Formal Verification of Embedded Real-Time Software in Component-Based Application Frameworks[1]

Pao-Ann Hsiung†, Win-Bin See‡, Trong-Yen Lee§, Jih-Ming Fu♯, and Sao-Jie Chen‡
†Dept. of Computer Science and Information Engineering, National Chung Cheng University
‡Dept. of Electrical Engineering, National Taiwan University
§Dept. of Electrical Engineering, Chung Cheng Institute of Technology, National Defense University
♯Dept. of Electronic Engineering, Cheng-Shiu Institute of Technology
†Chiayi–621, ‡Taipei–106, §Taoyuan–335, ♯Kaohsiung–833, Taiwan.

Abstract

Producing correct software is a premier goal for application frameworks that are targeted at Embedded Real-Time Systems because incorrect software are not only of no use but might also cause severe system damage. It is shown how formal verification can be elegantly, seamlessly, and scalably integrated into a component-based object-oriented application framework for embedded real-time systems. Two issues in such a technology integration are addressed: (1) the choice of a common system model, and (2) the integration of formal synthesis and model checking. Solutions are provided, respectively, in the form of: (1) proposing a new Formal Object-Oriented Model (FOOM), *and (2) the execution of model checkers within synthesis algorithms. Technically, we propose a compositional software verification framework, in which model checking is employed, with state-space reduction techniques adapted for embedded real-time software. A separate Verifier component is proposed for modular integration as illustrated by its implementation in the VERTAF application framework. An example illustrates the success of our approach and the benefits gained through integrating formal verification.*

1 Introduction

According to recent statistical studies, software accounts for as much as 80% of the functionalities in embedded real-time systems such as home appliances, information appliances, personal assistants, telecommunication gadgets, and transportation facilities. Software is also much more complex than hardware due to its inherent flexibilities and the infinite number of valuations possible for a variable. It is often found that an on-market real-time embedded system fails due to some simple software glitches, which could have been avoided if the software was thoroughly verified. All these facts show that verifying the correctness of software is a demanding and important issue in the design phase of an embedded real-time system.

To guarantee the correctness of embedded real-time software, instead of a case-by-case verification, the work presented in this article takes a pioneer step in introducing *formal verification* (FV) into a *component-based object-oriented application framework* (COAF). A systematic integration of FV technology with COAF technology will be our prime concern and our goal will be to make this integration not only *seamless*, but also *scalable*.

There are two issues to be addressed for the integration of FV and COAF technologies.

- *System Model*: On one hand, COAF perceives a system as a collection of interacting components or objects with possibly complex behaviors, while on the other hand, FV observes a system as a set of concurrent real-time tasks with formal syntax and precise semantics. If COAF and FV are to be integrated, which model do we use or do we invent a new model?
- *Design Methodology v/s Verification Framework*: COAF generates software by going through a complete design methodology, while FV analyzes software by going through a verification framework. How can the methodology and framework work together with mutual benefits and a common goal of generating correct executable software?

We propose the following solutions to the above issues.

- *Formal Object-Oriented Model* (FOOM): A trade-off between component-based object-oriented model

[1]This work was supported in part by the project grant NSC90-2215-E-194-009 from the National Science Council, Taiwan.
Correspondence E-mail: hpa@computer.org

and formal model is proposed as a combination of the two into a new *Formal Object-Oriented Model* (FOOM), which stratifies granularity into a coarser, user-apprehensible, *object-based design* abstraction level and a finer, technology-operable, *process-based verification* abstraction level. The execution semantics of an object is defined by one or more processes. This model will be described in details in Section 3.

- *Formal Synthesis and Model Checking* (FSMC): Formal methods have been applied to the synthesis [1, 4, 14] as well as the verification [2, 7, 9, 12, 13, 16] of embedded real-time systems. Using the same formal model such as *Timed Automata* [3] or *Time Free-Choice Petri Nets* [14], model checking procedures can be called from within synthesis algorithms. The basic framework will be *compositional* verification with modular packaging of verification techniques. Further details will be given in Section 4.

The proposed solutions to COAF-FV technology integration issues are currently being implemented in a component-based object-oriented application framework called VERTAF [16]. VERTAF generates code for embedded real-time systems using formal modeling and synthesis techniques. A separate software component called *Verifier* is being developed in VERTAF for encapsulating the proposed solutions.

This article is organized as follows. Section 2 will give a brief account of some previous work on COAF, FV, and their integration. Section 3 will describe the *Formal Object-Oriented Model* (FOOM), which can be used for both design and verification. Section 4 will discuss how the *Formal Synthesis and Model Checking* processes work hand-in-hand for a common goal. Section 5 will illustrate the feasibility and success of the proposed technology integration framework through its implementation as a *Verifier* component in the VERTAF application framework and through some application examples. Section 6 will conclude the article with some research directions for future work.

2 Previous Work

Currently, there are very few component-based object-oriented frameworks developed specifically to generate code for embedded real-time systems. In the following, we first summarize three of such frameworks.

Two recently proposed COAF are *Object-Oriented Real-Time System Framework* (OORTSF) [20, 23, 24] and SESAG [10]. 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. 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 were not described in the above two works. Another more recent framework called VERTAF [16] is an enhanced version of SESAG, incorporating software component technology, formal verification technology, industry standards such as *Unified Modeling Language* (UML) and Java.

As far as formal software synthesis is concerned, it was mainly performed for communication protocols [22], plant controllers [4], and real-time schedulers [1] because they generally exhibited regular behaviors. Recently, there has been some work on automatically generating code for embedded systems [5, 21, 25]. Lin [21] proposed an algorithm that generates a software program from a concurrent process specification through an intermediate safe Petri-Net representation by applying *quasi-static scheduling*. Sgroi et al. [25] proposed a software synthesis method for a more general Petri-Net framework, called *Free-Choice Petri Nets* (FCPN). A necessary and sufficient condition was given for a FCPN to be schedulable. Schedulability was first tested for a FCPN and then a valid schedule generated by decomposing a FCPN into a set of *Conflict-Free* components. Code was finally generated from the valid schedule. Balarin et al. [5] proposed a software synthesis procedure for reactive embedded systems in the *Codesign Finite State Machine* (CFSM) [6] framework with the POLIS hardware-software codesign tool [6]. This work cannot be easily extended to other more general frameworks.

As far as formal software verification is concerned, Hsiung [12] gave solutions to the questions of *when*, *where*, and *how* software verification is to be performed for an embedded real-time system. In answer to the when question, the *Schedule-Verify-Map* method was proposed, which states that verification should be performed after scheduling. In answer to the where question, verification under system concurrency, instead of under process concurrency, was proposed, which is justified by the fact that system concurrency is generally much smaller than process concurrency. In answer to the how question, symbolic model checking was integrated with the other two solutions to provide a complete solution to the verification of embedded real-time software. Other work include the linear hybrid automata (LHA) model based hardware-software timing coverification techniques [11, 13], and the coverification strategy for automatic mapping to LHA [7].

3 Formal Object-Oriented Model

As a compromise between the object-oriented model used by engineers and the formal model used by scientists, a *Formal Object-Oriented Model* (FOOM) is proposed for

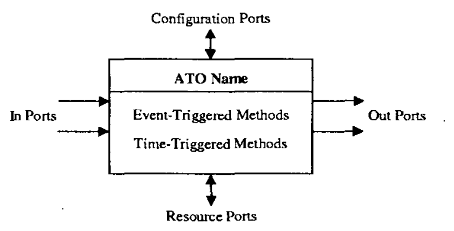

Figure 1. Autonomous Timed Object

introducing formal verification into an application framework. Syntactically, FOOM consists of a uniform representation called *Autonomous Timed Object* (ATO) for task specification by a software designer. Semantically, FOOM consists of a uniform representation called *Autonomous Timed Process* (ATP) for modeling the behavior of all tasks, which can be used for verification after transforming into other formal models.

3.1 Autonomous Timed Object

Autonomous Timed Object (ATO) incorporates advantageous features of two object models, namely *Port-Based Object* (PBO) [26] and *Time-triggered Message-triggered Object* (TMO) [19]. The basic structure of our newly proposed ATO is illustrated in Figure 1. 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. 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.

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. 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. 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, hence are more appropriate for embedded systems.

3.2 Autonomous Timed Process

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 or more corresponding ATP, which means there may be more than one ATP associated with a generic ATO in a system under design. The number of ATPs associated with a generic ATO usually depends on the number of use cases the ATO has.

Upon an ATO declaration, one or more new ATPs are created, which are then configured into instantiated object processes. 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, terminating 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 ATPs 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 ATPs and arcs in E represent the call relationships (event propagation) between two ATPs. This graph is useful for schedulability test, resource allocation, scheduling, and conflict resolution. Besides the event table, another table called the *Process Table* records all the ATPs in a system. 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.

4 Formal Synthesis and Model Checking

Formal synthesis is an analytic method by which a formally modeled system is made to satisfy a given logic specification. For example, a soft embedded real-time system, modeled by a set of *Time Free-Choice Petri Nets* (TFCPN), is made to satisfy a system property specification given in *Timed Reachability Specification* (TRS) logic [15].

Model checking is defined as an algorithmic procedure by which a system can be formally and automatically verified to check if it satisfies a given logic specification. For example, a concurrent real-time system modeled by a set of *Timed Automata* can be model checked for satisfaction of a *Timed Computation Tree Logic* (TCTL) specification [9].

Before going into the details of technology integration, some definitions are required and are described as follows. The sets of integers and non-negative real numbers are denoted by $\mathcal{N}$ and $\mathcal{R}_{\geq 0}$, respectively.

A timed automaton (TA) is composed of various *modes* interconnected by *transitions*. Variables are segregated into categories of *clock* and *discrete*. Clock variables increment at a uniform rate and can be reset on a transition, whereas discrete variables change values only when assigned a new value on a transition. A TA may remain in a particular mode as long as the values of all its variables satisfy a *mode predicate*, which is a conjunction of clock constraints and boolean propositions.

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: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg \eta_1$, where $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. ||

Let $B(C, D)$ be the set of all mode predicates over C and D.

Definition 2 *:* **Timed Automaton**
A *Timed Automaton* (TA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode, $E_i \subseteq M_i \times M_i$ is a set of transitions, $\tau_i : E_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values. ||

4.1 Technology Integration Framework

Our target problem is formulated as follows:

Definition 3 *:* **COAF-FV Technology Integration**
Given an embedded real-time system described in a *Component-Based Object-Oriented Application Framework* (COAF) using the *Formal Object-Oriented Model* (FOOM) along with a set of temporal constraints, generated software code is to be formally verified to satisfy all given constraints. ||

As a solution to the above problem, we propose the following technology integration framework. Given an embedded real-time system described using a set of ATOs $\{Q_1, Q_2, \ldots, Q_n\}$ and a set of constraints, the behavior of each ATO Q_i is modeled using one or more ATP $\{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ and a TCTL specification ϕ (c.f. Definition 4) is generated from the set of constraints.

Definition 4 *:* **Timed Computation Tree Logic Formula**
A timed computation tree logic formula has the following syntax: $\phi ::= \eta \mid \exists \Box \phi' \mid \exists \phi' \mathcal{U}_{\sim c} \phi'' \mid \neg \phi' \mid \phi' \vee \phi''$. Here, η is a mode predicate in $B(\cup_{i=1}^{n} C_i, \cup_{i=1}^{n} D_i)$, ϕ', ϕ'' are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$, and $c \in \mathcal{N}$. $\exists \Box \phi'$ means there exists a computation, from the current state, along which ϕ' is always true. $\exists \phi' \mathcal{U}_{\sim c} \phi''$ means there exists a computation, from the current state, along which ϕ' is true until ϕ'' becomes true, within the time constraint of $\sim c$. Traditional shorthands like $\exists \Diamond$, $\forall \Box$, $\forall \Diamond$, $\forall \mathcal{U}$, $\wedge$, and $\rightarrow$ can all be defined as in [9]. ||

As detailed in Table 1, we propose a *compositional* integration framework, which provides an elegant interaction between *software components* and *verification manipulators*. Here, a verification manipulator is a modular packaging of verification techniques such as state-space reduction and concurrent process merging.

First, a TCTL specification formula ϕ is generated by **Gen_TCTL**() procedure in Step (1) of Table 1. In Step (2), given a set of ATPs $ATP\_Set = \{P_{11}, P_{12}, \ldots, P_{1k_1}, P_{21}, \ldots, P_{2k_2}, \ldots, P_{n1}, \ldots, P_{nk_n}\}$, **Gen_TA**() generates a set of timed automata $ATA\_Set = \{A_{11}, A_{12}, \ldots, A_{1k_1}, A_{21}, \ldots, A_{2k_2}, \ldots, A_{n1}, \ldots, A_{nk_n}\}$, where A_{ij} corresponds to P_{ij}. In Step (3), ATA_Set is then scheduled using some scheduling algorithm *Sched_Alg* by the procedure **Schedule**() into another set of TA, $STA\_Set = \{A_{11}^s, A_{12}^s, \ldots, A_{1k_1}^s, A_{21}^s, \ldots, A_{2k_2}^s, \ldots, A_{n1}^s, \ldots, A_{nk_n}^s\}$. Only after scheduling do we start performing verification. Within our framework, *Sched_Alg* is taken as *quasi-static scheduling* [14, 25]. In Step (4), there is a **while** loop which iterates until the set STA_Set becomes a singleton (i.e., cardinality = 1), which implies that the global state-space of the set has been constructed. Within each iteration, **MROF**() in Step (5)

Table 1. Compositional Verification for COAF

```
Compositionally_Verify($ATP\_Set, Constraint\_Set$)
$ATP\_Set = \{P_{11}, \ldots, P_{1k_1}, P_{21}, \ldots, P_{n1}, \ldots, P_{nk_n}\}$;
$Constraint\_Set$; // Set of constraints
{
    $\phi$ = Gen_TCTL($Constraint\_Set$);                    (1)
    $ATA\_Set$ = Gen_TA($ATP\_Set$);                          (2)
    $STA\_Set$ = Schedule($ATA\_Set, Sched\_Alg$);            (3)
    while ($|STA\_Set| > 1$) {                                 (4)
        MROF($STA\_Set$);                                      (5)
        $\vec{r}$ = FBRS($STA\_Set$);                          (6)
        Reduce($STA\_Set, \vec{r}$);                           (7)
    }
    if(Model_Check($STA\_Set, \phi$)) return $Verified$;       (8)
    else return $Constraints\_Violated$;                       (9)
}
```

merges two most-related TA from STA_Set into one TA by representing their concurrent behavior (see Section 4.2. **FBRS()** in Step (6) searches for the best sequence $\vec{r}$ of reduction manipulators which reduces the current state-space the most. In Step (7), the actual reduction of state-space is performed. After the global state-space is constructed, it is model-checked by procedure **Model_Check()** in Step (8).

Since the proposed technology integration framework is *compositional*, the global state-space of a given set of TA is constructed *iteratively* such that in each iteration two TA are selected for merging into one TA, which represents the state-space of their concurrent behavior. After merging in each iteration, the intermediate state-spaces are then reduced using a sequence of reduction techniques. Thus, there are two decisions which affect verification scalability, namely *merge sequence* and *reduction sequence*

4.2 Merge Related Objects First

As solution for the first decision issue on merge sequence, we give details of the **MROF()** procedure in Step (5) of Table 1. Selecting a different pair of TA for merging in an iteration affects how large the intermediate state-space (also a TA) can grow.

Suppose we are given a set of ATOs $\{Q_1, Q_2, \ldots, Q_n\}$, whose behaviors are represented by the set of ATPs $\{P_{11}, P_{12}, \ldots, P_{1k_1}, P_{21}, \ldots, P_{2k_2}, \ldots, P_{n1}, \ldots, P_{nk_n}\}$, where the behavior of Q_i is represented by $\{P_{i1}, \ldots, P_{ik_i}\}$. Let the set of TA that model the set of ATPs be $\{A_{11}, \ldots, A_{1k_1}, \ldots, A_{n1}, \ldots, A_{nk_n}\}$. After scheduling with some algorithm, let the scheduled set of TA be $\{A^s_{11}, A^s_{12}, \ldots, A^s_{1k_1}, A^s_{21}, \ldots, A^s_{2k_2}, \ldots, A^s_{n1}, \ldots, A^s_{nk_n}\}$. We adopt a hierarchical merge strategy which includes both syntax-based and semantics-based methods as described in the following steps.

1. *Same Family*: This is a syntax-based method. Since the ATPs that represent the behavior of the same ATO are more related to each other than to the ATPs of other ATOs, we first merge all the ATPs of the same family (i.e., the same ATO). Notationally, $\{A^s_{i1}, \ldots, A^s_{ik_i}\}$ are merged into A^m_1. Thus, after this step, instead of $\sum_{1 \le i \le n} k_i$ TA, there are only n TA now.
2. *Near Relatives*: This is a semantics-based method. Degrees of proximity are calculated for each pair of ATOs based on the number of shared discrete variables, clock variables, synchronization labels, and the number of communication channels. Higher the number of shared variables and communication channels, higher is the proximity degree. The pair with the highest proximity is said to be *near relatives* and is merged first. Notationally, the following proximity function is defined for each pair of TA:

$$\pi(A_i, A_j) = \text{Num\_Shared\_Variables}(A_i, A_j) + \text{Num\_Channels}(A_i, A_j) \quad (1)$$

After the first step of merging same family TA, A^m_u and A^m_v are merged first if $\pi(A^m_u, A^m_v) = \max_{1 \le i < j \le n}\{\pi(A^m_i, A^m_j)\}$.

4.3 Find Best Reduction Sequence

A *reduction technique* is a procedure, which takes as input a state-space and reduces its size in terms of the number of modes and transitions. A state-space can be represented by a TA and a reduction technique can be implemented as a modularly packaged *manipulator*. We consider the following manipulators, which were implemented in the *State-Graph Manipulators* (SGM) tool [17, 18, 28, 27]. SGM is a comprehensive, high-level, real-time system verification tool. We briefly describe how the manipulators may be used in our technology integration framework. Details on each manipulator can be found in [17, 18, 28]. Experimental results will be given in Section 5.

We only consider four reduction manipulators for our experiments. (1) *Symmetry reduction* is very useful for our framework because our underlying FOOM is symmetric. (2) *Clock shielding* is useful because our target systems consists of *concurrent, real-time* software components with several clocks. (3) *Read-Write reduction* is useful because besides time-triggered methods, an ATO also has one or more event-triggered methods, which depend on some communication variables or channels. (4) *Internal transition bypass* is a useful reduction manipulator due to the composition nature of our framework.

The following is how we obtained a heuristically optimal reduction sequence for our framework.

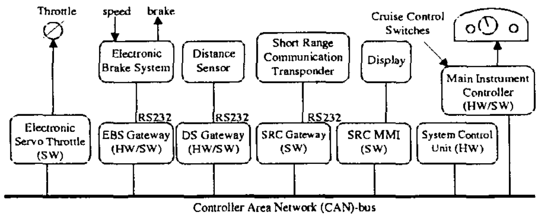

Figure 2. AICC Example: System Architecture (ATO)

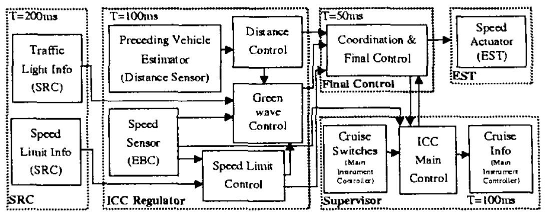

Figure 3. AICC Example: Call-Graph (ATP)

- If there is no clock variable, skip the shield clock reduction. If there is no discrete variable, skip the read-write reduction technique.
- Always perform symmetry reduction after read-write reduction because the information obtained from read-write reduction is useful for symmetry reduction.
- Perform internal transition bypass after read-write reduction and clock shielding because the information obtained by the other two techniques are useful for deciding if a transition is internal.
- Permute the reduction sequence by deciding when to perform symmetry reduction (after read-write).
- Generate the best sequence from the above experiments and heuristics.

We have compared the above heuristic method of obtaining the best reduction sequence with the theoretical method from [28]. The results are almost the same when the targets are fixed as embedded real-time systems in a framework.

5 Application Example

Autonomous Intelligent Cruise Controller (AICC) system application [8] 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 and 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 Fig. 2. The software development methodology used in [8] is based on sets of interconnected so-called *software circuits* (SC). Each SC has a set of input connectors where data are received and a set of output connectors where data are produced. We model the software

Table 2. AICC Example: Process Table

#	ATP	ATO	P^*	T^*	D^*
1	Traffic Light	SRC	200	10	400
2	Speed Limit	SRC	200	10	400
3	Proc Vehicle Est	ICCReg	100	8	100
4	Speed Sensor	ICCReg	100	5	100
5	Distance Ctrl	ICCReg	100	15	100
6	Green Wave Ctrl	ICCReg	100	15	100
7	Speed Limit Ctrl	ICCReg	100	15	100
8	Coordination	Final_Control†	50	20	50
9	Cruise Switches	Supervisor	100	15	100
10	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, * All times in milliseconds, P period, T execution time, D deadline, † Implemented in hardware.

circuits in [8] as *autonomous timed objects* in FOOM.

As shown in Fig. 3, there are five ATOs specified by the designer of AICC for implementing a *BASEMENT* system, namely *Short Range Communication* (SRC), *Intelligent Cruise Controller* (ICC) *Regulator*, *Final Control*, *Supervisor*, and *Electronic Servo Throttle* (EST). BASEMENT is a vehicle's internal real-time architecture developed in the *Vehicle Internal Architecture* project [8], within the *Swedish Road Transport Informatics Programme*. As observed in Fig. 3, each ATO may map to one or more ATP. *Call-Graph* and *Process Table* for AICC are shown in Fig. 3 and Table 2, respectively. There are 12 functions performed in 5 objects, out of which 11 functions are implemented in software. Thus, there are 11 ATPs in this system.

The 11 ATPs were scheduled by *Scheduler* component of VERTAF and transformed into 11 TA by the *Verifier* component of VERTAF based on the ATO description (Fig. 2), Call-Graph (Fig. 3), and Process Table (Table 2).

Based on the characteristics of ATP models in the AICC example, TA models are generated for each ATP. The proposed approaches to increasing verification scalability through different merge sequences and reduction sequences, as given in Section 4, were applied to the set of TA modeling the ATPs of AICC example. Table 3 gives results, which corroborate our claims.

Our experiments were performed on a Sun UltraSPARC-II 450 MHz machine with a single processor and 1 GB physical memory. We experimented with several different versions of the set of TA models. With respect to the number of TA, a full version consists of 11 TA, while a simplified version consists of 6 TA. Simplification was performed by removing some of the sensor tasks such as Speed Limit Info. System model consistency was preserved after the simplification by ensuring that the removed local execution paths do not affect any global verification results. Two types of communication models were considered: *shared memory* (SM) and *message passing* (MP). In the shared memory model, shared variables were used as primitives for data/control transfer. In the message passing model, send-receive events were used as primitives.

From Table 3, we make the following observations.

- The full version of 11 TA (Rows 1 and 2) required more CPU time and memory space compared to the simplified version of 6 TA (Rows 3–17). This fits our knowledge of larger state-spaces for highly concurrent systems.
- Message passing required more time and memory compared to shared memory (compare rows 3 with 6). This is because message passing uses events and broadcasting is expensive with event-based communication, while shared memory uses variables and broadcasting is automatic through concurrent memory reads.
- Fitting with our intuition, application of reduction techniques resulted in smaller state-spaces and lesser time and memory use (compare rows 3 with 5 and 6 with 7).
- mg_1 is a sequential merge according to the TA indices and mg_2 is a *near-relatives* merge as defined by Equation 1. Comparing rows 3 and 4, the second sequence gives a better result in terms of shorter CPU time and memory space utilizations compared to the first sequence. This corroborates our claims in Section 4.2.
- Comparing rows 7 to 17, the reduction sequence $\langle mg_1, rw, sm, sc, bit \rangle$ in row 14 gives the best results in terms of the smallest state-space size (i.e., number of modes and transitions). The CPU time and memory space usage are not the least (compare with row 10), because smaller state-spaces are sometimes obtained by spending a little extra time and space.
- The first case of 11 TA (Row 1) and without reduction could not execute to completion, which gives a general idea of how extremely large sized is the global system state-space.

All the above observations illustrate and corroborate our proposed techniques as described in Section 4. Thus, through this example we have shown how formal verification can be integrated into a complex component-based object-oriented application framework and applied to a real-world industrial example.

6 Conclusion

Using the proposed framework for technology integration of *Component-based Object-oriented Application Framework* (COAF) and *Formal Verification* (FV), a software engineer can be guaranteed a verified correct code for his/her application. Issues related to such a technology integration include deciding on a common system model and integrating formal synthesis and verification algorithms. Solutions were proposed in this work for each of the above issues, which include the proposal of a *Formal Object-Oriented Model* (FOOM) for system modeling and the compositional framework for technology integration. A software component called *Verifier* was implemented in the VERTAF application framework for formal verification of generated software. A real-world industrial example on cruise controller was given to illustrate the feasibility and success of our approach for integrating formal verification into an application framework. Future research directions related to this work include developing an API for users to implement his or her own reduction and verification techniques and exploring new reduction techniques based on design patterns.

References

[1] K. Altisen, G. Gobler, A. Pneuli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the Real-Time System Symposium (RTSS'99)*. IEEE Computer Society Press, 1999.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill. Modeling checking for real-time systems. In *Proceedings of the IEEE International Conference on Logics in Computer Science (LICS'90)*, 1990.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183 – 235, 1994.

[4] E. Asarin, O. Maler, A. Pneuli, and J. Sifakis. Controller synthesis for timed automata. In *Proceedings of System Structure and Control*. IFAC, Elsevier, July 1998.

[5] F. Balarin and M. Chiodo. Software synthesis for complex reactive embedded systems. In *Proceedings of International Conference on Computer Design (ICCD'99)*, pages 634 – 639. IEEE CS Press, October 1999.

[6] F. Balarin and et al. *Hardware-software Co-design of Embedded Systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[7] J.-M. Fu, T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen. Hardware-software timing coverification of distributed embedded systems. *IEICE Transactions on Information and Systems*, E83-D(9):1731–1740, September 2000.

Table 3. AICC Example: State-Space Reduction and Model Checking

#	n	Communication	Manipulator Sequence	#Modes	#Transitions	Time (s)	Memory (MB)
1	11	Shared Memory	$\langle mg_1 \rangle$	> 125 K	> 1869 K	N/A	Out of Memory
2	11	Shared Memory	$\langle mg_1, rw, sc, sm \rangle$	270	1138	50212.95	61.67
3	6	Message Passing	$\langle mg_1 \rangle$	19776	26677	1390.78	210.90
4	6	Message Passing	$\langle mg_2 \rangle$	19776	26677	234.38	76.64
5	6	Message Passing	$\langle mg_1, rw, sc, sm \rangle$	141	320	873.39	18.72
6	6	Shared Memory	$\langle mg_1 \rangle$	7912	16557	303.20	52.00
7	6	Shared Memory	$\langle mg_1, rw, sc, bit, sm \rangle$	101	290	182.49	5.69
8	6	Shared Memory	$\langle mg_1, sc, bit, rw, sm \rangle$	88	232	257.81	6.30
9	6	Shared Memory	$\langle mg_1, sc, rw, bit, sm \rangle$	94	258	212.48	5.52
10	6	Shared Memory	$\langle mg_1, rw, bit, sc, sm \rangle$	114	366	183.78	5.78
11	6	Shared Memory	$\langle mg_1, bit, rw, sc, sm \rangle$	108	369	201.35	6.17
12	6	Shared Memory	$\langle mg_1, bit, sc, rw, sm \rangle$	100	281	202.86	6.18
13	6	Shared Memory	$\langle mg_1, sm, rw, sc, bit \rangle$	349	3165	366.04	15.91
14	6	Shared Memory	$\langle mg_1, rw, sm, sc, bit \rangle^*$	71	180	192.52	6.21
15	6	Shared Memory	$\langle mg_1, rw, sc, sm, bit \rangle$	92	259	197.99	6.19
16	6	Shared Memory	$\langle mg_1, sm, sc, rw, bit \rangle$	321	3319	423.41	15.83
17	6	Shared Memory	$\langle mg_1, sc, rw, sm, bit \rangle$	82	198	237.55	6.18

mg_1: sequential merge, mg_2: near-relatives merge (Eq. 1)
rw: read-write, sc: shield-clock, bit: bypass internal transition, sm: symmetry, *: best reduction sequence

[8] H. A. Hansson, H. W. Lawson, M. Stromberg, and S. Larsson. BASEMENT: A distributed real-time architecture for vehicle applications. *Real-Time Systems*, 11(3):223–244, 1996.

[9] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the IEEE International Conference on Logics in Computer Science (LICS'92)*, 1992.

[10] 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)*, pages 138–147. IEEE Computer Society Press, September 1998.

[11] P.-A. Hsiung. Timing coverification of concurrent embedded real-time systems. In *Proceedings of the 7th IEEE/ACM International Workshop on Hardware Software Codesign (CODES'99, Rome, Italy)*, pages 110 - 114. ACM Press, May 1999.

[12] P.-A. Hsiung. Embedded software verification in hardware-software codesign. *Journal of Systems Architecture — the Euromicro Journal*, 46(15):1435–1450, December 2000.

[13] P.-A. Hsiung. Hardware-software timing coverification of concurrent embedded real-time systems. *IEE Proceedings — Computers and Digital Techniques*, 147(2):81–90, March 2000.

[14] P.-A. Hsiung. Formal synthesis and code generation of embedded real-time software. In *International Symposium on Hardware/Software Codesign (CODES'01, Copenhagen, Denmark)*, pages 208–213. ACM Press, New York, USA, April 2001.

[15] P.-A. Hsiung. Formal synthesis and control of soft embedded real-time systems. In *Proceedings of IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01)*, pages 35–50. Kluwer Academic Publishers, August 2001.

[16] P.-A. Hsiung, F.-S. Su, C.-H. Gao, S.-Y. Cheng, and Y.-M. Chang. Verifiable embedded real-time application framework. In *Proceedings of IEEE International Real-Time Technology and Applications Symposium (RTAS'01, WiP Session, Taipei, Taiwan)*, pages 109–110. IEEE Computer Society Press, May 2001.

[17] P.-A. Hsiung and F. Wang. A state-graph manipulator tool for real-time system specification and verification. In *Proceedings of the 5th Intl Conf on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 181–188, October 1998.

[18] P.-A. Hsiung and F. Wang. User-friendly verification. In *Proceedings of the 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.

[19] K. H. Kim. APIs for real-time distributed object programming. *IEEE Computer*, 33(6):72–80, June 2000.

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

[21] B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of IEEE/ACM Design Automation Conference (DAC'98)*, pages 502 – 505. ACM Press, June 1998.

[22] P. Merlin and G. Bochman. On the construction of submodule specifications and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5(1):1 – 25, January 1983.

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

[24] W.-B. See and C. S.-J. Object-oriented real-time system framework. *Domain-Specific Application Frameworks*, pages 327–338, 2000.

[25] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings IEEE/ACM Design Automation Conference (DAC'99)*. ACM Press, June 1999.

[26] 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*, 23(12), December 1997.

[27] F. Wang and P.-A. Hsiung. Efficient and user-friendly verification. *To appear in IEEE Transactions on Computers*.

[28] F. Wang and P.-A. Hsiung. Automatic verification on the large. In *Proceedings of the 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, pages 134–141, November 1998.