Modeling Real-Time Systems — Challenges and Work Directions

Joseph Sifakis
Joseph.Sifakis@imag.fr

Verimag, 2 rue Vignate, 38610 Gières, France

1 Introduction

1.1 Advanced Real-Time Systems

The evolution of information sciences and technologies is characterized by the extensive integration of embedded components in systems used in various application areas, from telecommunications to automotive, manufacturing, medical applications, e-commerce etc. In most cases, embedded components are real-time systems that continuously interact with other systems and the physical world. Integration and continuous interaction of software and hardware components makes the assurance of global quality a major issue in system design. The failure of a component may have catastrophic consequences on systems performance, security, safety, availability etc.

Building embedded real-time systems of guaranteed quality, in a cost-effective manner, raises challenging scientific and technological problems. Existing theory, techniques and technology are of little help as they fail to provide a global framework relating various design parameters to system dynamics and its properties. Contrary to conventional real-time systems, the development of advanced real-time systems, must take into account a variety of requirements about:

- Cost-effectiveness and time to market. These requirements are certainly the most important for advanced real-time systems usually embedded in mass market products. It is possible to improve quality by increasing costs and this has been often the case for conventional real-time applications. For example, the cost of the control equipment in a commercial aircraft is (still) a small percentage of the cost of the whole. On the contrary, for cellular phones even minimal optimizations of resources such as memory and energy or of time to market is of paramount importance.
- Fast evolving environments with rich dynamics e.g. in multimedia and telecommunication systems.
- Combination of hard and soft real-time activities which implies the possibility to apply dynamic scheduling policies respecting optimality criteria. Soft real-time is indeed harder than hard real-time as it requires that when necessary, some timing constraints are relaxed in some optimal manner, provided quality of service remains acceptable.

– Behavior which is dynamically adaptive, reconfigurable, reflexive, intelligent and "any fashionable buzzword used to qualify properties meaning that systems behave less stupidly than they actually do". Building systems meeting such properties is essential for quality assurance if we want to increase system interactivity and autonomy. Inventing new buzzwords does not help solving problems which are intrinsically hard. In fact, it is easy to understand that building systems enjoying such desirable properties amounts to designing controllers and thus, advanced controller design techniques for complex and heterogeneous systems are needed.
– Dependability covering in particular reliability, security, safety and availability. The dynamic nature and heterogeneity of advanced real-time systems makes most dependability evaluation techniques partial or obsolete.

Advanced real-time system developers lack theoretical and practical tools and enabling technology for dependable and affordable products and services. The emergence of such enabling technology requires tight and long term cooperation between researchers and practitioners. From a theoretical point of view, it raises foundational problems about systems modeling, design, analysis and control. The concept of control appears to be a key concept in advanced real-time systems engineering.

1.2 The role of modeling

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

– enhanced modifiability of the model and its parameters
– ease of construction by integration of models of heterogeneous components,
– generality by using genericity, abstraction, behavioral non determinism
– enhanced observability and controllability especially, avoidance of probe effect and of disturbances due to experimentation
– finally, possibility of analysis and predictability by application of formal methods.

Building models which faithfully represent complex systems is a non trivial problem and a prerequisite to the application of formal analysis techniques. Usually, modeling techniques are applied at early phases of system development and at high abstraction level. Nevertheless, the need of a unified view of the various lifecycle activities and of their interdependencies, motivated recently, the so called model-based approaches [Ves97,BALS99,HHK01,L$^+$01] which heavily rely on the use of modeling methods and tools to provide support and guidance for system development and validation. Modeling systems in the large is an important trend in software and systems engineering today.

Currently, validation of real-time systems is done by experimentation and measurement on specific platforms in order to adjust design parameters and hopefully achieve conformity to QoS requirements. The existence of modeling techniques for real-time systems is a basis for rigorous design and should drastically ease their validation.

The paper unifies results developed at Verimag over the past four years into a methodological framework for modeling advanced real-time systems. Most of the ideas are applicable to arbitrary systems modeling. Current trends in advanced real-time systems foreshadow trends in general systems as integration and interactivity increase.

We consider modeling as an activity integrated in the system development process, strongly related to design and synthesis. Based on this view, we identify some challenging problems and a related research agenda. *A central thesis is that a dynamic model of a real-time system can be obtained by adequately restricting the behavior of its application software with timing information.* We present a composition/decomposition methodology and experimental results illustrating the thesis. The methodology raises some very basic and challenging problems about relating functional to non functional (time dependent) aspects of the behavior by application of composition techniques. Its application requires building models in the large by composition of software components and is illustrated by results obtained within the Taxys project.

2 Challenges and open problems

We identify main challenging problems and work directions for their solution.

2.1 The divide between application software and real-time system

It is generally agreed that a main obstacle to the application of rigorous development techniques is the lack of methodology for relating application software and functional design to physical architecture and implementation.

At functional design level, a system is specified as a set of interacting components. The functional architecture adopts a decomposition which may be different from the one adopted by physical architecture. High level real-time languages such as ADA, Esterel and SDL, very often use simplifying assumptions about components behavior and their cooperation e.g. instantaneous and perfect communication of components, synchrony of interaction with the external environment or atomicity of actions. These are very useful abstractions that drastically simplify description. The lack of associated methods for correct implementation of the high level constructs and concepts may be a serious limitation for the effective use of high level languages.

The deep distinction between real-time application software and a corresponding real-time system resides in the fact that the former is immaterial and thus untimed. Its real-time properties depend on the speed of the underlying platform and the interaction with external environment. By the operational semantics of the language, it represents a reactive machine triggered by external stimuli. Even if there are actions depending on time e.g. timeouts, time is external and provided by the execution platform. Expiration of a timeout is an event that is treated in the same manner as any external event such as hitting an obstacle. The study of a real time system requires the use of timed models, that can describe the combined effect of both actions of the application software and time progress.

The transition from application software to implementation involves steps and choices that determine the dynamic behavior of the real-time system. These steps consist in

- partitioning the application software into parallel tasks or threads and mapping them to the physical architecture.
- ensuring resource management and task synchronization by means of synchronization primitives offered by the underlying platform
- finding adequate scheduling policies so that given quality of service requirements (non functional properties) are met; this requires in principle, taking into account the dynamics of the execution platform e.g. WCET for atomic actions and the dynamics of the external environment.

There exist today reasonably good methods, tools and technology for supporting functional system design and application software development activities. Nevertheless, the rigorous transition to implementation comes up against several problems, such as

- relating properties of the functional design with properties of the implementation e.g. it is desirable that functional properties are preserved by the implementation methods
- modeling the dynamics of an implementation for simulation, analysis and validation purposes
- evaluating the influence of design choices on non functional properties.

2.2 Synchronous vs. Asynchronous real-time

Current practice in real-time systems design follows two well-established paradigms.

The synchronous paradigm based on a synchronous execution model 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 computes its reaction to environment stimuli by propagating their effects through its components. The synchrony assumption says that system reaction is fast enough with respect to its external environment. This practically 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 and languages such as Esterel [BG92], Lustre [HCRP91], and Signal [BLJ91] adopt the synchronous paradigm. For correct implementation of these languages care should be taken to meet the synchrony assumption by guaranteeing not only termination of the actions performed in a step but also that their execution times have known upper bounds.

Synchronous programs are implemented by adopting scheduling policies that guarantee that within a step all the tasks make some progress even though high priority tasks get a larger share of CPU. They are used in signal processing, multimedia and automatic control. Safety critical applications are often synchronous and are implemented as single sequential tasks on single processors. Synchronous language compilers generate sequential code by adopting very simple scheduling principles.

The asynchronous paradigm does not impose any notion of global computation step in program or system execution. Asynchronous real-time is a still unexplored area, especially for distributed computing systems that are inherently dynamic and must adapt to accommodate workload changes and to counter uncertainties in the system and its environment.

A simple instance of the asynchronous paradigm is the one developed around the ADA 95 [Whe96] language and associated implementation techniques, essentially RMA techniques [HKO$^+$93,GKL91]. The latter provide a collection of quantitative methods that allow to estimate response times of a system composed of periodic or sporadic tasks with fixed priorities.

In asynchronous implementations, tasks are granted resources according to criteria based on the use of priorities. This is usually implemented by using RTOS and schedulers.

There exists a basic difference between the two paradigms. The synchronous one guarantees, due to the existence of global computation steps, that "everybody gets something". The asynchronous paradigm applies the principle that the "winner takes all", the winner being elected by application of scheduling criteria. This implies that in asynchronous implementations, a high priority external stimulus can be taken into account "as soon as possible" while in synchronous implementations reaction time is bounded by the execution time of a step. Nevertheless, "immediate" reaction to high priority external stimuli does not imply satisfaction of global real-time properties. Existing theory based on the use of analytic models, allows to guarantee simple real-time properties, typically meeting deadlines and is applicable only to simple task arrival models [LL73].

For advanced real-time applications it is desirable to combine the synchronous and asynchronous paradigm at both description and implementation levels. 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 implementation to cope with inherent limitations of each paradigm. For instance, for synchronous languages it is possible to get more efficient implementations that respect the abstract semantics, by scheduling components execution within a step and thus making the system sensitive to environment state changes within a step. Furthermore, for synchronous software it is possible to relax synchrony at implementation level by mapping components solicited at different rates to different non pre-emptible tasks coordinated by a runtime system.

Proposals of real-time versions of object-based languages such as Java [Gro00] and UML [Gro01], provide concepts and constructs allowing to mix the two paradigms and even to go beyond the distinction synchronous/asynchronous. In principle, it is possible to associate with objects general scheduling constraints to be met when they are executed. 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.

3 Modeling real-time systems

3.1 Component-based modeling

Definition: The purpose of modeling is to build models of software and systems which satisfy given requirements. We assume that models are built by composing components which are model units (building blocks) fully characterized by their interface. We use the notation $||$ to denote an arbitrary composition operation including simple composition operations a la CCS [Mil89] or CSP [Hoa85], protocols or any kind of "glue" used in an integration process: $C1||C2$ represents a system composed of two components $C1$ and $C2$. We assume that the meaning of $||$ can be defined by operational semantics rules determining the behavior of the composite system from the behavior of the components.

The modeling problem: Given a component C and a property P find a composition operation $||$ and a component C' such that the system $C||C'$ satisfies P.

Notice that by this definition, we consider that modeling does not essentially differ from design or synthesis. In practice, the property P is very often implicit as it may express the conjunction of all the requirements to be met by the composite system. The above definition provides a basis for hierarchical or incremental modeling as composite systems can themselves be considered as components. Thus, complex systems can be obtained by iterative application.

Incremental modeling: To cope with complexity, it is desirable that the model of a complex system is obtained incrementally by further restricting some initial model. This can be done in two manners:

- By integration of components, that is building a system $C1||C2\ldots||Cn$ by adding to $C1$ interacting components $C2\ldots Cn$ so that the obtained system satisfies a given property. We want, throughout the integration process, already established properties of components to be preserved. *Composability*, means that if a property P holds for a component C then this property holds in systems of the form $C||C'$ obtained by further integration. For example, if C is deadlock-free then it remains deadlock-free by integration. Composability is essential for building models which are by construction correct. Unfortunately, time dependent properties are non composable, in general [AGS00,AGS01,Lee00].

– By refinement, that is by getting from an abstract description C a more concrete (restricted) one C' in the sense that the behaviors of C' are "contained" in the behaviors of C. Refinement relations are captured as simulation relations modulo some observability criterion establishing a correspondence between concrete and abstract states and/or actions. We want, throughout the modeling process, refinement to be preserved by composition operators. That is, if components of a system are replaced by components refining them, then the obtained system is a refinement of the initial system. This property called *refinement compositionality*, is essential for relating models at different abstraction levels. Refinement compositionality should play an important role in relating application software to its implementations. Existing results are not satisfactory in that they deal only with preservation of safety properties. The application of constructive techniques requires stronger results e.g. preservation of progress properties.

3.2 About Timed Models

A real-time system is a layered system consisting of the application software implemented as a set of interacting tasks, and the underlying execution platform. It continuously interacts with an external environment to provide a service satisfying requirements, usually called QoS requirements. The requirements characterize essential properties of the dynamics of the interaction.

Models of real-time systems should represent faithfully the system's interactive behavior taking into account relevant implementation choices related to resource management and scheduling as well as execution speed of the underlying hardware. They are timed models as they represent the dynamics of the interaction not only in terms of actions but also in terms of time. Building such models is clearly a non trivial problem.

Timed models can be defined as extensions of untimed models by adding time variables which are state variables used to measure the time elapsed. They can be represented as machines that perform two kinds of state changes: transitions and time steps. Transitions are timeless state changes that represent the effect of actions of the untimed system; their execution may depend on and modify time variables. Time steps represent time progress and increase uniformly only time variables. They are specified by time progress conditions [BS00]: time can progress from a state by t if the time progress condition remains true in all intermediate states reached by the system. During time steps state components of the untimed model remain unchanged. There exists a variety of timed formalisms extensions of Petri nets [Sif77], process algebras [NS91] and automata [AD94]. Any executable untimed description e.g. application software, can be extended into a timed one by adding explicitly time variables or other timing constraints about action execution times, deadlines etc.

Timed models use a notion of logical time. Contrary to physical time, logical time progress can block especially as a result of inconsistency of timing constraints. The behavior of a timed model is characterized by the set of its runs, that is the set of maximal sequences of consecutive states reached by performing transitions or time steps. The time elapsed between two states of a run is computed by summing up the durations of all the time steps between them. For a timed model to represent a system, it is necessary that it is well-timed in the sense that in all runs time diverges.

As a rule, in timed models there exist states from which time cannot progress. If time can progress from any state of a timed model, then it is always possible to wait and postpone the execution of actions which means that it is not possible to model action urgency. Such models represent degenerated timed systems which are in fact untimed systems as time can be uniformly abstracted away. Action urgency at a state is modeled by disallowing time progress. This possibility of stopping time progress goes against our intuition about physical time and constitutes a basic difference between the notions of physical and logical time. It has deep consequences on timed systems modeling by composition of timed components.

Composition of timed models can be defined as extensions of untimed composition. They compose actions exactly as untimed composition. Furthermore, for time steps, a synchronous composition rule is applied as a direct consequence of the assumption about a global notion of time. For a time step of duration t to occur in a timed system, all its components should allow time progress by t. Well-timedness is not composable in general, especially when components have urgent synchronization actions.

3.3 Building the timed model

We present a methodology for building timed models of real-time systems as layered descriptions composed of

- Models of the tasks
- A synchronization layer
- A scheduler that controls execution so as to meet QoS requirements.

Timed models of tasks The application of the methodology requires compilation and WCET analysis tools. We discuss the principle of construction of the timed model without addressing efficiency or even effectiveness issues. We assume that code for tasks (execution units) is obtained by partitioning the application software and separate compilation. Furthermore, we assume that for a given implementation platform and by using analysis tools, the following information can be obtained about each task:

- Which sequences of statements are atomic during execution
- For atomic sequences of statements, estimates of execution times e.g. WCET or interval timing constraints with lower and upper bounds.

The code of a task C with this information constitutes its timed model C_T. The latter can perform the actions of C - which are atomic sequences of statements of C - and time steps.

Synchronization of timed tasks We assume that task synchronization is described in terms of synchronization primitives of the underlying platform such as semaphores and monitors, to resolve task cooperation and resource management issues. In principle, untimed tasks code with their synchronization constraints is a refinement of the application software.

We want to get compositionally the timed model corresponding to the system of synchronized tasks. For this, it is necessary to extend the meaning of synchronization operations on timed behavior. For example, if $C1$ and $C2$ is the code of two tasks and $C||C2$ represents the system after application of synchronization constraints, then we need a timed extension $||_T$ of $||$ to compose the timed models $C1_T$ and $C2_T$ of $C1$ and $C2$. $C1_T||_TC2_T$ is the timed model of $C1||C2$. In [BS00], it has been shown that untimed composition operators $||$ admit various timed extensions $||_T$ depending on the way urgency constraints (deadlines) of timed tasks are composed.

Extending untimed composition operators to timed ones is an important problem for the construction of the global timed model. It is desirable that essential properties of the timed components such as deadlock-freedom and well-timedness are preserved by timed composition. We have shown that usual composition operators for timed models do not preserve well-timedness [BGS00,BS00]. The existence of methodology and theory relating properties of untimed application software to the associated timed model is instrumental for getting correct implementations. Usually, correct implementation of the operator $||$ used in the software development language, requires a fairly complex definition of $||_T$. The reasons for this are twofold.

1. Whenever the interaction between untimed components is supposed to be instantaneous, the application of the same composition principle to timed components may lead to inconsistency such as deadlocks or timelocks. The obvious reason for this is that in the timed model (the implementation) component reactions to external stimuli do take time. When a component computes a reaction, its environment state changes must be recorded to be taken into account later. So, contrary to instantaneous untimed composition, timed composition needs memory to store external environment events.
2. In many composition operations for untimed systems, independent actions of components may interleave. Interleaving implicitly allows indefinite waiting of a component before achieving synchronization. It may happen that an untimed system is deadlock-free and the corresponding timed system has deadlocks. Adding timing constraints may restrict waiting times and destroy smooth cooperation between components (see example in [BST98,BS00]).

Composition of timed components should preserve the main functional properties of the corresponding untimed system. Otherwise, most of the benefit from formal verification of functional properties is lost. To our knowledge, this is a problem not thoroughly investigated and which cannot be solved by directly transposing usual untimed composition concepts. We need results for correct implementation of untimed interaction primitives relying on the concepts of protocol or architecture.

Scheduler modeling The real-time system model must be "closed" by a timed model of the external environment. We assume that this model characterizes the expected QoS from the system. That is, instead of using extra notation such as temporal logic or diagrams to express the QoS requirements, we represent them as a timed model. It can be shown that their satisfaction boils down as for untimed systems, to absence of deadlock in the product system obtained by composition of the environment and the real-time system models. This method applied to verify safety properties for untimed systems can be used to verify also liveness properties if the timed models are well-timed and this property is preserved by parallel composition.

Scheduler modeling is a challenging problem. Schedulers coordinate the execution of system activities so that requirements about their timed behavior are met, especially QoS requirements. We have shown that schedulers can be considered as controllers of the system model composed of its timed tasks with their synchronization and of a timed model of the external environment [AGS00,AGS01]. They apply execution strategies which satisfy, on the one hand timing constraints resulting from the underlying implementation, essentially execution times, and on the other hand QoS requirements represented by a timed model of the environment. Under some conditions, correct scheduling can be reduced to a deadlock avoidance problem.

More concretely, a scheduler monitors the state of the timed model and selects among pending requests for using common resources. The role of scheduler can be specified by requirements expressed as a constraint K, set of timed states of the scheduled system. The scheduler keeps the system states in a control invariant K' subset of K, set of states from which the constraint K can be maintained in spite of "disturbances" of the environment and of internal actions of the tasks. The existence of a scheduler maintaining K depends on the existence of non empty control invariants contained in K [MPS95,AGP$^+$99]. Control invariants can be characterized as fixpoints of monotonic functions (predicate transformers) representing the transition relation of the timed system to be scheduled. There exists a scheduler maintaining K iff there exist non empty fixpoints implying K. Computing such fixpoints is a non trivial problem of prohibitive complexity when it is decidable - essentially, for scheduling without preemption. This relation of scheduler modeling to controller synthesis explains the inherent difficulties in the application of model-based techniques to scheduling.

A methodology based on composability results has been studied in [AGS01] to circumvent the difficulties in scheduler modeling. It consists in decomposing the constraint K into a conjunction of two constraints $K = K_{sched} \wedge K_{pol}$. K_{sched} specifies the fact that timing requirements are satisfied. K_{pol} specifies scheduling policies for resource allocation and can be expressed as the conjunction of resource allocation constraints K^r, $K_{pol} = \bigwedge_{r \in R} K^r$, where R is the set of the shared resources. The constraint K^r can again be decomposed into two types of constraints $K^r = K^r_{resolve} \wedge K^r_{admit}$. $K^r_{resolve}$ specifies how conflicts for the acquisition of r are resolved while K^r_{admit} characterizes admission control policies and says when a request for r is taken into account by the scheduler.

According to results presented in [AGS01] such a decomposition allows to simplify the modeling problem of a scheduler maintaining $K_{sched} \wedge K_{pol}$ by proceeding in two steps. First, getting a scheduler that maintains K_{pol}; this does not require the application of synthesis algorithms. The second step aims at finding by restriction of this scheduler, a scheduler which maintains K_{sched} and requires in general, the application of synthesis or verification techniques.

The construction of the scheduler maintaining K_{sched} heavily relies on the use of **dynamic priority rules**. It uses composability results allowing to obtain incrementally the scheduler. A priority rule is a pair consisting of a condition and of an order relation on task actions. When the condition is true, the associated priority order is applied and its effect is to prevent actions of low priority to occur when actions of higher priority are enabled. It has been shown that non trivial scheduling policies and algorithms can be modeled, by using priority rules. Furthermore, there exist composabilty results that guarantee preservation of absence of deadlock and in some cases of liveness, by application of dynamic priority rules [BGS00].

Interesting research directions in scheduler modeling are:

– Composability: An important observation is that scheduling methods are not composable, in the sense that if a scheduler maintains $K1$ and a scheduler maintains $K2$ then even if they control access to disjoint sets of resources, their application does not in general maintain $K1 \wedge K2$. In [AGS00] a sufficient condition for scheduler composability is given. There is a lot to be done in that direction.
– Connections to the controller synthesis paradigm: This is a very interesting research direction, especially because controller synthesis provides a general theoretical framework that allows better understanding the scheduling problem and the inherent difficulties. Controller synthesis provides also the general framework for tackling more pragmatic and methodological questions.
– Combining model-based scheduling and scheduling theory: Scheduling theory proposes sufficient conditions for schedulability of simple tasks, usually periodic or sporadic, for specific scheduling policies. It is based on the use of analytic models that lead to more tractable analysis techniques at the price of approximation [ABR91,HKO$^+$93,EZR$^+$99]. Some combination and unification of the two approaches seems feasible and should give results of highest interest.

3.4 Application to modeling embedded telecommunication systems

The Taxys project, in collaboration with France Telecom and Alcatel Business Systems uses the principle given in 3.3 to build timed models of real-time systems. The purpose of this project is the analysis of embedded real-time applications used in cellular phones, to check QoS requirements. The latter are expressed as properties of the form “the time distance between an input signal and the associated reaction is bounded by some delay” or “the time distance between two consecutive output signals is within a given interval”.

The application software is written in Esterel extended with C functions and data. The design environment comprises a compiler used to generate C code from application software and a parametric code generator used to generate executable code for DSP's. The implementation consists of one task and does not use OS or scheduler. Scheduling issues are resolved by the compiler which determines causally correct orders of execution between processes.

The C code is structured as a reactive machine whose transitions are triggered by signals provided by the external environment. A transition occurrence changes atomically control state and data state by executing an associated C function. To build a timed model of the application, the C code is instrumented by adding timing constraints about execution times of C functions. These constraints are lower and upper bounds of execution times estimated by using an execution time analysis tool. The global timed model of the embedded real-time system is obtained by composition of the timed model of the C code with a timed model of the environment and an external events handler. The latter is used to store signals emitted by the environment. It is important to notice that while according to Esterel semantics, the interaction between an Esterel program and its environment is instantaneous, the composition between the timed application software and the timed environment requires the use of memory. In fact, instantaneous actions of the untimed model take time when executed and during this time the environment changes must be registered to be taken into account in the next application cycle. This is an instance of the general problem evoked in 3.3. Global timed models of the real-time system, are analyzed by using the Kronos [DOTY96,Yov97,BSS97] timing analysis tool.

The Esterel compiler has been engineered to generate the instrumented C code from pragmas describing timing constraints. Thus, the timed models of both the application software and of the environment can be described in Esterel appropriately extended with pragmas to express timing constraints. The obtained results are very encouraging [BPS00,CPP$^+$01,TY01b,TY01a] and we plan to apply this principle to modeling real-time systems with multitasking.

4 Discussion

Modeling is a central activity in systems development intimately related to design, programming and specification. We consider only modeling by using executable languages. Although we do not deny the interest of using notations based on logic and declarative languages, we do not consider them as essential in control dominated modeling. Some notation encompassing abstraction such as the various kinds of charts (message charts, sequence charts) is certainly useful for requirements expression. Nevertheless, we consider that the main challenges concern modeling by using executable languages at different levels from specification to implementation and for different execution models.

We believe that an important trend is relating implementation to models at different abstraction levels. For this, it seems inevitable that models are built by using or rather reusing components written in languages such as C and C++. We need modeling environments which support methodologies and tools for building models from heterogeneous components, synchronous and asynchronous. The use of C and C++ code in such environments requires methodology and tools for monitoring and instrumenting code execution, in particular to change abstraction, granularity of execution and execution model. Modeling systems in the large is an important and practically relevant work direction.

4.1 For a common conceptual basis

Building general modeling environments requires a common conceptual basis for component based modeling, a unifying notation with clean and natural operational semantics. Such a notation should distinguish 3 basic and complementary ingredients in models: components, interaction model and execution model. A model is a layered description consisting of a set of components, on which are applied successively an interaction model and an execution model. Interaction and execution models describe two complementary aspects of an architecture model.

Components are system units characterized by their interface specified as a set of interaction points. An interaction point corresponds to a shared action (event-based approach) or a shared variable (state-based approach). An interaction point can be conceived as a pair consisting of a name (port, signal, input, output) and a list parameters.

A system is modeled as a set of interacting components. A component state determines which interaction points can participate in an interaction and the values of their parameters. For given component states, an interaction between a set of components can be specified as a subset of compatible interacting points. The compatibility relation may be statically determined or may depend on the states of the components involved in the interaction. Usually, we distinguish between the initiator of the interaction and the other participating components as in some interaction models it is necessary to take into account the flow of causality. Interactions may be n-ary, in general. For general modeling languages there is no reason for restricting to interactions of particular arity.

The interaction model describes the effect of interactions on the states of the participating components. We need languages encompassing the following different interaction types.

- **blocking and non blocking interaction:** Blocking interactions are interactions that cannot take place unless all the involved interaction points are enabled for instance, synchronous message passing in CSP and CCS. Non blocking interactions can take place when the initiator of the interaction is enabled and involve all the enabled participants. Synchronous languages and hardware description languages use non blocking interaction.
- **atomic and non atomic interaction:** Atomic interaction means that interaction is completed without interference of other system events. Clearly, non atomic interaction can be modeled in terms of atomic interaction by using memory such as shared variables and fifo queues. It is desirable that

modeling languages offer primitives for the direct description of non atomic interaction mechanisms.

Some papers distinguish also between event-based and state-based interaction. We do not consider this distinction to be foundationally relevant although in practice it can be convenient to use rather one than the other type of interaction.

The execution model contains information about component concurrency control in a given architecture. It further constrains the behavior obtained by application of an interaction model by reducing non determinism inherent to interactive behavior. Its effect can be represented by adding extra components that restrict the behavior of the interacting components. For methodological reasons, it is very important to separate component interaction from execution control. Component interaction deals with satisfaction of functional properties, essentially safety properties, such as mutual exclusion. The execution model deals with component progress and sharing computing power. It determines the computation threads and is strongly related to fairness, scheduling, run to completion, time step properties. Usually, in existing languages, the execution model is strongly related to their interaction model: atomic interaction for synchronous languages and non atomic interaction for asynchronous languages. Nevertheless, it is desirable to dissociate interaction from execution aspects. An important trend in proposals for real-time versions of object-based languages, is associating with objects priorities and scheduling constraints.

4.2 Relating functional to non functional models

Relating untimed to timed, functional to non functional models by using as much as possible composability and compositionality is a very important work direction. Existing theory on system modeling and composition such as process algebra, has been developed on low level models e.g. transitions systems, and badly resists to extensions with data and time. For programs, we have compositionality results for simple sequential programs e.g. Hoare logic whose extension to concurrent systems leads to complicated methodologies. The concepts of composition studied so far, consider essentially composition as intersection of behaviors. They lend themselves to theoretical treatment but are not appropriate for component-based modeling. For the latter, we need composition concepts that modulo some assumptions about component properties, guarantee properties of the whole system. Results using assume/guarantee rules seem difficult to apply. We believe that for component-based modeling, we need "flexible" composition rules that compose the components behavior by preserving essential component properties such as deadlock-freedom. Contrary to usual composition, flexible composition operators seek consensus (avoid "clashes") between interacting components. A notion of flexible composition for timed systems have been studied in [BS00]. The basic idea is not to compose separately timing requirements and actions which may easily lead to inconsistency: a deadlock is reached if a component offers an urgent synchronization action that cannot be accepted by its environment without letting time pass. Instead, flexible composition adopts a "non Newtonian" or "relativistic" view of time by assuming that time is "bent by the action space". To avoid deadlock in the previous example, time is allowed to pass as long as no action is enabled. This may lead to violation of the individual timing constraints of components but guarantees well-timedness of the composed system.

Existing implementation theories do not preserve properties such as deadlock-freedom of components, or more general progress properties, especially when going from functional to non functional (timed) models. Usually, for timed models obtained by adding timing requirements to an untimed model, component deadlock-freedom is not preserved in either way. Timing can introduce deadlocks to deadlock-free untimed systems and deadlocks of untimed systems can be removed by adding timing constraints. In absence of any theoretical results about preservation of properties other than safety properties, it is questionable whether it is worthwhile verifying at functional model level properties which are not preserved by implementation techniques such as deadlock-freedom, liveness and fairness properties. We badly lack implementation theories for high level languages.

4.3 Scheduler and controller modeling

The distinction between interaction and execution models determines two kinds of modeling problems of uneven difficulty. Modeling interaction seems to be an easier and in any case a better understood problem than modeling execution. We consider the latter to be a still unexplored problem, especially in the case of timed models. To describe execution models, we need languages with powerful constructs allowing compositional description of dynamic priorities, preemption and urgency. The distinction between synchronous and asynchronous execution should be captured by using execution models.

Schedulers are components that implement execution models. Their modeling deserves special attention as their role is crucial for correct implementation. Scheduler modeling can be considered as an instance of a more general problem, that of finding controllers which maintain given constraints. The development of design and modeling methodologies for controllers leads to explore connections to control theory. This is certainly a very interesting research direction confirmed by trends in some areas such as multimedia where control algorithms are used to solve traffic management problems. Connection to controller synthesis problems may also bring solutions to all the hard and still not very well understood problems of building systems enjoying "magic" properties such as adaptivity, reflectivity, survivability, and why not, intelligence.

Acknowledgment: I would like to thank my colleagues of Verimag and J. Pulou of France Télécom R&D for constructive comments and criticism.

References

[ABR91] N.C. Audsley, A. Burns, and M.F. Richardson. Hard real-time scheduling: the deadline monotonic approach. In *Workshop on real-time operating systems and software*, 1991.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AGP$^+$99] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. RTSS 1999*, pages 154–163. IEEE Computer Society Press, 1999.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In M. Joseph, editor, *proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[AGS01] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing" (to appear)*, 2001.

[BALS99] H. Ben-Abdallah, I. Lee, and O. Sokolsky. Specification and analysis of real-time systems with Paragon. In *Annals of Software Engineering*, 1999.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGS00] S. Bornot, G. Gößler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.

[BLJ91] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BPS00] V. Bertin, M. Poize, and J. Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference on Real Time Systems*, pages 157–164, 2000.

[BS00] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.

[BSS97] A. Bouajjani, S.Tripakis, and S.Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. IEEE Real-Time Systems Symposium, RTSS'97*. IEEE Computer Society Press, 1997.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Proc. COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[CPP$^+$01] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS = ESTEREL + KRONOS. a tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer-Verlag, 2001.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[EZR$^+$99] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/software codesign of embedded systems — the SPI workbench. In *Proc. Int. Workshop on VLSI, Orlando, Florida*, 1999.

[GKL91] M. Gonzáles, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priorities. In *Proc. RTSS 1991*, 1991.

[Gro00] Real-Time Java Working Group. International j consortium specification. Technical report, International J Consortium, 2000.

[Gro01] OMG Working Group. Response to the omg rfp for schedulability, performance, and time. Technical Report ad/2001-06-14, OMG, June 2001.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHK01] T. A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Embedded control systems development with Giotto. In *Proc. LCTES 2001 (to appear)*, 2001.

[HKO$^+$93] M.G. Härbour, M.H. Klein, R. Obenza, B. Pollak, and T. Ralya. *A Practitioner's Handbook for Real-Time Analysis.* Kluwer, 1993.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[L$^+$01] E. A. Lee et al. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, University of California at Berkeley, 2001.

[Lee00] E.A. Lee. What's ahead for embedded software. *Computer, IEEE*, pages 18–25, September 2000.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS'95*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. CAV'91*, volume 575 of *LNCS*. Springer-Verlag, July 1991.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In *Proc. 3rd Intl. Symposium on Modeling and Evaluation*, pages 75–93. IFIP, North Holland, 1977.

[TY01a] S. Tripakis and S. Yovine. Modeling, timing analysis and code generation of PATH's automated vehicle control application software. In *Proc. CDC'01 (to appear)*, 2001.

[TY01b] S. Tripakis and S. Yovine. Timing analysis and code generation of vehicle control software using taxys. In *Proc. Workshop on Runtime Verification, RV'01*, volume 55, Paris, France, July 2001. Elsevier.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 11–21, 1997.

[Whe96] D. Wheeler. *ADA 95. The Lovelace Tutorial.* Springer-Verlag, 1996.

[Yov97] S. Yovine. KRONOS: A verification tool for real-time systems. *Software Tools for Technology Tranfer*, 1(1+2):123–133, 1997.