Building Models of Real-Time Systems from Application Software

JOSEPH SIFAKIS, STAVROS TRIPAKIS, ASSOCIATE MEMBER, IEEE, AND SERGIO YOVINE

Invited Paper

We present a methodology for building timed models of real-time systems by adding time constraints to their application software. The applied constraints take into account execution times of atomic statements, the behavior of the system's external environment, and scheduling policies. The timed models of the application obtained in this manner can be analyzed by using time analysis techniques to check relevant real-time properties.

We show an instance of the methodology developed in the TAXYS project for the modeling and analysis of real-time systems programmed in the Esterel language. This language has been extended to describe, by using pragmas, time constraints characterizing the execution platform and the external environment. An analyzable timed model of the real-time system is produced by composing instrumented C-code generated by the compiler. The latter has been re-engineered in order to take into account the pragmas. Finally, we report on applications of TAXYS to several nontrivial examples.

Keywords*—*Automatic code instrumentation, correct implementation, modeling, real-time systems, synchronous and asynchronous execution, timing analysis.

I. INTRODUCTION

Modeling plays a central role in systems engineering. The use of models can profitably replace experimentation on actual systems with incomparable advantages, such as:

1) enhanced modifiability of the model and its parameters;
2) ease of construction by integration of models of heterogeneous components;
3) generality by using abstraction and behavioral nondeterminism;
4) enhanced observability, controllability, and avoidance of probe effect or of disturbances due to experimentation;
5) possibility of analysis and predictability by application of formal methods.

Manuscript received December 20, 2001; revised August 31, 2002.This work was supported in part by the European IST project "Next TTA" under Project IST-2001-32111.

The authors are with Verimag, 38610 Gières, France (e-mail: Joseph.Sifakis@imag.fr; Stavros.Tripakis@imag.fr; Sergio.Yovine@imag.fr).

Digital Object Identifier 10.1109/JPROC.2002.805820

Currently, validation of real-time systems is done by experimentation and measurement on specific platforms in order to adjust design parameters and, it is hoped, achieve conformity with requirements. The existence of modeling techniques is a basis for rigorous design and should drastically ease validation. Modeling systems in the large is an important trend in software and systems engineering today, as demonstrated by the so-called model-based approaches [9], [10], [19], [25]. Nevertheless, building models that faithfully represent real-time systems is not a trivial problem. For this reason, models are often used at early phases of system development, at high abstraction level, and they do not easily carry through the entire design life cycle.

This paper unifies results developed at Verimag (Gières, France) over the past four years into a methodology for modeling real-time systems. The methodology is based on the thesis that a timed model of a real-time system can be obtained by adequately restricting the behavior of its application software with time constraints characterizing the execution platform and the external environment (e.g., execution times, task arrival times, or scheduling policies) [31].

The paper presents the methodology, discusses problems related to its feasibility, and describes its application to synchronous real-time systems. It is organized as follows. Section II presents current practice and challenges in real-time system development. The methodology considers modeling as an activity integrated in the system development process. Getting faithful models requires a clear understanding of the implementation process and the possibility of relating the application software with its run-time behavior. We discuss the general problem of establishing a connection between the application software with its implementation and explain why current practice does not address the problem in a satisfactory way.

Section III presents the general modeling framework. It discusses general methodological aspects about how the application software can be related to its implementation and proposes a general notion of correct implementation.

Section IV presents results of the TAXYS project, an application of the general methodology to synchronous real-time systems developed in Esterel. It provides a simple example and benchmarks from nontrivial case studies.

Section V discusses perspectives of application to multithreaded asynchronous real-time systems and reports on ongoing work in that direction.

II. Current Practice and Challenges

A. The Divide Between Application Software and Real-Time System

Application software is usually written in some high-level programming language, such as C, Java, ADA-95 [36], SDL [20], Esterel [5], or Lustre [17]. To cope with the complexity of applications, software is decomposed into components. Conceptually, the programmer reasons in terms of a model of computation while developing the software. This model is either explicit in languages with formal semantics or implicitly assumed by the programmer. This *high-level* model is based on abstractions about the behavior and interaction of components. Such abstractions include concurrent execution, instantaneous computation, zero delay, and perfect communication between components and/or between components and the external environment, atomicity of actions, and so on. Indeed, these are very useful abstractions that drastically simplify description. Moreover, they are necessary for platform independence, which is crucial to software portability and reuse. Finally, they often reduce the complexity of analysis, which is especially important during the design phase.

Application software must be implemented on a particular platform. Usually, the implementation process involves more than compiling. For instance, different parts of the application software, developed by different teams, are sometimes compiled separately. Then, the various executables need to be integrated, on either a single-processor platform or a multiprocessor platform with some communication medium (e.g., a bus) linking the processors. In any case, implementation compromises the abstractions of the high-level programming model: components must be executed on one or a few processors (thus, sequentially rather than concurrently), computation and communication takes time, and so on. Therefore, implementation involves resolving a number of issues not always resolved at application software level, such as resource allocation (e.g., distribution of tasks to different processors, scheduling policy) or task communication and synchronization (e.g., shared memory, semaphores, queues).

From the preceding discussion, it becomes apparent that the divide between application software and implementation (real-time system) resides in the fact that the high-level model of computation (the model of the application software) is, in general, different from the *low-level* model of computation (the one of the real-time system). Software is immaterial and, ideally, platform independent; therefore, the high-level model often uses a *logical-time* axis, for example, a partial or total ordering of events. The implementation runs on a platform and interacts with its environment in *real time*; thus, the low-level model uses a real-time axis, for example, positive reals to express the delay between two events. It is worth noting that, even if the application software contains statements depending on real time, e.g., timeouts, they are essentially not different from awaiting an external event, such as hitting an obstacle.

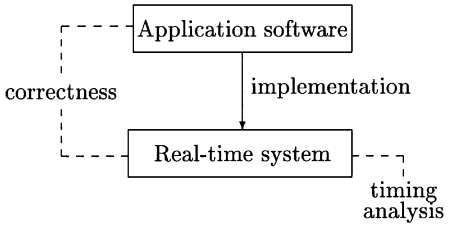

Fig. 1 The implementation process and main challenges.

Since abstractions break down during implementation, it is not at all obvious that a real-time system preserves the properties of its application software. For example, we may have verified absence of deadlocks using a high-level model of the application software which assumes actions take zero time. Nevertheless, the real-time system may have deadlocks, owing to the fact that computations take a nonzero amount of time and result in desynchronizations. The first challenge, therefore, is to check that a real-time system is correct with respect to its application software. We call this the *correctness* problem (see Fig. 1). To check correctness formally, we must first build models of both the application software and the real-time system. Also, since these models use different time axes (logical versus real time), a framework must be developed that relates the two and encompasses a notion of correctness.

Preserving the properties of the application software not only means preserving a set of *functional* (untimed) properties. It is also necessary to verify *nonfunctional* properties. For example, it is critical in most control applications that the system quickly reacts to changes of the environment. This can be expressed as a property of the form "whenever event a occurs, event b will follow at most after x time units." This is a nonfunctional property, because the time units are measured in real time. We call the problem of checking such nonfunctional properties the *timing analysis* problem. Like correctness, timing analysis also requires building a model of the real-time system.

As presented above, both the correctness and timing analysis problems are *analysis* problems. They can also be turned into *synthesis* problems, where we look for methods that help resolving the choices that need to be made during implementation. Some of these choices are imposed by external constraints; for example, the execution platform and network may be fixed because of power consumption or economic reasons. Many other choices, however, are often left to the programmer or the compiler. For example, the programmer must decide which concurrent components to group into a single sequential thread. The compiler must decide, in case there are two independent computations, in what order to perform them. Such decisions are often taken in an arbitrary or *ad hoc* manner. Instead, they should be *guided* by correctness and timing requirements.

In Section III, we propose a modeling framework that addresses the correctness and timing analysis problems. We do not address the synthesis problem, although we recognize it as an important challenge [1].

B. Synchronous and Asynchronous Real Time

Current practice in real-time systems design follows two well-established paradigms, namely, synchronous and asynchronous.

The synchronous paradigm [4] has been developed in order to better control reaction times and interaction with the external environment. It assumes that a system interacts with its environment by performing global computation steps. In a step, the system reacts to environment stimuli by propagating their effects through its components in a well-defined order (*causality order*). The *synchrony assumption* states that the system's reaction is fast enough with respect to the environment. In practice, this means that environment changes occurring during a step are treated at the next step and implies that responsiveness and precision are limited by step duration. Hardware description languages (such as VHDL) and the so-called synchronous languages (such as Esterel [5], Lustre [17], and Signal [16]), adopt the synchronous paradigm. These languages are used, among others, in signal processing and automatic control applications.

Synchronous programs are typically implemented as a single task that executes a read/compute/write loop. Simple scheduling policies are employed to resolve concurrency of components, e.g., by serialization of the causal order mentioned above. This is typically done only once, at compile-time, avoiding scheduling overhead at run-time. Moreover, an operating system is often unnecessary. For correct implementation, care should be taken to verify the synchrony assumption, for instance, by ensuring that the execution time of a reaction is not too long.

The asynchronous paradigm arose from the *multitasking* execution model. It does not impose any notion of global execution step. The concurrent components (threads, tasks, or processes) proceed each at its own pace and communicate, for instance, by message passing. Therefore, this paradigm is particularly suitable for distributed systems. Languages such as ADA-95 [36], C, and Java adopt the asynchronous paradigm. When concurrency operators are not explicit, they are provided through the use of thread libraries. Note that these languages are more general purpose than the synchronous languages mentioned above.

Implementation of asynchronous languages typically relies on an operating system. The latter is responsible for scheduling, which is usually based on static priorities. Real-time scheduling theory (e.g., [11], [12], [18], and [32]) provides techniques, such as *rate-monotonic* analysis [26], that guarantee satisfaction of simple time constraints, such as *deadlines*. Unfortunately, these results are often applicable only to simple models, and are difficult to generalize.

Neither of the two paradigms faces the implementation challenges in a satisfactory manner. One of the difficulties in the synchronous paradigm is that the synchrony assumption is not always easy to meet, in particular when high responsiveness to the environment is required. Another drawback is that modularity cannot be easily handled; for instance, it is hard to compile synchronous programs separately and then link them together or with nonsynchronous implementations. The latter is a problem in practice, since in large projects, software is usually provided by different teams. On the other hand, the asynchronous paradigm results in less predictable implementations, which are hard to analyze.

For advanced real-time applications, it is desirable to combine the synchronous and asynchronous paradigm for both application software and implementation. We need programming and specification languages combining the two description styles, as some applications have loosely coupled subsystems composed of strongly synchronized components.

Even in the case where purely synchronous or asynchronous programming languages are used, it is interesting to mix synchronous and asynchronous implementations to cope with inherent limitations of each paradigm. For instance, for synchronous languages, making the scheduling of components within a step more sophisticated can result in a system that is more sensitive to environment changes. It is also possible to relax synchrony at implementation level by mapping components solicited at different rates to different nonpreemptable tasks.

Proposals of real-time versions of object-based languages such as Java [21], [28] and UML [15], provide concepts and constructs allowing to mix the two paradigms and even to go beyond the distinction between synchronous and asynchronous. In principle, it is possible to associate with objects general scheduling constraints to be met at run-time. The concept of dynamic scheduling policy should allow combining the synchronous and asynchronous paradigms or, most importantly, finding intermediate policies corresponding to tradeoffs between these two extreme policies. The development of technology enabling such a practice is certainly an important work direction.

III. Modeling Framework

In this section, we propose a modeling framework that allows relating the properties of the application software and those of the implementation. In our discussion, we will limit ourselves to single-processor implementations.

A. The Elements of the Framework

The elements of our modeling framework are depicted in Fig. 2.

The model of the application software is shown at the top of the figure. The application software is made up of a number of *components*, which are conceptually executing *concurrently*. In the figure, a distinction is made between components responsible for the interaction with the environment (shaded area on the left of the figure) and components performing the computation (shown on the right). Sometimes, owing to the abstractions made at the high level (e.g., assumptions such that the software is always *receptive* to the environment), the environment interface components

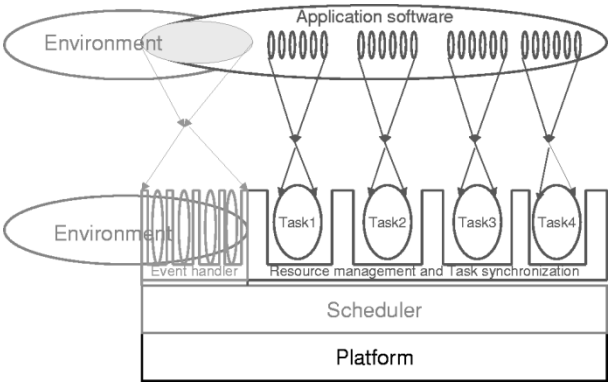

Fig. 2 The elements of the modeling framework.

are not explicit, but implied by the semantics of the language. The same is true concerning the interaction between concurrent components.

During implementation, the concurrent components are mapped into one or more *tasks*. A task is a sequential process. Since tasks are sequential, when several high-level concurrent components are mapped into the same task, their concurrency must be resolved. This is a scheduling problem. The order of execution of the concurrent components within a task is determined either at compile-time (as is typically done in synchronous languages) or at run-time (e.g., ROOM [29]). A second level of scheduling is required when there are more than one tasks. This appears as "Scheduler" in Fig. 2. The scheduling policy is usually provided by the operating system (e.g., *fixed-priority* scheduling) and the choice is to apply the policy to the current set of tasks (e.g., assign a priority). It may also be the case, however, that the scheduler is "custom-designed" for a particular application (e.g., a time-triggered cyclic scheduler).

Schedulers are timed systems that manage shared resources so as to respect the timing requirements of the tasks and of the environment. Typical timing requirements are deadlines about the action completion and task arrival times. Usually, schedulers apply scheduling policies to choose among pending requests for access to resources. Methods for modeling schedulers have been proposed in [2].

The interactions between components of the application software must also be explicitly implemented. For components grouped into the same task, interactions are often implemented by the compiler, using shared variables read and written by the components. In general, the intertask communication mechanisms provided by the operating system or any other middleware (e.g., Java Virtual Machine) can be used. The implementation of component interactions appears as "Resource management and Task synchronization" in Fig. 2.

The interface with the environment needs to be implemented as well. Concerning inputs, this is typically done using two techniques. The first is the use of *interrupts*, and can be seen as *environment-driven*: whenever some external device detects a change in the environment, it raises an interrupt, and an interrupt handler is called. The second technique is the use of *sampling*, and can be seen as *program driven*: whenever the program is ready to accept new inputs, it calls some device-driver routines that gather data from the sensor devices, and use them in their computation. Concerning outputs, no standard method exists. Sometimes, the outputs are written throughout the computation by calling special device-driver routines. Sometimes, the outputs are gathered and written at the same time, at the end of the computation or at the end of a period.

In Fig. 2, a distinction is made between the environment and the execution platform. Although this distinction reflects reality, it is sometimes not important at modeling level. Indeed, for reasons of complexity, the platform cannot be modeled precisely, and is therefore abstracted as a nondeterministic "player" that interacts with the running program in a similar way as the external environment. In our methodology, the execution platform will be abstracted merely as a set of nondeterministic delays. Accurate bounds on these delays are crucial for faithful modeling.

B. Correctness

We now propose a formal modeling framework that allows us to reason precisely about correctness of a real-time system (implementation) with respect to its application software (specification). In summary, both the application software and the real-time system are seen as *reactive machines*, which consume inputs from the environment and produce outputs. Nevertheless, these machines operate in different time domains. Therefore, additional functions are needed in order to relate the two time domains. We introduce *untiming* functions, which map real-time inputs and outputs into logical-time ones. In what follows, T will denote the logical time domain and T_R the real-time domain.

Inputs and Outputs: Let In be the set of all *input values* and Out the set of all *output values*. We assume that In and Out are *union closed*; that is, they are power sets of a given set. For example, if the application is event triggered, then if a and b are events, then $\{a, b\}$ is also an event. Union closure is assumed for technical reasons and is not a restrictive assumption.

Inputs and outputs are partial functions on time domains. We denote by $[X \to \mathrm{In}]$ (resp., $[X \to \mathrm{Out}]$) the set of all inputs (resp., outputs) on the time domain X.

The Model of the Application Software: We view the application software as a function

$$f : [T \to \mathrm{In}] \to 2^{[T \to \mathrm{Out}]}. \tag{1}$$

The output of f is, in general, a set, meaning that the application-software is allowed to be nondeterministic. This helps, for instance, to model multithreaded programs.

The Model of the Real-Time System: Similarly, we view the real-time system as a function

$$f_R : [T_R \to \mathrm{In}] \to 2^{[T_R \to \mathrm{Out}]}. \tag{2}$$

The nondeterminism of f_R reflects the uncertainty in the behavior of the execution platform.

The Untiming Functions: Since the program and the implementation are not generally defined on the same time domain, we need some way to relate the behaviors on the two different time domains. This is done with the untiming functions.

The function

$$\psi_{\text{in}} : [T_R \to \text{In}] \to [T \to \text{In}] \tag{3}$$

maps every input on the real-time axis T_R to some input on the logical-time axis T. We have a similar function for outputs, namely

$$\psi_{\text{out}} : [T_R \to \text{Out}] \to [T \to \text{Out}]. \tag{4}$$

Examples of untiming functions are given in Section III-C.

We consider that all the functions f, f_R, ψ_{in}, and ψ_{out} are transducers, i.e., they are monotonic with respect to the prefix order: if ϕ is any one of these functions, for any ρ and $0 \leq t \leq t'$, $\phi(\rho[0,t])$ is the prefix of some function in the set $\phi(\rho[0,t']))$, where $\rho[0,t]$ is the restriction of ρ in the interval [0, t].

Correctness: Correctness is defined with respect to an environment $E \subseteq [T_R \to \text{In}]$ and untiming functions ψ_{in} and ψ_{out}. Given a model of the application software f and a model of the real-time system f_R, we say that f_R *implements* f, with respect to E, ψ_{in}, and ψ_{out}, if

$$\forall \rho \in E, \quad \psi_{\text{out}}(f_R(\rho)) \subseteq f(\psi_{\text{in}}(\rho)) \tag{5}$$

where the definition of ψ_{out} is extended from inputs/outputs to sets of inputs/outputs in the natural way.

The notion of correctness is illustrated by Fig. 3.

As in any implementation relation, we should avoid having trivial implementations. Therefore, we require that

$$\forall \rho \in E, \quad f(\psi_{\text{in}}(\rho)) \neq \emptyset \Longrightarrow \psi_{\text{out}}(f_R(\rho)) \neq \emptyset. \tag{6}$$

Timing Analysis: Timing analysis is performed with respect to a given environment E and a given set of *timing requirements*, P. Formally, P is a relation between inputs and outputs in the real-time domain, that is, $P \subseteq [T_R \to \text{In}] \times [T_R \to \text{Out}]$. The meaning is that, given an input $\rho : T_R \to \text{In}$, an output $\pi : T_R \to \text{Out}$ is legal iff $(\rho, \pi) \in P$.

Then, given a model of the real-time system f_R, f_R *satisfies* P, if

$$\forall \rho \in E, \quad \forall \pi \in f_R(\rho), \quad (\rho, \pi) \in P. \tag{7}$$

C. Modeling Methodology

We now propose a methodology for obtaining the elements of the formal modeling framework f, f_R, ψ_{in}, ψ_{out}, and E. The methodology is based on the principle that pieces of software that represent system behavior (timed or untimed) are models. Complex models can be obtained by composition of software components. In principle, there are no specific requirements about the languages used to write the software (e.g., formal semantics, model of execution). Nevertheless, the specific features of the language used may have an important impact on the possibility to construct more or less efficiently the models and to apply verification techniques.

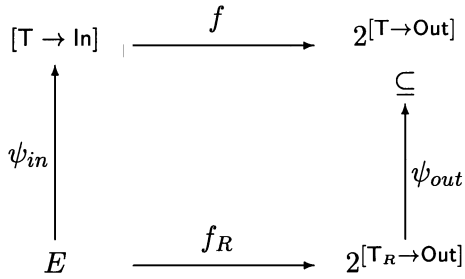

Fig. 3 Illustration of the correctness notion.

The application software is a reactive program. Thus, the function f can be defined by the correspondence between (logical-time) inputs and outputs when the program is executed.

The function f_R can be obtained by building a model according to the decomposition shown in Fig. 2. Models of tasks and their interaction and of the execution platform can be composed by adding time constraints about durations of atomic actions in tasks and of platform primitives. This requires, in particular, decomposing tasks into atomic (nonpreemptable) sequences of statements, and estimating their execution times on the target platform. Adding timing information about execution times (e.g., lower and upper bounds) allows abstracting from all the details about the underlying platform.

The functions ψ_{in} and ψ_{out} relate inputs and outputs on the real-time axis to inputs and outputs on the logical axis.

ψ_{in} abstracts away from real time by mapping into the same logical instant independent events or events close enough in the real-time scale. The logical ordering takes into account causal relations between events. Thus, ψ_{in} can be considered as an abstract specification of an input event handler (see Section IV-B)

If ρ is an input of the real-time system, then $\psi_{\text{in}}(\rho)$ defines a *time base*, that is, an increasing sequence of instants $(t_j)_j$ in the following manner. As ψ_{in} is a transducer, under some technical continuity conditions we have that, for any natural number j, there exists a minimal real number t_j, such that $\psi_{\text{in}}(\rho(0,t_j]) = \psi_{\text{in}}(\rho)(j)$, where $\rho(0,t_j]$ is the restriction of ρ in the interval (0, t_j]. In other words, the logical input at instant j depends only on the input values in the interval (0, t_j].

In practice, the time base $(t_j)_j$ can be defined by imposing *separability* constraints which guarantee that significant changes of the environment are not mapped into the same logical instant. These constraints can be any combination of: 1) constraints restricting environment state changes within two consecutive instants of the time base, e.g., values of a variable or the integral of some variable remain within some bounds; and 2) time constraints maintaining the distance between two successive instants of the time base close enough to separate significant input changes, e.g., sampling.The notion of a *separator* event is useful to define untiming

functions. An event is a separator if distinct occurrences of this event are mapped into distinct logical instants.

For $T = \mathbb{N}$ and $T_R = \mathbb{R}$, some examples of untiming input functions are the following.

1) A very simple case comprises sampling functions where a periodic event is a separator; that is, for any ρ and $n \in \mathbb{N}$, $\psi_{\text{in}}(\rho)(n) = \rho(n \cdot \delta)$, where δ is the sampling period. This is a typical situation in time-triggered systems [23].
2) Another simple case corresponds to the situation where all the input events are separators. This means that the application receives the input events one by one in the order they arrive; that is, for any ρ and $n \in \mathbb{N}$, $\psi_{\text{in}}{}'(\rho)(n) = \rho(t_n)$, where $(t_n)_n$ are the arrival times of all the events.

Synchronous languages often assume that the logical input at instant j depends only on the inputs in the interval $(t_{j-1}, t_j]$, i.e.,

$$\psi_{\text{in}}(\rho(0, t_j]) = \psi_{\text{in}}(\rho)(j) = \psi_{\text{in}}(\rho(t_{j-1}, t_j]). \quad (8)$$

If the input ρ takes finitely many values in any interval of the time base, then $\psi_{\text{in}}(\rho)(j)$ is taken to be equal to $\bigcup_{t_{j-1} < \tau \leq t_j} \rho(\tau)$, i.e., the logical input is the union all the events that occurred in the jth interval.

The untiming function ψ_{out} can be defined in a similar manner.

The model E of the environment is obtained by using more or less strong abstractions, which reduce the complex (and often not precisely known) dynamics of the environment to a set of input/output behaviors subject to real-time constraints. For example, we could model the environment as a set of inputs arriving periodically or with a minimum interarrival time. Although E is not, strictly speaking, part of the real-time system's model, it is necessary to close a system's description in order to study the dynamics of the interaction.

D. Tool Support

The application of the previously described methodology even to simple real-time systems is not tractable without tool support. It is possible, under some conditions, to relieve the user from tedious modeling work by automating model generation.

Depending on the language in which the application software is written, it is sometimes possible to establish a correspondence between atomic sequences of actions at task level (target code) and sequences of statements at application software level (source code). In such a case, the source code can be annotated with time constraints on the execution time of the target code on the given platform. Such constraints can be derived using, for instance, execution time estimation techniques [27].

There are tools, such as METAH [8] and GIOTTO [19], that provide support for building and analyzing models of real-time software. Our methodology, implemented in the tool TAXYS [6], [13], is based on the idea that compilation or synthesis techniques can be used for this purpose. In TAXYS, the compiler is engineered to generate instrumented code, where the added statements encode the execution of the real-time model, according to the annotated time constraints.

IV. TAXYS

The structure of TAXYS is depicted in Fig. 4. A typical embedded real-time program is decomposed into a control part, written in Esterel, and a data manipulation and computation part, written in C. The program is compiled with the Esterel compiler SAXO-RT [35] that generates sequential C code. The execution platform is a monoprocessor hardware architecture [e.g., a digital signal processor (DSP) without a real-time operating system] where the Esterel program runs as a single task.

A. Overview of the Tool

The application software is composed of the Esterel program and a high-level description of the event handler. The latter determines the way inputs are taken into account by the program to generate an image of the state of the environment that is consistent with the synchronous semantics. The event handler is specified as a buffer characterized by its size and the list of separator input events. This information is used to determine which input events are consumed by the program at each reaction.

The real-time system consists of the compiled program and the event handler, which are composed with the environment that produces the inputs. Conceptually, the behavior of each component is modeled as a timed automaton [3], and the model of the overall system is obtained by an appropriate composition operator. To extract a timed model from the application software, we assume that time elapses only when the program is executing the data manipulation functions written in C. These functions are annotated with intervals defined by the best and the worst execution times, which can be estimated with existing techniques (e.g., profiling, static analysis, etc.) for analyzing software performance. The assumption that the C code that implements the control structure takes no time is not restrictive. It is straightforward to build a finer model that takes into account the execution time of the control part of the code.

The behavior of the environment is also given as an annotated Esterel program. Since synchronous programs are deterministic, Esterel has been extended with a nondeterministic choice statement (called *npause*) to model nondeterministic behavior of the environment [35]. This statement is not to be used for programming the application software. Clearly, having the same language for the program and the environment is not a fundamental issue. Nevertheless, it greatly simplified the development of the tool suite and enhanced its usability.

The program and the environment are compiled with SAXO-RT. The compiler has been re-engineered to use the annotations (pragmas) and the specification of the event handler to appropriately instrument the implementation (C code). The role of the instrumentation is to substitute the actual execution times of the C functions by the intervals provided by the annotations. This is done in such a way

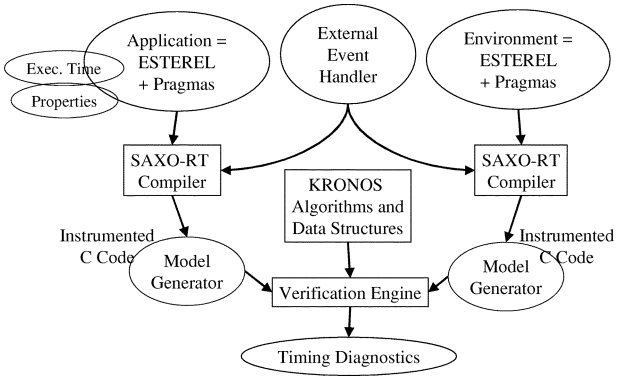

Fig. 4 TAXYS.

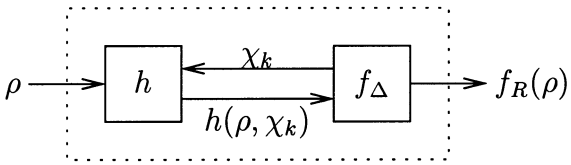

Fig. 5 TAXYS model of the real-time system.

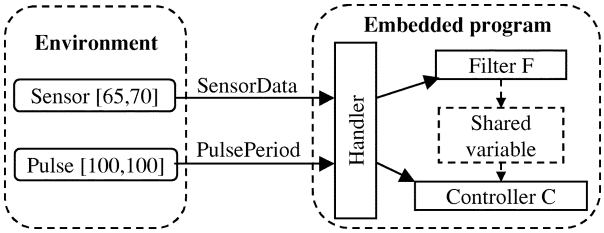

Fig. 6 Simple example.

that the execution of the instrumented C code generates the timed automaton model of the system. In other words, the instrumented implementation is an implicit representation of the timed model (as the implementation is an implicit representation of the synchronous model).

The instrumented implementation is linked with the KRONOS [14] timing verification library to obtain the on-the-fly model checker of the real-time system. This allows checking correctness and verifying timing requirements.

B. Checking Correctness

TAXYS checks correctness [see Condition (5)] by verifying a reachability property on the timed model of the real-time system. What follows is an explanation of how this is actually done.

Let f be the model of the Esterel program. We denote by f_Δ the function modeling the execution of the program according to the time constraints derived from the annotations associated with the C functions. f_Δ characterizes the timed automaton constructed by the compiler. Hence, for any $in \in \mathrm{In}$, and $\chi \in T_R$, we have that

$$f_\Delta(in, \chi) = (f(in), \chi') \tag{9}$$

where $\chi' \geq \chi$ is the time that the output $f(in)$ is produced when the input value in is provided at time χ (i.e., $\chi' - \chi$ is the execution time for in).

The model f_R of the real-time system is the result of the composition of the event handler h and of the function f_Δ as depicted in Fig. 5. We assume that $(\chi_k)_k$ is the time base defined by the response times of f_Δ. That is, if an input value is provided at time χ_k, then f_Δ will terminate the corresponding computation at time χ_{k+1}, which is also the start time of the next computation step. We assume that the event handler computes in zero time an output $h(\rho, \chi_k) \in \mathrm{In}$. Thus, the model f_R of the real-time system is such that, for any input $\rho \in E$

$$f_R(\rho) = (f_\Delta(h(\rho, \chi_k), \chi_k))_k = (f(h(\rho, \chi_k)), \chi_{k+1})_k. \tag{10}$$

In our case, the function ψ_{out} simply forgets real-time information, i.e.,

$$\psi_{\mathrm{out}}(f_R(\rho)) = (f(h(\rho, \chi_k)))_k. \tag{11}$$

Thus, the correctness condition, of (5) is reduced to

$$\forall \rho \in E.\, \forall k \in \mathbb{N}.\, f(h(\rho, \chi_k)) \subseteq f(\psi_{\mathrm{in}}(\rho)(k)). \tag{12}$$

Thus, it is sufficient to show that the event handler satisfies

$$\forall \rho \in E.\, \forall k \in \mathbb{N}.\, h(\rho, \chi_k) \subseteq \psi_{\mathrm{in}}(\rho)(k). \tag{13}$$

Let $(t_j)_j$ be the time base induced by $\psi_{\mathrm{in}}(\rho)$ and defined by the arrival times of the separator events. To respect the semantics of Esterel, the function ψ_{in} is such that

$$\psi_{\mathrm{in}}(\rho)(j) = \psi_{\mathrm{in}}(\rho(t_{j-1}, t_j]) = \bigcup_{t_{j-1} < \tau \leq t_j} \rho(\tau). \tag{14}$$

The event handler model records all the events occurred in the interval $(t_{j-1}, t_j]$. Notice that by definition, there is only one separator in any interval. The contents of the event handler are consumed at times χ_k. To respect the correctness condition (13), the event handler at time χ_k should contain all the events in the interval $(t_{k-1}, t_k]$. Thus, $t_k \leq \chi_k$. Furthermore, $\chi_k < t_{k+1}$ as the event handler should not contain two separators in the same interval.

The model of the event handler is a buffer with a particular error state. This state is reached when the environment generates a separator and there is already a separator in the buffer.

Hence, checking correctness amounts to verifying that the error state is not reachable.

C. A Simple Example

We illustrate the approach with a simple example depicted in Fig. 6. The program is composed of two parallel modules that control some physical device. The filter F is triggered by the input SensorData cyclically emitted by a sensor with a minimum delay of 65 ms and a maximum of 70 ms between each occurrence. F performs some computation using the data and stores the result in a shared variable (which is not modeled). This computation takes between 20 and 25 ms. The computed value is then used by the controller C to approximate the state of the device at that moment and apply the desired control. The controller C is periodically triggered

```
module Program:
procedure F()(), C()() ;
input SensorData, PulsePeriod ;
output Data, Control ;
loop
  await SensorData
      %{SD:=age(SensorData)}% ;
  call F()
      %{[20,25]}% ;
  emit Data
end loop
||
loop
  await PulsePeriod
      %{PP:=age(PulsePeriod)}% ;
  call C()
      %{[10,15], PP≤40 ∧ SD≤85}% ;
  emit Control ;
end loop
end module
```

```
module Environment:
output SensorData, PulsePeriod ;
loop
  npause
      %{65≤X≤70, X:=0}% ;
  emit SensorData
end loop
||
loop
  npause
      %{Y=100, Y:=0}% ;
  emit PulsePeriod ;
end loop
end module
```

Fig. 7 (Left) Program. (Right) Environment.

by the input PulsePeriod. Computing the control takes between 10 and 15 ms. Controls should be applied every 100 ms with an admissible delay of at most 40 ms. The value used to compute the control must be less than 85 ms old.

The Esterel code of the program is depicted in Fig. 7. The comments between %{ and }% are the annotations carrying the timing information. The execution times are given as intervals. Variables SD and PP are clocks as in the timed automata theory, that is, they are continuous variables that progress at equal rate. The assignment SD:=age(SensorData) sets SD to the time elapsed since the arrival in the buffer of the event SensorData consumed in the current reaction. The constraint PP≤40 ∧ SD≤85 expresses the requirements: the deadline for emitting the control is 40 ms and the data used to compute it is at most 85 ms old. The code of the environment is depicted in Fig. 7. Variables X and Y are clocks that are reset to 0 each time the environment emits SensorData and PulsePeriod, respectively.

The corresponding (extended) timed automata models are depicted in Fig. 8 and 9. Recall that the automata are not constructed explicitly. Their behavior is generated on the fly by the verification engine (see Fig. 4) by executing the instrumented C code produced by the compiler SAXO-RT. Dotted arrows in Fig. 8 and 9 represent eager transitions that must happen as soon as they become enabled. The "react" transition is the beginning of a reaction that starts as soon as there is an event in the buffer. Notice that, between the two possible orderings which are consistent with the semantics of Esterel, the sequential code generated by the compiler SAXO-RT schedules the filter F first when both events are simultaneously present. The event handler is shown in Fig. 10. The capacity of the buffer is one, and both events are specified to be separators. The clock t is used to record the time elapsed since the arrival of the last event or, equivalently, to measure the age of the event.

To verify the correctness of this example, TAXYS runs in less than a second, generates about 300 symbolic states and concludes that the real-time system is correct (i.e., the "error" state of the event handler is not reached) and that the timing requirements are satisfied. A symbolic state is composed of the state of the program (i.e., a valuation of the signals SensorData and PulsePeriod as present or not, and a control point), the state of the event handler (i.e., a configuration of the queue), the state of the environment (similar to the program), and a constraint on the clocks (i.e., a difference bounds matrix structure used by KRONOS.)

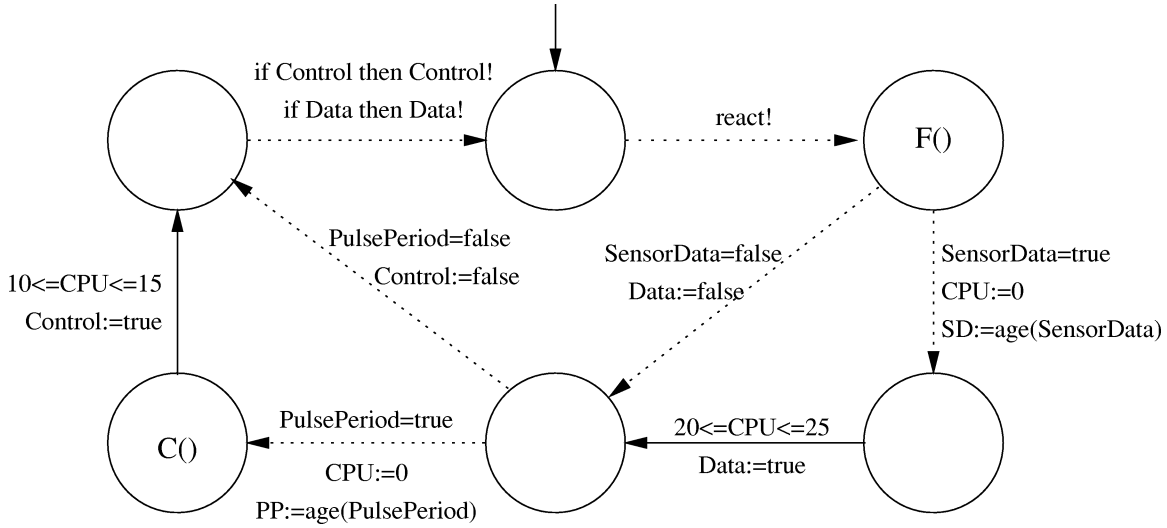

Fig. 8 Program's model.

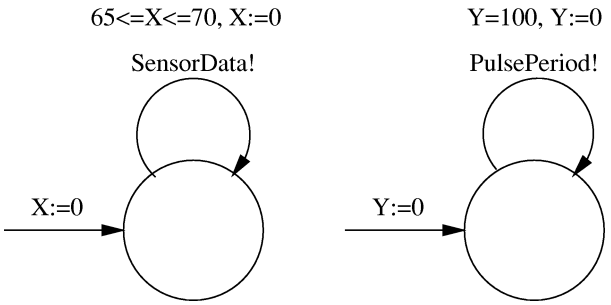

Fig. 9 Environment's model.

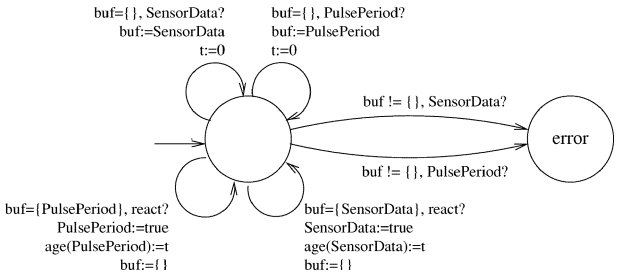

Fig. 10 Handler's model.

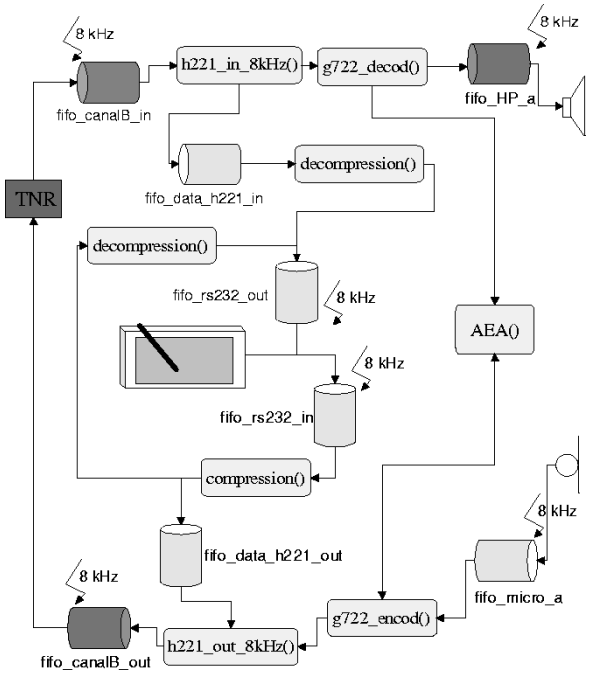

Fig. 11 ISDN prototype phone.

D. Experimental Results

TAXYS has been applied in several industrial case studies:

Radio Link of a GSM Terminal: This case study is reported in [7]. It consists in the programming and verification of the radio link of a global system for mobile communications (GSM) terminal developed by Alcatel. We describe here a small part of the application, which is composed of two modules. When the event Prepar arrives, the first module takes 50 ms to prepare the radio front end of the mobile terminal in order to receive data. Then, when the event Receipt arrives, it goes through a demodulation phase, that takes between 80 and 100 ms, followed by a decoding phase that finishes in 20 ms. The second module is triggered by the event Freq and calculates the frequencies on which the data are going to be received, and completes in 40 ms. The computations are subject to the time constraints annotated in the code. The code of the full application developed by Alcatel consists of 815 lines of Esterel and 48 000 lines of C. The application was validated for 62 test environments provided by Alcatel. Four scenarios were found to lead to deadline violations caused by a wrong scheduling of calls. These errors were corrected by slightly changing the Esterel code.

ISDN Prototype Phone: This case study is reported in [13]. It deals with a prototype phone carrying simultaneously voice and data produced by a graphic tablet, implemented on a 32 million instructions per second DSP. The prototype has an audio input channel sampled at 8 kHz that is connected to the microphone, an RS232 input channel carrying data from the graphic tablet, and an input channel sampled at 8 kHz to retrieve audio and graphic data sent by the network (TNR) (see Fig. 11). Processing audio data consumes 3900

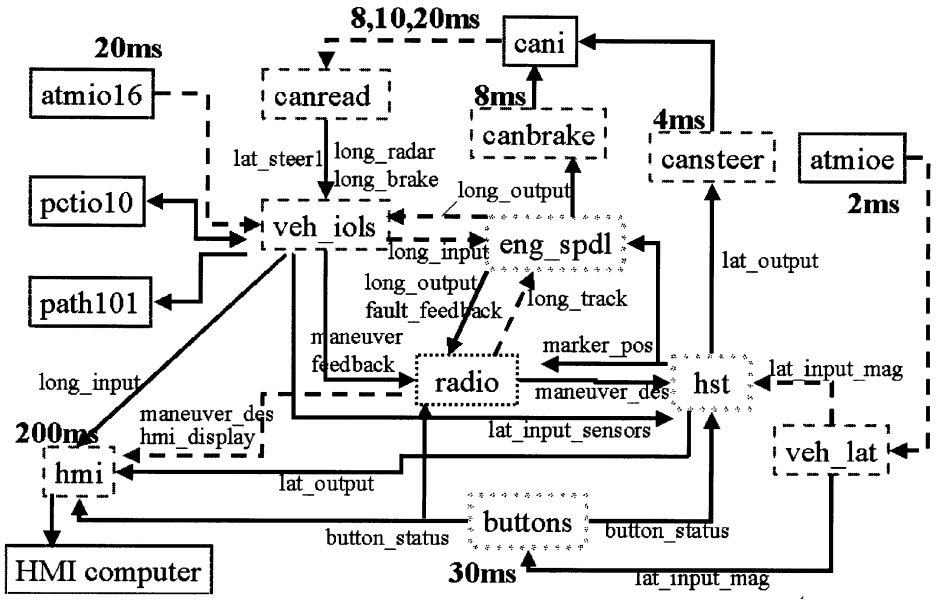

Fig. 12 Automated vehicle control software architecture.

Table 1
Experimental Results for the Prototype Phone.

Name	Buff. size	Symb. States	Verif. Time	Diagnostic
$\mathbf{ISDN}_1$	5	2 200	1.27 s	incorrect
$\mathbf{ISDN}_2$	6	10 849	5 s	OK
$\mathbf{ISDN}_3$	5	15 894	6.29 s	incorrect
$\mathbf{ISDN}_4$	6	633 472	10 mn 47 s	OK
$\mathbf{ISDN}_5$	5	22 695	13.6 s	incorrect
$\mathbf{ISDN}_6$	6	$> 10^7$	?	aborted

central processing unit (CPU) cycles over the 4000 CPU cycles available every 125 μs. Graphic data are compressed by a vectorization algorithm that consumes sporadically between 15 000 and 20 000 CPU cycles. The program consists of 258 lines of Esterel and 3000 lines of C code. We have used TAXYS to analyze the relationship between the size of the input buffers and the arrival rate of graphic data. We have analyzed the behavior of the system with three different environment models. For each one, we have experimented with buffers of sizes 5 and 6. In $\mathbf{ISDN}_1$ and $\mathbf{ISDN}_2$, the environment model is composed of two strictly periodic and independent tasks (the first carrying audio data at 8 kHz and the second the graphic tablet data at 100 Hz). In $\mathbf{ISDN}_3$ and $\mathbf{ISDN}_4$, the second task is aperiodic and emits bursts at rates varying nondeterministically between 25 and 100 Hz. In $\mathbf{ISDN}_5$ and $\mathbf{ISDN}_6$, there is an additional periodic task that models switching between several audio modes. In all cases, the 8-kHz periodic event is specified to be separator. The results presented in Table 1 show that the buffer needs to be of at least size 6 for the implementation to be correct. The current prototype was unable to handle the complexity of $\mathbf{ISDN}_6$.

Automated Vehicle Control Software: This case study is reported in [33]. It involves the software developed by the PATH Advanced Vehicle Control and Safety Systems project at the University of California at Berkeley [34]. This software is responsible for controlling a set of cars moving autonomously in a *platoon* formation (one car behind the other, with a small distance, e.g., 4–6 m, between them), on the highway and at high speed (e.g., 65 mi/h). The software consists of a set of processes running concurrently on a PC, reading data from various sensors (e.g., radar, speedometer, accelerometer, magnetometer), writing to actuators (throttle, brake, and steering), and using radio to communicate data to other vehicles. Fig. 12 shows the tasks and their interactions. Each arrow labeled with a variable name means that the originator of the arrow updates the variable, and the target of the arrow reads the variable. Periodic tasks are those labeled with a period in milliseconds. Event-driven tasks are those with dashed arrow pointing into them, labeled with the name of the variable the task sets a trigger for. The control part of the system has been reprogrammed in Esterel. Fig. 13 shows the Esterel code of two of the tasks. The size of the buffer is one, and `atmioeI` is the separator event. We have verified with TAXYS that the implementation is correct and that all periodic tasks complete their execution before the next period. TAXYS explored the entire reachable set (2022 symbolic states) in a less than a second.

V. Conclusion

We propose a modeling methodology for real-time systems. The methodology is based on the composition of timed models obtained by instrumenting software components used in the implementation. TAXYS is an application of the methodology to the simple case of synchronous real-time applications implemented as a single task. TAXYS combines three major advantages.

1) It is easy to use because the model of the real-time system is generated automatically by compilation of

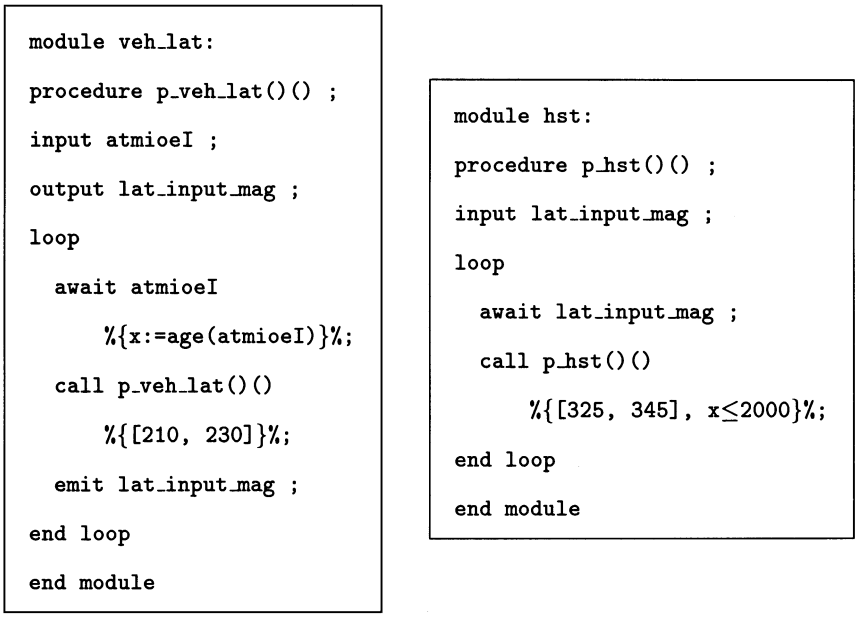

Fig. 13 Esterel program augmented with timing information.

annotated programs. The user has to learn only a minimal annotation language to express timing specifications.

2) The generated model faithfully represents the behavior of the real-time system. This is because the model of the latter is the code generated by the Esterel compiler, instrumented with statements that model the passage of time.
3) Possibility of analysis as the generated model is simple enough and is based on well-founded semantics.

The application of the general modeling methodology to other languages is more difficult and raises some nontrivial problems.

First, the modeling methodology is implicitly related to an implementation methodology for building the real-time system as a succession of steps involving the development of software components and their integration. The lack of a clearly defined implementation methodology is an obstacle to the application of the modeling methodology.

Second, when the application software is written in general purpose languages, such as C or Java, without built-in reactive execution semantics, model generation requires analysis to identify observable states and associated computation steps. The analysis task can be further hardened by features such as multiple threads and dynamic process creation.

Finally, a key issue for the application of the methodology is the use of adequate composition operators for modeling software consisting of heterogeneous components, such as synchronous and asynchronous, or event triggered and time triggered [24], [22]. Related to this is the problem of correctly adding time constraints to untimed models. "Correct" means, for instance, to avoid artifacts of the mathematical model (e.g., *zeno* behaviors) that do not correspond to real phenomena. Also, in order to have a modular modeling approach, it is desirable that time constraints are added in a compositional manner [30].

REFERENCES

[1] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, "A framework for scheduler synthesis," in Proc. 20th IEEE Real-Time Syst. Symp., 1999, pp. 154–163.

[2] K. Altisen, G. Goessler, and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *J. Real-Time Syst.*, vol. 23, pp. 55–84, 2002.

[3] R. Alur and D. L. Dill, "A theory of timed automata," *Theoret. Comput. Sci.*, vol. 126, pp. 183–235, 1994.

[4] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[5] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems," in *Proc. 40th IEEE Conf. Decision and Control*, vol. 3, 2001, pp. 2875–2880.

[7] V. Bertin, M. Poize, J. Pulou, and J. Sifakis, "Toward validated real-time software," presented at the 12th Euromicro Conf. Real-Time Systems, Stockholm, Sweden, June 2000.

[8] P. Binns and S. Vestal, "Scheduling and communication in metah," in *Proc. Real-Time Syst. Symp.*, 1993, pp. 194–200.

[9] ——, "Formalizing software architectures for embedded systems," in *Lecture Notes in Computer Science, Embedded Software*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2211, pp. 451–468.

[10] J. R. Burch, R. Passeronne, and A. Sangiovanni-Vincentelli, "Using multiple levels of abstractions in embedded software design," in *Lecture Notes in Computer Science, Embedded Software*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2211, pp. 324–343.

[11] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Reading, PA: Addison-Wesley, 2001.

[12] G. Buttazzo, *Hard Real-Time Computing Systems*. Norwell, MA: Kluwer, 1997.

[13] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys: A tool for the development and verification real-time embedded systems," in *Lecture Notes in Computer Science, Computer Aided Verification*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2102, pp. 391–395.

[14] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The tool Kronos," in *Lecture Notes in Computer Science, Hybrid Systems III: Verification and Control*. Heidelberg, Germany: Springer-Verlag, 1996, vol. 1066, pp. 208–219.

[15] "Response to the OMG RFP for Schedulability, Performance, and Time Revised Submission," Open Management Group, OMG doc. no. ad/2001-06-14, 2001.

[16] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with signal," *Proc. IEEE*, vol. 79, pp. 1321–1336, Sept. 1991.
[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proc. IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.
[18] M. G. Harbour, M. H. Klein, R. Obenza, B. Pollak, and T. Ralya, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Norwell, MA: Kluwer, 1993.
[19] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Lecture Notes in Computer Science, Embedded Software*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2211, pp. 166–184.
[20] "Specification and Description Language (SDL)," International Telecommunication Union–Standardization Sector, Genève, Z-100, 1999.
[21] "Real Time Core Extensions for the Java Platform," J-Consortium, Specification no. T1-00-01, 2000.
[22] H Kopetz, "The temporal specification of interfaces in distributed real-time systems," in *Lecture Notes in Computer Science, Embedded Software*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2211, pp. 223–236.
[23] H. Kopetz and G. Grunsteidl, "TTP—A protocol for fault-tolerant real-time systems," *Computer*, vol. 27, no. 1, pp. 14–23, Jan. 1994.
[24] E. Lee and A. Sangiovanni-Vincentelli, "A unified framework for comparing models of computation," *IEEE Trans. Computer-Aided Design*, vol. 17, pp. 1217–1229, Dec. 1998.
[25] E. Lee and Y. Xiong, "System-level types for component-based design," in *Lecture Notes in Computer Science, Embedded Software*, vol. 2211. Heidelberg, Germany, 2001, pp. 237–253.
[26] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.
[27] P. Puschner and A. Burns, "A review of WCET analysis," *Real Time Syst.*, vol. 18, no. 2/3, 2000.
[28] G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley, 2000.
[29] M. Saksena, P. Freedman, and P. Rodziewicz, "Guidelines for automated implementation of executable object oriented models for real-time embedded control systems," in *Proc. 18th IEEE Real-Time Syst. Symp.*, 1997, pp. 240–251.
[30] S. Bornot and J. Sifakis, "An algebraic framework for urgency," *Inform. Comput.*, vol. 163, pp. 172–202, 2000.
[31] J. Sifakis, "Modeling real-time systems—Challenges and work directions," in *Lecture Notes in Computer Science, Embedded Software*. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2211, pp. 373–389.
[32] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Norwell, MA: Kluwer, 1998.
[33] S. Tripakis and S. Yovine, "Timing analysis and code generation of vehicle control software using Taxys," in *Electronic Notes in Theoretical Computer Science, RV'2001 Runtime Verification*, vol. 55, 2001.
[34] P. Varaiya, "Smart cars on smart roads: Problems of control," *IEEE Trans. Automat. Contr.*, vol. 38, pp. 195–207, Feb. 1993.
[35] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou, "Efficient compilation of Esterel for real-time embedded systems," in *Int. Conf. Compilers, Architectures, and Synthesis for Embedded Syst.*, 2000, pp. 2–8.
[36] D. Wheeler, *ADA 95: The Lovelace Tutorial*. New York: Springer-Verlag, 1996.

Joseph Sifakis is CNRS Researcher and Director of the at Verimag laboratory, Grenoble, France. He worked on both theoretical and practical aspects of concurrent systems specification and verification. He contributed to the development of verification methods and tools by model checking for untimed and timed systems.

His current research work aims at developing methods for modeling real-time systems with emphasis on composability and compositionality techniques.

Stavros Tripakis (Associate Member, IEEE) received the B.Sc. degree in computer science from the University of Crete, Heraklion, Greece, in 1992, and the Ph.D. degree in formal verification and synthesis for timed automata from Verimag, Grenoble, France, in 1998.

He was a Postdoctoral Researcher at the University of California, Berkeley, in 1999 and 2000. He currently holds a CNRS Researcher position at Verimag. His research interests include formal methods, networks, and embedded systems.

Sergio Yovine graduated in computer science from Escuela Superior Latino Americana de Informática, Buenos Aires, Argentina, in 1989. He received the Ph.D. degree in computer science from Institut National Polytechnique de Grenoble, Grenoble, France, in 1993.

He was Research Engineer at the University of California, Berkeley, in 1997 and 1998. He currently holds a CNRS Researcher position at Verimag, Grenoble, France. He works on modeling and analysis of real-time and hybrid systems.