through a case-by-case construction.

1. A message in a sequence diagram is mapped to a set of Petri net nodes, including an incoming arc, a transition, an outgoing arc, and a place. If it is an initial message, no incoming arc is generated. If a message has a guard, the guard is associated to the incoming arc.
2. For each set of concurrent messages in a sequence diagram, a fork transition is first generated, which is then connected to a set of places that lead to a set of message mappings as described in Step (1) above.
3. If messages are sent in a loop, the Petri-nets corresponding to the messages in the loop are generated as described in Step (1) and connected in the given sequential order of the messages. The place in the mapping of the last message is identified with the place in the mapping of a message that precedes the loop, if any. This is called a branch place. The loop iteration guard is associated with the incoming arc of the first message in the loop, which is also an outgoing arc of the branch place. Another outgoing arc of the branch place points to a transition outside the loop, which corresponds to the message that succeeds the loop.
4. Different sequence diagrams are translated to different Petri-nets. If a Petri net has an ending transition which is the same as the initial transition of another Petri net, they are concatenated by merging the common transition.
5. Sequence diagrams that are inter-related by precedence constraints are first translated individually into independent Petri nets, which are then combined with a connecting place, that may act as a branch place when several sequence diagrams have a similar precedent.
6. An ending transition is appended to each generated Petri-net because otherwise there will be tokens that are never consumed resulting in infeasible scheduling.

By applying the above mapping procedure, all user-specified sequence diagrams are translated and combined into a compact set of Petri nets. All kinds of temporal constraints that appear in the sequence diagrams such as time-out, time interval between two events (sending and receiving of messages), periods and deadlines associated with a message, and timing guards on messages are translated into guard constraints on arcs in the generated Petri nets. This set of RTPN or CCPN is then input to QDS or EQSS, respectively, for scheduling. Details on the scheduling procedures can be found in [11], [12], and [28]. The

basic strategy is to decompose each Petri net into conflict-free components that are scheduled individually for satisfaction of memory constraints. A conflict-free component is a reduction of a Petri net into one without any branch place. This is EQSS. QDS applies EQSS first and then because the resulting memory satisfying schedules may have some sequencing flexibilities, they are taken advantage of for satisfaction of temporal constraints. Finally, we have a set of feasible schedules, each of which corresponds to a particular behavior configuration of the system. A behavior configuration of a system is a feasible computation that results from the concurrent behaviors of the conflict-free components of its constituent Petri nets. For example, a system with two Petri nets, N_1 and N_2, which have two conflict-free components each, namely N_{11},N_{12}, and N_{21}, N_{22}, can have totally at most four different behavior configurations: $N_{11}||N_{21}$, $N_{12}||N_{21}$, $N_{11}||N_{22}$, and $N_{12}||N_{22}$.

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. Further, for verification purposes, an extended timed automaton is also generated by following the set of transition sequences. For uniformity, this scheduler automaton can be viewed as a timed statechart for the generated scheduler class and thus the scheduler is just another object in the system. Code generation becomes a lot easier with this uniformity.

For our running EGS 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 EGS is schedulable.

2.3 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) [29], 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.

Automation in formal verification of user-specified UML models of real-time embedded software is achieved in VERTAF by the following implementation mechanisms.

1. User-specified timed statecharts are automatically mapped to a set of ETA.
2. User-specified extended sequence diagrams are automatically mapped to a set of Petri nets that are scheduled and then a scheduler ETA is automatically generated.
3. Using the state-graph merge manipulator in SGM, all the ETA resulting from the above two steps are merged into a single state-graph representing the global system behavior.
4. User-specified OCL properties and system-defined properties are automatically translated into TCTL specification formulas.
5. The system state-graph and the TCTL formulas obtained in the previous two steps are then input to SGM for model checking.
6. When a property is not satisfied, SGM generates a counterexample, which is then automatically translated into a UML sequence diagram representing an erratic trace behavior of the system. This approach provides a seamless interface to VERTAF users such that the formal models are all hidden and the users need to interact only with what they have specified in UML models.

Design complexity is a major issue in formal verification, which leads to unmanageable and exponentially large state-spaces. Both engineering paradigms and scientific techniques are applied in VERTAF to handle the state-space size explosion issue. The applied techniques include (1) Model Construction Guidelines, (2) Architectural Abstractions, (3) Functional Abstractions, and (4) State-Space Reductions. Due to page-limit, we have not elaborated on the individual techniques.

For our running EGS example, the ETA for each statechart were generated and then merged with the scheduler ETA. For illustration, we show in Figure 5 the ETA that is generated by VERTAF corresponding to the controller statechart of Figure 3. The other 6 ETA are omitted due to page-limit. 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.

2.4 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.

The main issue in this phase occurs when a software class is not deployed on any hardware component or not deployed on any specific hardware device type,

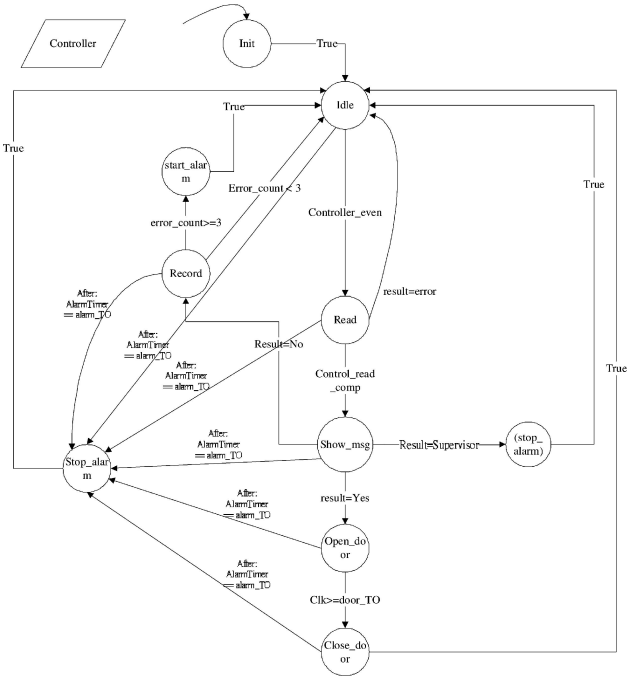

Fig. 5. ETA for Controller in EGS

for example the type of microcontroller to be used is not specified. Currently, VERTAF adopts an interactive approach whereby the designer is warned of this lack of information and he/she is requested to choose from a list of available compatible device types for the deployment. An automatic solution to this issue is not feasible because estimates are not easy without further information about the non-deployed software classes.

Another 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 also warned in case of such conflicts.

2.5 Code Generation

There are basically three issues in this phase including hardware portability, software portability, and temporal correctness. We adopt a 3-tier approach for code generation: a hardware abstraction layer, an OS with middleware layer, and a scheduler with temporal monitor, which solves the above three issues, respectively. Currently supported underlying hardware platforms include ARM-based, StrongARM-based, 8051-based, and Lego RCX-based Mindstorm systems. For hardware abstraction, VERTAF supports MicroHAL and the embedded version of POSIX. For OS, VERTAF supports MicroC/OS, Linux, and eCOS. For

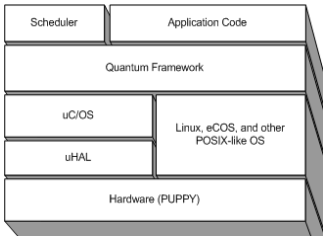

Fig. 6. Multi-Tier Code Architecture

middleware, VERTAF is currently based on the Quantum Framework [38]. For 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. As shown in Figure 6, this multi-tier approach decouples application code from the OS through the middleware and from the hardware platform through the OS and hardware abstraction layer.

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 also takes the role of a real-time executive kernel.

For our running example, the final application code consisted of 6 activeobjects derived from the statecharts and 1 activeobject representing the scheduler. Makefiles were generated for linking in the API of the 6 hardware classes and configuration files were generated for the StrongARM microprocessor with MicroC/OS II and embedded Linux. There were totally 2,300 lines of C code, out of which the designer had to write only around 300 lines of code.

3 AICC Cruiser Application

An application developed with VERTAF is AICC (Autonomous Intelligent Cruise Controller) system application [7], which had been developed and installed in a Saab automobile by Hansson et al. The AICC system can receive information from road signs and adapt the speed of the vehicle to automatically follow speed limits. Also, with a vehicle in front cruising at lower speed the AICC adapts the speed and maintains safe distance. The AICC can also receive information from the roadside (e.g. from traffic lights) to calculate a speed profile which will reduce emission by avoiding stop and go at traffic lights. The system architecture consisting of both hardware (HW) and software (SW) is shown in Figure 7. The software development methodology used in [7] is based on sets of interconnected so-called software circuits. Each software circuit has a set of input connectors where data are received and a set of output connectors where data

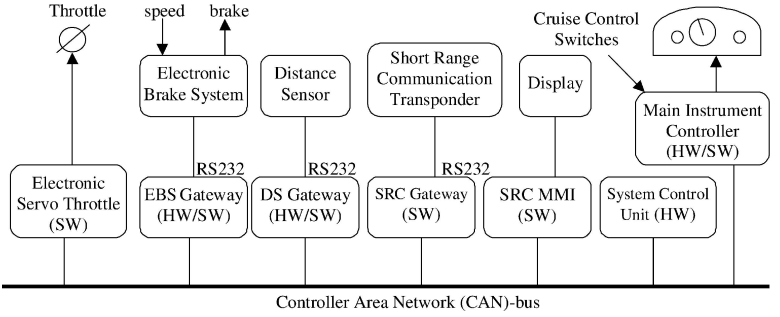

Fig. 7. AICC System Architecture

are produced. We model the software circuits in [7] as active application domain objects in VERTAF.

As shown in Figure 8, there are five domain objects specified by the designer of AICC for implementing a Basement system. Basement is a vehicle's internal real-time architecture developed in the Vehicle Internal Architecture (VIA) project [7], within the Swedish Road Transport Informatics Programme. As observed in Figure 8, each object may correspond (map) to one or more tasks. The tasks and the Call-Graph are as shown in Table 1 and Figure 8, respectively. There are totally 12 tasks performed by 5 application domain objects. There were 21 application framework objects specified by the designer. Totally, 26 objects were in the final program code generated. The average integration time per object was 0.5 day and the average learning time was amortized as 0.1 day for each designer using the framework. Without using the framework, the average integration time was 2 days for each object. This application took 5 days for 3 real-time system designers using VERTAF. The same application took the same designers 20 days to complete development a second time. The significant decrease in design time was due to the high degree of automation in VERTAF.

4 Conclusion

An object-oriented component-based application framework, called VERTAF, was proposed for real-time embedded systems application development. 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 since new specification languages, scheduling algorithms, etc. can easily be integrated into it. Future extensions will include support for share-driven scheduling algorithms. More applications will also be developed using VERTAF. VERTAF will be enhanced in the future by considering more advanced features of real-time applications, such as: network delay, network protocols, and on-line task scheduling. Performance

Table 1. AICC Tasks

Index	Task Description	Object	p	e	d
1	Traffic Light Info	SRC	200	10	400
2	Speed Limit Info	SRC	200	10	400
3	Proceeding Vehicle Estimator	ICCReg	100	8	100
4	Speed Sensor	ICCReg	100	5	100
5	Distance Control	ICCReg	100	15	100
6	Green Wave Control	ICCReg	100	15	100
7	Speed Limit Control	ICCReg	100	15	100
8	Coordination & Final Control	Final_Control	50	20	50
9	Cruise Switches	Supervisor	100	15	100
10	ICC Main Control	Supervisor	100	20	100
11	Cruise Info	Supervisor	100	20	100
12	Speed Actuator	EST	50	5	50

SRC: Short Range Communication, ICCReg: ICC Regulator, EST: Electronic Servo Throttle, p: period, e: execution time, d: deadline

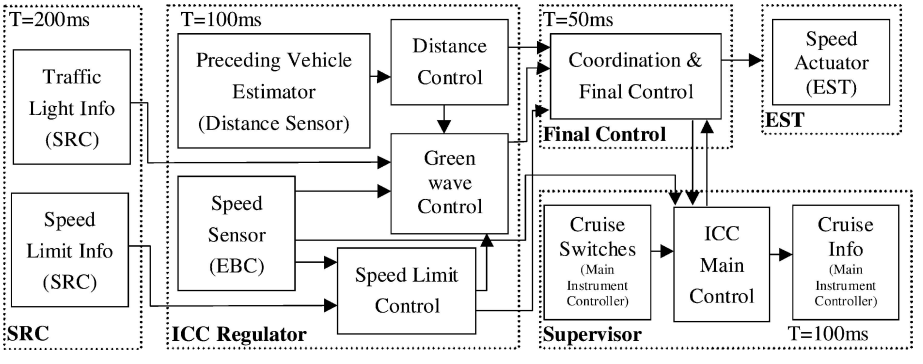

Fig. 8. AICC Call-Graph

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 into VERTAF in the future.

References

1. R. Alur and D. Dill, "Automata for modeling real-time systems," Theoretical Computer Science, Vol. 126, No. 2, pp. 183-236, April 1994.
2. T. Amnell, E. Fersman, L. Mokrushin, P. Petterson, and W. Yi, "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 (FORMATS, Marseille, France), September 2003.

3. E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in Proceedings of the Logics of Programs Workshop, LNCS Vol. 131, pp. 52-71, Springer Verlag, 1981.
4. E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking, MIT Press, 1999.
5. B. P. Douglass, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Addison Wesley Longman, Inc., Reading, MA, USA, November 1999.
6. M. Fayad and D. Schmidt, "Object-oriented application frameworks," Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, October 1997.
7. H. A. Hansson, H. W. Lawson, M. Stromberg, and S. Larsson, "BASEMENT: A distributed real-time architecture for vehicle applications," Real-Time Systems, Vol. 11, No. 3, pp. 223-244, 1996.
8. P.-A. Hsiung, "RTFrame: An object-oriented application framework for real-time applications," in Proceedings of the 27th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'98), pp. 138-147, IEEE Computer Society Press, September 1998.
9. P.-A. Hsiung, "Embedded software verification in hardware-software codesign," Journal of Systems Architecture - the Euromicro Journal, Vol. 46, No. 15, pp. 1435-1450, Elsevier Science, November 2000.
10. P.-A. Hsiung and S.-Y. Cheng, "Automating formal modular verification of asynchronous real-time embedded systems," in Proceedings of the 16th International Conference on VLSI Design, (VLSI'2003, New Delhi, India), pp. 249-254, IEEE CS Press, January 2003.
11. P.-A. Hsiung and C.-Y. Lin, "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 (CODES+ISSS'2003, Newport Beach, CA, USA), pp. 114-119, ACM Press, October 2003.
12. P.-A. Hsiung, C.-Y. Lin, and T.-Y. Lee, "Quasi-dynamic scheduling for the synthesis of real-time embedded software with local and global deadlines," in Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'2003, Tainan, Taiwan), February 2003.
13. A. Knapp, S. Merz, and C. Rauh, "Model checking timed UML state machines and collaboration," in Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS Vol. 2469, pp. 395-414, Springer Verlag, September 2002.
14. T. Kuan, W.-B. See, and S.-J. Chen, "An object-oriented real-time framework and development environment," in Proceedings OOPSLA'95 Workshop #18, 1995.
15. S. Kodase, S. Wang, and K. G. Shin, "Transforming structural model to runtime model of embedded software with real-time constraints," in Proceedings of Design, Automation and Test in Europe Conference, Munich, Germany, pp. 170-175, March 2003.
16. L. Lavazza, "A methodology for formalizing concepts underlying the DESS notation," Software Development Process for Real-Time Embedded Software Systems, EUREKA-ITEA project (http://www.dess-itea.org), D 1.7.4, December 2001.
17. W.-S. Liao and P.-A. Hsiung, "FVP: A formal verification platform for SoC," in Proceedings of the 16th IEEE International SoC Conference, Portland, Oregon, USA, IEEE CS Press, September 2003.

18. C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real time environment," Journal of the Association for Computing Machinery, Vol. 20, pp. 46-61, January 1973.
19. D. de Niz and R. Rajkumar, "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, San-Diego, California, USA, pp. 133-143, June 2003.
20. J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in Proceedings of the International Symposium on Programming, LNCS Vol. 137, pp. 337-351, Springer Verlag, 1982.
21. J. Rumbaugh, G. Booch, and I. Jacobson, The UML Reference Guide, Addison Wesley Longman, 1999.
22. M. Samek, Practical Statecharts in C/C++ Quantum Programming for Embedded Systems, CMP Books, 2002.
23. D. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software," Handbook of Programming Languages, Vol. I, 1997.
24. W.-B. See and S.-J. Chen, "Object-oriented real-time system framework," in Domain-Specific Application Frameworks (M. E. Fayad and R. E. Johnson, eds.), ch. 16, pp. 327-338, John Wiley, 2000.
25. B. Selic, "Modeling real-time distributed software systems," in Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems, pp. 11-18, 1996.
26. B. Selic, "An efficient object-oriented variation of the statecharts formalism for distributed real-time systems," in Proceedings of the IFIP Conference on Hardware Description Languages and Their Applications, 1993.
27. B. Selic, G. Gullekan, P. T. Ward, Real-time Object Oriented Modeling, John Wiley and Sons, Inc., 1994.
28. F.-S. Su and P.-A. Hsiung, "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, Colorado, USA), pp. 211-216, ACM Press, May 2002.
29. F. Wang and P.-A. Hsiung, "Efficient and user-friendly verification," IEEE Transactions on Computers, Vol. 51, No. 1, pp. 61-83, January 2002.
30. S. Wang, S. Kodase, and K. G. Shin, "Automating embedded software construction and analysis with design models," in Proceedings of International Conference of Euro-uRapid, Frankfurt, Germany, December 2002.