Available online at www.sciencedirect.com
ScienceDirect
Computer Languages, Systems & Structures 34 (2008) 153–169
COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems

Pao-Ann Hsiung*, Shang-Wei Lin

Department of Computer Science and Information Engineering, National Chung-Cheng University, Chiayi, Taiwan, ROC

Received 15 November 2006; accepted 1 June 2007

Abstract

Currently available application frameworks that target the automatic design of real-time embedded software are poor in integrating functional and non-functional requirements for mobile and ubiquitous systems. In this work, we present the internal architecture and design flow of a newly proposed framework called *Verifiable Embedded Real-Time Application Framework* (VERTAF), which integrates three techniques namely software component-based reuse, formal synthesis, and formal verification. Component reuse is based on a formal unified modeling language (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 real-time operating systems (RTOS)-specific application code or automatically generated real-time executive with application code. Formal verification integrates a model checker kernel from state graph manipulators (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 increases design productivity.

Keywords: Application framework; Code generation; Real-time embedded software; Formal synthesis; Formal verification; Scheduling; Software components; UML modeling

1. Introduction

With the proliferation of embedded mobile and ubiquitous systems in all aspects of human life, we are making greater demands on these systems, including more complex functionalities such as pervasive computing, mobile computing, 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 [1–21]. 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. Motivated by the above status-quo, this work demonstrates how the integration of software engineering

* Corresponding author. Tel.: +886 5 2720411; fax: +886 5 2720859.
E-mail addresses: hpa@computer.org, pahsiung@cs.ccu.edu.tw (P.-A. Hsiung).

1477-8424/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2007.06.002

techniques such as software component reuse, formal software synthesis techniques such as scheduling and code generation, and formal verification technique such as model checking can be realized in the form of an integrated design environment targeted at the acceleration of real-time embedded software construction.

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

Several issues are encountered in the development of an integrated design framework. First and foremost, we need to decide upon an architecture for the framework. 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 it as an object-oriented application framework [22], 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) [23] 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 [4].
3. *System models*: For scheduling, we use variants of Petri nets (PN) [6,7] and for verification, we use Extended Timed Automata (ETA) [7,24], 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 [6,7] that generate program schedules for a single processor. For verification, we employ symbolic model checking [25–27] 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 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, RTOSs, and hardware device configurations. We call the two phases as front-end and back-end phases. The front-end phase is further divided into three sub-phases, namely UML modeling phase, real-time embedded software scheduling phase, and formal verification phase. There are two sub-phases in the back-end phase, namely component mapping phase and code generation phase. We will now present the details of each phase in the rest of this section illustrated by a running example called Entrance Guard System with Mobile and Ubiquitous Control (EGSMUC). EGSMUC is a real-time embedded system that controls any entrance with a programmable electronic lock installed. Two ways of control accesses are allowed: (a) registered users can be authenticated locally at the entrance itself, and (b) guest users may obtain a remote authentication through master acknowledgment. Here, a master could be the owner of the building to which the entrance system is protecting and he or she can have mobile and ubiquitous control access to EGSMUC. The master can grant entry access to the guest user irrespective of how he or she is connected to EGSMUC (mobile access) and also irrespective of where he or she is located (ubiquitous access). We will model EGSMUC and VERTAF will automatically synthesize and verify the code for the system.

2.1. UML modeling

UML [23] 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).

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.

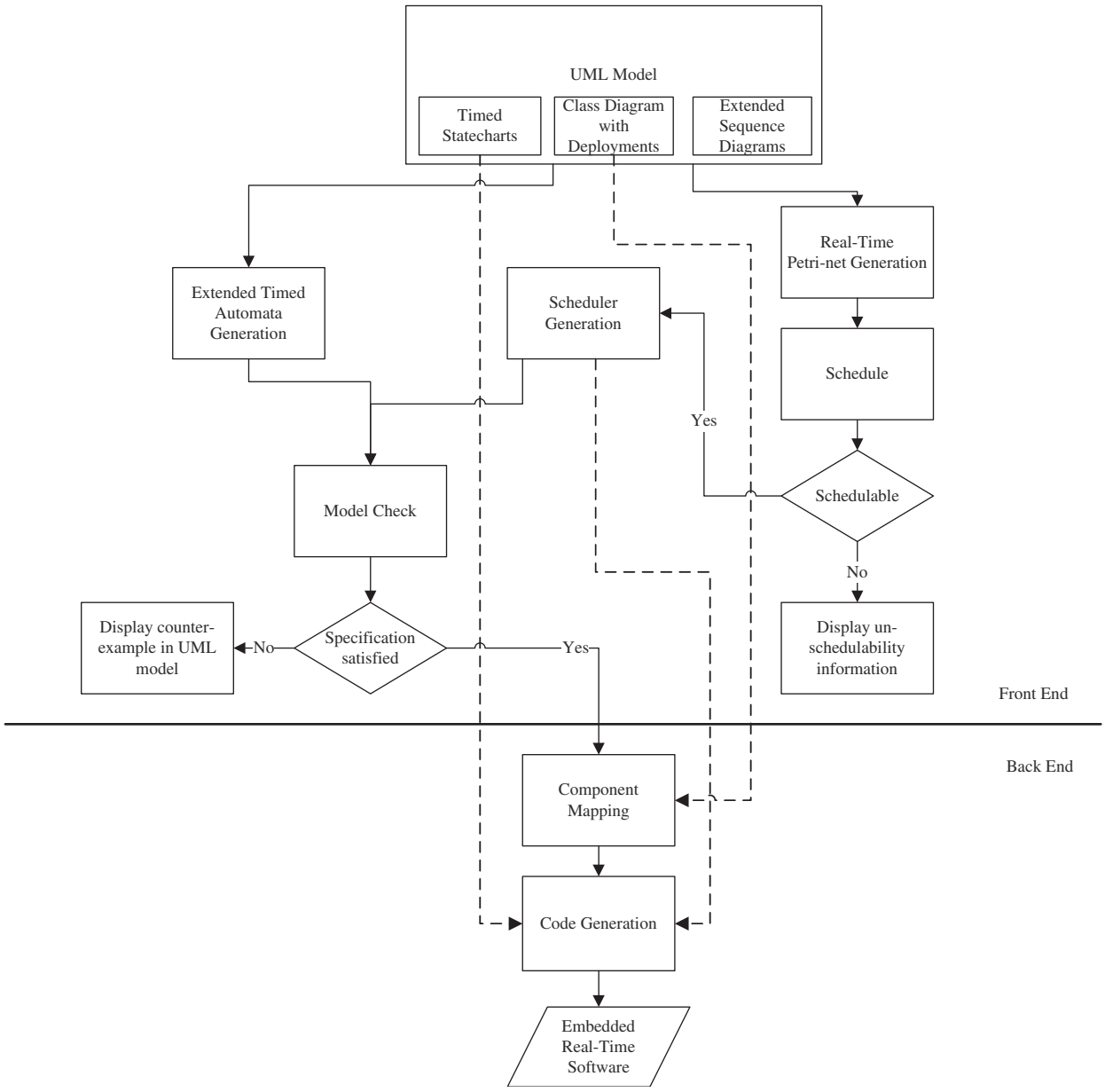

Fig. 1. Design and verification flow of VERTAF.

- *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 PN models that are used for scheduling.

For our running EGSMUC example, the system class diagram with deployment is shown in Fig. 2, a timed statechart for the system controller class is shown in Fig. 3, and an extended sequence diagram for one of the use cases dealing with guest entry and master acknowledgment is shown in Fig. 4.

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

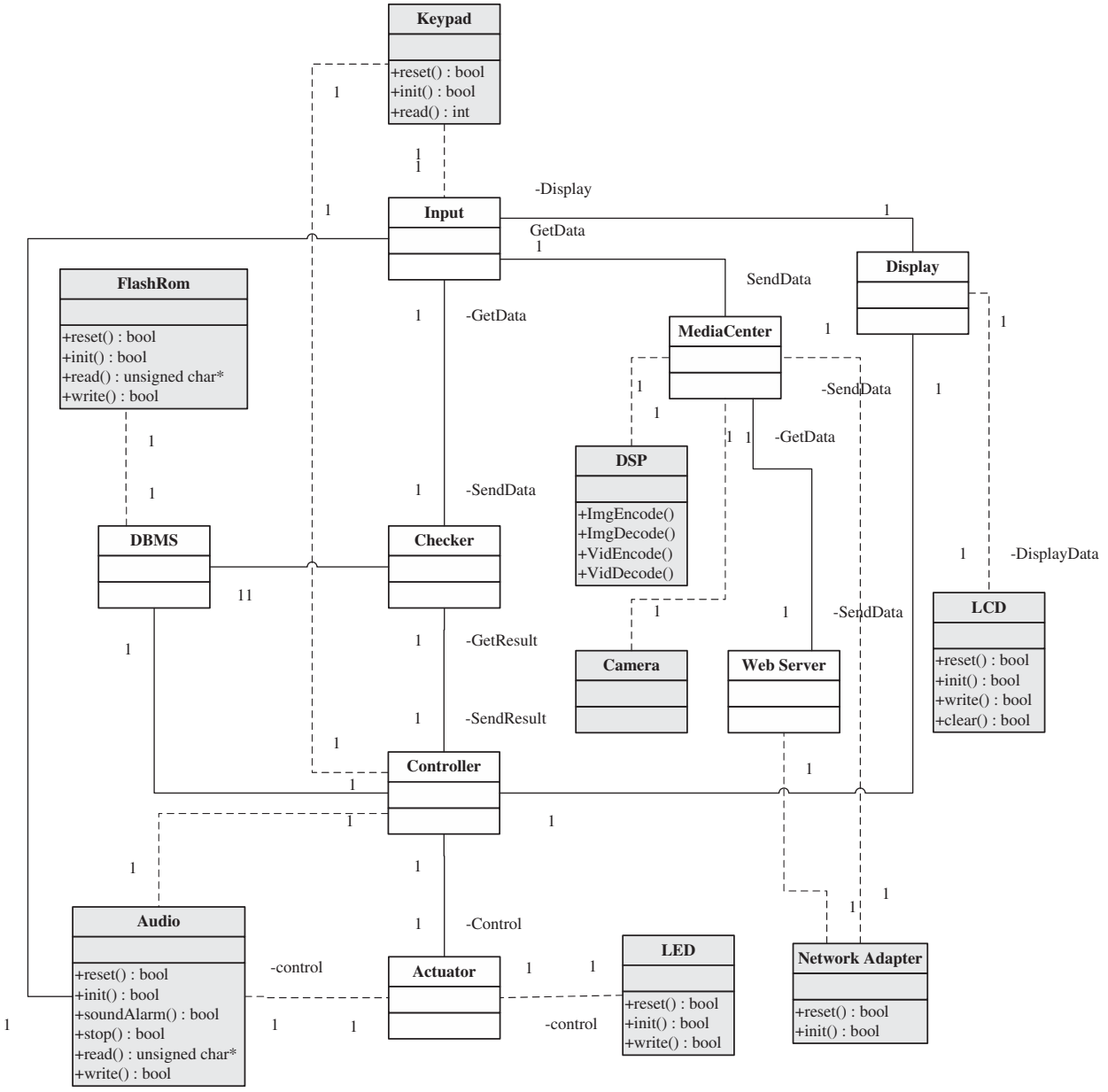

Fig. 2. Class diagram with deployment for Entrance Guard System with Mobile and Ubiquitous Control.

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.
- 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

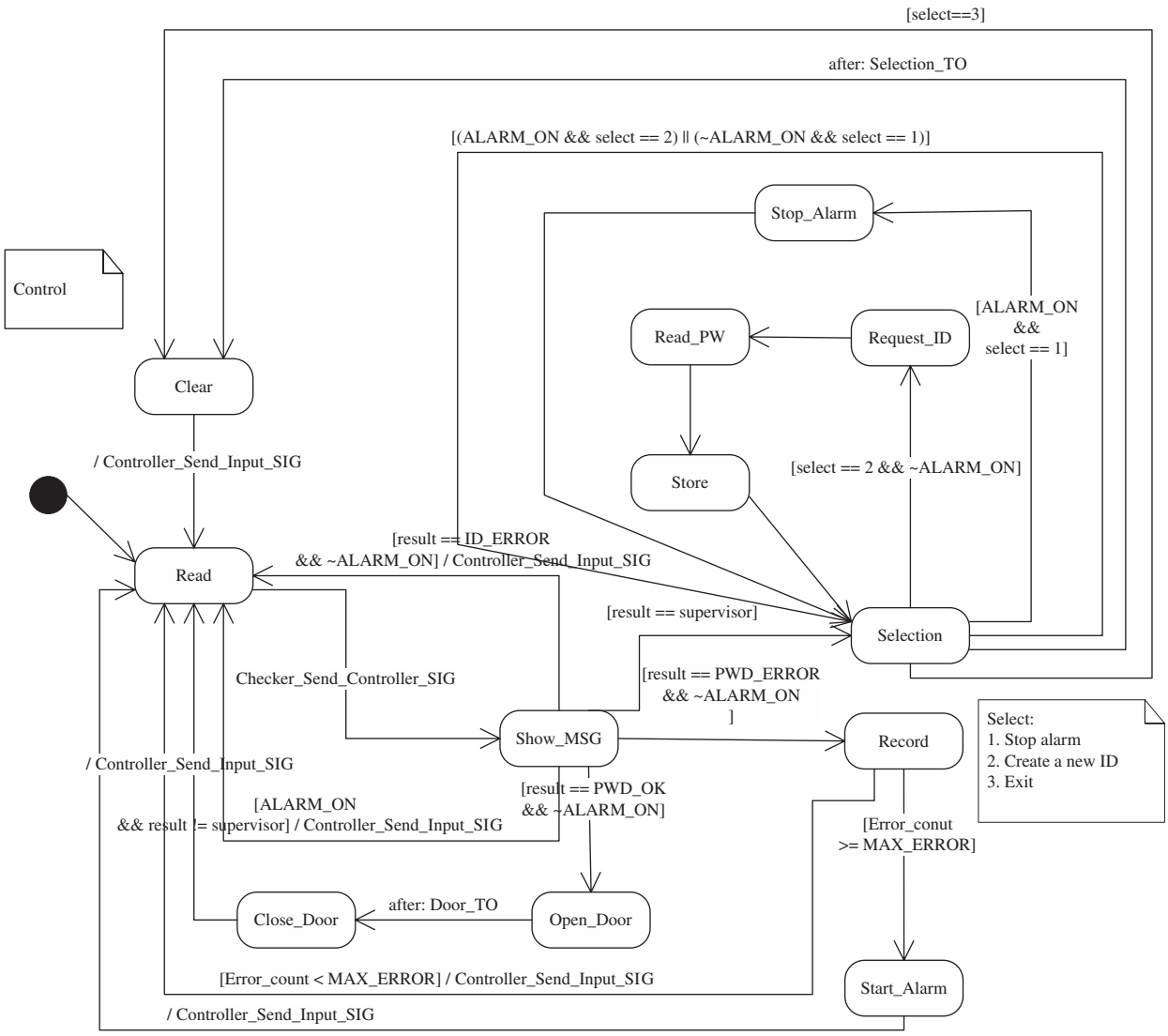

Fig. 3. Timed statechart for controller in Entrance Guard System with Mobile and Ubiquitous Control.

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 $\{D_{seq} | D_{seq} = \langle C', M \rangle, C' \subseteq C\}$,

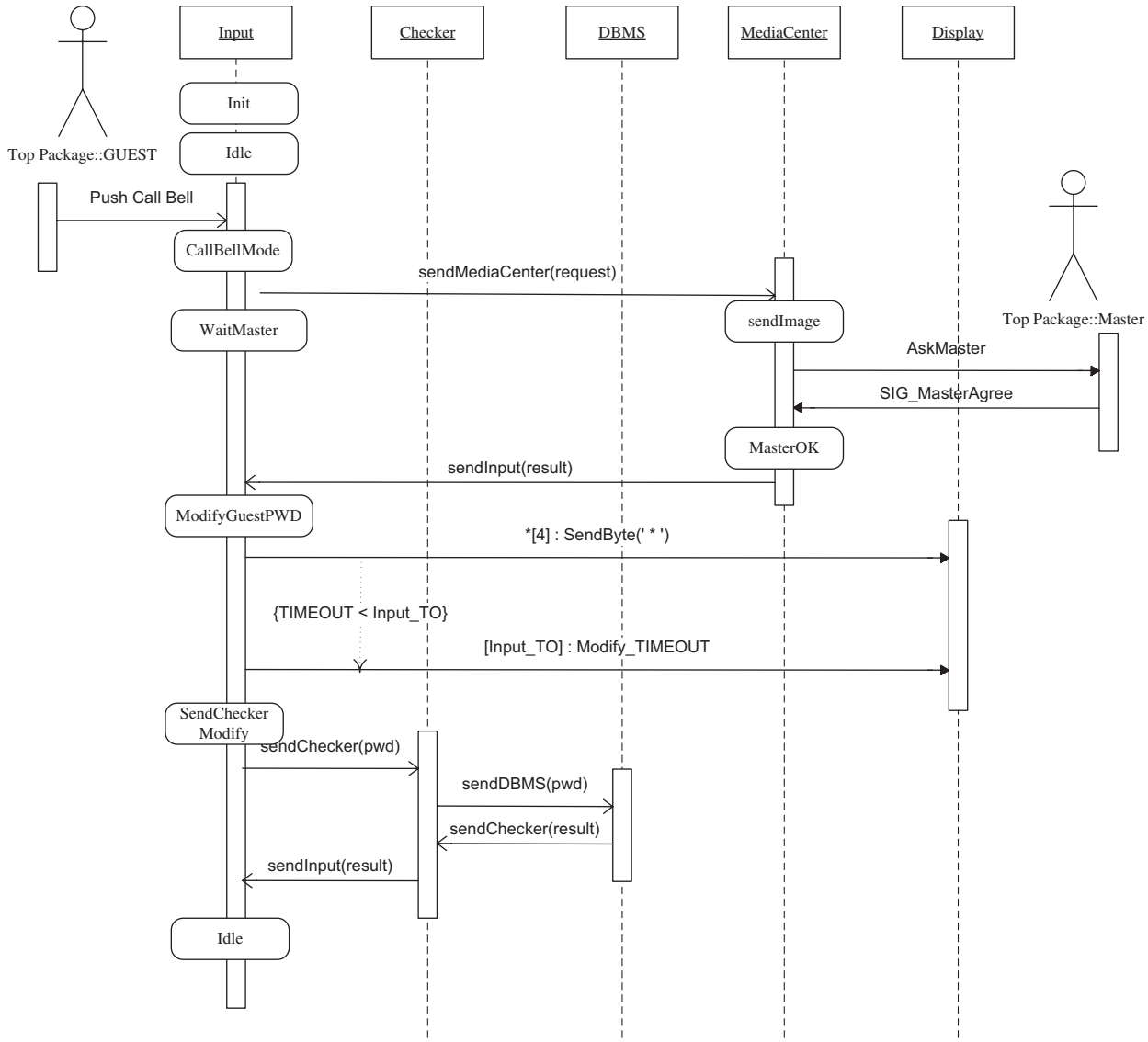

Fig. 4. An extended sequence diagram for Entrance Guard System with Mobile and Ubiquitous Control.

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, $D_{schart}(c_0) \times \cdots \times D_{schart}(c_{|C|}) \times B(D^1_{seq}, \ldots, D^k_{seq})$ denotes the system behavior semantics, where B is the scheduler ETA as formalized in Section 2.2.

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) [6,7] 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 [29] 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 PN, RTPN, or CCPN. RTPN enhances the standard PN with code execution characteristics associated with transitions. Given a standard PN $N = \langle P, T, \phi \rangle$, 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 PN 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 PNs 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 PNs. If a PN has an ending transition which is the same as the initial transition of another PN, they are concatenated by merging the common transition.
5. Sequence diagrams that are inter-related by precedence constraints are first translated individually into independent PNs, 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 PN 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 PNs. 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 PNs. 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 [6,7], and [28]. The basic strategy is to decompose each PN into conflict-free components that are scheduled individually for satisfaction of memory constraints. A conflict-free component is a reduction of a PN 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 PNs. For example, a system with two PNs, 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 EGSMUC example, as shown in Fig. 5, a single PN is generated from the user-specified set of statecharts, which is then scheduled using QDS. In this example, scheduling is required only for the timers associated with the actuator, the controller, and the input object. After QDS, we found that EGSMUC is schedulable.

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) [20], 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 PN 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 the following.

- *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,

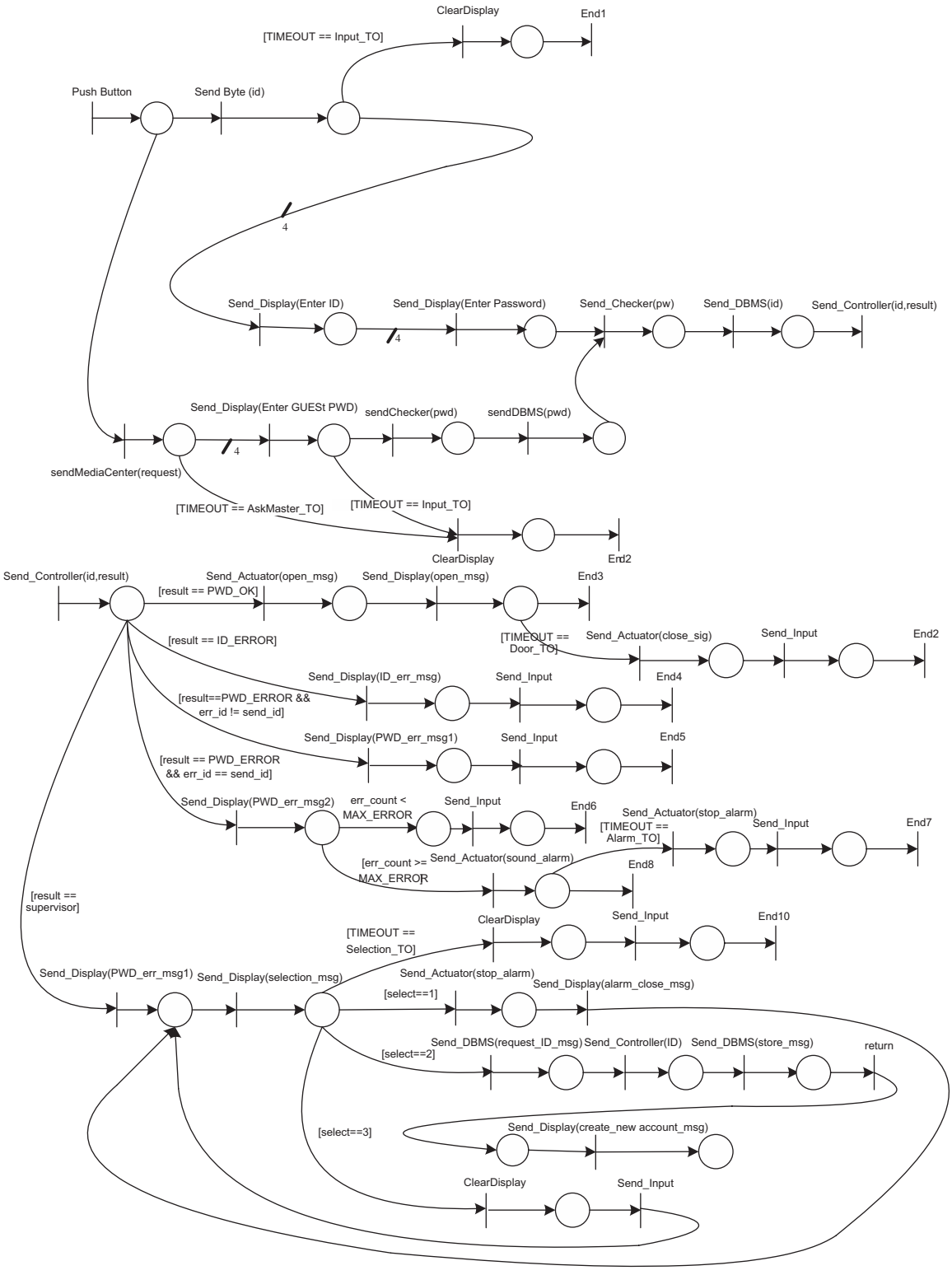

Fig. 5. Petri net for Entrance Guard System with Mobile and Ubiquitous Control.

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 [30,31]. For the purpose of automation, we have proposed and implemented the automatic generation of assumptions and guarantees for each ETA based on their interface traces, which are then verified individually [5]. 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 sub-properties for each software component [19]. Related issues such as memory reference, object reference, and reentrance [32] 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 [20]. 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 UML models, users not following the guidelines are warned of the possible intricacies. Upon completion of model construction, first PN 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 EGSMUC example, the ETA for each statechart were generated and then merged with the scheduler ETA. For illustration, we show in Fig. 6 the ETA that is generated by VERTAF corresponding to the controller statechart of Fig. 3. There are seven other ETA in this system example. All ETA were input to SGM and AGR was applied. Reduction techniques were then applied to each state-graph obtained from AGR. OCL constraints were then translated into TCTL and verified by the SGM model checker kernel.

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, 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.

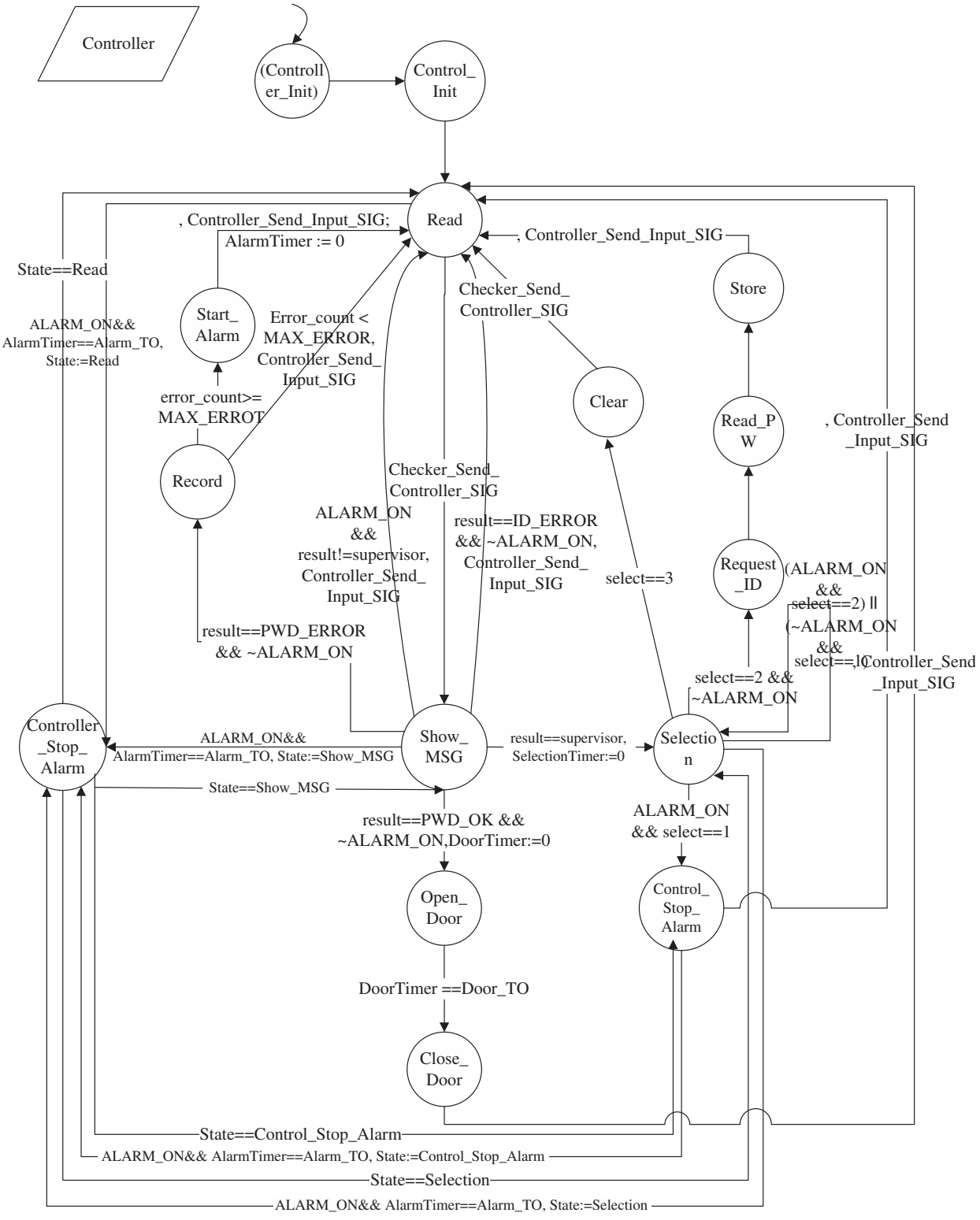

Fig. 6. ETA for controller in Entrance Guard System with Mobile and Ubiquitous Control.

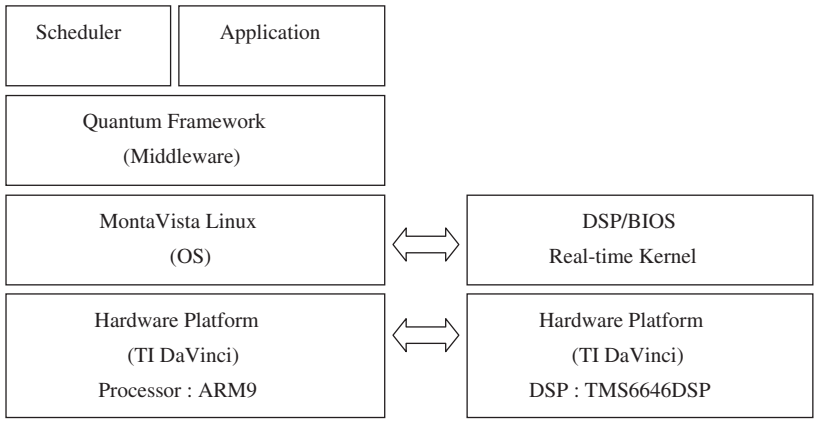

Fig. 7. Multi-tier code architecture.

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.

For our running EGSMUC example, all software classes in the class diagram given in Fig. 2 are deployed on one or more hardware or software classes supported by VERTAF.

2.5. Code generation

There are basically three issues in this phase including hardware portability, software portability, and temporal correctness. We adopt a multi-tier approach for code generation: an operating system layer, a middleware layer, and an application with scheduler layer, which solves the above three issues, respectively. Currently supported underlying hardware platforms include dual core ARM-DSP based, single core ARM, StrongARM, or 8051 based, and Lego RCX-based Mindstorm systems. For hardware abstraction, VERTAF supports MicroHAL and the embedded version of POSIX. For operating systems, VERTAF supports MontaVista Linux, MicroC/OS, Embedded Linux, and eCOS. For middleware, VERTAF is currently based on the Quantum Framework [13]. 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. A sample configuration is shown in Fig. 7, where the multi-tier approach decouples application code from the operating system through the middleware and from the hardware platform through the operating system layer.

Each ETA that is generated either from UML statecharts or from the scheduled PN (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 seven ActiveObjects derived from the statecharts and one ActiveObject representing the scheduler. Makefiles were generated for linking in the API of the eight hardware classes and configuration files were generated for the ARM-DSP dual microprocessor platform called DaVinci from Texas Instruments with MontaVista Linux as its operating system on the ARM processor and DSP/BIOS real-time kernel as the operating system on the DSP TMS6646DSP processor. There were totally 2754 lines of C code for the full EGSMUC system, out of which the system designers had to write only around 170 lines of C code, which is only 6.2% of the full system code.

3. Analysis and evaluation

For the running example EGSMUC, we now analyze why VERTAF is capable of generating a significant part of the system implementation code, thus alleviating the designer from the tedious and error-prone task of manual coding.

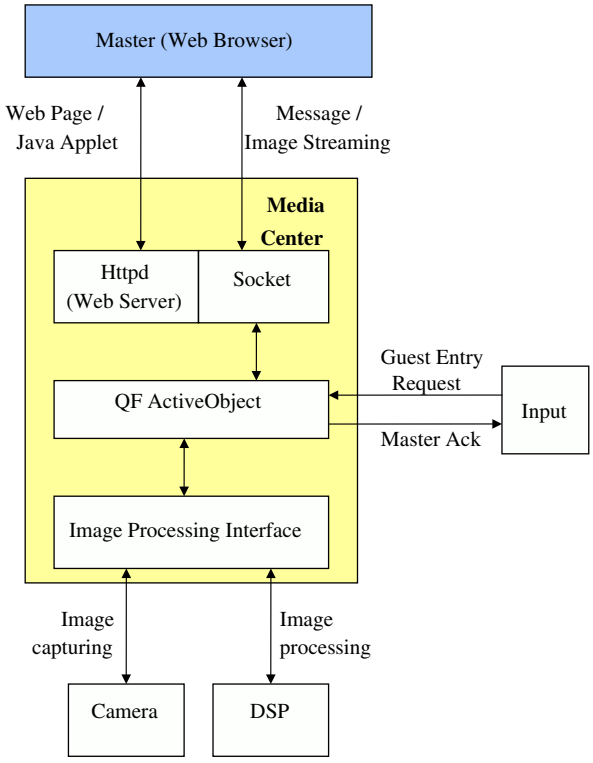

Fig. 8. Code structure for media center in EGSMUC.

Due to its application framework architecture, VERTAF supports software components that are commonly found in mobile, ubiquitous, real-time, and embedded application domains. We classify the components supported by VERTAF into the following.

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

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

For illustration purposes, we show how the media center class in the EGSMUC example was implemented using VERTAF. The media center class is responsible for getting acknowledgment from a mobile master ubiquitously, which means whenever a guest wants to enter the building that the EGSMUC system is guarding, the media center notifies

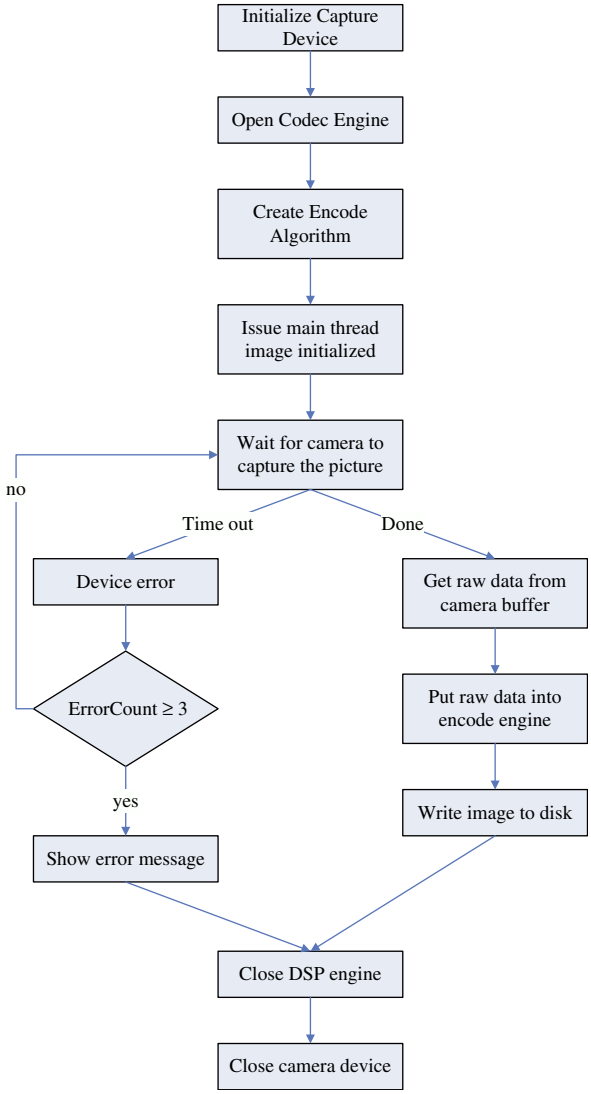

Fig. 9. Flow chart for media center code in EGSMUC.

the DSP class to use the camera to capture an image of the guest and then send the guest image to a master (the owner of the building or house). The master can send an acknowledgment through the web after which the guest can enter the building. A password is setup by a guest so that the guest can enter the building within the span of time set by the master beforehand.

Fig. 4 shows gave the sequence diagram that a user needs to specify in order for VERTAF to generate corresponding code. The architecture of the code generated by VERTAF is shown in Fig. 8, where QF ActiveObject is an active object from the Quantum Framework. The code consists of three parts, namely a web server, a QF ActiveObject, and an image processing interface. The web server allows a master to connect to EGSMUC using a web browser that can run Java applets. The applet opens a socket connection between the media center and the client machine of the master. The image of the guest requesting entrance is captured and processed through the image processing interface. When a master acknowledges, the guest is notified through the input class. The control and data flows of the media center are automatically generated by VERTAF and the user has to merely specify the sequence diagrams as shown in Fig. 4 and deploy the related classes to hardware or software components in the class diagram as shown in Fig. 2. This is exactly the reason why VERTAF can save a lot of coding and design efforts.

Fig. 9 shows the detailed flow of image capture and processing as implemented in the TI DaVinci platform, which has several image and video codecs. After a capture device is initialized, a required codec engine is opened and an algorithm created to run on the DSP. The camera is initialized and if the camera times out for three times, an error message is shown and the program exits. Otherwise, the raw data from the camera buffer is read, input to encode engine and finally the image is written to disk. The DSP engine and the camera device are then closed.

There were totally 16 objects in the final application generated by VERTAF, out of which the user or designer had to only model 7 classes. The remaining nine classes included components from all the four categories as described at the start of Section 3. Empirical results obtained from comparing two different implementations of the EGSMUC system, one using VERTAF, and one without using VERTAF, showed that not only the user written code reduced to 6.2% and the number of objects reduced to 44%, but the total time required to develop the application also reduced by more than 60%. The average learning time for each designer using VERTAF was approximately 0.1 day. The experimental and empirical results all show that VERTAF is beneficial to designers of real-time embedded software with mobile and ubiquitous control access.

4. Conclusion

An object-oriented component-based application framework, called VERTAF, was proposed for the development of real-time embedded system applications with mobile and ubiquitous control access. It was a result of the integration of three different technologies: software component reuse, formal synthesis, and formal verification. Starting from user-specified UML models, automation was provided in model transformations, scheduling, verification, and code generation. VERTAF can be easily extended 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 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] Amnell T, Fersman E, Mokrushin L, Petterson P, Yi W. TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of the 1st international workshop on formal modeling and analysis of timed systems (FORMATS), September 2003.
[2] Douglass BP. Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns. Reading, MA, USA: Addison Wesley Longman, Inc.; 1999.
[3] Hsiung PA. 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). Silver Spring: IEEE Computer Soc Press; September 1998. p. 138–47.
[4] Hsiung PA. Embedded software verification in hardware–software codesign. Journal of Systems Architecture—the Euromicro Journal 2000;46(15):1435–50.
[5] Hsiung PA, Cheng SY. Automating formal modular verification of asynchronous real-time embedded systems. In: Proceedings of the 16th international conference on VLSI design (VLSI'2003). Silver Spring: IEEE Computer Soc Press; January 2003. p. 249–54.
[6] Hsiung PA, Lin CY. 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). New York: ACM Press; October 2003. p. 114–9.
[7] Hsiung PA, Lin CY, Lee TY. 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). February 2003.
[8] Knapp A, Merz S, Rauh C. Model checking timed UML state machines and collaboration. Proceedings of the 7th international symposium on formal techniques in real-time and fault-tolerant systems. Lecture notes in computer science, vol. 2469. Berlin: Springer; September 2002. p. 395–414.
[9] Kuan T, See WB, Chen SJ. An object-oriented real-time framework and development environment. In: Proceedings OOPSLA'95 workshop #18, 1995.
[10] Kodase S, Wang S, Shin KG. Transforming structural model to runtime model of embedded software with real-time constraints. In: Proceedings of design, automation and test in europe conference. March 2003. p. 170–5.
[11] Lavazza L. A methodology for formalizing concepts underlying the DESS notation. Software development process for real-time embedded software systems, EUREKA-ITEA project; December 2001. ⟨http://www.dess-itea.org/⟩ D: 1.7.4.
[12] de Niz D, Rajkumar R. Time weaver: a software-through-models framework for embedded real-time systems. In: Proceedings of the international workshop on languages, compilers, and tools for embedded systems, June 2003. p. 133–143.
[13] Samek M. Practical statecharts in $C/C++$ quantum programming for embedded systems. CMP Books; 2002.

[14] Schmidt D. Applying design patterns and frameworks to develop object-oriented communication software. In: Peter S, editor. Handbook of programming languages, vol. I, MacMillan Computer Publishing; 1997.
[15] See WB, Chen SJ. Object-oriented real-time system framework. New York: Wiley; 2000 p. 327–38 [Chapter 16].
[16] Selic B. Modeling real-time distributed software systems. In: Proceedings of the 4th international workshop on parallel and distributed real-time systems. 1996. p. 11–8.
[17] Selic B. 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.
[18] Selic B, Gullekan G, Ward PT. Real-time object oriented modeling. New York: Wiley; 1994.
[19] T.Y. Shen, Assume-guarantee based formal verification of hierarchical software designs, Master's thesis, Department of CSIE, National Chung Cheng University; July 2003.
[20] Wang F, Hsiung PA. Efficient and user-friendly verification. IEEE Transactions on Computers 2002;51(1):61–83.
[21] Wang S, Kodase S, Shin KG. Automating embedded software construction and analysis with design models. In: Proceedings of international conference of Euro-uRapid, December 2002.
[22] Fayad M, Schmidt D. Object-oriented application frameworks. In: Communications of the ACM, special issue on object-oriented application frameworks, vol. 40, October 1997.
[23] Rumbaugh J, Booch G, Jacobson I. The UML reference guide. Reading: Addison-Wesley; 1999.
[24] Alur R, Dill D. Automata for modeling real-time systems. Theoretical Computer Science 1994;126(2):183–236.
[25] Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic. Proceedings of the logics of programs workshop, Lecture notes in computer science, vol. 131. Berlin: Springer; 1981. p. 52–71.
[26] Clarke EM, Grumberg O, Peled DA. Model checking. Cambridge: MIT Press; 1999.
[27] Queille JP, Sifakis J. Specification and verification of concurrent systems in CESAR. Proceedings of the international symposium on programming, Lecture notes in computer science, vol. 137. Berlin: Springer; 1982. p. 337–51.
[28] Su FS, Hsiung PA. 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). New York: ACM Press; May 2002. p. 211–216.
[29] Liu C, Layland J. Scheduling algorithms for multiprogramming in a hard-real time environment. Journal of the Association for Computing Machinery 1973;20:46–61.
[30] Henzinger TA, Qadeer S, Rajamani SK. Decomposing refinement proofs using assume-guarantee reasoning. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD'00), 2000. p. 245–52.
[31] Zulkernine M, Seviora RE. Assume-guarantee supervisor for concurrent systems. In: Prceedings of the 15th international parallel and distributed processing symposium. April 2001. p. 1552–60.
[32] Szyperski C. Component software: beyond object-oriented programming. Reading: Addison-Wesley; 2002.