VERTAF: An Application Framework for the Design and Verification of Embedded Real-Time Software

Pao-Ann Hsiung, *Member*, *IEEE*, Shang-Wei Lin, Chih-Hao Tseng, Trong-Yen Lee, Jih-Ming Fu, and Win-Bin See, *Member*, *IEEE*

Abstract—The growing complexity of embedded real-time software requirements calls for the design of reusable software components, the synthesis and generation of software code, and the automatic guarantee of nonfunctional properties such as performance, time constraints, reliability, and security. Available application frameworks targeted at the automatic design of embedded real-time software are poor in integrating functional and nonfunctional requirements. To bridge this gap, we reveal the design flow and the internal architecture of a newly proposed framework called *Verifiable Embedded Real-Time Application Framework* (VERTAF), which integrates software component-based reuse, formal synthesis, and formal verification. A formal UML-based *embedded real-time object* model is proposed for component reuse. Formal synthesis employs quasi-static and quasi-dynamic scheduling with automatic generation of multilayer portable efficient code. Formal verification integrates a model checker kernel from SGM, by adapting it for embedded software. The proposed architecture for VERTAF is component-based and allows plug-and-play for the scheduler and the verifier. Using VERTAF to develop application examples significantly reduced design effort and illustrated how high-level reuse of software components combined with automatic synthesis and verification can increase design productivity.

Index Terms—Application framework, code generation, embedded real-time software, formal synthesis, formal verification, scheduling, software components, UML modeling.

1 Introduction

Driven by the demand for new and complicated features, embedded systems are becoming more complex, which makes their correctness very difficult to verify. Further, embedded systems are also becoming more pervasive, which makes guaranteeing their correctness all the more important. Currently, the design of embedded real-time software is supported partially by modelers, code generators, analyzers, schedulers, and frameworks [3], [14], [27], [28], [29], [30], [31], [34], [35], [36], [37], [40], [47], [48], [49], [50], [51], [52], [53], [58], [59]. Nevertheless, the technology for a completely integrated design and verification environment is still relatively immature. Furthermore, the methodologies for design and for verification are also poorly integrated, relying mainly on the experiences of embedded software engineers. 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 techniques such as model checking can be realized in the form of an integrated design environment targeted at the acceleration of embedded real-time 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* [17], which is a "semicomplete" 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 based on system requirements, rather than the designer. Other issues encountered in architecting an application framework for embedded real-time software are as follows:

1. To allow software component reuse, how do we define the syntax and semantics of a reusable component? How must a designer specify system requirements such that they can be automatically synthesized and verified? How can the existing reusable components be integrated with the user-specified components into a feasible working system?

- *P.-A. Hsing, S.-W. Lin, and C.-H. Tseng are with the Department of Computer Science and Information Engineering, National Chung Cheng University, 160, San-Hsing, Min-Hsiung, Chiayi, Taiwan-621, ROC. E-mail: hpa@computer.org, {lsw92, tch92}@cs.ccu.edu.tw.*
- *T.-Y. Lee is with the Department of Electronic Engineering, National Taipei University of Technology, 1 Sec. 3, Chung-Hsiao E. Rd. Taipei 10608, Taiwan ROC. E-mail: tylee@ntut.edu.tw.*
- *J.-M. Fu is with the Department of Electronic Engineering, Cheng Shiu University, 840, Chengcing Rd., Niaosong, Kaohsiung County 833, Taiwan, ROC. E-mail: marsfuh@csu.edu.tw.*
- *W.-B. See is with the Aerospace Industrial Development Corporation, #27-3 Dahung Street, Taichung, Taiwan, 407, ROC. E-mail: winbinsee@ms.aidc.com.tw.*

Manuscript received 18 Feb. 2003; revised 7 July 2004; accepted 19 July 2004. Recommended for acceptance by B. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118302.

2. What is the control-data flow of the automatic design and verification process? When do we verify and when do we schedule a system under design?
3. What kinds of model can be used for each design phase, such as scheduling and verification of a system under design?
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 the handling capabilities of a machine used for designing the system?
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) [45] 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, as described in Section 3.1.4.
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 [28].
3. *System Models*: For scheduling, we use variants of *Petri Nets* (PN) [30], [31] and for verification, we use *Extended Timed Automata* (ETA) [38], both of which are automatically generated from user-specified UML models that follow our restrictions and guidelines. The generation procedures are detailed in Sections 3.2 and 3.3.
4. *Design Automation*: For synthesis, we employ quasi-static and quasi-dynamic scheduling methods [30], [31] that generate program schedules for a single processor. For verification, we employ symbolic model checking [10], [11], [42] 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, as described in Section 3.3.
5. *Portable Efficient Multilayered Code*: For portability, a multilayered approach is adopted in code generation. To compensate 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. Sections 3.4 and 3.5 provide code generation details.

In summary, this work illustrates how an application framework may integrate all the above proposed design and verification solutions for satisfying both functional and nonfunctional requirements. Our implementation has resulted in a *Verifiable Embedded Real-Time Application Framework* (VERTAF) whose features include formal modeling of embedded real-time 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 related work on the construction of application frameworks for embedded real-time system design. Section 3 describes the details of the design and verification flow in VERTAF along with an illustration example. Section 4 describes the five software components of VERTAF. Section 5 presents the experimental results of two application examples. Section 6 gives the final conclusions with some future work.

2 Previous Work

Though object-oriented technology has been applied to the design of real-time systems in several areas, such as language design [1], [7], [20], [32], verification and analysis [8], [19], distributed system design [22], [33], [48], [50], [51], [52], and embedded system design [47], [53], [60], there has been very little work on the development of application frameworks for real-time application design. Two known frameworks are *Object-Oriented Real-Time System Framework* (OORTSF) [35], [49] and SESAG [27], which are simple frameworks that have been applied to avionics software development. In these frameworks, some design patterns related to real-time application design were proposed and code automatically generated. Nevertheless, there are still some scheduling and real-time synchronization issues not addressed such as asynchronous event handling and protocol interfacing. VERTAF is an enhanced version of SESAG, incorporating software component technology, synthesis, formal verification, and standards such as UML.

Other related toolsets for the design and verification of systems include the B-toolkit [4], SCR toolset [24], NIMBUS [57], and SCADE Suite [16]. The B-toolkit takes *abstract machines* as system models and applies theorem proving for proof-obligation generation and verification. The SCR toolset uses the SPIN model checker, PVS-based TAME theorem prover, a property checker, and an invariant generator for the formal verification of a real-time embedded system specified using the SCR tabular notation. It supports the generation of test cases through the TVEC toolset. NIMBUS is a specification-based prototyping framework for embedded safety-critical systems. It allows execution of software requirements expressed in RSML with various models of the environment such as physical hardware, RSML models or user input scripts. NIMBUS supports model checking through a variety of model checkers and a framework based on Tame by SCR, as well as theorem proving using PVS. Lustre-based [21] SCADE Suite from Esterel Technologies uses Safe State Machines (SSM) for requirement specification and automatically generates DO-178B Level A [13] compliant and verified C/Ada code for avionics systems. Nondeterminism is not allowed by SSM in SCADE.

Worldwide research projects targeting embedded real-time embedded software design include the MoBIES project [36], [40], [59] supported by USA DARPA, the HUGO project [34] by Germany's Ludwig-Maximilians-Universität München, the DESS project [37] supported by Europe's EUREKA-ITEA, and the TIMES project [3] by the Uppsala University of Sweden. In the DARPA supported MoBIES (*Model-Based Integration of Embedded Systems*) project, there are several subprojects that cover varied parts of the embedded software design process. For example, Kodase et al. [36] and Wang et al. [59] proposed AIRES (*Automatic Integration of Reusable Embedded Software*), which focuses on automatically generating a runtime model from a structural model through several metamodels: software architecture, platform, runtime, and performance metamodels. AIRES has been effectively applied to avionics and automotive applications. Further, de Niz and Rajkumar proposed *Time Weaver* [40], which is a software-through-models framework that focuses on capturing para-functional properties into models of different semantic dimensions such as event flow, deployment, timing, fault tolerance, modality, and concurrency.

Knapp et al. have been developing HUGO [34] that focuses on model checking statecharts against collaborations. Code generation is also possible by HUGO, but scheduling is not performed and, thus, the generated code might not satisfy user-specified temporal constraints. The DESS project by EUREKA-ITEA [37] is another effort at defining a methodology for the design of real-time and embedded systems, which provides guidelines for incorporating various kinds of tools into the design process and how formal methods may be exploited. Neither real implementation of the concepts nor any toolset is provided by DESS. Lastly, TIMES [3] is a set of tools for the symbolic schedulability analysis and code synthesis of predictable real-time systems. No features of embedded systems are considered in TIMES and the input model is a set of timed automata and not the engineer-friendly UML model.

VERTAF differs from academic research-oriented project application frameworks and from commercial code generating frameworks mainly in the following aspects.

- *System Models*: In contrast to the use of ad hoc system models in AIRES, Time Weaver, B-toolkit, SCR toolset, NIMBUS, and SCADE, VERTAF uses standard models such as UML with stereotype extensions for design specification, Petri nets for synthesis, and extended timed automata for verification, which allow compatibility with other tools.
- *Formal Synthesis*: Synthesis consists of two phases: scheduling and code generation. Commercial and academic application frameworks either rely on manual refinements or automatically generate embedded software code without guarantee on satisfaction of temporal or spatial constraints, whereas VERTAF tries to find a schedule that satisfies user-defined timing constraints. If no feasible schedule exists, VERTAF illustrates the location of constraint violations in the original user-specified UML diagrams.
- *Formal Verification*: Commercial tools due to product marketing strategies and academic application frameworks due to lack of interdiscipline expertise normally leave the verification of generated embedded code to the user who invokes another tool for verification. The problem is that there is a gap between design and verification and this causes problems when some design errors are detected in verification, but cannot be easily illustrated in the original design models. Similar to tools such as Statemate and Esterell, VERTAF has a built-in model checker, hence does not have this problem.

Due to the requirement for great precision in the specification and design of real-time embedded systems, formal verification is especially desirable and feasible in proving correctness of such systems. Currently, there is no known application framework that incorporates automatic formal verification into its design process. VERTAF takes this step and, thus, needs to solve several issues such as methodology flow, automation procedure, and abstraction methods as described in Section 1. There are several formal verification methods that can be applied, such as model checking, process algebra, theorem proving, and other logic-related techniques. In VERTAF, we use *model checking* [10], [11], [42], which automates formal verification and generates counterexamples on property violation. Given a real-time system description S and a temporal property specification ϕ, model checking answers whether S satisfies ϕ. A counterexample is produced if the property is not satisfied. A real-time system is modeled by a set of *Extended Timed Automata* (ETA) [38], which extends timed automata [2] with discrete variables and by allowing synchronization between transitions of different automata through common labels. A temporal property is specified in *Timed Computation Tree Logic* (TCTL) [25].

3 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 Fig. 1, VERTAF provides solutions to the various issues introduced in Section 1.

In Fig. 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 refer to the two phases as front-end and back-end phases. The front-end phase is further divided into three subphases, namely, UML modeling phase, embedded real-time software scheduling phase, and formal verification phase. There are two subphases in the back-end phase, namely, component mapping phase

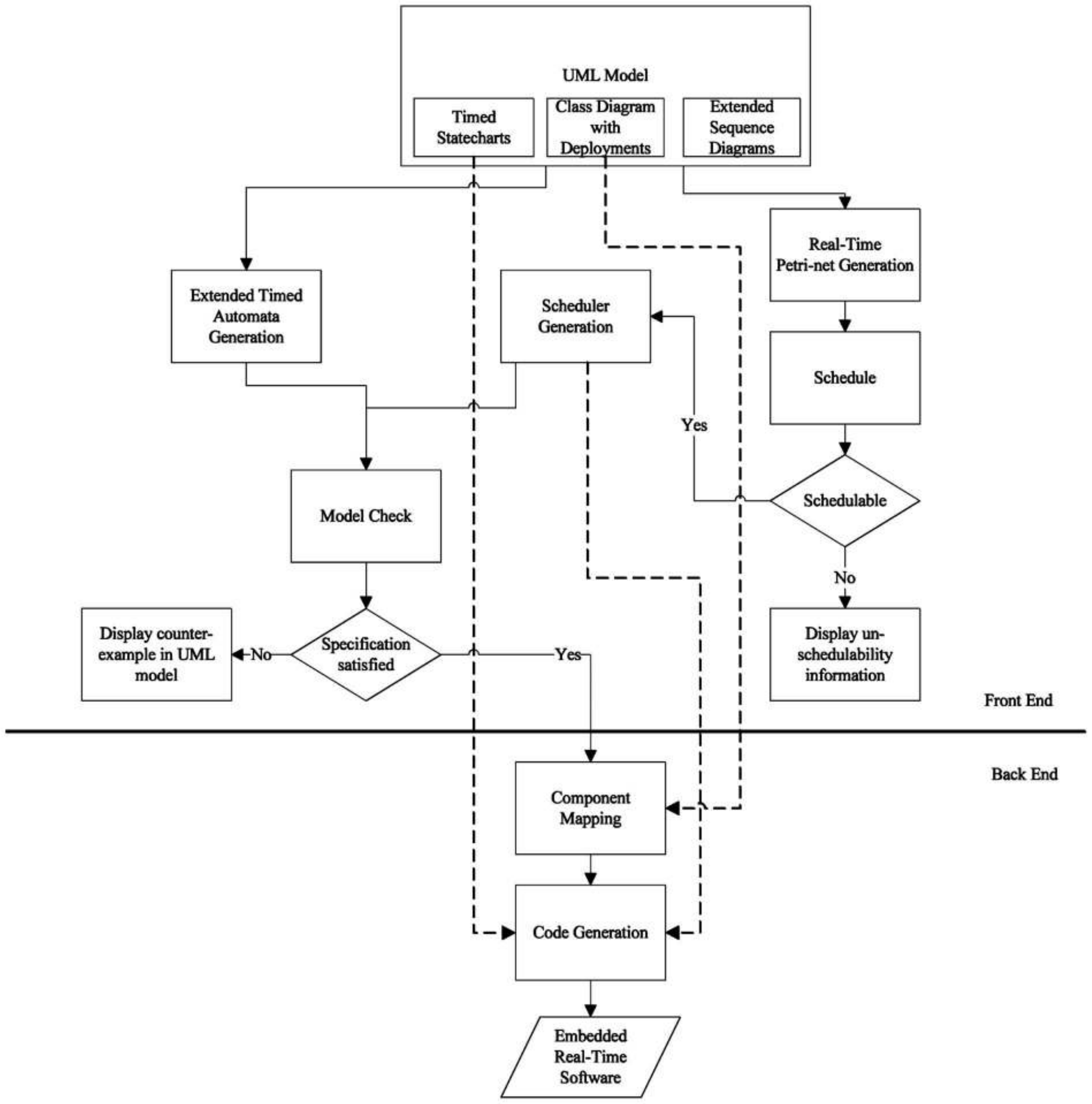

UML Model
Timed Statecharts
Class Diagram with Deployments
Extended Sequence Diagrams
Real-Time Petri-net Generation
Extended Timed Automata Generation
Scheduler Generation
Schedule
Yes
Model Check
Schedulable
No
Display counter-example in UML model
No
Specification satisfied
Yes
Display un-schedulability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

Fig. 1. Design and Verification Flow of VERTAF.

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.

3.1 UML Modeling

UML [45] is one of the most popular modeling and design languages in the industry. It has become the de facto standard. 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 specializations of it are always required for targeting any specific application domain. In VERTAF, the three UML diagrams are restricted as well as enhanced, with guidelines for specifying synthesizable and verifiable system models.

3.1.1 *Class Diagrams with Deployment*

A designer-specified class diagram represents the core part of automatically generated software code and architecture. In VERTAF, we classify classes into two types: software classes that are either specified by a designer from scratch or reused from library components and hardware classes that represent supported hardware components. Besides the relationships found in a typical class diagram such as

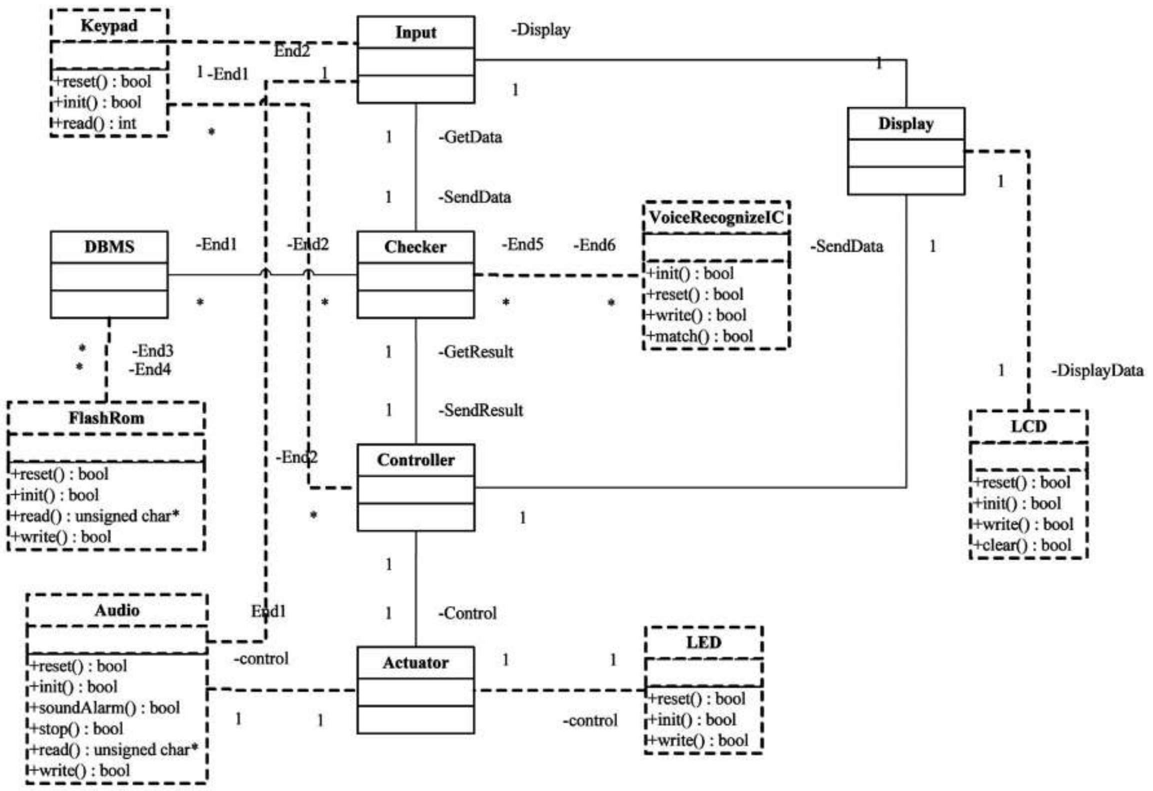


Fig. 2. Class diagram with deployments for entrance guard system.

aggregation, generalization, and association, we introduce a deployment relationship between a software class and a hardware class, which can be used to specify a hardware component on which a software object is deployed. For example, a control program is deployed on 8051 microcontroller or display software is deployed on an LCD.

In VERTAF, besides *event-triggered methods*, another type called *time-triggered methods* is used to model real-time tasks. Keywords such as *period* and *deadline* distinguish time-triggered methods from event-triggered ones. Time-triggered methods can be *one-shot* or *periodic* and are started, stopped, and restarted by actions in the statechart that represents the object's behavior.

For the EGS example, the class diagram with deployments is shown in Fig. 2, where the dotted boxes represent hardware classes, the regular ones are software classes, the dotted lines represent deployments, and the solid lines represent associations. There are six software classes which are deployed on six hardware classes.

3.1.2 Timed Statecharts

A UML statechart represents the behavior of an object and is used for generating the behavioral code architecture of the object. Besides the original time-out specification in statecharts, we allow more complex temporal behavior specifications such as multiple clocks, and clock check and reset. These enhancements are the same as those found in timed automata [2].

Another addition to the standard statecharts is the set of keywords associated with time-triggered methods, namely, *start*, *stop*, and *reset*, which starts, stops, or restarts a time-triggered method, respectively. A keyword *time-out* is introduced to specify the temporal deadline for an action in a statechart.

Fig. 3 illustrates a timed statechart for the Controller Class in EGS example. It has two levels of hierarchy to model timer interrupts. Five other statecharts, one for each of the five classes (Fig. 2), are omitted here due to space constraints.

3.1.3 Extended Sequence Diagrams

Sequence diagrams represent the use-cases of an embedded software application and contain temporal and other information related to how a user may use the system. In VERTAF, sequence diagrams are mainly used for scheduling the different tasks performed by objects. Due to poor expressiveness in the original UML diagrams, we have defined control structures in sequence diagrams, including concurrency, conflict, and composition, which aid in formalizing the semantics and in mapping to Petri nets.

Another enhancement to the standard sequence diagrams is the *state-markers* that are inserted into the life axis of an object. A state-marker on the life axis of an object A has the same label as a state in the statechart C_A of the object A, in which the messages following the state-marker in the sequence diagram are sent or received by the object. State-markers explicitly relate a sequence diagram to the states in a set of statecharts that represent the behavior of the object instances appearing in that sequence diagram. They aid in user comprehension of the sequence diagrams

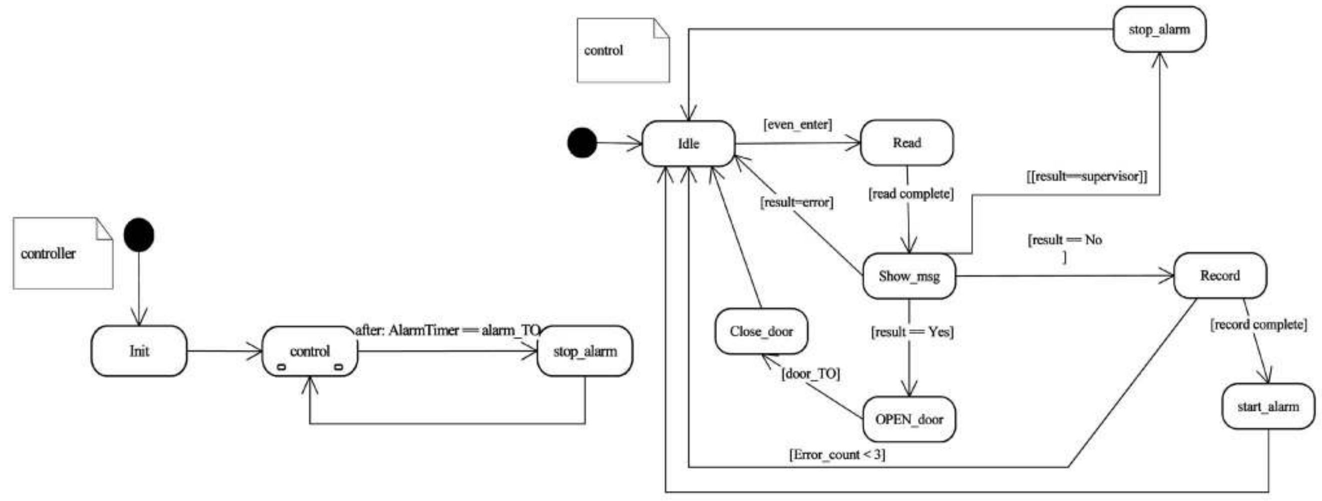


Fig. 3. Timed statechart for the controller class in EGS.

for further maintenance and also help in the scheduling process of VERTAF.

Fig. 4 illustrates one of the five sequence diagrams for EGS, which depicts a normal scenario where a legal user gains entry into the building after successful voice recognition. A state-marker *Show_msg* on the *Controller* object life axis indicates that the controller sends the *Send_Actuator()* message only when it is in the *Show_msg* state.

3.1.4 Model Preciseness and Design Guidelines

UML is well-known for its informal and general-purpose semantics. The enhancements described in the foregoing sections 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 embedded real-time software will execute and, thus, generated code will adhere to designer intent more precisely.
- 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, hence either *control structures* may be used in a sequence diagram or a set of nonoverlapping sequence diagrams may be interrelated with *precedence constraints*.
- Ensure the logical correctness of the relationships between class diagram and statecharts and between statecharts and sequence diagrams. The former is represented by actions and events in statecharts that correspond to object methods in class diagram. The latter is represented by state-markers in sequence diagrams corresponding to Statechart states.

3.1.5 System Model

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 embedded real-time software to be designed and verified by VERTAF. The formal definition of a system model is as follows.

Definition 1. *Embedded real-time software system model.*

Given a class diagram $D_{class} =< C, \delta >$, a statechart $D_{schart}(c) =< Q, q_0, \tau >$ for each class c in C, and a set of sequence diagrams $\{D_{seq} | D_{seq} =< C', M >, C' \subseteq C\}$, where C is a set of classes, δ is the mapping for 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, an *embedded real-time 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 a system S is defined as the parallel composition of all statecharts with behavior represented by sequence diagrams. Notationally, $D_{schart}(c_1) \times \ldots \times D_{schart}(c_{|C|}) \times B({D_{seq}}^1, \ldots, {D_{seq}}^k)$ denotes the system behavior semantics, where B is the scheduler ETA as formalized in Section 3.2.

3.2 Embedded Real-Time Software Scheduling

There are two issues in embedded real-time 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) [30], [31] 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.

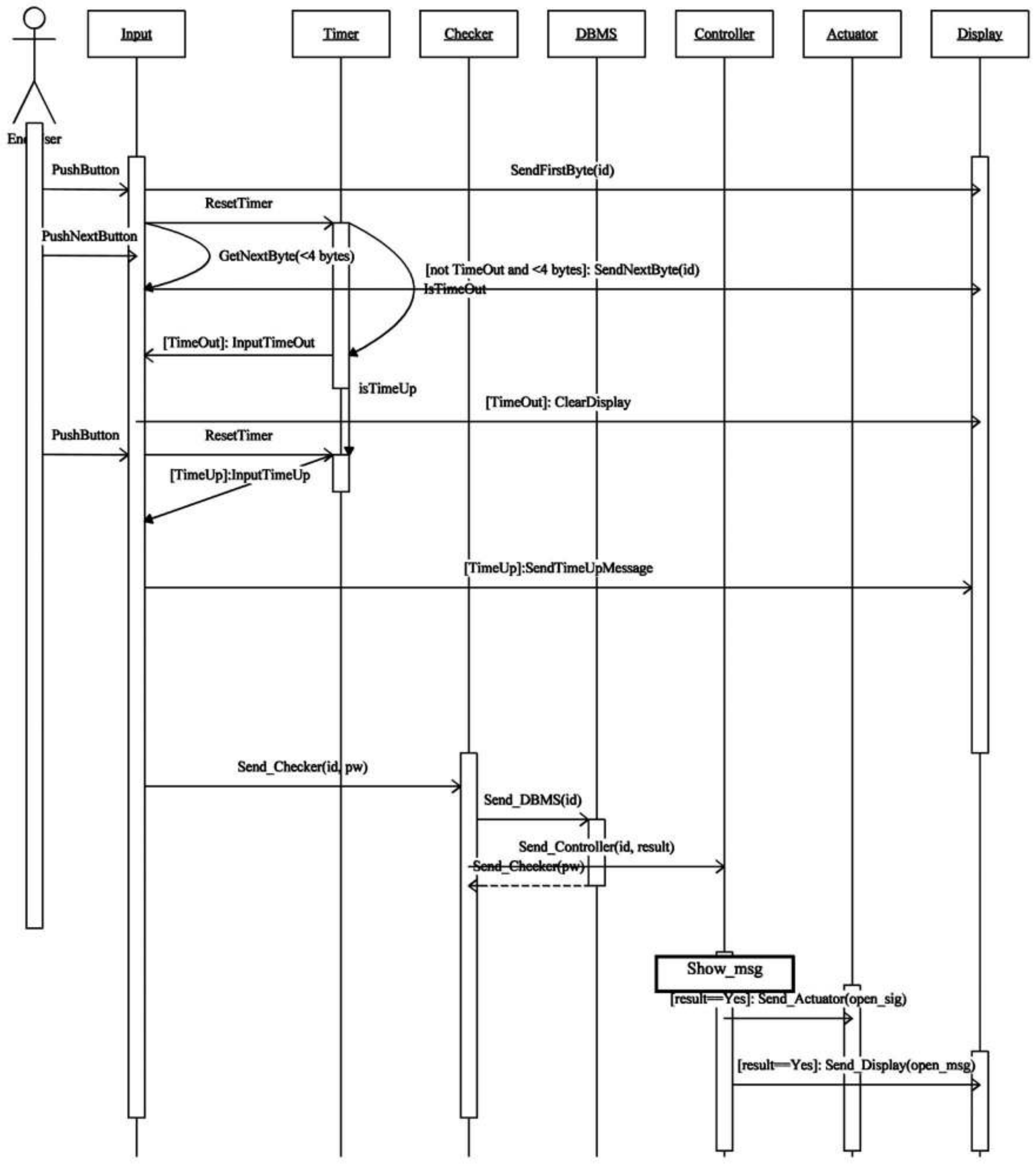


Fig. 4. A sequence diagram for EGS.

- With RTOS: *Extended quasi-static scheduling* (EQSS) [55] with *real-time scheduling* [39] 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.

QDS and EQSS are both static scheduling techniques which will be discussed later in this section after illustrating how their input Petri nets are automatically generated from UML models. There have been several work on formalizing UML semantics by translating UML diagrams into formal models such as Petri nets [6], [15], [18], [46]. However, the translated models are mostly targeted for either verification [15], [18], [46] or performance evaluation [6]. The colored Petri net and stochastic models that are most popularly used in such formalizations can become quite complex to handle. Further, the target systems for most related work are general object-oriented systems that do not consider deployments. In contrast, VERTAF uses simple RTPN/CCPN models for scheduling purposes that include real-time characteristics determined by user-given hardware deployments in the class diagram. The translated Petri nets are closer to their final embedded software implementations.

The RTPN enhances the standard Petri net with code execution characteristics associated with transitions. Given

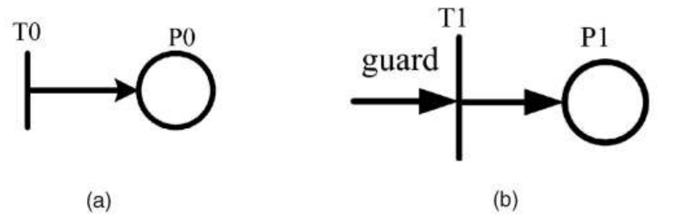


Fig. 5. Mapping a message to a Petri net. (a) PN for initial message. (b) PN for intermediate message.

a standard Petri net $N = < P, T, \phi >$, where P is a set of places, T is a set of transitions, and ϕ is a weighted flow relation between places and transitions, $N_R = < N, \chi, \pi >$ 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. Both RTPN and CCPN allow nonfree choices at transitions [55], but do not allow the computations from a branch place to synchronize at some later place. For synthesizability, both RTPN and CCPN only allow a loop that has at least a single token in some place along the loop. 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. Examples are shown in Fig. 5.
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 (Fig. 5b). Fig. 6 illustrates the mapping for two concurrent messages, SendActuator() and SetTimer(), which implement a typical time-out mechanism.
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 (Fig. 5) 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. Fig. 7 shows a typical example of a 4-byte text entry loop. A timer interrupt is also modeled by an intermediate place having an outgoing arc to a transition TimeOut1 that is fired when a previously set timer expires.
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. The two Petri nets in Fig. 8a have a common transition T1, which can be merged as shown in Fig. 8b.
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. An example is shown in Fig. 9, where each independent Petri net is in a frame. Branch place P2 is used to combine them.
6. An ending transition is appended to each generated Petri net because otherwise, there will be an infinite accumulation of nonconsumed tokens, which will result in infeasible scheduling because tokens represent memory space.

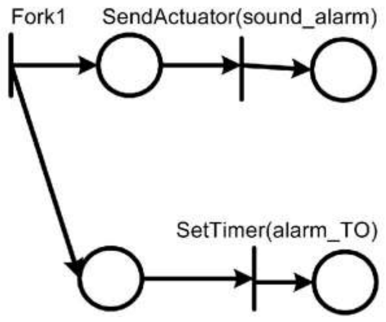


Fig. 6. Mapping concurrent messages with a fork transition in the Petri net.

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 [30], [31], [55]. 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 in total 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

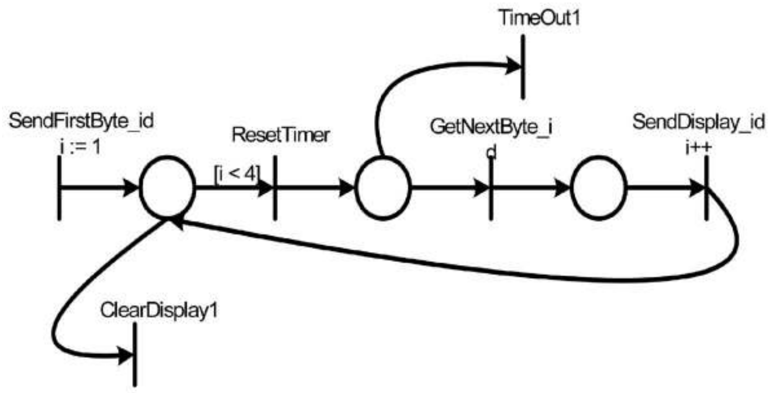


Fig. 7. Mapping a message loop with timer interrupt to a Petri net.

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, as shown in Fig. 10, 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.

3.3 Formal Verification

Embedded real-time systems often have complex control schemes and high level of concurrency among various device controllers, which often results in error conditions that require very precise timing of inputs (e.g., an input interrupt arrives at exactly the moment that a certain state transition is taking place in a firmware). Static analysis, because all possible execution sequences are considered, is much more suitable for detecting these error conditions than testing. Further, model-based design of such systems is especially suitable for formal verification because the models can be made into precise semantic representations for verification. Here, we apply *model checking* [10], [11], [42], which is an automatic state-based analysis procedure that can show if a system satisfies a temporal property or violates it in some counterexample. It requires a formal system model and a formal specification of a temporal property. UML models are mapped to ETA [38] and OCL properties to TCTL formulas [25].

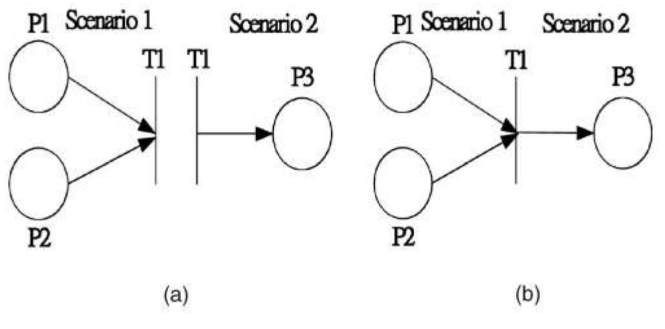


Fig. 8. Concatenating two Petri nets into one. (a) Two Petri nets. (b) Concatenated PN.

In VERTAF, as shown in Fig. 11, 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. The three types of states in statecharts are mapped into ETA entities as follows: Each basic state is mapped to an ETA state. An OR-state is mapped to the set of ETA states corresponding to the states within the OR-state and additional ETA transitions are added to account for statechart transitions that cross hierarchy levels. An AND-state is mapped to two or more concurrent ETA corresponding to the parallelism in the AND-state. Labels are used for synchronizing the concurrent ETA. Details on the hierarchy flattening scheme can be found in [34], [37]. Clock variables and constraints appearing in statecharts can be directly copied to ETA. For a time-out value of TO on a transition t of a statechart, a temporary clock variable x is used to represent a corresponding timer. Variable x is reset on all incoming ETA transitions to the mapped source state of t. A time invariant $x \leq \text{TO}$ is added to the mapped source state of t. A triggering condition $x = \text{TO}$ is added to the ETA transition that corresponds to t.

As shown in Fig. 11, once we have the set of ETA models generated from the user-specified UML statecharts, they 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) [58], 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 2) user-defined properties specified in the *Object Constraint Language* (OCL) as defined by OMG in its UML specifications. All of these properties

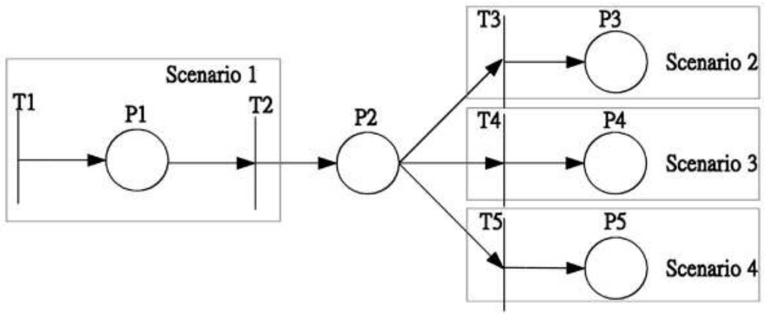


Fig. 9. Combining Petri nets that correspond to sequence diagrams with precedence.

are automatically translated into TCTL specifications for verification by SGM.

As shown in Fig. 11, except for the edges marked by SGM, all other transformations are automated in VERTAF, including the generation of ETA models from statecharts, Petri nets from sequence diagrams, scheduler automaton after scheduling, TCTL properties from OCL constraints, and the final counterexamples, if any, in terms of sequence diagrams.

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. Described in the following is a summary of the applied techniques.

- *Model Construction Guidelines*: The kind of specification models that a designer inputs to any tool always has a great effect on how the tool performs, thus guidelines aid designers to get the most out of a tool. Some typical guidelines that a VERTAF user is suggested to follow are:
  1. reuse existing components as much as possible,
  2. maximize the explicit definition of all hardware deployments in the class diagram,
  3. a class should have only one statechart representing its behavior,
  4. a statechart should have no more than four levels of hierarchy,
  5. make explicit the relations among all sequence diagrams, and
  6. both event-triggered and time-triggered methods in each class should appear somewhere in its statechart or sequence diagram.
- *Architectural Abstractions*: An assume-guarantee reasoning (AGR) based approach is adopted, whereby a complex verification task of a system is broken down into several smaller verification tasks of constituent subsystems. The theory of AGR is beyond scope here, but details can be found in [26], [61]. For the purpose of automation, we have proposed and implemented the automatic generation of assumptions and guarantees for each

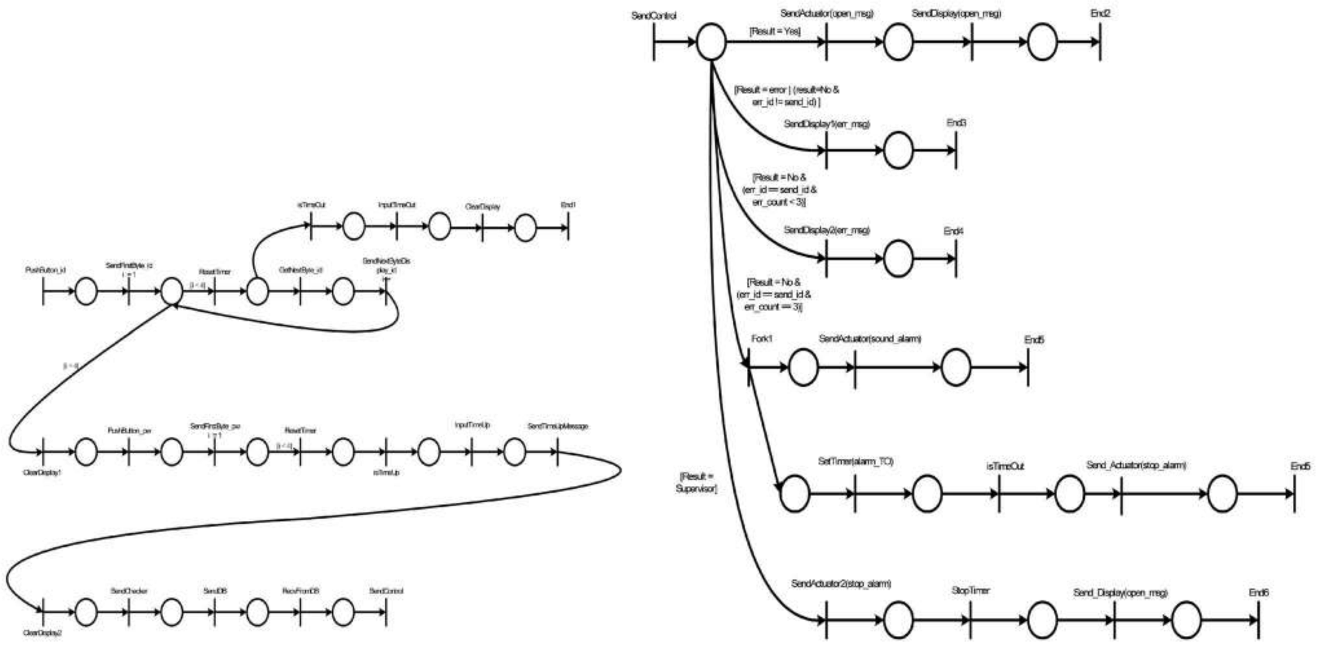


Fig. 10. Petri net model of the sequence diagrams in EGS.

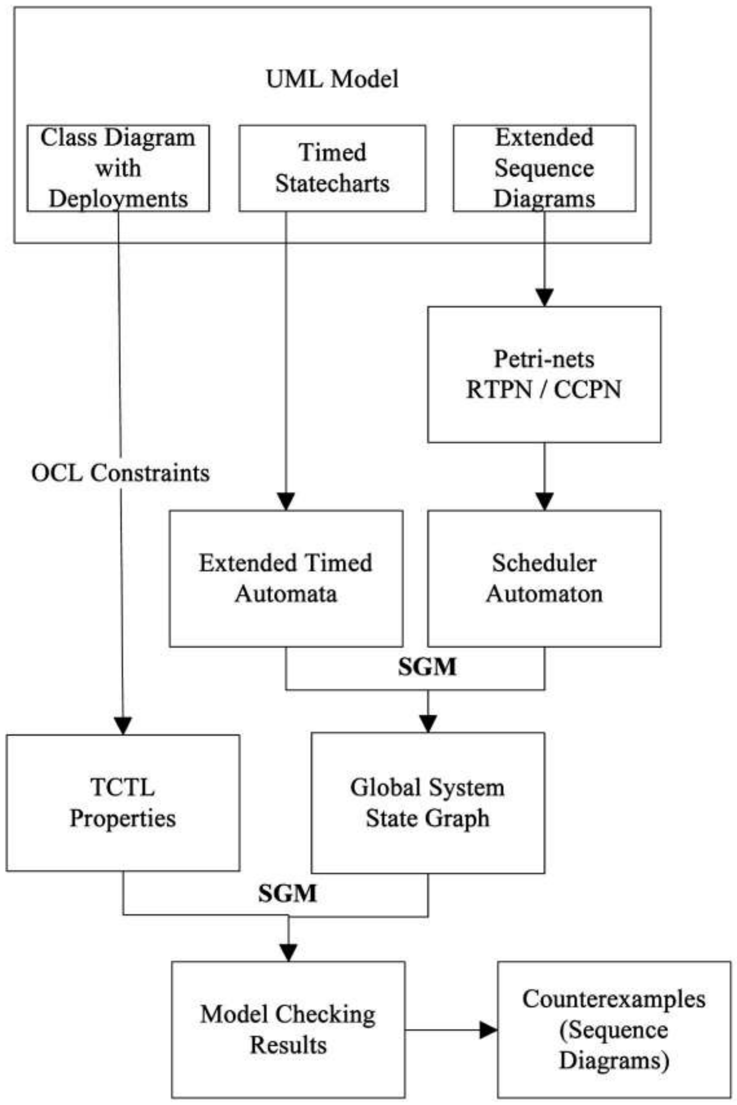


Fig. 11. Automation in VERTAF.

ETA based on their interface traces, which are then verified individually [29]. This divide and conquer approach overcomes the exponential state-space size issue to a significant extent. The benefit of AGR becomes limited when we are trying to verify properties that cross-cut the entire system. Thus, VERTAF users are suggested to decompose their properties into several smaller parts. The formal verification of component-based software is made feasible through the hierarchical decomposition of system properties into subproperties for each software component [53]. Related issues such as memory reference, object reference, and reentrance [56] are handled using a *Call-Graph* which records all component invocations.

- *Functional Abstractions*: The smaller tasks of verifying each module obtained in the architectural abstraction step is further simplified through a series of user-guided functional abstractions, including communication abstraction (communication methods such as protocols are individually verified), bit-width abstraction (instead of a 32-bit wide bus, a 1-bit or 2-bits abstract model is used), transactor models (an abstract model of other components in the system is used to verify a specific functionally detailed component), transaction-level verification (both hardware and software signals are abstracted), and assertion-based verification (only interface assertions are verified).
- *State-Space Reductions*: Several of the state-space reduction manipulators provided by SGM have been either directly applied to the ETA models generated in VERTAF or modified for adaptation to embedded systems. Since the scope here does not allow us to go into details of the reduction techniques, we merely list the techniques available and refer designers to related work [58]. The techniques applicable are: read-write reduction, discrete variable hiding reduction, clock shielding reduction, internal transition bypassing, and timed symmetry reduction.

The above abstraction techniques are applied to a user-specified UML model as follows: While constructing the

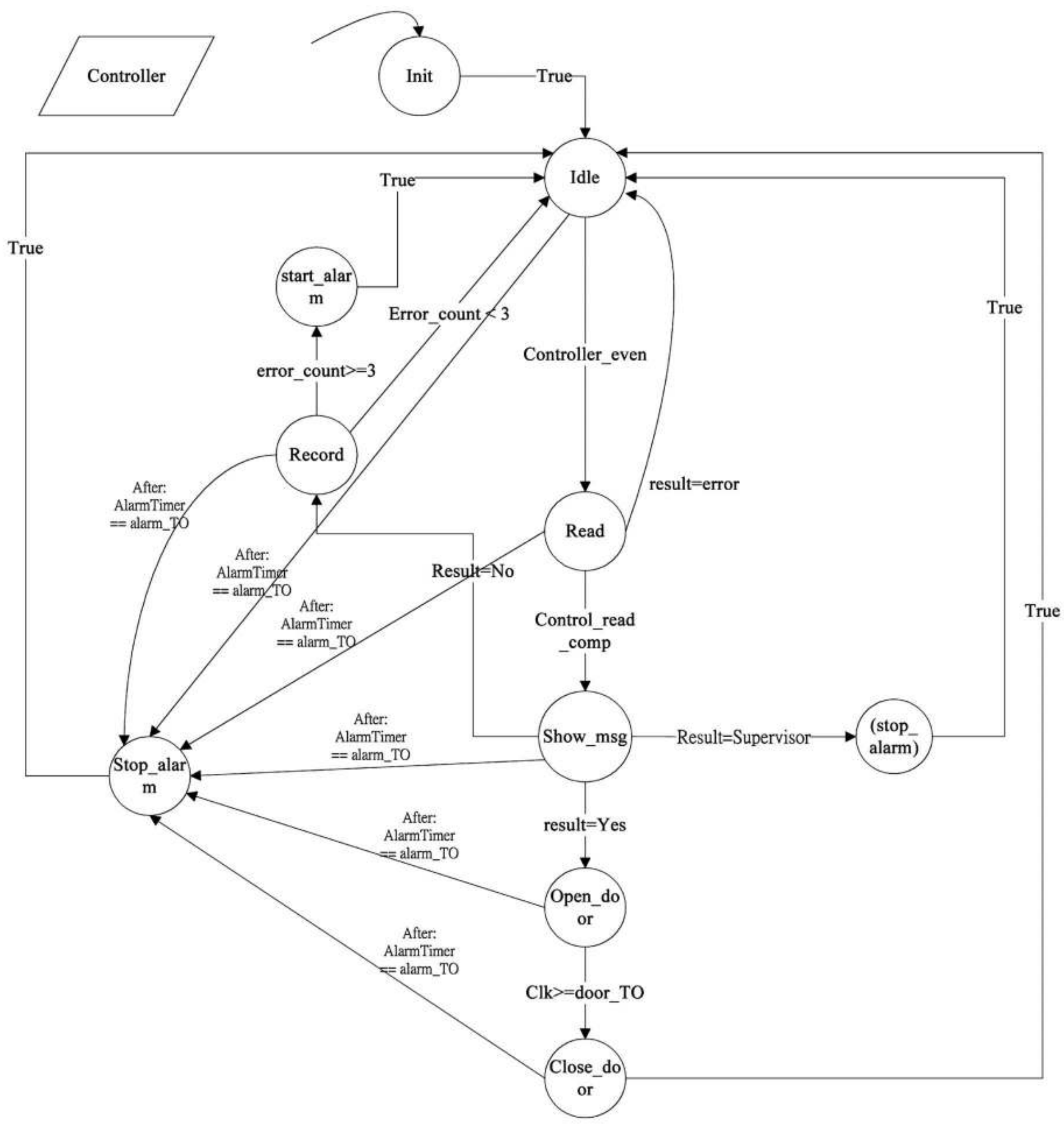


Fig. 12. Extended timed automaton for controller in EGS.

UML models, users not following the guidelines are warned of the possible intricacies. Upon completion of model construction, first Petri net models are generated, which are then scheduled to produce feasible system schedules that are represented by a scheduler ETA. Then, for each ETA generated from the statecharts, its assumptions and guarantees are generated. The guarantees of an ETA are verified by first merging the ETA with functional abstractions of the other ETA in the system and then reducing the state-spaces of the merged state-graph using SGM reduction manipulators. We can see that not only is verification automated but abstraction techniques such as AGR and state-space reductions are also automatically performed, which makes VERTAF scalable to large applications.

For our running EGS example, the ETA for each statechart were generated and then merged with the scheduler ETA. For illustration, we show in Fig. 12 the ETA that is generated by VERTAF corresponding to the controller statechart of Fig. 3. The other six 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 [44] 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

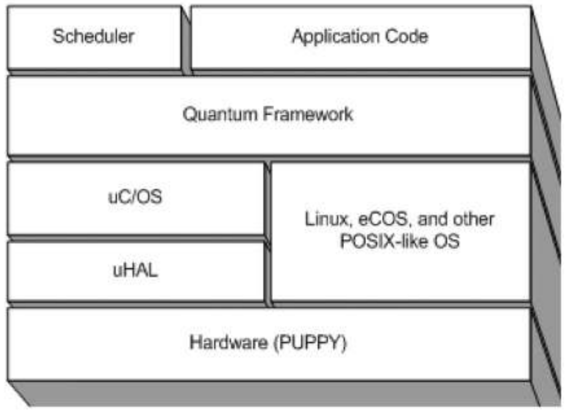


Fig. 13. Multitier architecture of VERTAF code generation.

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 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, 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 software performance profiling estimates are not easy without further hardware deployment information about the nondeployed 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.

3.5 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 middleware, VERTAF is currently based on the *Quantum Framework* [47]. 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 Fig. 13, this multitier approach decouples application code from the OS through the middleware and from the hardware platform through the OS and hardware abstraction layer.

Each ETA 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 safety-critical systems, generated code must undergo standards compliance testing and verification. We are currently trying to integrate Safecharts [12], a safety extended version of UML Statecharts, into VERTAF for conforming to safety standards such as DO-178B.

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

4 VERTAF Components

Fig. 14 illustrates the *component-based architecture* of VERTAF, which consists of five components, namely, *Implanter*, *Modeler*, *Scheduler*, *Verifier*, and *Generator*.

First, a VERTAF user identifies the objects that are specific to an application and then specifies the three kinds of UML models, namely, class diagram with deployments, timed statecharts, and extended sequence diagrams using the *Implanter* component of VERTAF. Real-time and embedded constraints are specified in OCL. Second, the models are then transformed by the *Modeler* component into corresponding ETA, Petri nets, and TCTL formulas. Third, the *Scheduler* schedules the Petri nets using QDS/EQSS and generates a scheduler ETA for verification and a scheduler ActiveObject for code generation. Fourth, the *Verifier* proves the feasibility of the scheduled set of ETA by showing if they satisfy all the given real-time and embedding constraints. AGR and other abstraction techniques are also applied by Verifier. Finally, the *Generator* is used to generate the application code from user-defined class details, the ActiveObjects corresponding to statecharts, and the scheduler ActiveObject.

The architecture of VERTAF has clear interfacing among the five components; hence, any of the components can be modified individually with minimal effect on the other parts of the framework. This modular structure also allows replacement of different components such as a new scheduler, a new verifier, or a new code generator to meet the demands of varied users.

Besides the above framework designer view of VERTAF, an application designer or framework user is mainly concerned with the modeling of objects in an application. Our logical model of an application object is expressed as an Embedded Real-Time Object (ERTO) as depicted in Fig. 15. ERTO is a merger and extension of the Port-Based Object (PBO) [54] and Time-triggered Message-triggered Object (TMO) [33]. ERTO incorporates the interface model of PBO with the methods model of TMO. The syntax of ERTO is defined in the left part of Fig. 15 consisting of data

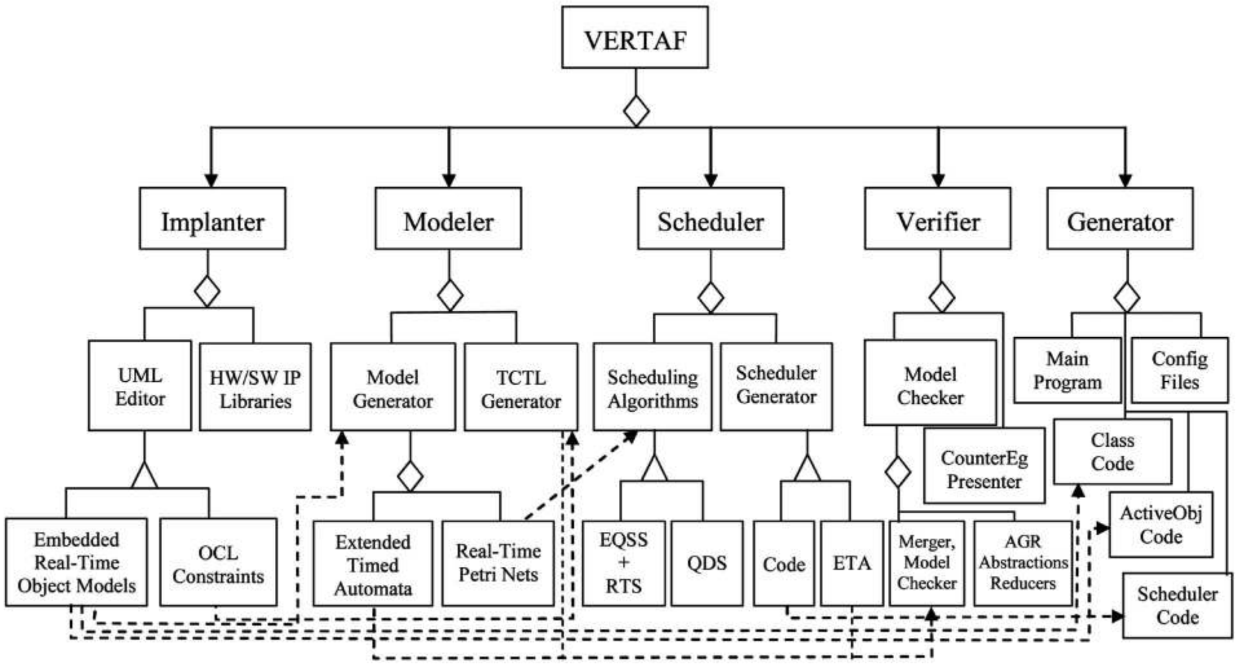


Fig. 14. Component view of VERTAF.

attributes, event-triggered methods, and time-triggered methods. The semantics of ERTO is defined in the right part of Fig. 15, consisting of hardware deployments, a timed statechart, and communication mechanisms such as queues, timers, and messages. Class-specific constraints are given in OCL. Communication with other objects is restricted to in and out ports. Configuration ports are used to reconfigure generic components for use with specific hardware. Resource ports connect to sensors and actuators.

5 Applications and Experimental Results

Two application examples developed using VERTAF are presented in this section: an *avionics* application consisting of 24 tasks (10 ERTO) used to control an aircraft and a *cruiser* application consisting of 12 tasks (five ERTO) used to control the vehicle speed under different circumstances. The benefits of using VERTAF in developing the two examples have been evaluated by the authors and the results show a marked improvement in design productivity and efficiency.

5.1 Avionics Application

This is an avionics system application called digital flight control [5]. A summary of the 24 tasks executed by 10 ERTO is shown in Table 1 [5], [41], [43]. The hardware resource for executing these tasks is the SIFT (*Software-Implemented Fault Tolerance*) computer [43] with eight processors, each having an instruction execution rate of 0.5 MIPS and an address space of 64 Kbytes. The UML models were input to VERTAF, formal ETA and Petri net models generated, and then QDS was applied. The system was schedulable

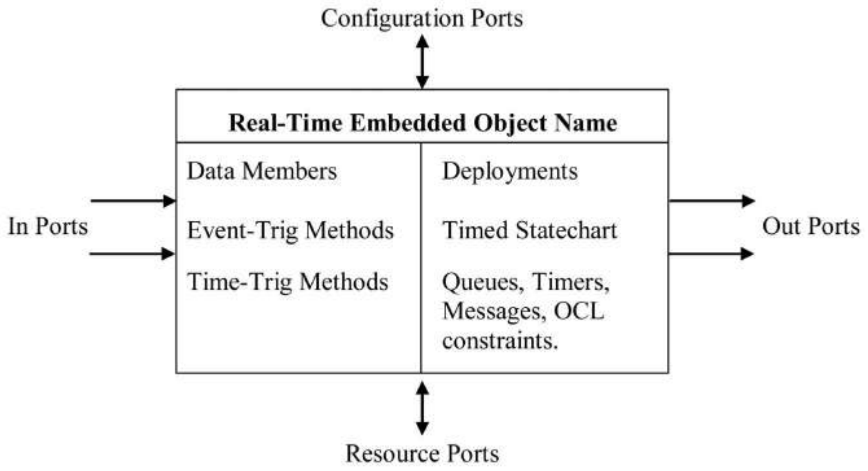


Fig. 15. Embedded real-time object.

TABLE 1
Avionics Digital Flight Control Tasks [5]

Index	Task Description	Execution Time (ms)	Period (ms)	Utilization	Memory
1	Attitude Control	2.456	50.00	0.04912	2,075
2	Flutter Control	0.276	4.00	0.06900	92
3	Gust Control	0.116	4.17	0.02784	60
4	Autoland	0.684	6.25	0.10944	1,025
5	Autopilot	0.400	200.00	0.00200	250
6	Attitude Director	5.120	33.33	0.15360	1,310
7	Inertial Navigation	2.700	40.00	0.06750	2,250
8	VOR/DME	1.540	200.00	0.00700	300
9	Omega	1.600	200.00	0.00800	505
10	Air Data	0.400	200.00	0.00200	135
11	Signal Processing	3.500	5000.00	0.00070	315
12	Flight Data	11.040	200.00	0.5520	550
13	Airspeed	1.098	62.50	0.01757	430
14	Graphics Display	7.950	125.00	0.06360	6,250
15	Text Display	3.800	100.00	0.03800	9,340
16	Collision Avoidance	0.064	1.49	0.04288	1,150
17	Onboard Communication	0.056	4.00	0.01400	705
18	Offboard Communication	0.310	250.00	0.00124	687
19	Data Integration	0.720	250.00	0.00288	1,300
20	Instrumentation	5.584	200.00	0.02792	1,900
21	System Management	4.640	2000.00	0.00232	950
22	Life Support	4.640	2000.00	0.00232	950
23	Engine Control	7.194	30.30	0.23740	1,500
24	Executive	0.400	200.00	0.00200	1,100

and a scheduler generated. A *Call-Graph* as shown in Fig. 16 depicts the invocation relationships among objects and is used in VERTAF for scheduling the RTPN generated from sequence diagrams. In total, there were 10 application domain objects (five designed by each of two designers) and 35 application framework objects (20 and 15 objects specified by the two designers, respectively). Thus, in total, there are 45 objects in the final program code generated. The average integration time per object was 0.4 day and the learning time was amortized as 0.1 day for each designer using the framework. The average integration time was two days for an object developed by one designer without using the framework. The initial effort at developing the application using VERTAF took only one week for two real-time system designers. The same two designers took five weeks in total to design the same application a second time without using VERTAF. The order for the experiments was determined by minimizing the amount of knowledge the second experiment needed from the first.

5.2 AICC Cruiser Application

Another application developed with VERTAF is AICC (*Autonomous Intelligent Cruise Controller*) [23], 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 hardware (HW) and software (SW) is shown in Fig. 17.

As shown in Fig. 18, 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 [23], within the *Swedish Road Transport Informatics Programme*. As observed in Fig. 18, each object may correspond (map) to one or more tasks. The tasks and the *Call-Graph* are as shown in Table 2 and Fig. 18, respectively. In total, there are 12 tasks performed by five application domain objects. There were 21 application framework objects specified by the designer. In total, 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 two days for each

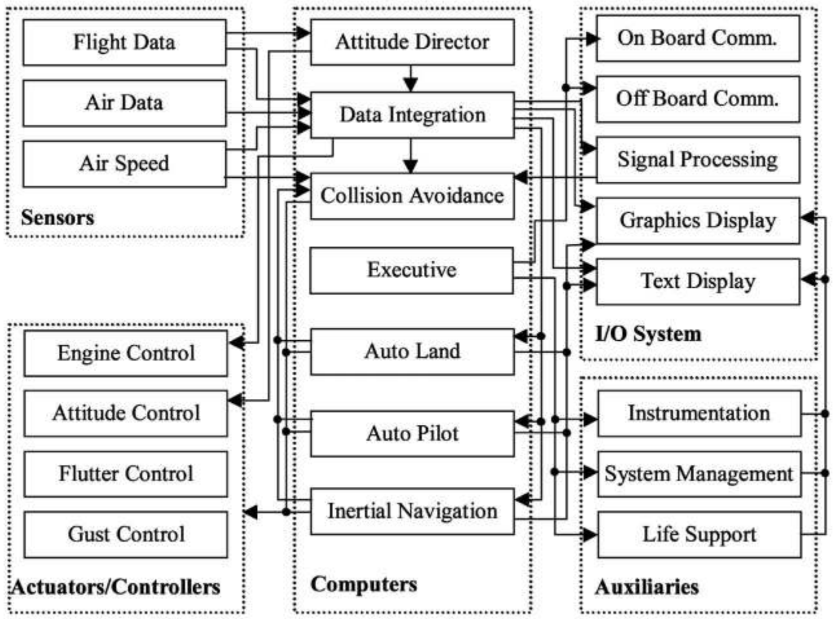


Fig. 16. Avionics system call-graph.

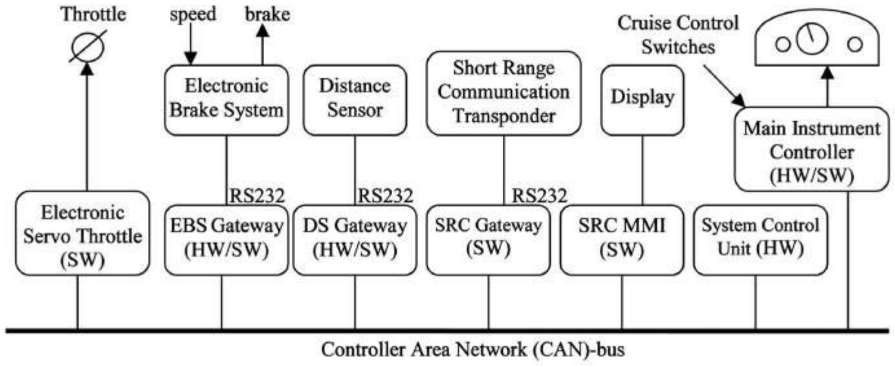


Fig. 17. AICC system architecture.

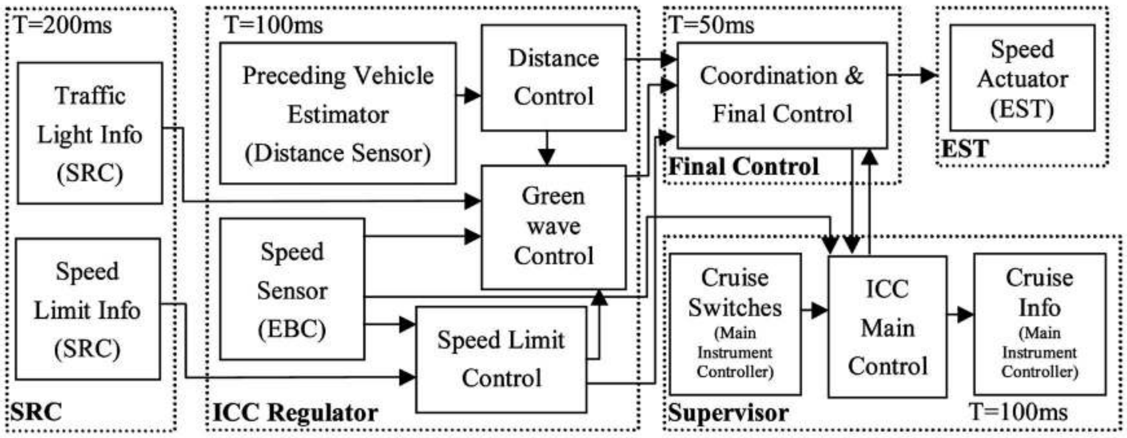


Fig. 18. AICC call-graph.

TABLE 2
AICC Tasks

Index	Task Description	Object	Period (ms)	Exec Time (ms)	Deadline
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*

object. In the initial effort, the application took five days for three real-time system designers using VERTAF. The same application took the same designers 20 days to complete development a second time without using VERTAF. The order was chosen because it was fairer than the reverse one in terms of learning effects.

6 Conclusion

An object-oriented component-based application framework, called VERTAF, was proposed for embedded real-time 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. A new *Embedded Real-Time Object* model was proposed for users to implant application domain objects into the application framework. A modular architecture was constructed for VERTAF, including five components: *Implanter*, *Modeler*, *Scheduler*, *Verifier*, and *Generator*. Besides an illustrative entrance guard system example, two industrial application examples were developed using VERTAF. Both the examples have shown how design time is significantly reduced due to a large extent of object and code reuse from VERTAF. 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 online 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 into VERTAF in the future. More abstractions required for successful model checking such as the abstraction of numeric constants [9] will also be integrated into VERTAF in the future.

Acknowledgments

This work was supported in part by project grants NSC91-2213-E-194-008, NSC92-2213-E-194-003, NSC91-2215-E-194-008, NSC92-2218-E-194-009, and NSC91-2213-E-230-002 from the National Science Council of Taiwan.

References

[1] B. Achauer, "Objects in Real-Time Systems: Issues for Language Implementers," *ACM OOPS Messenger,* vol. 7, pp. 21-27, Jan. 1996.
[2] R. Alur and D. Dill, "Automata for Modeling Real-Time Systems," *Theoretical Computer Science,* vol. 126, no. 2, pp. 183-236, Apr. 1994.
[3] T. Amnell, E. Fersman, L. Mokrushin, P. Petterson, and W. Yi, "TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems," *Proc. First Int'l Workshop Formal Modeling and Analysis of Timed Systems (FORMATS),* Sept. 2003.
[4] B-toolkit, B-core (UK) Ltd., http://www.b-core.com/, 2002
[5] J.A. Bannister and K.S. Trivedi, "Task Allocation in Fault-Tolerant Distributed Systems," *Acta Informatica,* vol. 20, no. 3, pp. 261-281, 1983.
[6] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analyzable Petri Net Models," *Proc. Third Int'l Workshop Software and Performance (WOSP '02),* pp. 35-45, July 2002.
[7] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java.* Addison Wesley, Jan. 2000.
[8] J. Browne, "Object-Oriented Development of Real-Time Systems: Verification of Functionality and Performance," *ACM OOPS Messenger,* special issue on object-oriented real-time systems, vol. 7, pp. 59-62, Jan. 1996.
[9] Y. Choi, S. Rayadurgam, and M.P.E. Heimdahl, "Automatic Abstraction for Model Checking Software Systems with Interrelated Numeric Constraints," *Proc. Eighth European Software Eng. Conf. and Ninth ACM SIGSOFT Symp. Foundation of Software Eng.,* Sept. 2001.
[10] E.M. Clarke and E.A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," *Proc. Logics of Programs Workshop,* 1981.
[11] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, 1999.
[12] H. Dammag and N. Nissanke, "Safecharts for Specifying and Designing Safety Critical Systems," *Proc. 18th IEEE Symp. Reliable Distributed Systems,* Oct. 1999.
[13] DO-178B: Software Considerations in Airborne Systems and Equipment Certification, RTCA, 1992.
[14] B.P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison Wesley, Nov. 1999.

[15] M. Elkoutbi and R.K. Keller, "Modeling Interactive Systems with Hierarchical Colored Petri Nets," *Proc. Advanced Simulation Technologies Conf.,* pp. 432-437, Apr. 1998.

[16] Esterel Technologies, http://www.esterel-technologies.com/, 2003.

[17] M. Fayad and D. Schmidt, "Object-Oriented Application Frameworks," *Comm. ACM,* special issue on object-oriented application frameworks, vol. 40, Oct. 1997.

[18] O. Fengler, W. Fengler, and T. Hummel, "Verification Method for Modeling Cooperating Processes with Colored Sequence Diagrams," *Proc. 23rd IASTED Int'l Conf. Modelling, Identification, and Control (MIC '04),* Feb. 2004.

[19] M. Gergeleit, J. Kaiser, and H. Streich, "Checking Timing Constraints in Distributed Object-Oriented Programs," *ACM OOPS Messenger,* special issue on object-oriented real-time systems, vol. 7, pp. 51-58, Jan. 1996.

[20] A. Grimshaw, A. Silberman, and J. Liu, "Real-Time Mentat, A Data-Driven Object-Oriented System," *Proc. IEEE Globecom Conf.,* pp. 141-147, Nov. 1989.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proc. IEEE,* vol. 79, no. 9, pp. 1305-1320, 1991.

[22] D. Hammer, L. Welch, and O. vanRoosmalen, "A Taxonomy for Distributed Object-Oriented Real-Time Systems," *ACM OOPS Messenger,* special issue on object-oriented real-time systems, vol. 7, pp. 78-85, Jan. 1996.

[23] 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.

[24] C.L. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements," *Proc. 10th Int'l Conf. Computer-Aided Verification,* pp. 526-531, 1998.

[25] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-Time Systems," *Proc. IEEE Logics in Computer Science,* 1992.

[26] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Decomposing Refinement Proofs Using Assume-Guarantee Reasoning," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '00),* pp. 245-252, 2000.

[27] P.-A. Hsiung, "RTFrame: An Object-Oriented Application Framework for Real-Time Applications," *Proc. 27th Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS '98),* pp. 138-147, Sept. 1998.

[28] P.-A. Hsiung, "Embedded Software Verification in Hardware-Software Codesign," *J. Systems Architecture-the Euromicro J.,* vol. 46, no. 15, pp. 1435-1450, Nov. 2000.

[29] P.-A. Hsiung and S.-Y. Cheng, "Automating Formal Modular Verification of Asynchronous Real-Time Embedded Systems," *Proc. 16th Int'l Conf. VLSI Design, (VLSI '03),* pp. 249-254, Jan. 2003.

[30] P.-A. Hsiung and C.-Y. Lin, "Synthesis of Real-Time Embedded Software with Local and Global Deadlines," *Proc. First ACM/IEEE/IFIP Int'l Conf. Hardware-Software Codesign and System Synthesis (CODES+ISSS '03),* pp. 114-119, Oct. 2003.

[31] 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," *Proc. Ninth Int'l Conf. Real-Time and Embedded Computing Systems and Applications (RTCSA '03),* Feb. 2003.

[32] Y. Ishikawa, H. Tokuda, and C.W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," *ACM SIGPLAN Notices, Proc. ECOOP/OOPSLA '90 Confs.,* vol. 25, pp. 289-298, Oct. 1990.

[33] K.H. Kim, "APIs for Real-Time Distributed Object Programming," *Computer,* vol. 33, no. 6, pp. 72-80, June 2000.

[34] A. Knapp, S. Merz, and C. Rauh, "Model Checking Timed UML State Machines and Collaboration," *Proc. Seventh Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems,* Sept. 2002.

[35] T. Kuan, W.-B. See, and S.-J. Chen, "An Object-Oriented Real-Time Framework and Development Environment," *Proc. OOPSLA '95 Conf. Workshop 18,* 1995.

[36] S. Kodase, S. Wang, and K.G. Shin, "Transforming Structural Model to Runtime Model of Embedded Software with Real-Time Constraints," *Proc. Design, Automation and Test in Europe Conf.,* pp. 170-175, Mar. 2003.

[37] L. Lavazza, "A Methodology for Formalizing Concepts Underlying the DESS Notation," Software Development Process for Real-Time Embedded Software Systems, EUREKA-ITEA project D1.7.4, http://www.dess-itea.org, Dec. 2001.

[38] W.-S. Liao and P.-A. Hsiung, "FVP: A Formal Verification Platform for SoC," *Proc. 16th IEEE Int'l SoC Conf.,* Sept. 2003.

[39] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real Time Environment," *J. ACM,* vol. 20, pp. 46-61, Jan. 1973.

[40] D. de Niz and R. Rajkumar, "Time Weaver: A Software-Through-Models Framework for Embedded Real-Time Systems," *Proc. Int'l Workshop Languages, Compilers, and Tools for Embedded Systems,* pp. 133-143, June 2003.

[41] M. Potkonjak and W. Wolf, "A Methodology and Algorithms for the Design of Hard Real-Time Multitasking ASICs," *ACM Trans. Design Automation of Electronic Systems,* vol. 4, no. 4, pp. 430-459, Oct. 1999.

[42] J.-P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR," *Proc. Int'l Symp. Programming,* 1982.

[43] R. Ratner, E. Shapiro, H. Zeidler, S. Wahlstrom, C. Clark, and J. Goldberg, "Design of a Fault-Tolerant Airborne Digital Computer," *Computational Requirements and Technology,* vol. 2, SRI Final Report, NASA Contract NAS1-10920, 1973.

[44] E.E. Roubtsova, J. van Katwijk, W.J. Toetenel, C. Pronk, and R.C.M. de Rooij, "Specification of Real-Time Systems in UML," *Electronic Notes in Theoretical Computer Science (ENTCS),* vol. 39, no. 3, 2000.

[45] J. Rumbaugh, G. Booch, and I. Jacobson, *The UML Reference Guide.* Addison Wesley Longman, 1999.

[46] J.A. Saldhana and S.M. Shatz, "UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis," *Proc. Int'l Conf. Software Eng. and Knowledge Eng. (SEKE),* pp. 103-110, July 2000.

[47] M. Samek, *Practical Statecharts in C/C++ Quantum Programming for Embedded Systems.* CMP Books, 2002.

[48] D. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," *Handbook of Programming Languages,* vol. I 1997.

[49] W.-B. See and S.-J. Chen, "Object-Oriented Real-Time System Framework," *Domain-Specific Application Frameworks,* M.E. Fayad and R.E. Johnson, eds., pp. 327-338, 2000.

[50] B. Selic, "Modeling Real-Time Distributed Software Systems," *Proc. Fourth Int'l Workshop Parallel and Distributed Real-Time Systems,* pp. 11-18, 1996.

[51] B. Selic, "An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems," *Proc. IFIP Conf. Hardware Description Languages and Their Applications,* 1993.

[52] B. Selic, G. Gullekan, and P.T. Ward, *Real-Time Object Oriented Modeling.* John Wiley and Sons, 1994.

[53] T.-Y. Shen, "Assume-Guarantee Based Formal Verification of Hierarchical Software Designs," master's thesis, Dept. of Computer Software and Information Eng., Nat'l Chung Cheng Univ., July 2003.

[54] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. Software Eng.,* vol. 23, no. 12, Dec. 1997.

[55] F.-S. Su and P.-A. Hsiung, "Extended Quasi-Static Scheduling for Formal Synthesis and Code Generation of Embedded Software," *Proc. 10th IEEE/ACM Int'l Symp. Hardware/Software Codesign (CODES '02),* pp. 211-216, May 2002.

[56] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2002.

[57] J.M. Thompson, M.P.E. Heimdahl, and S.P. Miller, "Specification-Based Prototyping for Embedded Systems," *Proc. Seventh ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 163-179, Sept. 1999.

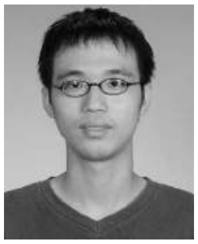
[58] F. Wang and P.-A. Hsiung, "Efficient and User-Friendly Verification," *IEEE Trans. Computers,* vol. 51, no. 1, pp. 61-83, Jan. 2002.

[59] S. Wang, S. Kodase, and K.G. Shin, "Automating Embedded Software Construction and Analysis with Design Models," *Proc. Int'l Conf. Euro-uRapid,* Dec. 2002.

[60] L.R. Welch, "A Metrics-Driven Approach for Utilizing Concurrency in Object-Oriented Real-Time Systems," *ACM OOPS Messenger,* vol. 7, pp. 70-77, Jan. 1996.

[61] M. Zulkernine and R.E. Seviora, "Assume-Guarantee Supervisor for Concurrent Systems," *Proc. 15th Int'l Parallel and Distributed Processing Symp.,* pp. 1552-1560, Apr. 2001.

Pao-Ann Hsiung received the BS degree in mathematics and the PhD degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 1991 and 1996, respectively. From 1993 to 1996, he was a teaching assistant and system administrator in the Department of Mathematics, National Taiwan University. From 1996 to 2000, he was a postdoctoral researcher at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. From February 2001 to July 2002, he was an assistant professor in the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, ROC. He is currently an associate professor. He was the recipient of the 2001 ACM Taipei Chapter Kuo-Ting Li Young Researcher for his significant contributions to design automation of electronic systems. This award is given annually to only one person under the age of 36, conducting research in Taiwan. Dr. Hsiung was also a recipient of the 2004 Young Scholar Research Award given by National Chung Cheng University to five young faculty members per year. He is on the editorial board of the *International Journal of Embedded Systems* (Inderscience Publishers) and has guest edited two special issues in 2004 and 2005 for that journal. He has been on the program committee of and chaired several international conferences. He has published more than 90 papers in international journals and conferences. His main research interests include: System-on-Chip (SoC) design and verification, embedded software synthesis and verification, real-time system design and verification, hardware-software codesign and coverification, and component-based object-oriented application frameworks for real-time embedded systems. He is a member of the IEEE.



Shang-Wei Lin received the BS degree in management information system from National Chung Cheng University, Chiayi, Taiwan, ROC, in 2002. He is currently working toward the PhD degree in the Department of Computer Science and Information Engineering at National Chung Cheng University, Chiayi, Taiwan, ROC. He is a teaching and research assistant in the Department of Computer Science and Information Engineering at National Chung Cheng University. His research interests include formal verification, scheduling, and object-oriented software synthesis.

Chih-Hao Tseng received the BS degree in computer science and information engineering from the National Taiwan University of Science and Technology, Taipei, Taiwan, ROC, in 2003. He is currently working toward the MS degree in the Department of Computer Science and Information Engineering at the National Chung Cheng University. He is a teaching and research assistant in the Department of Computer Science and Information Engineering at the National Chung Cheng University. His research interests include object-oriented design techniques in system syntheses and hardware-software codesign.

Trong-Yen Lee received the PhD degree in electrical engineering from the National Taiwan University, Taipei, Taiwan ROC, in 2001. Since 2002, he has been a member of the faculty in the Department of Electronic Engineering, National Taipei University of Technology, where he is currently an assistant professor. His research interests include the hardware-software codesign of embedded systems, SoC testing, and a software synthesis tool on embedded systems.

Jih-Ming Fu received the BS degree in computer science from the Tamkang University, Taipei County, Taiwan, ROC, in 1988, and the PhD degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 2001. Currently, he is an assistant professor in the Department of Electronic Engineering, Cheng Shiu University, Kaohsiung, Taiwan, ROC. His main research interests include distributed real-time system framework, hardware-software codesign and coverification, object-oriented design techniques in system synthesis, embedded software synthesis and verification, and real-time system design and verification.

Win-Bin See received the PhD degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 2003. He is currently with the Aerospace Industrial Development Corporation, Taichung, Taiwan, ROC. His main research interests include embedded system software design, System-on-Chip (SoC) harware-software codesign, object-oriented software framework for embedded systems, system and software engineering. He is a member of the ACM, the IEEE, the IEEE Computer Society, and the IEEE Communication Society.