VERTAF: An Object-Oriented Application Framework for Embedded Real-Time Systems[*]

Pao-Ann Hsiung[1], Trong-Yen Lee[2], Win-Bin See[3], Jih-Ming Fu[4], and Sao-Jie Chen[3]
[1]*National Chung Cheng University, Chiayi, Taiwan, ROC*
[2]*Chung Cheng Institute of Technology, National Defense University, Taiwan, ROC*
[3]*National Taiwan University, Taipei, Taiwan, ROC*
[4]*Cheng Shiu Institute of Technology, Kaohsiung, Taiwan, ROC*
[1]*E-mail: hpa@computer.org*

Abstract

Embedded real-time applications are often built from scratch on a trial-and-error basis, which leads to sub-optimal designs with latent errors that are not detectable in early stages of use or deployment and often incurs prolonged time-to-market. A new application framework called Verifiable Embedded Real-Time Application Framework (VERTAF) is proposed for embedded real-time application development, with the aim of reducing design errors and increasing design productivity. VERTAF is an integration of three technologies, namely object-oriented technology, software component technology, and formal verification technology. VERTAF consists of five software components: Implanter, Modeler, Scheduler, Verifier, and Generator. Experiences of using VERTAF show a significant increase in design productivity through design reuse, and a significant decrease in design time and effort through design verification. An example shows a relatively low design effort on the part of the designer using VERTAF.

1. Introduction

Current technology in designing *embedded real-time software* (ERTS) is quite immature such that software engineers tend to use a very rudimentary trial-and-error design technique, developing everything from scratch, applying only unit-testing, and producing sub-optimal software. To remedy the situation, this work tries to automate and systematize the design process so that ERTS is designed *correctly* and *efficiently*. In this process, we must analyze and solve various design issues such as the ease of user input, high-level design reuse, and satisfaction of resource and temporal constraints.

In solution to the above three issues, we propose the integration of three technologies as follows. First, for modular user-friendly input of system requirements, *object-oriented* (OO) modeling is required. Second, for automating the design process through the integration of reusable off-the-shelf modules, *software component* (SC) technology is required. Third, to design embedded real-time software such that all specified resource and temporal constraints are satisfied, *formal verification* (FV) is required. The three technologies are all integrated into an application framework called *Verifiable Embedded Real-Time Application Framework* (VERTAF), which can *efficiently* produce *verifiable* embedded real-time software, such that design productivity is increased, design time and error are reduced, and heuristically optimal designs are produced.

We will now briefly introduce *object-oriented application frameworks* (OOAF) and *embedded real-time systems*. OOAF is a reusable, "semi-complete" application that can be specialized to produce custom applications [2]. OOAF are application-domain specific reuse methods, such as user interfaces or real-time avionics. Examples include MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, Javasoft's RMI and implementation of OMG's CORBA. OOAF has the highest level of reuse in the design of an application. The primary benefits of OOAF stem from the modularity, reusability, extensibility, and inversion of control they provide to developers.

An *embedded real-time system* is generally specified as a collection of *tasks*, which might share resources and interact with the system in which it is installed or with the environment. The tasks are usually independent and periodic. Execution time, period, deadline, type of priority and resource requirements are specified for each task. To statically guarantee satisfaction of all timing constraints, the tasks must be scheduled using *priority-based* scheduling algorithms such as rate-monotonic (RM) [3], earliest-deadline first (EDF) [3], mixed-priority (MP) [3], pin-wheel, etc. or using *timed-based* scheduling algorithms.

[*] This work is supported by research project grant NSC-90-2215-E-194-009 from National Science Council, Taiwan, ROC

Real-time systems must satisfy stringent temporal constraints which formal verification can prove. VERTAF makes a pioneer step in incorporating formal verification into the design process. There are several formal methods that can be applied for formal verification of real-time systems such as model checking, process algebra, theorem proving, and other logic-related techniques. Here, we will use a formal method, called *model checking*, that is gradually gaining popularity among the industries and academia alike. Given a real-time system description S and a temporal property specification ϕ, model checking answers if S satisfies ϕ. A real-time system is modeled by a set of *Timed Automata* (TA) [4], a timed extension of conventional automata. A temporal property is specified using *Timed Computation Tree Logic* (TCTL) [5].

Using VERTAF, a developer can devote more time and effort to the actual application tasks, instead of real-time system peculiarities. Even program verification can be accomplished automatically by VERTAF since it has integrated formal verification into its design process. VERTAF is modularized into five software components that can be used at different stages of application development. The components are called *Implanter*, *Modeler*, *Scheduler*, *Verifier*, and *Generator*.

The article organization is as follows. Section 2 gives some previous and related work on applying object-oriented technology to real-time system design. Section 3 describes the five components of VERTAF using a *Components* view. Section 4 illustrates how a designer may use VERTAF to actually develop an embedded real-time application. Section 5 presents the experimental results of two different examples developed using VERTAF. Section 6 gives the final conclusions.

2. Previous Work

Although object-oriented technology has been applied to the design of real-time systems in several proposed work [6] - [12], there have been very few works on the development of OOAF for real-time application design. Two recently proposed OOAF are *Object-Oriented Real-Time System Framework* (OORTSF) [13], [14], [15] and SESAG [16], [17]. OORTSF and SESAG are simple frameworks that have been applied to avionics software development. Some design patterns were proposed related to real-time application design. Code can be automatically generated. But, there are still some scheduling and real-time synchronization issues left not handled such as asynchronous event handling and protocol hooking. The flexibility of specifying real-time objects, the ease of using the frameworks, the benefits of applying them, and other issues related to OOAFs are not described in the two works. VERTAF is, in fact, a newer more enhanced version of SESAG, incorporating software component technology, formal verification technology, industry standards such as *Unified Modeling Language* (UML) and Java, and multi-level reuse. According to the knowledge of the authors, besides OORTSF and SESAG, there is practically no other work on *enterprise application frameworks* devoted to real-time application development. As far as *middleware integration frameworks* for real-time applications are concerned, there has been a TAO Real-Time ORB proposed by Schmidt recently [18].

As far as software components are concerned, Stewart has recently proposed port-based object models, framework process models, state variable table for inter-process communication, and code-generation for both *Real-Time Operating System* (RTOS) and real-time executive environments [19]. All of the above concepts and implementations proposed by Stewart aid in developing robust, reusable software components with well-defined uniform interfaces.

3. VERTAF Components

Figure 1 illustrates the components of VERTAF. We use the industry standard *Unified Modeling Language* (UML) [20] for illustration. VERTAF consists of five basic software components: *Implanter*, *Modeler*, *Scheduler*, *Verifier*, and *Generator*. The given sequential order is the sequence in which they are used. In general, a user may use the five components of VERTAF as follows. Given a software application to be designed, an engineer identifies and specifies the objects that are specific to the application using the *Implanter* component. Real-time and embedding constraints are specified within the application objects. The application objects are then transformed by the *Modeler* component into standard uniform process models, each of which represents a real-time task. *Scheduler* checks the schedulability of the tasks and schedules them using some scheduling algorithm. *Verifier* proves the feasibility of the scheduled set of tasks by showing if they satisfy all the given real-time and embedding constraints. Finally, *Generator* is used to generate the application code based on the decisions made in the other components.

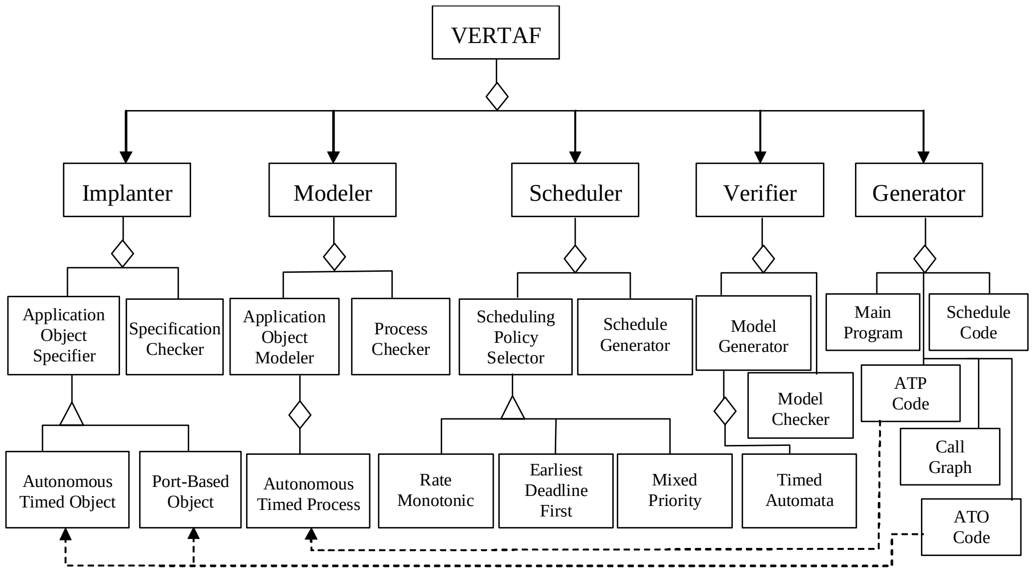

Figure 1. Component view of VERTAF

3.1. Implanter

The *Implanter* component acts as the main interface between application domain objects and VERTAF. *Application domain objects* are those objects that constitute the application that the designer desires to design. Through the use of Implanter, a designer can spend most of his/her efforts and time on the organization of application-specific details, instead of re-implementing objects representing real-time tasks.

A standard uniform object model is provided in Implanter such that all embedded real-time tasks can be specified using that model. In the following, we will describe the newly proposed *Autonomous Timed Object* (ATO) model for application domain object specification.

ATO incorporates advantageous features of two object models, namely *Port-Based Object* (PBO) [19] and *Time-triggered Message-triggered Object* (TMO) [21]. PBO is suitable for modeling embedded objects with standard interfaces such as in, out, and resource ports. Like PBO, ATO also adopts a standard interface for objects. Unlike PBO, ATO need not be independent and ATO methods (functions) need not be of a single type (the *cycle* method for both periodic and aperiodic tasks). Only the external interface of an ATO is adopted from PBO, while the internal methods are adaptations of those defined in TMO.

The basic structure of our newly proposed ATO is illustrated in Figure 2. There are four types of ports leading to and from an ATO, namely *configuration*, *in*, *out*, and *resource* ports. An ATO is initialized through the configuration ports. Instantiation is required because an ATO may be a generic class or a generic component. For example, a protocol stack component specified as an ATO may contain some parameters (counters, timers, access rates, …) which need to be assigned constant values before the protocol stack is deployed for use. After instantiation, an ATO may be configured either as a periodic or an aperiodic task. For aperiodic task configuration, it may be activated through resource ports that are connected to sensors or through events implemented in shared memory. For periodic task configuration, ATO is activated by a timer implemented in VERTAF. Upon activation, ATO reads data from in ports, executes corresponding methods, computes results, and writes data on out ports. ATO interface is suitable for modeling embedded objects due to its generic format.

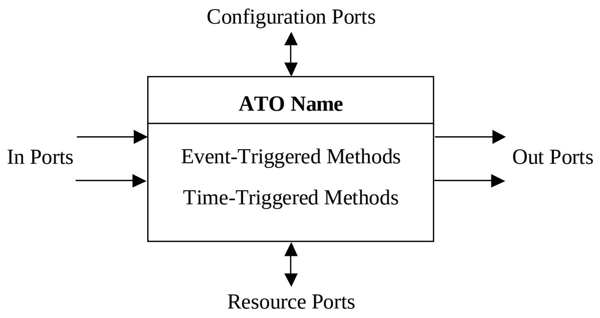

Figure 2. Autonomous Timed Object

Within ATO, there are two types of methods, namely *Event-Triggered Methods* (ETM) and *Time-Triggered Methods* (TTM). ETM are conventional object methods that execute only when called by another object, that is, it is triggered by a method call. ETM is used for modeling aperiodic task execution, since aperiodic tasks are also triggered by some in-coming event. TTM are object methods that were created due to the requirement of timely and predictable behavior from real-time systems. TTM are also called spontaneous methods in TMO. Execution of TTM does not require any in-coming event; TTM merely starts execution upon reaching a pre-specified time point. As far as inter-ATO interactions are concerned, ETM is one way of interacting, and another way is through global and local state variable tables as defined in the PBO model. State variable tables have lesser overhead when implemented in shared memory than message passing mechanisms. Thus, they are more appropriate for embedded systems.

3.2. Modeler

Every syntactic model must have a *semantic* model, which controls precisely how the model must behave in a dynamic environment. Corresponding to the ATO model, we next define its dynamic behavior using an *Autonomous Timed Process* (ATP) model. Each instance of an ATO has one corresponding ATP, which means there may be more than one ATP associated with a generic ATO in a system under design. The number of ATP associated with a generic ATO usually depends on the number of use cases the ATO has.

Figure 3 illustrates a basic ATP. Upon an ATO declaration, a new ATP is created, which is then configured into an instantiated object process. A newly created process, being unaware of the current system state, is updated through its in ports. This updated state is a stable state in which a process resides until it receives an interrupt. There are two types of interrupts that an ATP can receive: event and timer. An event interrupt indicates an aperiodic or sporadic task, and a corresponding event-triggered method is executed. A timer interrupt indicates a periodic task, and a corresponding time-triggered method is executed. After each method execution, all related temporal constraints are checked for violation or satisfaction. If a constraint is violated, then the ATP enters an *Error* state. ATP is reset by an error handling routine and then enters *Updated* state. A kill signal may be received before or after method execution, which terminates the process.

A standard uniform process model in the form of ATP increases the predictability of an embedded real-time application and also its ease of analysis and its verification scalability. In contrast to the *framework process* defined for PBO, ATP is not independent. When an ATP receives an event, it knows which ATP is the generating source of the event. All such events passed among ATP are recorded in an *Event Table*, such that a record consists of the source ATP, the destination ATP, the event type, and the associated variable values. The event table can also be represented as a *Call-Graph*, which is a directed graph $G = (V, E)$, where nodes in V represent ATP and arcs in E represent the call relationships (event propagation) between two ATP. This graph is useful for schedulability test, resource allocation, scheduling, and conflict resolution.

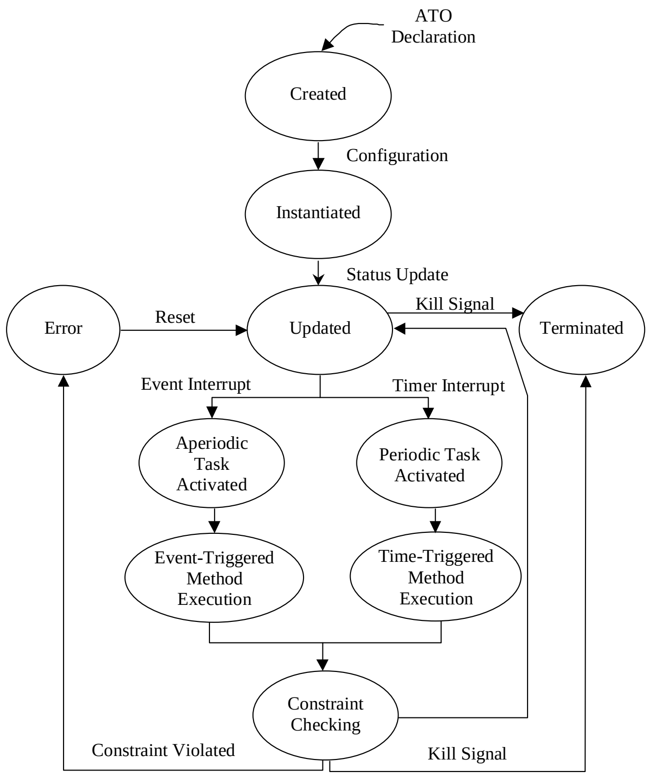

Figure 3. Autonomous Timed Process

Besides the event table, another table called the *Process Table* records all the ATP constructed by the *Modeler*. A record in the process table consists of the ATP index, the associated ATO methods, and the execution time, period, deadline, type of priority (fixed or dynamic), and resource requirements for each method. The resource requirement is specified as a real-numbered vector, where each element corresponds to some system resources such as memory, processor utilization, etc. and the real-number corresponds to the amount of each resource required by the particular ATO method. System resources are specified by the designer within *Implanter* through ATO instantiation of application domain objects.

3.3. Scheduler

The *Call-Graph* and the *Process Table* generated by *Modeler* are scheduled into a feasible application by *Scheduler*. There are several priority-based scheduling policies such as *rate-monotonic* (RM), *earliest-deadline first* (EDF), *mixed priority* (MP), *pin-wheel* (PW), etc. It is sometimes evident from the application as to which scheduling policy should be applied. But, in most applications, the prime concern is the satisfaction of the timing constraints, irrespective of which scheduling algorithm is applied. *Scheduler* includes a design pattern similar to the *Strategy Pattern* [1] adapted to real-time systems. In this pattern, one and only one scheduling algorithm must be chosen from the set of all scheduling algorithms for scheduling the ATP or tasks characterized in the *Call-Graph* and the *Process Table*.

Scheduler component mainly consists of two parts: a *Policy Selector* (PS) and a *Schedule Generator* (SG). The designer can choose to assign a particular scheduling policy he/she deems fit or the designer can also choose to allow VERTAF determine automatically the right choice. The decision is made by performing schedulability tests using each scheduling algorithm. One of the scheduling algorithms in the successful cases is then selected as the automatic decision result. When more than one algorithm can perform the scheduling, the selection can be arbitrary or based on some criteria such as the shortest schedule length (i.e., the shortest scheduled time). Currently, VERTAF leaves this option to the designer. *Schedule Generator* generates the actual start/end timing of each ATP based on the schedule policy chosen and on the *Call-Graph* constraints such as precedence relationships.

In priority-based scheduling, a well-known problem arises, namely the *priority inversion problem*, which occurs when a high priority task is blocked from execution due to some required resource being held by a low priority task, while a middle priority task with a long execution time preempts the low priority task resulting in the high priority task exceeding its deadline. We adopt the *priority inheritance approach* [22] to solve this problem.

3.4. Verifier

The ATP model allows all tasks to have a uniform dynamic behavior representation. ATP can be viewed as a finite state machine. In *Verifier* component, each ATP is further translated into a *Timed Automaton* (TA) [4]. In the following, the set of integers and non-negative real numbers are denoted by $\boldsymbol{N}$ and $\boldsymbol{R}_{\geq 0}$, respectively.

Definition 1: **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := false \mid x{\sim}c \mid x{-}y{\sim}c \mid d{\sim}c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim\, \in \{\leq, <, =, \geq, >\}$, $c \in \boldsymbol{N}$, $d \in D$, and η_1, η_2 are mode predicates.

Let $B(C, D)$ represent the set of all mode predicates over C and D. A TA is composed of various *modes* interconnected by *transitions*. Variables are distinguished into *clock* and *discrete*. Clock variables increment at a uniform rate and can be reset on a transition. Discrete variables change values only when assigned a new value

```
Symbolic_Mcheck(S, φ)
Set of TA S;
TCTL formula φ;
{
    Let Reach = Unvisited = {R_init};
    While (Unvisited ≠ NULL) {
        R′ = Dequeue(Unvisited);
        For all out-going transition e of R′ {
            R″ = Successor_Region(R′, e);
            If R″ is consistent and R″ ∉ Reach
        {
            Reach = Reach ∪ {R″};
            Queue(R″, Unvisited); }}}
    Label_Region(Reach, φ);
    Return L(R_init);
}
```

Figure 4. Symbolic Model Checking

on a transition.

Definition 2: **Timed Automaton**
A *Timed Automaton* (TA) is an 8-tuple $A = (M, m^0, C, D, X, E, T, R)$ such that: M is a finite set of modes, $m^0 \in M$ is the initial mode, C is a set of clock variables, D is a set of discrete variables, $X\colon M \rightarrow B(C, D)$ is an invariance function that labels each mode with a condition true in that mode, $E \subseteq M{\times}M$ is a set of transitions, $T\colon E \rightarrow B(C, D)$ defines the transition triggering conditions, and $R\colon E \rightarrow 2^{C\cup(D\times N)}$ is an assignment function that maps each transition to a set of assignments such as resetting clock variables and setting discrete variables to integer values.

A real-time system is often modeled as a network of communicating TA. The TA may share global variables including clock and discrete. State-spaces of a real-time system modeled by a set of TA are generally very large and grows exponentially with the large time constant and the system degree of concurrency.

A temporal constraint can be specified using *Timed Computation Tree Logic* (TCTL) [5].

Definition 3: Timed Computation Tree Logic (TCTL)
A timed computation tree logic formula has the following syntax: $\phi ::= \eta \mid \exists\Box\phi \mid \exists\phi\mathbf{U}_{\sim c}\phi' \mid \neg\phi \mid \phi\vee\phi'$ (1)
Here, η is a mode predicate (Definition 1), ϕ, ϕ' are TCTL formulae, $\sim\, \in \{<, \leq, =, \geq, >\}$, and $c \in \boldsymbol{N}$. $\exists\Box\phi$ means there exists a computation, from the current state, along which ϕ is always true. $\exists\phi\mathbf{U}_{\sim c}\phi'$ means there exists a computation, from the current state, along which ϕ is true until ϕ' becomes true, within the time constraint of ~c. Traditional shorthands like $\exists\Diamond$, $\forall\Box$, $\forall\Diamond$, $\forall\mathbf{U}$, $\wedge$, and $\rightarrow$ can all be defined [5].

The resulting set of TA and a TCTL specification can be then verified using the popular *model checking* technique. As shown in Figure 4, given a set of timed automata *S* modeling a real-time system and a TCTL property specification ϕ, model checking answers if *S* satisfies ϕ. A model checker can be implemented using a labeling algorithm, which labels each region (collection of states) recursively with a Boolean condition on the satisfaction of intermediate formulae of a given TCTL specification. There is extensive theory on model checking, interested readers may refer to [5] for further details.

Some formal verification tools that perform model checking include UPPAAL [23], KRONOS [24], and SGM [25], [26]. A kernel portion of such a model checker is implemented in VERTAF as the *Verifier* component. Thus, an application can automatically verify if the set of specified ATP satisfy any given real-time constraint. It must be noted here that VERTAF allows verification only after scheduling. The reason behind such a restriction is that after scheduling the degree of non-determinism in the dynamic behavior of a system is much lesser than that before scheduling. This approach allows more scalable verification for embedded real-time applications as has been emphasized in [27].

3.5. Generator

Generator component of VERTAF is responsible for generating the OO code for an embedded real-time application under development by a designer. The codes of all the previously described four components are used in this component to generate the final application code. It consists of six parts: VERTAF main program, implanter code, modeler code, scheduler code, verifier code, and execution code.

Two different types of main programs can be generated depending on whether there is a *Real-Time Operating System* (RTOS) or not. When there is an RTOS installed in an embedded system, a set of ATP is generated by the modeler. The ATPs execute as conventional real-time processes within the installed RTOS. When there is no RTOS, ATP is not generated by the modeler because there is no OS to handle and execute them. Instead, a real-time executive takes charge by scheduling and executing the ATO specified by a user. Another difference is that processes can be preemptive when there is an RTOS, whereas the methods in an ATO, once started, must execute to completion. Process and event tables are generated in both cases for schedulability analysis.

The main OO program maintains a global clock, which is used for recording progress in the developed system or application. It also contains exception handler for error recovery, fault handling, and other mechanisms to handle exceptions such as constraint violations. The main program ensures that the system is always in an acceptable state by monitoring for constraint violations such as missed deadlines.

4. Application Development

After an overview of the VERTAF framework, this section describes how one actually designs a real-time application using VERTAF. Figure 5 illustrates the development strategy (central column) in context of both the Components (left column) and the Classes (right column) of VERTAF. Rectangular boxes represent processes to be accomplished either by the user or by VERTAF. Ovals represent class instantiations. There are three phases in the application development of VERTAF, namely, *Specification*, *Integration*, and *Generation*.

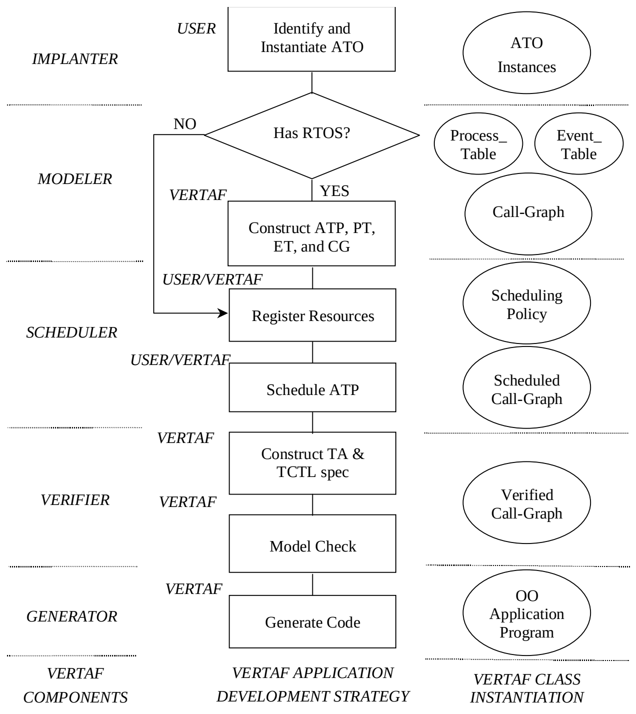

Figure 5. Application Development in VERTAF

- **Specification**: A user of VERTAF begins with specifying his/her system. The user may either define application domain objects (such as in the AICC example presented in Section 5) or directly input tasks by instantiating the *Process_Table* class and the *Call_Graph* class. When application domain objects are specified, *Process_Table* and *Call_Graph* are instantiated in VERTAF by the *Modeler* component.
- **Integration**: At this stage, we have the *Process_Table* and *Call_Graph* instantiated either by the user or by *Modeler* in VERTAF. VERTAF then distinguishes passive objects from active ones. This distinction is required so that implicit resources may be identified. Any object that does not actively call other object methods is called a passive object; otherwise it is an active one. ATP from the *Process_Table* and their interdependencies from the *Call_Graph* are scheduled

using a user-specified scheduling policy or through automatic decision by VERTAF.

- **Generation**: After integration, we already have the skeleton for a feasible application where tasks have been explicitly distinguished, registered, scheduled, resources allocated without conflicts, and timing constraints satisfied through scheduling. Code generation is carried out by VERTAF as described in Section 3.5.

5. Application and Experimental Results

An industrial application example developed using VERTAF is presented in this section: a *cruiser* example consisting of 12 tasks used to control the vehicle speed under different circumstances. The benefits of using VERTAF in developing the example have been evaluated and the results show a marked improvement in design productivity and efficiency.

AICC (*Autonomous Intelligent Cruise Controller*) system application [29] was developed and installed in a Saab automobile by Hansson et al. The AICC system can receive information from road signs and adapt the speed of the vehicle to automatically follow speed limits. Also, with a vehicle in front cruising at lower speed the AICC adapts the speed and maintains safe distance. The AICC can also receive information from the roadside (e.g. from traffic lights) to calculate a speed profile which will reduce emission by avoiding stop and go at traffic lights. The system architecture consisting of both hardware (HW) and software (SW) is as shown in Figure 6. The software development methodology used in [29] is based on sets of interconnected so-called *software circuits* (SC). Each SC has a set of input connectors where data is received and a set of output connectors where data is produced. We model the software circuits in [29] as *active application domain objects* in VERTAF.

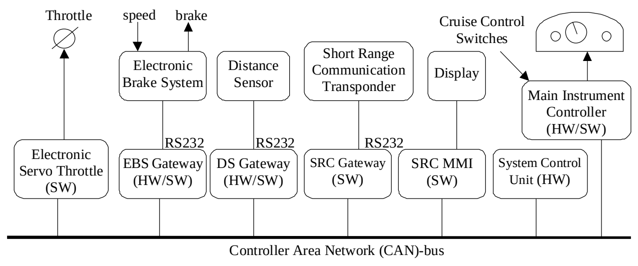

Figure 6. AICC Example: System Architecture

As shown in Figure 7, 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 [29], within the *Swedish Road Transport Informatics Programme*. Each object may correspond (map) to one or more tasks. *Process Table* and *Call-Graph* generated by the *Modeler* component are as shown in Table 1 and Figure 7, respectively. There are totally 12 tasks performed in 5 objects. Two different resources were identified in VERTAF, namely, SRC and Display. This application took 5 days for a real-time system designer using VERTAF. The same application took the same designer 20 days to complete development. This significant decrease in design time was because VERTAF automatically extracted the tasks and constraints from the object specifications.

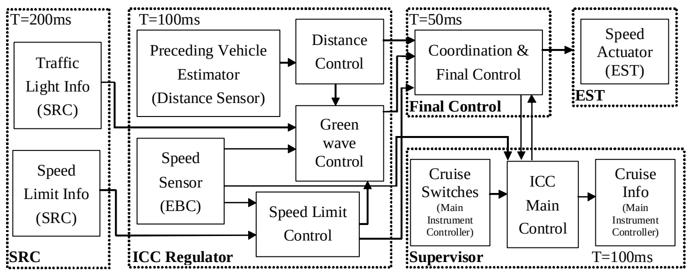

Figure 7. AICC Example: Call-Graph

Table 1. AICC Example: Process Table

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

P: Period, E: Execution Time, D: Deadline, *in ms, SRC: *Short Range Communication*, ICCReg: *ICC Regulator*, EST: *Electronic Servo Throttle*

6. Conclusion

An object-oriented application framework, called VERTAF, was proposed for embedded real-time systems application development. It was a result of the integration of three different technologies: *object-oriented* technology, *software component* technology, and *formal verification* technology. The integration resulted in verifiable objects and components, and thus eliminated design errors at an early stage. Different levels of re-use, including object-level and component-level, increased design productivity and decreased overall design effort and time.

A new *Autonomous Timed Object* model was proposed for users to implant real-time objects into the framework and a corresponding *Autonomous Timed Process* model was proposed for modeling its dynamic behavior and for system verification. Totally, five components were developed and presented in VERTAF, including *Implanter*, *Modeler*, *Scheduler*, *Verifier*, and *Generator*. An application example was developed using VERTAF, which showed how design time is significantly reduced due to a large extent of object and code reuse from VERTAF.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

[2] R. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming*, Vol. 1, pp. 22–35, June 1988.

[3] C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard-real time environment,” *Journal of the Association for Computing Machinery*, Vol. 20, pp. 46–61, January 1973.

[4] R. Alur and D. Dill, “Automata for modeling real-time systems,” *Theoretical Computer Science*, Vol. 126, No. 2, pp. 183–236, April 1994.

[5] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, “Symbolic model checking for real-time systems,” in *Proceedings IEEE Logics in Computer Science*, 1992.

[6] A. Attoui and M. Schneider, “An object oriented model for parallel and reactive systems,” in *Proceedings Real-Time Systems Symposium*, pp. 84–93, December 1991.

[7] D. Hammer, L. Welch, and O. van Roosmalen, “A taxonomy for distributed object-oriented real-time systems,” *ACM OOPS Messenger*, Vol. 7, pp. 78–85, January 1996.

[8] Y. Ishikawa, H. Tokuda, and C. W. Mercer, “Object-oriented real-time language design: Constructs for timing constraints,” *ACM SIGPLAN Notices*, *ECOOP/OOPSLA'90 Proceedings*, Vol. 25, pp. 289–298, October 1990.

[9] B. Achauer, “Objects in real-time systems: Issues for language implementors,” *ACM OOPS Messenger*, Vol. 7, pp. 21–27, January 1996.

[10] L. R. Welch, “A metrics-driven approach for utilizing concurrency in object-oriented real-time systems,” *ACM OOPS Messenger*, Vol. 7, pp. 70–77, January 1996.

[11] 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, January 1996.

[12] 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, January 1996.

[13] W.-B. See and S.-J. Chen, “Object-oriented real-time system framework,” in *Domain-Specific Application Frameworks* (M. E. Fayad and R. E. Johnson, eds.), ch. 16, pp. 327–338, John Wiley, 2000.

[14] W.-B. See and S.-J. Chen, “High-level reuse in the design of an object-oriented real-time system framework,” in *Proceedings International Computer Symposium*, pp. 363–370, December 1996.

[15] T. Kuan, W.-B. See, and S.-J. Chen, “An object-oriented real-time framework and development environment,” in *Proceedings OOPSLA'95 Workshop #18*, 1995.

[16] P.-A. Hsiung, “RTFrame: An Object-Oriented Application Framework for Real-Time Applications,” in *Proceedings of the 27th International Conference on Technology of Object-Oriented Languages and Systems* (TOOLS'98), pp. 138–147, IEEE Computer Society Press, September 1998.

[17] P.-A. Hsiung, “Object-Oriented Application Framework Design for Real-Time Systems,” in *Proceedings of the 4th International Symposium on Real-Time and Media Systems* (RAMS'98), pp. 221–227, September 1998.

[18] D. Schmidt, “Applying design patterns and frameworks to develop object-oriented communication software,” *Handbook of Programming Languages*, Vol. I, 1997.

[19] D. B. Stewart, R. A. Volpe, and P. K. Khosla, “Design of dynamically reconfigurable real-time software using port-based objects,” *IEEE Transactions on Software Engineering*, Vol. 23, No. 12, December 1997.

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

[21] K. H. Kim, “APIs for Real-Time Distributed Object Programming,” *IEEE Computer*, Vol. 33, No. 6, pp. 72–80, June 2000.

[22] L. Sha, R. Rajkumar, and J. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *Technical Report CMU-CS-87-181*, Computer Science Department, Carnegie Mellon University, November 1987.

[23] J. Bengtsson, K. Larsen, F. Larsson, P. Petterson, Y. Wang, and C. Weise, “New Generation of UPPAAL,” *Proceedings of the International Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

[24] C. Daws, A. Olivers, S. Tripakis, and S. Yovine, “The tools KRONOS,” *Hybrid System III*, Lecture Notes in Computer Science, Vol. 1066, pp. 208–219, 1996.

[25] P.-A. Hsiung and F. Wang, “User-friendly verification,” in *Proceedings of IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques For Distributed Systems and Communication Protocols & Protocol Specification, Testing, And Verification*, (FORTE/PSTV '99), October 1999.

[26] F. Wang and P.-A. Hsiung, “Efficient and User-Friendly Verification,” *IEEE Transactions on Computers*, Vol. 51, No. 1, pp. 61–83, January 2002.

[27] P.-A. Hsiung, “Embedded Software Verification in Hardware-Software Codesign,” *Journal of Systems Architecture — the Euromicro Journal*, Vol. 46, No. 15, pp. 1435–1450, Elsevier Science, December 2000.

[28] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*, Addison Wesley, USA, January 2000.

[29] 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.