Formal verification of real-time embedded software in an object-oriented application framework

P.-A. Hsiung, T.-Y. Lee, J.-M. Fu and W.-B. See

Abstract: With the rapid escalation in design complexity of real-time embedded software, application frameworks have become an almost indispensable tool because they greatly ease the work of a designer by performing tedious tasks on behalf of a designer and by reusing semi-complete application codes. To ensure code quality and reliability, computer-aided analysis is also performed for the generated application software in some frameworks. However, when the target is real-time embedded systems, the correctness of the software in terms of satisfying all user-given real-time and embedded constraints becomes a primary objective for such frameworks. To guarantee correctness, formal verification in the form of model checking is a viable solution due to its full automation capability. Nevertheless, little is known from either the existing literature or industrial experience on how formal verification can be integrated into an object-oriented application framework, whose primary purpose was previously only to design and generate application software. This work contributes to the state-of-art technology by showing how a design framework and a verification framework can be integrated. Three main issues are tackled: (i) what to verify?; (ii) when to verify?; and (iii) how to verify? As a solution to these three issues the authors propose a mapping from the object-oriented model to a formal model, a schedule-verify-map strategy and a compositional verification methodology, respectively. These have been implemented in a component-based framework and experiments performed to illustrate their feasibility. Due to the incorporation of industry *de-facto* standards such as real-time unified modelling language and real-time Java, in the proposed techniques it should now be possible for an engineer to gain access to theoretically proven formal verification technologies that would otherwise be considered to be inaccessible to an engineer unskilled in verification techniques.

1 Introduction

According to industry statistics, software accounts for as much as 70% of the total functionalities in real-time embedded systems including home appliances, information appliances, personal assistants, wearable computers, telecommunication gadgets and transportation facilities. The main reason for the widespread use of embedded software is its greater design flexibility compared to hardware, but along with this advantage a designer is also burdened with greater complexity in ensuring its correctness due to the large number of valuations possible for a given software 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 formally verified before deployment. Software glitches cost the US economy $59.5 billion each year, according to a recent study done by the US Department of Commerce's National Institute of Standards and Technology. All these facts go to show that verifying the correctness of a software is a demanding and important issue in the design phase of a real-time embedded system.

To overcome the significant complexity issues in the design of real-time embedded software, a designer often resorts to using an object-oriented application framework, which is a semi-complete application that allows code reuse and automatic generation of the final software code. Currently available application frameworks such as SESAG [1, 2] and OORTSF [3–5] for designing real-time embedded software, do not verify the correctness of the generated code. The consequences of deploying faulty software range from simple system failures such as a malfunctioning microwave oven to fatal disasters such as radiation leaks in a nuclear reactor. Thus, an application framework without verification is basically incomplete. As a solution, we show how formal verification can be integrated into an application framework called VERTAF [6, 7], which generates and verifies real-time embedded software code in Java and C.

To guarantee the correctness of the automatically generated real-time embedded software, instead of case-by-case verification, the work presented in this article takes a pioneering step in introducing formal verification into a component-based object-oriented application framework. Our goal will be to make this integration both seamless and scalable. By seamless integration, we mean a software designer using an application framework with verification

IEE Proceedings online no. 20041102

doi: 10.1049/ip-cdt:20041102

Paper first received 8th January and in revised form 30th July 2004

P.-A. Hsiung is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan

T.-Y. Lee is with the Department of Electronic Engineering, National Taipei University of Technology, Taipei, Taiwan

J.-M. Fu is with the Department of Electronic Engineering, Cheng Shiu University, Kaohsiung, Taiwan

W.-B. See is with Aerospace Industrial Development Corporation, Taichung, Taiwan

capabilities need neither be well versed in the theory of formal verification nor be forced to learn how to verify his/her target software. By scalable integration, we mean an application framework can adapt the embedded verification technology to a user's system specification.

There are several issues to be addressed for the seamless, scalable integration of formal verification and application framework technologies as described in the following.

- What to verify? On one hand, an application framework perceives a system as a collection of interacting components or objects with possibly complex behaviours, whereas, formal verification regards a system as a set of concurrent real-time tasks with formal syntax and precise semantics. There is a difference in the granularity and level of abstraction between the two system models. If application framework and formal verification are to be integrated, how do we generate a precise formal model from an imprecise object-oriented model?
- When to verify? The application framework is responsible for synthesising a real-time embedded software, whereas formal verification ensures the correctness of the same software. The exact step(s) in the design process where verification is to be performed is critical and affects both the design time and design results. Thus, the question here is: at what design step must verification be performed?
- How to verify? The application framework generates software by going through a complete design methodology, whereas formal verification analyses software by going through a verification methodology. The two methodologies might not be compatible in terms of data exchange formats and mutual requirements. Hence, how can the two methodologies work together with mutual benefits for a common goal of generating correct executable software?

Our proposed solutions to the above issues are illustrated in Fig. 1, described briefly as follows, and will be elaborated on in the rest of this article.

- Formal object-oriented model and timed automata. In VERTAF, an application designer uses the unified modelling language (UML) profile for schedulability, performance, and time specification and the real-time specification for Java to specify the formal object-oriented model (see Appendix A, Section 9.1) for a system. UML and Java are industry *de-facto* standards, but being general-purpose languages with real-time features added to them, they still lack a semantically precise behaviour model that is suitable for formal verification. For example, the exact time extensions of the UML statecharts are not clearly defined. Furthermore, UML is too general purpose, thus we need to constrain the user's model expressiveness such that whatever restricted models are specified by the user will eventually have a formal unambiguous semantics represented in a formal model such as timed automata [8], which are accepted by model checkers for timed systems. The formal object-oriented model is mapped into timed automata, which are then used as system models for our formal verification, thus solving the 'what to verify?' question. The mapping to timed automata will be described in Section 3.
- Verification after scheduling but before code generation. Scheduling is an essential step in the synthesis of real-time embedded software and verification could be performed either before or after scheduling. In response to the 'when to verify?' question, it is proposed in this work that verification be performed after scheduling but before code generation. Reasons for this proposal and justifications for it will be discussed in Section 4.
- Formal synthesis and model checking. Formal methods have been applied to the synthesis [9–15] as well as the verification [7, 16–20] of real-time embedded systems. Application frameworks are a good platform for the integration of these two classes of techniques. Using a common system model such as timed automata, model checking procedures can be called from synthesis algorithms. The basic framework will be compositional verification with modular packaging of verification techniques. Further details will be given in Section 5.

The above proposed solutions to the technology integration issues are currently being implemented in a component-based object-oriented application framework called VERTAF [6, 7]. VERTAF generates code for real-time embedded systems using formal modelling and synthesis techniques. A separate software component called Verifier is being developed in VERTAF for encapsulating the proposed solutions. Verifier is briefly described in Appendix D, Section 9.4. In the course of introducing the solutions, some running examples will be given to illustrate the feasibility and success of the `Verifier` component.

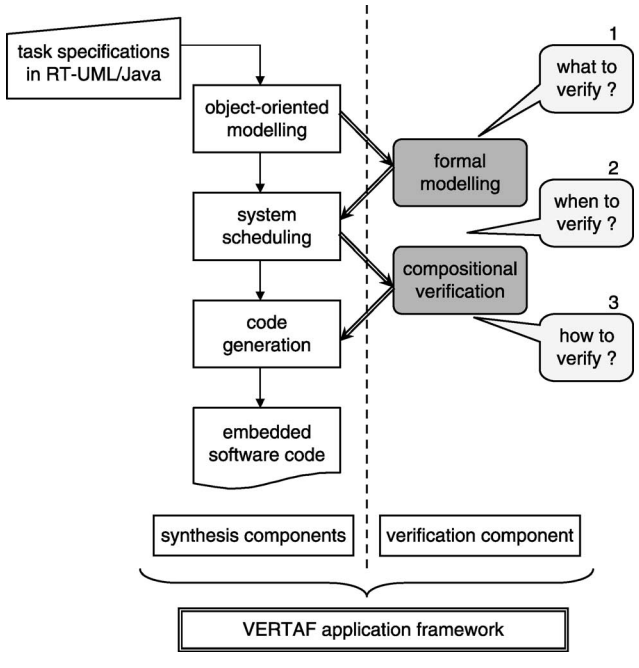

Fig. 1 *Synthesis and verification components in an application framework*

2 Previous work

Since our focus is on technology integration, we will concentrate on the previous work related to object-oriented application frameworks, formal synthesis and formal verification.

Currently, there are very few component-based object-oriented frameworks developed specifically for generating code for real-time embedded systems. In the following, we first summarise three such frameworks. Two recently proposed application frameworks are the object-oriented real-time system framework (OORTSF) [3–5] and SESAG [1, 2], which have been applied to the development of avionics software. Some design patterns related to real-time application design were proposed and code could be generated automatically. Scheduling and real-time synchronisation issues such as asynchronous event handling

and protocol hooking were not handled. Some other issues related to application frameworks such as the flexibility of specifying real-time objects, the ease of using the frameworks, and the benefits of applying them were also not described. Another more recent framework, called VERTAF [6, 7], is an enhanced version of SESAG, incorporating software component technology, formal verification technology, industry standards such as UML and Java, and multi-level reuse of code, design patterns, and framework architecture. As an example of middleware integration frameworks for real-time applications, TAO real-time object request broker was designed by Schmidt [21].

Cadena [22] is a design and verification framework for CORBA component-model-based avionic applications that use Boeing's Bold Stroke middleware. Cadena does not consider timing and scheduling in its design process. Some other worldwide research projects targeting embedded real-time software design include USA DARPA's MoBIES [23, 24], Germany LMU München's HUGO [25], Europe EUREKA-ITEA's DESS [26], and Sweden Uppsala University's TIMES [27]. These projects mainly use their own self-defined modelling techniques and do not consider scheduling in the design process.

Automatic generation of real-time embedded software code requires formal software synthesis, which has been mainly performed for communication protocols [28], plant controllers [10, 11, 13], and real-time schedulers [9, 14] because they generally exhibit regular behaviours. Recently, there has also been some work on automatically generating code for embedded systems [29–33] as described in the following. Lin [30, 31] 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. Later, Zhu and Lin [33] proposed a compositional version of the synthesis method that reduced the generated code size and was thus more efficient. Sgroi *et al*. [32] proposed a software synthesis method called quasi-static scheduling for a more general Petri net model, namely free-choice petri nets. A necessary and sufficient condition was given for a free-choice Petri net to be schedulable. Schedulability was first tested for a net and then a valid schedule generated by decomposing a net into a set of conflict-free components which were then individually and statically scheduled. Code was finally generated from the valid schedule. Later, Hsiung [12] integrated quasi-static scheduling with real-time scheduling to synthesise real-time embedded software. A synthesis method for soft real-time systems was also proposed by Hsiung [34]. The free-choice restriction was first removed by Su and Hsiung [35] in their work on extended quasi-static scheduling. Recently, Gau and Hsiung [36, 37] proposed a more integrated approach called time-memory scheduling based on reachability trees. Balarin *et al*. [29] proposed a software synthesis procedure for reactive embedded systems modelled using codesign finite state machine [38] and synthesised using the POLIS hardware-software codesign tool [38]. This work cannot be easily extended to other more general frameworks.

Besides software synthesis for Petri nets, synthesis has also been performed for other formal models such as timed automata. Given a dense real-time system modelled by a set of timed automata and a (temporal) property given as a formula in timed computation tree logic (TCTL) [16, 18], a controller is synthesised such that it restricts the behaviour of the system for satisfying the property. This is the controller synthesis problem for dense real-time systems. Recently, system parameters have also been taken into consideration for real-time controller synthesis [39]. Controller synthesis for plants (also called supervisor synthesis) was mainly performed in the discrete time domain, with a large portion of classical work done by Ramadge and Wonham [40, 41]. Around 1994, when timed automata was proposed as a dense-time model for real-time systems [8], controller synthesis was extended to dense real-time systems [10, 13, 14], to hybrid systems [15], and to multimedia scheduler synthesis [9].

Formal verification of general software that is found on personal computers or workstations is a formidable task, hence their correctness is generally validated through analysis techniques such as apportioning [42]. In contrast to general software, real-time embedded software can be more easily verified. For example, model checking [43–45] with assume-guarantee reasoning [46] is a viable method. Assume-guarantee reasoning partitions a complex software into modules that are individually verified to satisfy some guarantees under some assumptions. Then, the assumptions of each module are discharged (validated) through an analysis of the guarantees of all other modules. Finally, the system is verified to satisfy a property (implied by the system guarantee) under given system assumptions by a logical composition of all module guarantees. Details of this procedure can be found in [46]. Model checking real-time embedded software can be performed by first modelling the software as a set of timed automata, then they are scheduled according to user-given constraints, and finally verified through a labelling algorithm of model checking. Some details of this procedure can be found in [44]. Model checking tools for real-time systems modelled as timed automata include UPPAAL [47], KRONOS [48], SGM [49], and RED [50]. Other work include hardware-software timing coverification [20, 51] based on linear hybrid automata, and the coverification strategy for automatic mapping to linear hybrid automata [17].

3 Formal object-oriented model and timed automata

As introduced in Section 1, we need to solve the issue of 'what to verify?' during the technology integration between design and verification. It is shown in this Section how an object-oriented model used by design engineers can be related to the formal timed automata model used by verification engineers of real-time embedded systems. Since we emphasise formal verification, we will briefly touch upon the object-oriented model (leaving the details to the Appendix) and then go into details on the timed automata model.

A formal object-oriented model was proposed in VERTAF [6, 7], which consists of autonomous timed objects (ATO) and autonomous timed processes (ATP). An application designer specifies real-time embedded tasks by describing ATOs using the real-time profile of UML and the reference implementation of the real-time specification for Java. VERTAF automatically generates the corresponding semantic models called ATPs, which are used for scheduling and code generation. Further details on ATO and ATP can be found in Appendix A, Section 9.1 of this article.

A timed automaton 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 timed automaton may remain in a particular mode as long as the values of all

its variables satisfy a timed predicate, which is a conjunction of clock constraints, Boolean propositions, and synchronisation labels.

The following definitions formally define the timed automata model, where the sets of integers and non-negative real numbers are denoted by $\mathcal{N}$ and $\mathcal{R}_{\geq 0}$, respectively.

Definition 1: Timed predicate. Given a set C of clock variables, a set D of discrete variables, and a set Y of synchronisation labels, the syntax of a timed predicate η over C, D and Y is defined as:

$$\eta := \text{false} | x \sim c | x - y \sim c | d \sim c | s | \eta_1 \wedge \eta_2 | \neg \eta_1$$

where $x, y \in C$, $d \in D$, $s \in Y$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$ and η_1, η_2 are timed predicates.

Let $B(C, D, Y)$ be the set of all timed predicates over C, D, and Y. A timed automaton may go from one mode to another mode, that is, it performs a transition, when the triggering condition which is specified as a timed predicate, is satisfied by the current valuation of clock and discrete variables and a corresponding transition with the same synchronisation label is also triggered. On a transition, some clocks may be reset to zero and some discrete variables may be assigned new integer values.

Definition 2: Timed automaton. A timed automaton is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i)$ 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;
- Y_i is a set of synchronisation labels;
- $\chi_i : M_i \mapsto B(C_i, D_i, Y_i)$ 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;
- $\tau_i : E_i \mapsto B(C_i, D_i, Y_i)$ defines the transition triggering conditions;
- $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ 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.

Definition 3: System state. Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i | \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i),\ 1 \leq i \leq n\}$, a state s of system $\mathcal{S}$ is defined as a mapping from $\{1, \ldots, n\} \cup \bigcup_i C_i \cup \bigcup_i D_i$ to $\bigcup_{1 \leq i \leq n} M_i \cup \mathcal{N} \cup R_{\geq 0}$ such that:

- $\forall_i \in \{1, \ldots n\}$, $s(i) \in M_i$ is the mode of $\mathcal{A}_i$ in s;
- $\forall i, \forall x \in C_i, s(x) \in \mathcal{R}_{\geq 0}$ is the reading of clock x in s, such that $s(x) \models \wedge_i \chi_i(s(i))$, where $\models$ is a notation for satisfaction of predicates by a state;
- $\forall i, \forall d \in D_i, s(d) \in \mathcal{N}$ is the value of d in s, such that $s(d) \models \wedge_i \chi_i(s(i))$.

Definition 4: System transition. Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i | \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i),\ 1 \leq i \leq n\}$, and two system states s and s', there is a system transition from s to s' in $\mathcal{S}$, in symbols $s \rightarrow s'$, iff there is an i, $1 \leq i \leq n$ such that:

- $(s(i), s'(i)) \in E_i$;
- $s(i) \models \tau_i(s(i), s'(i))$;
- if $\tau_i(s(i), s'(i))$ does not have any synchronisation label, then for all $1 \leq j \leq n$ and $j \neq i$, $s(j) = s'(j)$;
- if $\tau_i(s(i), s'(i))$ has a synchronisation label $s \in Y_i$ such that $s \in Y_k$ for some $1 \leq k \leq n$, $k \neq i$ then for all $1 \leq j \leq n$, $j \neq i$, and $j \neq k$, $s(j) = s'(j)$ and $(s(k), s'(k)) \in E_k$;
- $\forall x \in X((x \in \rho_i(s(i),\ s'(i)) \Rightarrow s'(x) = 0) \wedge (x \notin \rho_i(s(i), s'(i)) \Rightarrow s'(x) = s(x)))$.

3.1 Generation of timed automata

We now show the relation between the object-oriented ATO-ATP models and the timed automata model. In general, the following four types of timed automata can be generated from a general ATP based on the input/output relationships among ATPs in 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. The translation process is described in Algorithm 1, where the `type()` function distinguishes the different types of ATP and corresponding timed automaton generated as shown in Figs. 2–5.

Algorithm 1: Generation of timed automata from ATP
`Gen_TA` (*ATP_Set*, *Call_Graph*)
ATP_Set = $\{P_{11}, P_{12}, \ldots, P_{1k_1}, P_{21}, \ldots, P_{2k_2}, \ldots P_{n1}, \ldots P_{nk_n}\}$;
Call_Graph;
{
Step 1: For each $ATP \in ATP\_Set$ {
Step 2: Switch (`type`(*ATP*, *Call_Graph*)) {
Step 3: Case Nt/P: ATA = `create_TA` (*NtP_TA*); break; //Fig. 2
Step 4: Case Nt/A: ATA = `create_TA` (*NtA_TA*); break; //Fig. 3
Step 5: Case T/P: ATA = `create_TA` (*TP*); break; //Fig. 4
Step 6: Case T/A: ATA = `create_TA` (*TA*); break; //Fig. 5
Step 7: Default: ATA = `create_TA` (*event_TA*);
}
Step 8: $ATA\_Set = ATA\_Set \cup \{ATA\}$;
}
Step 9: Return ATA_Set;
}

- Non-triggerable/passive (Nt/P). A non-triggerable/passive timed automaton is generated from a solitary ATP that has no input and no output event. System upkeep tasks that require no input and produce no output constitute an example of this type of ATP. Figure 2 illustrates such a timed automaton.
- Non-triggerable/active (Nt/A). A non-triggerable/active timed automaton is generated from an ATP that has no input

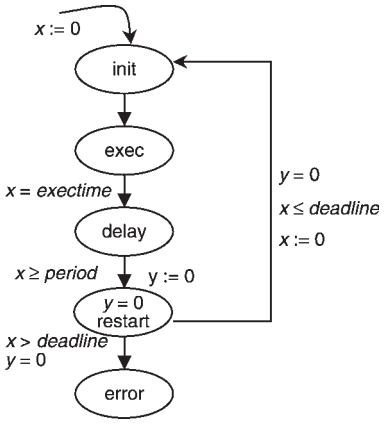

Fig. 2 *Non-triggerable/passive timed automaton*

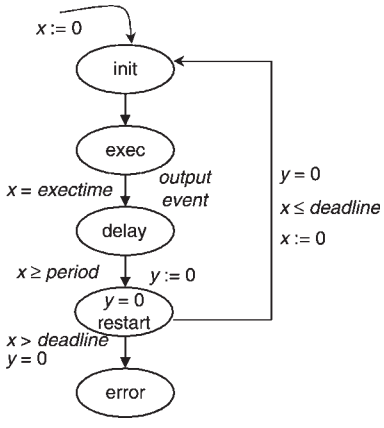

Fig. 3 *Non-triggerable/active timed automaton*

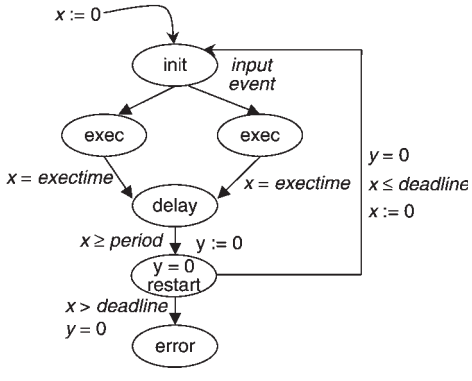

Fig. 4 *Triggerable/passive timed automaton*

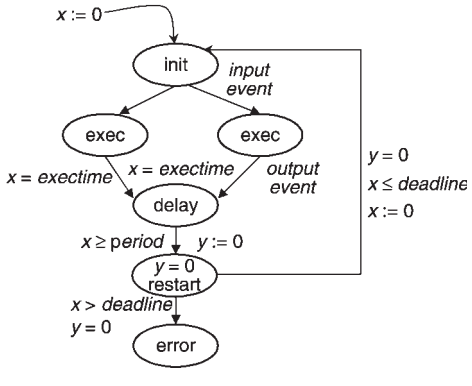

Fig. 5 *Triggerable/active timed automaton*

event, but produces some output event. A periodic stimuli generator task is an example of such an ATP. Figure 3 illustrates such a timed automaton.

- Triggerable/passive (T/P). A triggerable/passive timed automaton is generated from an ATP that has some input event but produces no output event. Final control and actuator tasks are examples of such ATPs. Figure 4 illustrates such a timed automaton.
- Triggerable/active (T/A). A triggerable/active timed automaton is generated from an ATP that has both input and output events. Most real-time tasks are of this type such as signal-processing, transmission protocols, etc. Figure 5 illustrates such a timed automaton.

In Figs. 2–5, x and y are clock variables, where x records the elapsed time since the start of an ATP execution and y forces an immediate state change upon entering the *restart* mode. Other variables such as *exectime*, *period*, *deadline* are the characteristics of an ATP execution. The *init* mode is used for initialisation of an ATP. It is noted here that the explicit modelling of status broadcast for an ATP is not necessary because it is assumed in the timed automata model that each timed automaton knows the status of all other timed automata. The *exec* mode represents the execution of an ATP. The *delay* mode enforces a delay before the task is restarted so that period constraints are obeyed. The *restart* mode simply checks if the elapsed time represented by the value of clock x has exceeded the deadline or not. If not, then the task is restarted, otherwise, an *error* mode is entered, which means the ATP has violated its deadline constraints. Lastly, *input event* and *output event* are events which trigger the execution of an ATP or events that are produced by an ATP for triggering an other ATP, respectively.

In summary, a designer specifies his/her system by describing ATOs, for which their semantic models ATPs are generated automatically. Finally, each ATP is converted into one of the above-described timed automata. ATP can be used for simulation, scheduling and code generation. Timed automata can be used for verification, synthesis and code generation. In Section 3.2, we will illustrate this design process using an example.

3.2 Autonomous intelligent cruise controller example

An autonomous intelligent cruise controller (AICC) system application has been developed and installed in a Saab automobile by Hansson *et al.* [52]. 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 a 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 and software is as shown in Fig. 6. AICC has a system bus called a controller area network bus, through which all the sensors, actuators and control systems communicate. The sensors are connected by serial RS-232 interfaces to the bus. The software development methodology used in [52] is based on sets of interconnected so-called software circuits. Each software circuit has a set of input connectors where data are received and a set of output connectors where data are produced. We model the software circuits in [52] as autonomous timed objects in the formal object-oriented model.

As shown in Fig. 7, there are five ATOs (the dotted blocks) 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 [52], within the Swedish road transport informatics programme. As observed in Fig. 7, each ATO (dotted block) may map to one or more ATP (solid-line blocks). The arrows represent function calls between ATPs. An ATP without an incoming arrow (function call) represents a time-triggered method, which executes periodically. Each ATO has a period T associated with it. The call-graph and process table for the AICC are shown in Fig. 7 and Table 1, respectively. There is a total of 12 functions performed in five objects, out of which 11 functions are to be implemented in software. Thus, there are 11 ATPs in this system. Here, SRC and display are identified as resources. This application took 5 days for three real-time system designers using VERTAF. The same

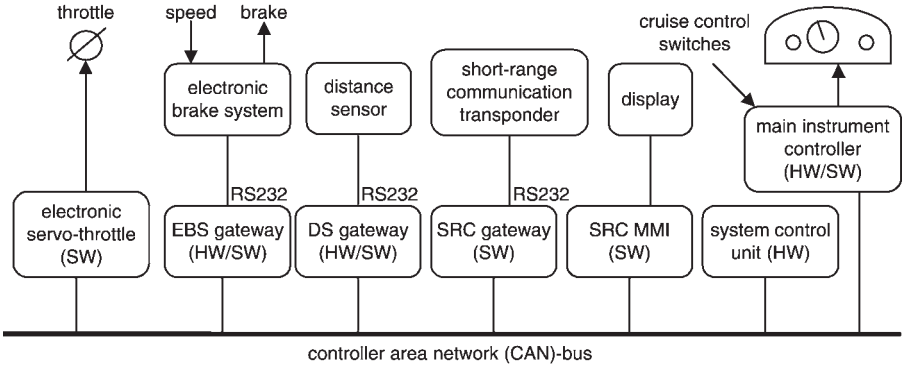

Fig. 6 *AICC Example: system architecture in which SW stands for software and HW for hardware*

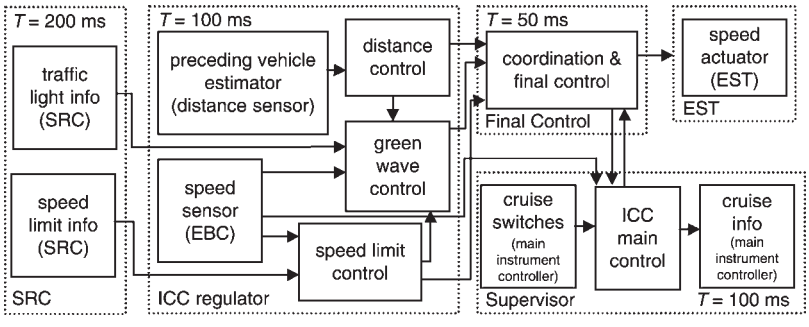

Fig. 7 *AICC example: call-graph with ATOs and ATPs*

Table 1: AICC example: process table

	ATP	ATO	Period*	Exec. time*	Deadline*
1	Traffic light info	SRC	200	10	400
2	Speed limit info	SRC	200	10	400
3	Preceding vehicle estimator	ICCReg	100	8	100
4	Speed sensor	ICCReg	100	5	100
5	Distance control	ICCReg	100	15	100
6	Green wave control	ICCReg	100	15	100
7	Speed limit control	ICCReg	100	15	100
8	Coordination & final control	Final_Control†	50	20	50
9	Cruise switches	Supervisor	100	15	100
10	ICC main control	Supervisor	100	20	100
11	Cruise info	Supervisor	100	20	100
12	Speed actuator	EST	50	5	50

SRC: short range communication, ICCReg: ICC regulator, EST: electronic servo-throttle
*All times are in milliseconds
†Implemented in hardware

Table 2: AICC example: timed automata models

	ATP	Triggerability	Type
1	Traffic light info	no	active
2	Speed limit Info	no	active
3	Preceding vehicle estimator	no	active
4	Speed sensor	no	active
5	Distance control	yes	active
6	Green wave control	yes	active
7	Speed limit control	yes	active
8	Coordination & final control	hardware	
9	Cruise switches	no	active
10	ICC main control	yes	active
11	Cruise info	yes	passive
12	Speed actuator	yes	passive

application took the same designers 20 days to complete development. This significant decrease in design time was because VERTAF automatically extracted the tasks and constraints from the object specifications.

The 11 ATPs were scheduled by the Scheduler component of VERTAF and then transformed into 11 timed automata by the Verifier component of VERTAF based on the system architecture description (Fig. 6), call-graph (Fig. 7), and process table (Table 1). Based on the characteristics of ATP models in the AICC example, timed automata models are generated as shown in Table 2. It is observed that the ATPs that are responsible for data collection or are reactive to environment changes are modelled as non-triggerable/active timed automata (ATP index: 1-4 and 9). Furthermore, the ATPs at the terminating end of an execution path are modelled as triggerable/passive timed automata (ATP index: 11 and 12). Lastly, all other ATPs are modelled as triggerable/active timed automata (ATP index: 5, 6, 7 and 10).

The above description correlates the ATO-ATP model with the timed automata model through a concrete real-world example. We will return to this example in Section 5.5, where the synthesis and verification results for this example will be described.

4 Verification after scheduling but before code generation

Real-time embedded software can be synthesised by going through three design phases, namely. (i) specification; (ii) scheduling; and (iii) code generation. Specification is the modelling of software as a set of communicating processes such as a network of timed automata. Scheduling is restricting the behaviour of a modelled software such that it satisfies all user-given functional, temporal and spatial constraints. Quasi-static scheduling [30–33] is an example of such scheduling. Code generation is producing real-time embedded software code in some programming language such as C. The main issue here is: when should software be verified?

As illustrated in Fig. 8, verification can be performed at three different stages: (i) after specification but before scheduling; or (ii) after scheduling but before code generation; or (iii) after code generation. On the one hand, at the point after specification but before scheduling, processes generally have some regions in their state-space which will be eventually eliminated by scheduling. On the other hand, at the point after code generation, the software code is generally implementation-dependent and contains coding technicalities that do not really contribute toward the actual behaviour of the software. Hence, we propose to verify software after scheduling but before code generation. In the rest of this Section we will make a comparison between conventional verification approaches and our proposed approach. Finally, we will illustrate this comparison using a distributed polling system example.

4.1 Conventional verification approaches

Theoretically, verifying the given processes can be done after either one of the stages during software synthesis Verification scientists try to verify processes immediately after process specification to find any specification errors. This is called the verify-schedule-map (VSM) approach (column 1 in Fig. 8). Design engineers try to verify the final program after code generation. This is called the schedule-map-verify (SMV) approach (column 3 in Fig. 8). Both of these approaches encounter different degrees of state-space explosion problems.

Verifying process specification explores unnecessary regions in the state-space that would eventually not even exist in the final software code. These regions are basically those that will be eliminated after scheduling. The problem becomes worse when the degree of non-determinism is high in the specification. The degree of non-determinism (δ_{ND}) is the maximum number of different possible behaviours that a system can have in any one state. Further details on how δ_{ND} affects verification are discussed in Appendix B, Section 9.2.

Verification of software program code also indulges in unnecessary state-space explosions and thus affects scalability in the number or size of processes verifiable. Software programs usually contain many auxiliary, implementation-dependent variables, that contribute towards neither the real behaviour of the software nor the satisfaction of specified real-time constraints by the software. As is well known, the state-space size explored during verification increases exponentially with the number of clock variables and largest integer constant used [16]. The state-space size also increases drastically with the number of free variables. Software programs generally contain a lot of variables, the number of which is not optimised either by the software synthesis procedure or by the software compiler.

In conclusion, both of the above approaches unnecessarily explore regions in the state-space that do not contribute towards the actual goal of verification. Thus, in

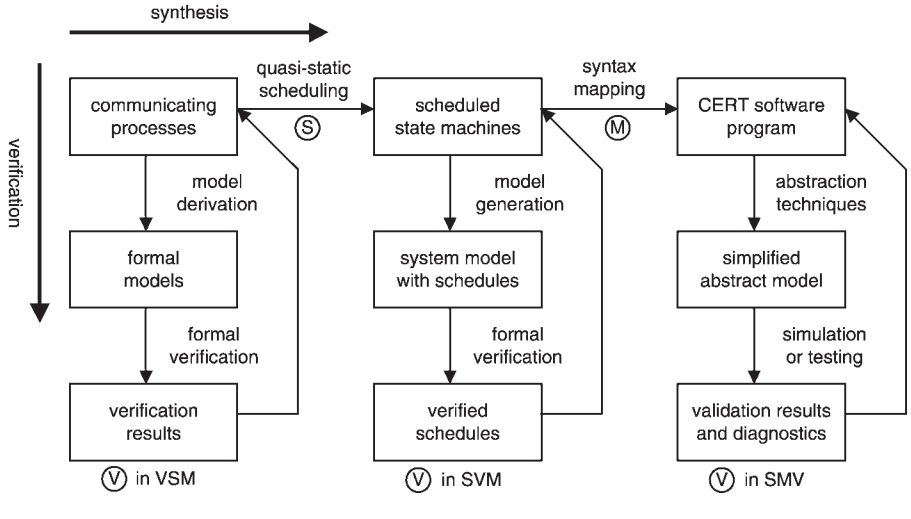

Fig. 8 *Options for verification stage in real-time embedded software synthesis*

Table 3: A comparison of the three verification approaches

Verification approach	Correctness	Feasibility	State-space size	Completeness
VSM	too sure	vague	exponentially large	more than complete
SVM	Sure	largely	reduced	complete
SMV	not sure	practical	small to medium	incomplete

the following Section an approach is proposed called schedule-verify-map (SVM) as illustrated by column 2 of Fig. 8.

4.2 Proposed SVM approach

To overcome the difficulties in verification presented in the preceeding Section, we propose an approach called SVM. In SVM, verification is performed after scheduling but before code generation. Since scheduling eliminates certain regions in the state-space, SVM will obviously explore a much smaller part of the state-space. The degree of reduction is analysed in Appendix B, Section 9.2. Since the target of verification is a set of scheduled processes and not program code, SVM will also search a smaller state-space than the engineers' approach (verification after code generation).

Comparing the two conventional approaches, VSM adopted by verification scientists and SMV adopted by design engineers, and our proposed SVM approach, we have the pros and cons of each summarised in Table 3. On comparison, it is observed that SVM is a good trade-off between practical feasibility (column 3) and verification completeness (column 5). Although VSM is more than complete, its practical feasibility is often hindered by the exponentially large state-space. SMV is the most practical among the three approaches, yet it is an incomplete validation process (mostly accomplished through simulation and testing that covers less than 100% of the system behaviour). A more detailed analysis on the SVM approach is presented in the following Section.

4.3 Distributed signal polling system example

This example illustrates not only how SVM explores a smaller state-space compared to VSM, but also how different scheduling techniques affect the sizes of the state-spaces explored for verification.

As illustrated in Fig. 9, this application is a distributed signal polling system that is generally located at each entry/exit gate of a parking lot. In this example, each process initialises a counter to 500 for the number of vacant parking spaces. Then, it starts to poll for any car-entry, car-exit, or check-count signal. When a signal is detected, appropriate actions are carried out. The counter value is decremented for a car-entry signal and incremented for a car-exit signal. the counter value is output for a check-count signal. After completing actions, the polling process is repeated. Here, we need to verify

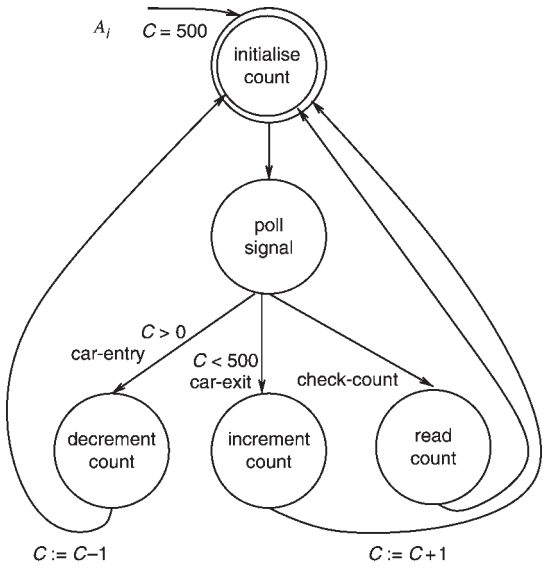

Fig. 9 *Distributed signal polling system*

that a car is never allowed entry when there are no vacant parking space available $(Count = 0)$.

Experiments were carried out for this example with and without scheduling, the details of which are shown in Table 4. Verification of a four-process signal polling system required exploring 78 000 modes with 205 000 transitions when no scheduling was applied (the VSM approach). This is a very large state-space, the construction of which requires a large amount of memory (178 MB) and time (9608 s). For the SVM approach, three scheduling techniques were applied to this example: (i) post-signal scheduling; (ii) pre-signal scheduling; and (iii) both post-signal and pre-signal scheduling. Post-signal scheduling is the scheduling of the processes that have detected signals concurrently. Pre-signal scheduling is the scheduling of processes before any signal detection is started. We observe from Table 4 that the three types of scheduling techniques result in different sizes for the state-space. Applying both post- and pre-signal scheduling results in the smallest state-space. Applying pre-signal scheduling results in a smaller state-space than applying post-signal scheduling. This is consistent with our intuition, pre-signal scheduling applies a much greater restriction on the behaviour of the processes than post-signal scheduling.

Table 4: The use of SVM and VSM into distributed signal polling system

n	Schedule	Approach	Modes	Trans	Memory, MB	Time, s
4	No	VSM	78 347	205 578	178.12	9608.36
4	Post-signal	SVM	1018	1255	6.57	34.20
4	Pre-signal	SVM	29	41	10.69	82.19
4	Post/pre-signal	SVM	22	26	5.48	27.06

5 Formal synthesis and model checking

Having proposed solutions to the issues of 'what to verify?' in Section 3, and 'when to verify?' in Section 4, we will discuss 'how to verify?' real-time embedded software in this Section. Based on the proposed solutions, we demonstrate how design and verification can be integrated into an application framework. The design and verification methodologies used in VERTAF are, respectively, formal synthesis and model checking.

Formal synthesis is defined as a design method by which a formally modelled real-time embedded system is scheduled and code synthesised to satisfy a set of real-time and memory specifications. Several of the research works in this area were discussed in Section 2. In contrast to conventional engineering-type synthesis methods, formal synthesis methods are precise and produce verifiable systems.

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 modelled by a set of timed automata can be model checked to see if it satisfies a given TCTL specification [18]. Recently, model checking has become a popular practical formal method which is gradually being accepted by the industry and used in collaboration with simulation techniques.

5.1 Compositional design and verification

Our target problem is formulated as follows:

Definition 5: Integration of verification into design. Given a real-time embedded system described in an object-oriented application framework using the formal object-oriented model along with a set of temporal and memory constraints, software that is automatically synthesised must be formally verified to satisfy all the given constraints.

Since the focus of this work is on verification, interested readers may refer to [7, 12, 35–37] for further details on how real-time embedded software is synthesised using scheduling.

As a solution to the above posed problem, we propose the following technology integration framework. Given a real-time embedded system described using a set of ATOs $\{Q_1, Q_2, \ldots, Q_n\}$ and a set of constraints, the behaviour of each ATO Q_i is modelled using one or more ATP $\{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ each of which is in turn represented by a timed automaton $\mathcal{A}_{ij} = (M_{ij}, m_{ij}^0, C_{ij}, D_{ij}, Y_{ij}, \chi_{ij}, E_{ij}, \tau_{ij}, \rho_{ij})$ and a TCTL specification ϕ (c.f. definition 6) is generated from the set of constraints.

Definition 6: TCTL formula. A TCTL formula has the following syntax:

$$\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi'\mathcal{U}_{\sim c}\phi'' \mid \neg\phi' \mid \phi' \vee \phi'' \qquad (1)$$

Here, η is a timed predicate in $B(\cup C_{ij}, \cup D_{ij}, \cup Y_{ij}), \phi'$, 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 such as $\exists\Diamond, \forall\Box, \forall\Diamond, \forall\mathcal{U}, \wedge$, and $\rightarrow$ can all be defined as in [18].

As detailed in algorithm 2, 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.

Algorithm 2: Compositional design and verification in application framework

```
Compositionally_Design_Verify (ATP_Set,
   Constraint_Set, Call_Graph, G)
ATP_Set = {P11,P12,...,P1k1,P21,...,P2k2,...Pn1,...Pnkn};
Constraint_Set; //set of constraints
Call_Graph; //call graph
G; //an empty graph
{
Step 1:  φ = Gen_TCTL(Constraint_Set);
           //φ: TCTL specification
Step 2:  ATA_ = Gen_TA(ATP_Set, Call_Graph);
           //ATA_Set: set of automata
Step 3:  STA_Set = Schedule(ATA_Set, Sched_Alg);
           //STA_Set: set of automata
Step 4:  while (|STA_Set| > 1) {
Step 5:    G = MROF(G, STA_Set);
             // merge related objects first
Step 6:    r = FBRS(G);    // find best reduction sequence
Step 7:    Reduce (G, r); //G: a state-graph
         }
Step 8:  If ((Counter_Eg = Model_Check(G, φ)) ==
           NULL) {
Step 9:    Code_Gen(STA_Set);
Step 10:   Return True;
         }
Step 11: Else return Counter_Eg;
}
```

First, a TCTL specification formula ϕ is generated by the `Gen_TCTL()` procedure in step 1 algorithm 2, which is adopted from the dense time extension of [53]. 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 $ATA\_Set = \{\mathcal{A}_{11}, \mathcal{A}_{12}, \ldots, \mathcal{A}_{1k_1}, \mathcal{A}_{21}, \ldots, \mathcal{A}_{2k_2}, \ldots \mathcal{A}_{n1}, \ldots, \mathcal{A}_{nk_n}\}$ such that $\mathcal{A}_{ij}$ models P_{ij}. The details of this step were given in Section 3.1. In step 3, ATA_Set is then scheduled using some scheduling algorithm $Sched\_Alg$ by the procedure `Schedule()` into another set of timed automata, $STA\_Set = \{\mathcal{A}_{11}^s, \mathcal{A}_{12}^s, \ldots, \mathcal{A}_{1k_1}^s, \mathcal{A}_{21}^s, \ldots, \mathcal{A}_{2k_2}^s, \ldots, \mathcal{A}_{n1}^s, \ldots, \mathcal{A}_{nk_n}^s\}$. Within our framework, $Sched\_Alg$ is taken as a timed version of extended quasi-static scheduling [12, 35]. Verification is performed only after scheduling. It is here that we use the strategy of verification after scheduling but before code generation, which was described in Section 4. 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 either merges the two most-related timed automata from STA_Set into a state-graph (defined later in Section 5.2) or merges one timed automaton with a given state-graph. `MROF()` will be given in detail in algorithm 3 in Section 5.2. Two timed automata in a set are said to be the most-related to each other if no other pair of timed automata shares a greater number of discrete variables, clock variables, and synchronisation labels. This simple definition can be further enhanced by taking other proximity factors into consideration such as the number of common neighbours, where a timed automaton is a neighbour of another timed automaton if they communicate through some variables or labels. Upon merging, the cardinality of STA_Set decrements by one. Next, `FBRS()` in step 6 searches for the best sequence $\boldsymbol{r}$ of reduction manipulators

which reduces the current state-space the most (described later in Section 5.3). In step 7, the actual reduction of the 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 timed automata is constructed iteratively such that in each iteration two timed automata are selected for merging into one timed automaton, which represents the state-space of their concurrent behaviour. 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 described as follows.

- Merge sequence: Section 5.2 answers the question of which two timed automata to merge in each iteration.
- Reduction sequence: Section 5.3 answers the question of what sequence of reduction techniques to apply to an intermediate state-space in each iteration.

5.2 Merge related objects first

As a solution for the first decision issue on the selection of a merge sequence, details of the `MROF()` procedure from step 5 of algorithm 2 are given as follows. The selection of a pair of timed automata for merging in an iteration affects how large the intermediate state-space can grow. We use a state-graph to symbolically represent a state space, as defined in definition 7 and implemented in the state-graph manipulators (SGM) tool [54–56], which is a comprehensive, high-level, real-time system verification tool.

Definition 7: State-graph. Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i \mid \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i),\ 1 \leq i \leq n\}$, a state-graph is a tuple $G = (R, r_0, \beta, F, \alpha)$, where:

- R is a set of symbolic regions, where a region is a set of system states (c.f. definition 3);
- r_0 is an initial region in R;
- $\beta: R \rightarrow \{\langle OBDD_r, DBM_r\rangle \mid r \in R\}$ characterises each region r with the following such that all system states in r satisfy them:
  (i) $OBDD_r$ is an ordered binary decision diagram (OBDD) [57] satisfied by all the discrete variable valuations;
  (ii) DBM_r is a difference-bound matrix (DBM) [58, 59] satisfied by all the clock variable valuations;
- F is a set of system transitions (c.f. definition 4);
- $\alpha: F \rightarrow \{\langle r_s, r_d, I, E, \gamma\rangle\}$ characterises each system transition $f \in F$ with the following:
  (i) $r_s \in R$ is the source region from which f originates;
  (ii) $r_d \in R$ is the destination region for f;
  (iii) $I \subseteq \{1, \ldots, n\}$ is a set of indices of processes which make a transition as defined in definition 4;
  (iv) $E \subseteq \bigcup_{1 \leq i \leq n} E_i$ is a set of process transitions which constitute f as defined in definition 4;
  (v) γ is a permutation of process indices in the set $\{1, \ldots, n\}$, which is used for timed symmetry reduction as described in Section 5.3.

The selection of timed automata for merging is classified into syntax-based and semantics-based as described in the following.

- Syntax-based. This is a sequential merging of timed automata, based on their indices. Two timed automata that have the lowest indices are first merged into a state-graph and then the timed automaton with the minimum index is merged into the newly constructed state-graph from a previous iteration. The criteria for merging can also be size-based, that is, merging is performed first for the timed automata that have the smallest sizes (in ascending order) or for the timed automata that have the largest sizes (in descending order).
- Semantics-based. A proximity relationship is calculated for each pair of timed automata based on some sharing factors such as the number of shared discrete variables, clock variables, synchronisation labels and the number of common neighbours. A timed automaton is a neighbour of another timed automaton if they communicate through some variables or labels.

In the proposed formal object-oriented model, one ATO may map to more than one ATP and each ATP has a corresponding timed automaton. Thus, the number of timed automata in a system is equal to the number of ATPs. Suppose we are given a set of ATOs $\{Q_1, Q_2, \ldots, Q_n\}$, whose behaviours 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 behaviour of Q_i is represented by $\{P_{i1}, \ldots, P_{ik_i}\}$. Let the set of timed automata that model the set of ATPs be $\{\mathcal{A}_{11}, \ldots, \mathcal{A}_{1k_1}, \ldots, \mathcal{A}_{n1}, \ldots, \mathcal{A}_{nk_n}\}$. After scheduling with some algorithm, let the scheduled set of timed automata be $\{\mathcal{A}^s_{11}, \mathcal{A}^s_{12}, \ldots, \mathcal{A}^s_{1k_1}, \mathcal{A}^s_{21}, \ldots, \mathcal{A}^s_{2k_2}, \ldots \mathcal{A}^s_{n1}, \ldots, \mathcal{A}^s_{nk_n}\}$,
In this work, 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 behaviour of the same ATO are more related to each other than with the ATPs of other ATOs, we first merge all the ATPs of the same family (i.e. the same ATO). Notationally, $\{\mathcal{A}^s_{i1}, \ldots, \mathcal{A}^s_{ik_i}\}$ are merged into $\mathcal{A}^m_1$. Thus, after this step, instead of $\sum_{1 \leq i \leq n} k_i$ timed automata, there are now only n automata.
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, synchronisation labels and the number of communication channels. The higher the number of shared variables and communication channels, the higher is the proximity degree. The pair with the highest proximity are said to be near-relatives and are merged first. Notationally, the following proximity function is defined for each pair of timed automata:

$$\pi(\mathcal{A}_i, \mathcal{A}_j) = \text{Num\_Shared\_Variables}(\mathcal{A}_i, \mathcal{A}_j) + \text{Num\_Channels}(\mathcal{A}_i, \mathcal{A}_j) \quad (2)$$

Algorithm 3: The merge related objects first (MROF) procedure
`MROF`(G_1, TA_Set)
a state-graph G_1;
set of timed automata TA_Set;
$//\mathcal{A}_i = (M'_i, m'^0_i, C'_i, D'_i, Y_i, \chi'_i, E'_i, \tau'_i, \rho'_i) \in TA\_Set, i \geq 1$
{
Step 1: `Select_Graphs`(G_1, G_2, TA_Set);
$//G_1, G_2$:selected state-graphs
Step 2: Let $Reach = Unvisited = \{R_{\text{init}}\}$;
$//Reach$, $Unvisited$: set of regions,
$//R_{\text{init}}: r'^0_1 \times r'^0_2$, where r'^0_i is an initial region of G_i.
Step 3: While $(Unvisited \neq NULL)$ {
Step 4: $R' =$ `Dequeue`$(Unvisited)$; $//R'$: a region
Step 5: For all out-going system transition f {

Step 6: $R'' = \texttt{Successor\_Region}(R', f);$)
//R'': a region
Step 7: If R'' is consistent and $R'' \notin Reach$ {
Step 8: $Reach = Reach \cup \{R''\}$;
Step 9: $Trans = Trans \cup \{f\}$;
// $Trans$: set of transitions
Step 10: Queue(R'', $Unvisited$);
}
}
}
Step 11: return G; //$G = (Reach, R_{init}, \beta, Trans, \alpha)$:
a state-graph

After the first step of merging same family timed automata, $\mathcal{A}_u^m$ and $\mathcal{A}_v^m$ are are merged first if $\pi(\mathcal{A}_u^m, \mathcal{A}_v^m) = \max_{1 \le i \le j \le n}\{\pi(\mathcal{A}_i^m, \mathcal{A}_j^m)\}$.

The details of the MROF() procedure are given in algorithm 3. First, either two timed automata are selected for merging from a given set of timed automata *TA_Set* or if a state-graph G_1 is given as input parameter for MROF(), then one automata G_2 is selected from *TA_Set* for merging with G_1 (step 1). In this procedure, three data-structures are maintained: (i) a queue of regions (*Unvisited*); (ii) a set of reachable regions (*Reach*); and (iii) a set of system transitions (*Trans*). *Unvisited* keeps a record of which regions are yet to be explored, *Reach* keeps a record of all the regions reached, and *Trans* keeps a record of all the system transitions connnecting the regions in *Reach*. The procedure starts from an initial region, R_{init}, which is a Cartesian product of the initial regions (modes) of the selected state-graphs (timed automata). Initially, the initial region is queued in *Unvisited* and recorded in *Reach* (step 2). A region, R', is dequeued from *Unvisited* and corresponding to each outgoing transition e of R' a successor region R'' is constructed by the function Successor_Region(R', e) (steps 4, 5 and 6). If R'' is consistent and is not already in *Reach*, then it is recorded in *Reach* and queued in *Unvisited* for further exploration of its successors (steps 7 and 8). The corresponding transition is also recorded in *Trans* (step 9). The procedure loops until all regions in the queue have been explored (steps 3–10). Finally, a state-graph constructed from *Reach* and *Trans* is returned (step 11).

5.3 Find the best reduction sequence

A reduction technique is a procedure, which takes as input a state-graph and reduces its size in terms of the number of modes and transitions. A state-space can be represented by a state-graph (definition 7) and a reduction technique can be implemented as a modularly packaged manipulator. We consider the following manipulators, which were implemented in the SGM tool: timed symmetry reduction, clock shielding, read-write reduction, and bypass internal transition. The description of these manipulators and their use in our technology integration framework are detailed in Appendix C, Section 9.3. Experimental results on the running AICC example, which was introduced in Section 3.2, will be given in Section 5.5.

The reduction manipulators can all be applied to a state-graph, but the sequence in which the manipulators are applied has an impact on the verification scalability. Although the manipulators all appear to be orthogonal to each other, the information obtained or deleted by one manipulator may be either useful or harmful to another manipulator. There is a theoretical analysis of how an appropriate sequence can be obtained in [56]. The given method needs to perform at least four iterations of compositional state-space construction before the best sequence is obtained. Since it is not quite feasible in terms of the time required for complex systems to iterate four times before deciding on an appropriate sequence, we propose a more heuristic method based on engineering experiences.

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

- 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.
- Permutate 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 a theoretical method from [56]. The results are the same for real-time embedded systems.

5.4 Model checking

The details of the model checking procedure Model_Check() called in step 8 of algorithm 2 are given in algorithm 4. In this Model_Check() procedure, first a given TCTL specification formula ϕ is decomposed into a set of subspecifications, which are topologically sorted in the nesting order, and enlisted in L (step 1). According to this order, each subspecification ϕ' is used for labeling each region in the state-graph (steps 2 and 3). A label is added to a region if the OBDD and DBM of the region satisfies ϕ' (step 4). Each label merely indicates the predicate satisfaction of a region with respect to a given TCTL specification represented by the label. Finally, if the initial region r_0 does not have a label for ϕ (i.e. G does not satisfy ϕ) (step 5), then a counter-example is generated and returned (step 6). Otherwise, NULL is returned (step 7).

Algorithm 4: Model checking procedure
Model_Check(G, ϕ)
a state-graph G;
TCTL formula ϕ;
{
Step 1: $L = \texttt{SubSpecList}(\phi)$;
Step 2: For each subspec $\phi' \in L$ {
Step 3: For each region $r \in R$ //R: regions in G
Step 4: If ($r \models \phi'$) Add_Label(r); }
Step 5: If ($label(r_0, \phi) ==$ FALSE):
//label for ϕ assigned to initial region r_0
Step 6: Return Counter_Eg(G, ϕ);
Step 7: Else return NULL;
}

5.5 Verifying AICC

Coming back to the running example introduced and modelled in Section 3.2, we will now illustrate how the AICC can be compositionally verified using the techniques presented in the preceeding Sections

The timed automata models that were generated for the AICC example (as listed in Table 2), were input to our proposed compositional verification framework, which has been implemented in the Verifier component of VERTAF (see Appendix D, Section 9.4). The proposed approaches to

Table 5: AICC example: state-space reduction and model checking

	n	Comm	Sequence	Regions	Transitions	Time, s	Memory, MB
1	11	SM	$\langle mg_1 \rangle$	> 125 000	> 1869 000	N/A	out of memory
2	11	SM	$\langle mg_1, rw, sc, sm \rangle$	270	1138	50 212.95	61.67
3	6	MP	$\langle mg_1 \rangle$	19 776	26 677	1390.78	210.90
4	6	MP	$\langle mg_2 \rangle$	19 776	26 677	234.38	76.64
5	6	MP	$\langle mg_1, rw, sc, sm \rangle$	141	320	873.39	18.72
6	6	SM	$\langle mg_1 \rangle$	7912	16 557	303.20	52.00
7	6	SM	$\langle mg_1, rw, sc, bit, sm \rangle$	101	290	182.49	5.69
8	6	SM	$\langle mg_1, sc, bit, rw, sm \rangle$	88	232	257.81	6.30
9	6	SM	$\langle mg_1, sc, rw, bit, sm \rangle$	94	258	212.48	5.52
10	6	SM	$\langle mg_1, rw, bit, sc, sm \rangle$	114	366	183.78	5.78
11	6	SM	$\langle mg_1, bit, rw, sc, sm \rangle$	108	369	201.35	6.17
12	6	SM	$\langle mg_1, bit, sc, rw, sm \rangle$	100	281	202.86	6.18
13	6	SM	$\langle mg_1, sm, rw, sc, bit \rangle$	349	3165	366.04	15.91
14	6	SM	$\langle mg_1, rw, sm, sc, bit \rangle^*$	71	180	192.52	6.21
15	6	SM	$\langle mg_1, rw, sc, sm, bit \rangle$	92	259	197.99	6.19
16	6	SM	$\langle mg_1, sm, sc, rw, bit \rangle$	321	3319	423.41	15.83
17	6	SM	$\langle mg_1, sc, rw, sm, bit \rangle$	82	198	237.55	6.18

n: number of timed automata, Comm: communication model, Sequence: manipulator sequence, SM: shared memory, MP: message passing mg_1: sequential merge, mg_2: near-relatives merge, *rw*: read-write, *sc*: shield-clock, *bit*: bypass internal transition, *sm:* symmetry
*best reduction sequence

increasing verification scalability through different merge and reduction sequences, as given in Section 5, were applied to the set of timed automata, and the results are tabulated in Table 5, which corroborates 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 timed automata models (columns 2 and 3 of Table 5), as described in the following.

- System model: With respect to the number of timed automata, a full version consists of 11 timed automata, while a simplified version consists of six timed automata with indices 1,4,6,7,10,12 from Table 2. 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.
- Communication model: Two types of communication models were considered: shared memory (SM) and message passing (MP). In the SM model, shared variables were used as primitives for data/control transfer. In the MP model, send-receive events were used as primitives.

From the results in Table 5, we make the following observations.

- The full version of 11 timed automata (rows 1 and 2) required much more CPU time and memory space compared to the simplified version of six timed automata (rows 3–17). This is in agreement with our knowledge of highly concurrent systems having larger state-spaces.
- With respect to the communication model, MP required more time and memory compared to SM (compare row 3 with row 6). This is because MP uses events and broadcasting is expensive with event-based communication, whereas SM uses variables and broadcasting is automatic through concurrent memory reads.
- Each system version was executed twice: first without reduction and then with reduction. In agreement with our intuition, application of reduction techniques resulted in smaller state-spaces and lower time and memory usages (compare rows 3 with 5 and 6 with 7).
- We also experimented with different merge sequences as described in Section 5.2: mg_1 is a sequential merge according to the timed automata indices and mg_2 is a near relatives merge as defined by (2). Comparing rows 3 and 4, the second sequence gives a better result in terms of a shorter CPU time and memory space utilisations compared to the first sequence. This corroborates our claims in Section 5.2.
- We experimented with different reduction sequences as described in Section 5.3. Comparing rows 7–17, 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 a minimum (compare with row 10), because smaller state-spaces are sometimes obtained by spending a little extra time and space.
- The first version of 11 timed automata (row 1) could not execute to completion without reduction, which gives a general idea of how extremely large sized the global system state-space is.

All the above observations illustrate and corroborate our proposed techniques as described in Section 5. Thus, through this example we have shown how formal verification can be integrated into a complex object-oriented application framework and applied to a real-world industrial example.

6 Conclusions

Using the proposed solutions to the issues in technology integration of object-oriented application framework and formal verification, a software engineer can be guaranteed a verified correct code for his/her application. Issues related to such a technology integration include deciding on what to verify (system model), when to verify (selecting the stage in a design process for verification) and how to verify (integrating formal synthesis and verification algorithms).

Solutions were proposed in this work for each of the above issues, which include the generation of timed automata from formal object-oriented models, the strategy of verification after scheduling but before code generation for greater verification scalability, and the compositional framework for technology integration. A software component called Verifier is implemented in the VERTAF application framework for formal verification of generated software. A real-world industrial example of a cruise controller was given to illustrate the feasibility and success of our approach for integrating formal verification into an application framework. A smaller example was given to illustrate why verification should be performed after scheduling but before code generation.

Future research directions related to this work include developing an API for users to implement their own reduction and verification techniques. Furthermore, design patterns can also be used as a basis upon which new reduction techniques can be invented and applied to the verification of real-time embedded software. Due to the high complexity of real-time embedded systems, hierarchical verification based on the assume-guarantee principle [60] will be integrated into VERTAF in the near future.

7 Acknowledgment

This work was supported by project grants NSC90-2215-E-194-009, NSC91-2213-E-194-008, NSC91-2215-E-194-008, NSC92-2213-E-194-003, NSC92-2218-E-194-009 from the National Science Council, Taiwan, ROC.

8 References

1 Hsiung, P.-A.: 'Object-oriented application framework design for real-time systems'. Proc. 4th Int. Symp. on Real-Time and Media Systems (RAMS), September 1998, pp. 221–227
2 Hsiung, P.-A.: 'RTFrame: An object-oriented application framework for real-time applications'. Proc. 27th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS), September 1998, pp. 138–147
3 Kuan, T., See, W.-B., and Chen, S.-J.: 'An object-oriented real-time framework and development environment'. Proc. OOPSLA'95, Workshop #18, 1995
4 See, W.-B., and Chen, S.-J.: 'High-level reuse in the design of an object-oriented real-time system framework'. Proc. Int. Computer Symp., December 1996, pp. 363–370
5 See, W.-B., and Chen, S.-J.: 'Object-oriented real-time system framework' (Wiley, 2000), Chapter 16, pp. 327–338
6 Hsiung, P.-A., Lee, T.-Y., See, W.-B., Fu, J.-M. and Chen Chen, S.-J.: 'VERTAF: An object-oriented application framework for embedded real-time systems'. Proc. 5th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC), Washington, DC, USA, April 2002, pp. 322–329
7 Hsiung, P.-A., Su, F.-S., Gao, C.-H., Cheng, S.-Y., and Chang, Y.-M.: 'Verifiable embedded real-time application framework'. Proc. IEEE Int. Real-Time Technology and Applications Symp. (RTAS), Taipei, Taiwan, May 2001, pp. 109–110
8 Alur, R., and Dill, D.L.: 'A theory of timed automata', *Theor. Comput. Sci.*, 1994, **126**, pp. 183–235
9 Altisen, K., Gößler, G., Pneuli, A., Sifakis, J., Tripakis, S., and Yovine, S.: 'A framework for scheduler synthesis'. Proc. Real-Time System Symp. (RTSS), 1999
10 Asarin, E., Maler, O., and Pneuli, A.: 'Symbolic controller synthesis for discrete and timed systems', *Lect. Notes Comput. Sci.*, 1995, **999**, pp. 1–20
11 Asarin, E., Maler, O., Pneuli, A., and Sifakis, J.: 'Controller synthesis for timed automata'. Proc. Conf. on System Structure and Control, IFAC, July 1998
12 Hsiung, P.-A.: 'Formal synthesis and code generation of embedded real-time software'. Proc. Int. Symp. on Hardware/Software Codesign (CODES), Copenhagen, Denmark, April 2001, pp. 208–213
13 Maler, O., Pnueli, A., and Sifakis, J.: 'On the synthesis of discrete controllers for timed systems', *Lect. Notes Comput. Sci.*, 1995, **900**, pp. 229–242
14 Wong-Toi, H., and Hoffman, G.: 'The control of dense real-time discrete event systems'. Technical Report STAN-CS-92-1411, Stanford University, Stanford, CA, 1992
15 Wong-Toi, H.: 'The synthesis of controllers for linear hybrid automata'. Proc. Int. Conf. CDC, 1997
16 Alur, R., Courcoubetis, C., Halbwachs, N., and Dill, D.: 'Modeling checking for real-time systems'. Proc. IEEE Int. Conf. on Logics in Computer Science (LICS), 1990
17 Fu, J.-M., Lee, T.-Y., Hsiung, P.-A., and Chen, S.-J.: 'Hardware-software timing coverification of distributed embedded systems', *IEICE Trans. Inf. Syst.*, 2000, **83**, (9), pp. 1731–1740
18 Henzinger, T.A., Nicollin, X., Sifakis, J., and Yovine, S.: 'Symbolic model checking for real-time systems'. Proc. IEEE Int. Conf. on Logics in Computer Science (LICS), 1992
19 Hsiung, P.-A.: 'Embedded software verification in hardware-software codesign', *J. Syst. Archit.*, 2000, **46**, (15), pp. 1435–1450
20 Hsiung, P.-A.: 'Hardware-software timing coverification of concurrent embedded real-time systems', *IEE Proc., Comput. Digit. Tech.*, 2000, **147**, (2), pp. 81–90
21 Schmidt, D.: 'Applying design patterns and frameworks to develop object-oriented communication software', Handbook of Programming Languages (1997), vol. I
22 Hatcliff, J., Deng, X., Dwyer, M.B., Jung, G., and Ranganath, V.P.: 'Cadena: An integrated development, analysis, and verification environment for component-based systems'. Proc. 25th Int. Conf. on Software Engineering (ICSE), OR, USA, May 2003, pp. 160–172
23 Kodase, S., Wang, S., and Shin, K.G.: 'Transforming structural model to runtime model of embedded software with real-time constraints'. Proc. Design, Automation and Test in Europe Conf. (DATE), Germany, March 2003, pp. 170–175
24 de Niz, D., and Rajkumar, R.: 'Time Weaver: A software-through-models framework for embedded real-time systems'. Proc. Int. Workshop on Languages, Compilers, and Tools for Embedded Systems, June 2003, pp. 133–143
25 Knapp, A., Merz, S., and Rauh, C.: 'Model checking timed UML state machines and collaboration', *Lect. Notes Comput. Sci.*, 2002, pp. 395–414
26 Lavazza, L.: 'A methodology for formalizing concepts underlying the DESS notation'. Software Development Process for Real-Time Embedded Software Systems, EUREKA-ITEA project (http://www.dess-itea.org), D 1.7.4, December 2001
27 Amnell, T., Fersman, E., Mokrushin, L., Petterson, P., and Yi, W.: 'TIMES: a tool for schedulability analysis and code generation of real-time systems'. Proc. 1st Int. Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS), Marseille, France, September 2003
28 Merlin, P., and Bochman, G.V.: 'On the construction of submodule specifications and communication protocols', *ACM Trans. Program. Lang. Syst.*, 1983, **5**, (1), pp. 1–25
29 Balarin, F., and Chiodo, M.: 'Software synthesis for complex reactive embedded systems'. Proc. Int. Conf. Computer Design (ICCD), October 1999, pp. 634–639
30 Lin, B.: 'Efficient compilation of process-based concurrent programs without run-time scheduling'. Proc. Design Automation and Test Europe (DATE), February 1998, pp. 211–217
31 Lin, B.: 'Software synthesis of process-based concurrent programs'. Proc. IEEE/ACM Design Automation Conf. (DAC), June 1998, pp. 502–505
32 Sgroi, M., Lavagno, L., Watanabe, Y., and Sangiovanni-Vincentelli, A.: 'Synthesis of embedded software using free-choice Petri nets'. Proc. IEEE/ACM Design Automation Conf. (DAC), June 1999
33 Zhu, X., and Lin, B.: 'Compositional software synthesis of communicating processes'. Proc. Int. Conf. on Computer Design (ICCD), October 1999, pp. 646–651
34 Hsiung, P.-A.: 'Formal synthesis and control of soft embedded real-time systems'. Proc. IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems (FORTE), August 2001, pp. 35–50
35 Su, F.-S., and Hsiung, P.-A.: 'Extended quasi-static scheduling for formal synthesis and code generation of embedded software'. Proc. 10th IEEE/ACM Int. Symp. on Hardware/Software Codesign (CODES), CO, USA, May 2002, pp. 211–216
36 Gau, C.-H., and Hsiung, P.-A.: 'Time-memory scheduling and code generation of real-time embedded software'. Proc. 8th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA), Tokyo, Japan, March 2002, pp. 19–27
37 Hsiung, P.-A., and Gau, C.-H.: 'Formal synthesis of real-time embedded software by time-memory scheduling of colored time Petri nets', *Electron. Notes Theor. Comput. Sci.*, April 2002
38 Balarin, F., *et al.*: 'Hardware-software co-design of embedded systems: the POLIS approach' (Kluwer, 1997)
39 Hsiung, P.-A.: 'Synthesis of parametric embedded real-time systems'. Proc. Int. Computer Symp. Workshop on Computer Architecture, 2000, pp. 144–151
40 Ramadge, P.J., and Wonham, W.M.: 'Supervisory control of a class of discrete event processes', *SIAM J. Control Optim.*, 1987, **25**, pp. 206–230
41 Ramadge, P.J., and Wonham, W.M.: 'The control of discrete event systems', *Proc. IEEE*, 1989, **77**, pp. 81–98
42 Iyer, S., and Ramesh, S.: 'Apportioning: a technique for efficient reachability analysis of concurrent object-oriented programs', *IEEE Trans. Softw. Eng.*, 2001, **27**, (11), pp. 1037–1056
43 Clarke, E.M., and Emerson, E.A.: 'Design and synthesis of synchronization skeletons using branching time temporal logic', *Lect. Notes Comput. Sci.*, 1981, **131**, pp. 52–71
44 Clarke, E.M., Grumberg, O., and Peled, D.A.: 'Model checking' (MIT Press, Cambridge, MA, 1999)
45 Queille, J.-P., and Sifakis, J.: 'Specification and verification of concurrent systems in CESAR', *Lect. Notes Comput. Sci.*, 1982, **137**, pp. 337–351
46 Pasareanu, C.S., Dwyer, M.B., and Huth, M.: 'Assume-guarantee model checking of software: A comparative case study', *Lect. Notes Comput. Sci.*, 1999, **1680**, pp. 168–183

47 Bengtsson, J., Larsen, F., Larsson, K., Petterson, P., Wang, Y., and Weise, C.: 'New generation of UPPAAL'. Proc. Int. Workshop on Software Tools for Technology Transfer (STTT), July 1998
48 Daws, C., Olivers, A., Tripakis, S., and Yovine, S.: 'The tools KRONOS', *Lect. Notes Comput. Sci.*, 1996, **1066**, pp. 208–219
49 Wang, F., and Hsiung, P.-A.: 'Efficient and user-friendly verification', *IEEE Trans. Comput.*, 2002, **51**, (1), pp. 61–83
50 Wang, F.: 'Efficient data-structure for fully symbolic verification of real-time software systems', *Lect. Notes Comput. Sci.*, 2000, **1785**, pp. 157–171
51 Hsiung, P.-A.: 'Timing coverification of concurrent embedded real-time systems'. Proc. 7th IEEE/ACM Int. Workshop on Hardware Software Codesign (CODES), Rome, Italy, May 1999, pp. 110–114
52 Hansson, H.A., Lawson, H.W., Stromberg, M., and Larsson, S.: 'BASEMENT: A distributed real-time architecture for vehicle applications', *Real-Time Syst.*, 1996, **11**, (3), pp. 223–244
53 Roubtsova, E.E., van Katwijk, J., Toetenel, W.J., Pronk, C., and de Rooij, R.C.M.: 'Specification of real-time systems in UML', *Electron. Notes Theor. Comput. Sci.*, 2000, **39**, (3)
54 Hsiung, P.-A., and Wang, F.: 'A state-graph manipulator tool for real-time system specification and verification'. Proc. 5th Intl. Conf. on Real-Time Computing Systems and Applications (RTCSA), October 1998, pp. 181–188
55 Hsiung, P.-A., and Wang, F.: 'User-friendly verification'. Proc. IFIP, TC6/WG6.1 Joint Int. Conf. on Formal Description Techniques For Distributed Systems and Communication Protocols & Protocol Specification, Testing, And Verification (FORTE/PSTV), October 1999
56 Wang, F., and Hsiung, P.-A.: 'Automatic verification on the large'. Proc. 3rd IEEE High-Assurance Systems Engineering Symp. (HASE), November 1998, pp. 134–141
57 Bryant, R.E.: 'Graph-based algorithms for Boolean function manipulation'. *IEEE Trans. Comput.*, 1986, **35**, (8)
58 Alur, R., Courcoubetis, C., Dill, D., Halbwachs, N., and Wong-Toi, H.: 'An implementation of three algorithms for timing verification based on automata emptiness'. Proc. IEEE Int. Conf. Real-Time Systems Symp. (RTSS), 1992
59 Dill, D.: 'Timing assumptions and verification of finite-state concurrent systems', *Lect. Notes Comput. Sci.*, 1989, **407**
60 Hsiung, P.-A., Cheng, S.-Y., and Lee, T.-Y.: 'Compositional verification of synchronous real-time embedded systems'. Proc. 2002 VLSI Design/CAD (VLSI) Symposium, Taitung, Taiwan, August 2002, pp. 187–190
61 Stewart, D.B., Volpe, R.A., and Khosla, P.K.: 'Design of dynamically reconfigurable real-time software using port-based objects', *IEEE Trans. Softw. Eng.*, 1997, **23**, (12)
62 Kim, K.H.: 'APIs for real-time distributed object programming', *IEEE Comput.*, 2000, **33**, (6), pp. 72–80

9 Appendices

9.1 Appendix A: formal object-oriented model [6]

As a compromise between the object-oriented model used by system engineers and the formal model used by system analysts, a formal object-oriented model is proposed for introducing formal verification into an application framework. The syntax and semantics of the model are presented in the rest of this section. In terms of syntax, the model consists of a uniform representation called an autonomous timed object (ATO) for task specification by a software designer. Semantically, the model consists of a uniform representation called an autonomous timed process (ATP) for modelling the behaviour of all tasks, which can be used for verification after transforming into other formal models.

9.1.1 An ATO: An ATO incorporates advantageous features of two object models, namely port-based object (PBO) [61] and time-triggered message-triggered object (TMO) [62]. PBO is suitable for modelling embedded objects with standard nterfaces such as in, out and resource ports. Like PBO, ATO also adopts a standard interface for objects. Unlike PBOs, ATOs 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. TMO is a syntactically simple and natural but semantically powerful extension of the conventional object structure. Its basic structure consists of four parts: (i) an object-data-store section; (ii) an environment-access-capability section; (iii) a spontaneous-method section; and (iv) a service-method section. TMO has been used for developing APIs for real-time distributed object programming. A distinctive feature of TMO is the spontaneous method, which is triggered by a timer, instead of by an event.

Algorithm 5: ATO for AICC supervisor

```
ATO supervisor
{
Step 1:  in double speed;     //input port
Step 2:  in int ICC_control;
Step 3:  signal int switch_sig, info_sig;     //internal
           signals
Step 4:  out int final_control;     //output port
Step 5:  res display_LCD;     //resource port
Step 6:  conf display_size;     //configuration port
Step 7:  ttm poll_switch(100, 100, switch_sig);
Step 8:  ttm show_info(100, 100, info_sig);
Step 9:  ttm gen_control(100,100,switch_sig,info_sig,speed,
           ICC_control, final_control);

}
```

The basic structure of our proposed ATO is illustrated in Fig. 10. There are four types of ports leading to and from an ATO, namely: (i) configuration; (ii) in; (iii) out; and (iv) resource ports. An ATO is initialised 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 a 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 modelling embedded objects due to its generic format. An example ATO for the AICC supervisor object (refer to Fig. 7 and Table 1) is illustrated in algorithm 5.

Within ATO, there are two types of methods, namely event-triggered methods and time-triggered methods. Event-triggered methods are conventional object methods that execute only when called by another object, that is, it is triggered by a method call. It is used for modelling aperiodic task execution, since aperiodic tasks are also triggered by some incoming event. Time-triggered methods are object methods that were created due to the requirement of a timely and predictable behaviour from real-time systems. They are

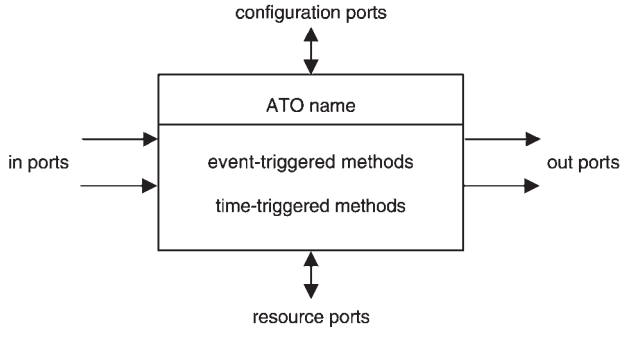

Fig. 10 *An ATO*

also called spontaneous methods in TMO. The execution of time-triggered methods does not require any incoming event, instead it is started upon reaching a prespecified time point. As far as inter-ATO interactions are concerned, using an event-triggered method 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 a smaller overhead when implemented in a shared memory than message passing mechanisms. Thus, they are more appropriate for embedded systems.

9.1.2 An ATP: 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 behaviour using an 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 (just as in UML).

Figure 11 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: (i) event; and (ii) 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 an updated state. A kill signal may be received before or after method execution, which terminates the process.

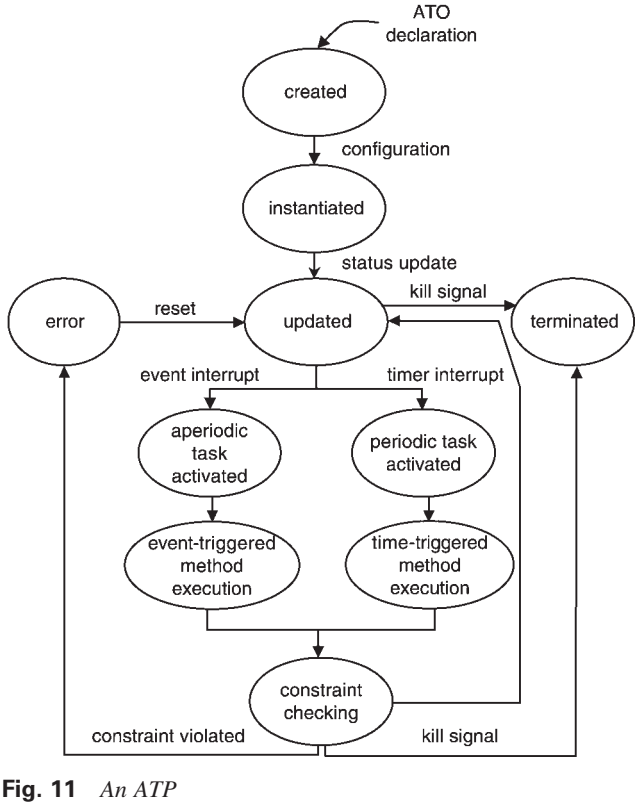

Fig. 11 *An ATP*

A standard uniform process model, in the form of ATP, increases the predictability of a real-time embedded 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 $G_C = (V_C, E_C)$, where nodes in V_C represent ATPs and arcs in E_C 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. The resource requirement is specified as a real-numbered vector, where each element corresponds to some system resources such as memory, processor utilisation, etc. and the real-number corresponds to the amount of each resource required by the particular ATO method.

9.2 Appendix B: analysis of the verification approaches

To analyse the advantage of the schedule-verify-map (SVM) strategy in comparison with conventional approaches, we first define the concept of state non-determinism in a system, and then use it to quantify the benefits obtained by SVM.

Definition 8: State non-determinism. Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i | \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leq i \leq n\}$, and a state s of system $\mathcal{S}$, the state non-determinism of s is defined as the total number of system transitions $(s \rightarrow s')$, whose occurrence at s is non-deterministic (arbitrarily decided), where s' is a successor system state. Notationally, we have the following definition for the state non-determinism $(\psi(s))$at *s:*

$$\psi(s) = \Pi_{1 \leq i \leq n} \omega(s(i)) \tag{3}$$

where $\omega(m)$ is the number of outgoing transitions of a mode m, which are non-deterministic.

In general, not all state non-determinism $(\psi(s))$ at a state (s) can be quasi-statically scheduled. We denote by $\psi_{qss}(s)$ those non-determinism that can be quasi-statically scheduled. In notations:

$$\psi_{qss}(s) = \Pi_{1 \leq i \leq n} \omega_{qss}(s(i)) \tag{4}$$

where $\omega_{qss}(m)$ is the number of outgoing transitions of a mode m, which are non-deterministic and can be quasi-statically scheduled.

Considering the overall effect of quasi-static scheduling on verification complexity, we have the following results. First, given a state s of a system S, since all quasi-statically scheduled non-determinisms have been eliminated before SVM, the reduction obtained is a multiplicative factor of $\psi_{qss}(s)$ for a state s. Second, along a computation run of a system (that is, a sequence of alternating states and system transitions), the combined effect of state non-determinisms at a state s and a successor state s' is multiplicative. This means that the combined non-determinism is

$\psi_{qss}(s) \times \psi_{qss}(s')$. Thus, the overall reduction effect of quasi-static scheduling on a system behaviour can be quantified by the total number of non-determinisms, $\Psi_{qss}(\mathcal{S})$, resolved by quast static scheduling for a system $\mathcal{S}$, as follows.

$$\Psi_{qss}(\mathcal{S}) = \Pi_{s \in \textbf{\textit{Reach}}(\mathcal{S})} \psi_{qss}(s)$$

$$= \Pi_{s \in \textbf{\textit{Reach}}(\mathcal{S})} \Pi_{1 \leq i \leq n} \omega_{qss}(s(i)) \quad (5)$$

where ***Reach*** $(\mathcal{S})$ is the set of reachable states of system $\mathcal{S}$.

Here, the resolution of a set of non-determinisms at a state, s, means instead of considering all possible successor states, s', due to the concurrent non-determinisms in each process, quasi-static scheduling has fixed (that is, scheduled) only one of the successor states as a valid scheduled state, where $s \rightarrow s'$. Hence, the total number of computation runs that SVM explores is $\Psi_{qss}(\mathcal{S})$ times less than that explored by the VSM approach for a system $\mathcal{S}$.

Taking limits on $\Psi_{qss}(\mathcal{S})$, we find that it is a double exponential term in the number of system processes, n, and in the size of the reachable state-space $|\textbf{\textit{Reach}}(\mathcal{S})|$, as given in the following:

$$\Psi_{qss}(\mathcal{S}) \rightarrow (\delta_{ND})^{n \times |\textbf{\textit{Reach}}(\mathcal{S})|} \quad (6)$$

where δ_{ND} is the maximum degree of non-determinism of all processes, $p_1, p_2, \ldots, p_n$. This shows a double exponential decrease in the number of computation runs that need be explored by SVM compared to VSM.

The above was an analytical comparison between the proposed SVM approach and the VSM approach. For a comparison between SVM and SMV, it is difficult to analyse theoretically since each final generated program code might contain different number of auxiliary variables and data structures. Nevertheless, the number of computation runs explored by SMV will be definitely larger than that by the SVM approach due to an increase in state-space size with an increase in the number of variables.

9.3 Appendix C: state-graph reduction techniques

In the following, we assume the system under design is modelled by a set of timed automata $\{A_i | \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leq i \leq n\}$ and the state-graph under reduction is $G = (R, r_0, \beta, F, \alpha)$.

9.3.1 Timed symmetry reduction:

A system modelled by a set of timed automata is said to be symmetric if the automata differ at most by their indices, that is, they are identical on permutation of their indices. For such symmetric systems, a reduction in the size of the state-graphs can be obtained through identification of symmetric regions. A set R_s of regions in a state-graph is said to be symmetric if there exists a permutation $\gamma : I \rightarrow I$ on the automata indices $I = \{1, \ldots, n\}$ such that $\epsilon(\beta)(r), \gamma) = \beta(r')$, for all $r \neq r' \in R_s$, where is defined as follows:

$$\epsilon(\beta(r), \gamma) = \langle \epsilon(OBDD_r, \gamma), \epsilon(DBM_r, \gamma) \rangle \quad (7)$$

where $\epsilon(OBDD_r, \gamma)$ and $\epsilon(DBM_r, \gamma)$ are OBDD and DBM with indices permuted according to γ, respectively.

One of the regions in a set of symmetric regions is preserved in a state-graph and the other regions are deleted. System transitions connected to the deleted regions are then connected to the one representative region left from the symmetric set. Each of these redirected transitions is associated with a permutation label γ, e.g. $\gamma = (2, 1)$, which means 'permute index 1 with index 2 and *vice-versa*'. These redirected transitions can be further reduced by a symmetry-based procedure similar to the one for the system states.

The representative region, that is left in a symmetry set, is selected by first sorting all variables according to some arbitrary order and then selecting the region that has the least value for the first differentiating variable. Reduction of all symmetric sets of regions leaves one representative in each set and thus the size of a state-graph is reduced. For example, suppose two system states M_1 and M_2 are identical in all respects, except for their two respective invariants $h = 1 \wedge k = 2 \wedge x \geq 0$ and $h = 2 \wedge k = 1 \wedge x \geq 0$, as implemented by OBDDs and DBMs, with h and k as discrete variables and x as a clock variable. Upon application of the permutation $\gamma = (2, 1)$, the latter becomes identical to the former. If $\langle x, h, k \rangle$ is the order selected for the variables, then M_1 will be selected as the representative system state and M_2 will be deleted. Thus, we have one less nodes in the state-graph.

This reduction technique is very useful for our integration framework because the underlying formal object-oriented model is symmetric. All objects and functions have a uniform ATO-ATP representation. This manipulator results in large reductions when applied to systems developed using our framework.

9.3.2 Clock shielding:

Given a state-graph, if the values of a clock variable are different in two regions, but the clock is either never read or is always reset before being read, then that clock variable should not be a distinguishing factor between the two regions. In plain terms, those two regions can be considered to be the same if they differ only in the values of that clock variable. For such an identification of the two regions, the clock is shielded in both the regions by ignoring its values.

Formally, a clock variable $c \in \cup_i C_i$ can be shielded in the DBM representation of clock constraints, DBM_r, of a region $r \in R$ if either of the following conditions is satisfied:

- $\exists$ a region $r' \in R$ such that:
  (i) r' is reachable from r, i.e. there exists a path or a sequence of regions connected by system transitions from r to r' in the state-graph G;
  (ii) c is reset along an incoming transition $f \in F$ of r', i.e. $\exists f \in F$, $\alpha(f) = \langle r_s, r_d, I, E, \gamma \rangle$, $r_d = r'$, $c \in \rho_i(e)$, for some $i \in I$, $e \in E \cap E_i$;
  (iii) along all paths from r to r', neither does c appear in any invariant of a system state nor in any triggering condition of a system transition, i.e. $c \notin \chi_i s(i))$, $\forall i \in \{1, \ldots, n\}$, $\forall s \in r''$, $\forall r''$ in all paths from r to r' and $c \notin \tau_i(e), \forall i \in I, \forall e \in E, \forall f = \langle r_s, r_d, I, E, \gamma \rangle$ that occur in all paths from r to r'.
- For all regions r' reachable from r, neither does c appear in any invariant of a system state in r' nor does it appear in any triggering condition of a system transition reachable from r, i.e. $c \notin \chi_i s(i))$,$\forall_i \in \{1, \ldots, n\}$, $\forall s \in r'$, and $c \notin \tau_i(e)$, $\forall i \in I$, $\forall e \in E$, $\forall f = \langle r_s, r_d, I, E, \gamma \rangle$ with $r_d = r'$.

Reduction through clock shielding is useful for our framework because: (i) our target systems are concurrent with many software components and objects modelled by timed automata; and (ii) they are real-time. Hence, there are many clocks within a real-time embedded system, on which the clock shielding manipulator can be applied. Furthermore, each ATO also has one or more time-triggered method(s), which also depend on clocks. Thus, this manipulator becomes handy in reducing state-space sizes.

9.3.3 Read-write reduction:

Whereas the clock shielding technique focuses on clock variable valuations, the read-write reduction technique is targeted at discrete variables. Recall that in the compositional approach (see algorithm 2), an intermediate state-graph is obtained in each merge iteration, which represents the state-space of a partial system. Based on this distinction between a partially composed system and a yet uncomposed system part, read-write reduction analyses the exact values which discrete variables do or do not take in each region of a composed state-graph.

Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i \,|\, \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leq i \leq n\}$, let S_C be the composed part and $S \backslash S_C$ be the uncomposed part of system S. For a given discrete variable $d \in \cup_{1 \leq i \leq n} D_i$, let $D_{S_C}(d)$ be the set of values written to d by processes in S_C, but not by processes in $S \backslash S_C$. With the above notations, the following theorem constitutes the core rationalisation for the read-write reduction technique.

Theorem 1: Given a composed part S_C of system S, a discrete variable d, and a finite run segment $\langle r_0, r_1, \ldots, r_k \rangle$ such that for all $i, 0 \leq i < k$, region r_i goes to region r_{i+1} without making an assignment to d on any transition of any process in S_C, then the following hold:

- if there is an assignment $d := c$; on an incoming transition of region r_0, then for all $i, 0 \leq i < k, r_i \models \wedge_{c' \in D_{S_C}(d) \backslash \{c\}} d \neq c'$;
- if there is a triggering condition $d = c$ and no assignment on an incoming transition of region r_0, then for all $i, 0 \leq i < k, r_i \models \wedge_{c' \in D_{S_C}(d) \backslash \{c\}} d \neq c'$;
- if there is a triggering condition $d \neq c$ and no assignment on an incoming transition of region r_0, then for all $i, 0 \leq i < k, r_i \models d \neq c$.

Proof: The results follow easily from the following three facts: (i) a discrete variable changes value only when assigned a new value by an assignment on a transition; (ii) all uncomposed processes in $S \backslash S_C$ will never write values to d from the set of values $D_{SC}(d)$; and (iii) all composed processes in S_C will not assign any value to d along the run segment.

For implementation, each of the deductions on discrete variable valuations from theorem 1 was conjuncted with the OBDD (invariant) of each region in a state-graph. This conjunction can be done either as a post-processing after the state-graph is constructed or on-the-fly while it is being constructed. By considering such deducted values of discrete variables, system transitions that have contradicting triggering conditions, cannot possibly happen and are thus deleted from the state-graph. Reduction of the state-graph is thus achieved.

This manipulator is useful for our framework because besides time-triggered methods, an ATO also has one or more event-triggered method(s), which depend on some type of communication variables or channels. Thus, they also have many discrete variables and the read-write reduction technique can thus reduce intermediate state-graph sizes.

9.3.4 Bypass internal transition:

Recall again that in the compositional approach (algorithm 2), two state-graphs are selected for merging into a single state-graph, which is then reduced in each iteration. Thus, at any stage of the merge-reduction process, there is always a composed (merged) part of the system and an uncomposed (yet-to-be merged) part. Some behaviours that are internal to a composed part may be invisible to the processes in the uncomposed part. Based on this insight, a composed state-graph can be reduced by eliminating such internal behaviours. A formal treatment is given in the following.

Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modelled by a set of n timed automata, $\{\mathcal{A}_i \mid \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, Y_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leq i \leq n\}$, let S_C be the composed part and $S \backslash S_C$ be the uncomposed part of system S. Let $D_{S_C} \subseteq \cup_{1 \leq i \leq n} D_i$ be the set of discrete variables accessed (read or written) by the processes in S_C only and not by any process from $S \backslash S_C$. A discrete variable d is said to be internal with respect to S_C and a given TCTL specification ϕ if the following two conditions are satisfied: (i) $d \in D_{S_C}$; and (ii) d is not in ϕ.

For example, all local variables of a process are internal in any state-graph.

A system transition reads and writes discrete variables through its triggering condition and assignments, respectively. A system transition is said to be internal to S_C if it accesses only internal variables. Accessing any clock variable, either local or global, results in a non-internal transition because the elapse of time is in general a globally visible action.

Once all internal transitions are identified in a state-graph, each one of them is bypassed by per-forming the following steps:

- for each internal transition $f = \langle r_s, r_d, I, E, \gamma \rangle \in F$;
- if $\beta(r_s) \rightarrow \beta(r_d)$ and $\beta(r_s) \rightarrow \phi$ then:
  (i) for each successor transition $f' = \langle r_d, r'_d, I', E', \gamma' \rangle \in F$
  (ii) add a new transition $f'' = \langle r_s, r'_d, I', E', \gamma' \rangle$ in F,
- delete transition f from F.

Once all the internal transitions are bypassed, there may be some regions which become unreachable and are thus deleted from the state-graph. An example of such an unreachable region is one that has only a single incoming internal transition.

Bypass internal transition is again a useful reduction manipulator for our framework due to its inherently compositional characteristic.

9.4 Appendix D: verifier: a VERTAF component

The proposed solutions to design and verification technology integration issues are being implemented in an object-oriented application framework called VERTAF [6, 7]. VERTAF generates code for real-time embedded systems using formal modelling and synthesis techniques. A separate software component called Verifier is being developed in VERTAF for encapsulating the proposed solutions. Verifier has several parts as follows.

- model generator. This part is responsible for automatically generating timed automata models. When input a set of ATPs, model generator transforms it into a set of timed automata as de-scribed in Section 3.1. If the input consists of a set of ATPs and a schedule as generated by the scheduler component of VERTAF, then the model generator produces a set of timed automata within which the schedule information is integrated. Interested readers may refer to [19] for further details on schedule information integration. This part eliminates the learning curve that an engineer might have to undergo if he/she had to learn how to model using timed automata without prior experience.
- Merge manipulator. This part is responsible for producing a state-graph from two component timed

automata, which represents their concurrent behaviours. As discussed in Section 5.2, different sequences for merging are possible and all have been implemented in this part of Verifier.

• Reduction manipulators. Within the Verifier component, we are currently integrating the SGM tool [35, 49, 54, 55] which is a high-level, modularised verification tool for real-time systems. SGM allows complete user-flexibility in applying varied manipulator sequences to an application as a result of which verification scalability is increased easily. This part consists of the state-graph reduction techniques implemented as modular manipulators from SGM. All reduction manipulators discussed in Appendix C can be used to reduce state-graphs derived from the merge manipulator.

• Model checker. This part is responsible for actually verifying if a reduced state-graph satisfies a given TCTL specification. As discussed in Section 5.4, a compositional model checker is developed. The main difference of this model checker from the conventional ones found in UPPAAL [47] or KRONOS [48] tools is that it is compositional. It can be applied to intermediate state-spaces to possibly get answers before the complete global state-space is constructed.