Real-Time Constraints

Pao-Ann Hsiung[†]

Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC

E-mail: hpa@computer.org

Abstract

Real-time constraints are restrictions on the timings of events, such that they occur *on-time*. A system with real-time constraints is called a *real-time system*. Not merely the performance of such systems, but also their feasibility depends on the satisfaction of real-time constraints. *Hard* constraints must be satisfied for system correctness, while the violation of *soft* constraints only degrades a system. Specification of real-time constraints requires either some extensions of programming languages through annotations and logic expressions, or the use of temporal logics or some formal declarative language with temporal constructs. Stringent timing restrictions complicate system design and verification. The time model can be dense or discrete, thus giving different methods for real-time system synthesis and verification. This article surveys the *specification*, *design*, and *verification* of real-time constraints and systems. The most important technique for guaranteeing real-time, namely scheduling, is briefly surveyed. Different system models are presented for handling real-time constraints such as Petri nets, timed automata, process algebra and object-oriented model. Design techniques for real-time hardware systems and for real-time software applications are introduced and discussed. Hardware-software codesign is also introduced. Verification techniques for real-time constraints, such as *model checking*, are also presented.

[†] The author will be moving to the Department of Computer Science and Information Engineering, National Chung-Cheng University, Chiayi, Taiwan, starting February 2001. The work was supported by several research grants from the National Science Council, Taipei, Taiwan.

Contents

Abstract **1**

1 Introduction **3**

2 Un-timed and Real-Time Systems **5**

2.1 Petri Nets **6**

2.2 Timed Automata **9**

2.3 Process Algebra **10**

2.4 Object-Oriented Techniques **11**

3 Real-Time Constraints **12**

3.1 Real-Time Languages **13**

3.2 Constraints Checking and Real-Time Logic **14**

4 Real-Time System Design **17**

4.1 Real-Time Scheduling **17**

4.2 Hardware System Design **20**

4.3 Software System Design **21**

4.3.1 Formal Software Synthesis 21

4.3.2 Object-Oriented Application Frameworks 22

4.3.3 Real-Time Operating Systems 23

4.3.4 Hardware-Software Co-design 23

5 Real-Time System Verification **24**

6 Conclusion and Future Work **27**

References **28**

1. Introduction

Should *real-time* be fast? No, that is a myth! In fact, real-time is *just in time*. For example, an avionics flight control system must spend 2.5 ms for attitude control every 50 ms period and 0.28 ms for flutter control every 4 ms period. The period restriction must be strictly abided, no faster and no slower, otherwise an airplane might crash! These just-in-time aspects constitute what are called *real-time constraints*.

With the increased use of intelligent everyday-life systems, such as electric home appliances, office automation contrivances, and medical equipments, it has become increasingly important to assimilate knowledge on how real-time constraints can be *specified*, *modeled*, *implemented*, and *verified*. Without trying to be exhaustive, this article tries to cover most aspects of real-time constraints, such that both a novice and an expert may benefit from referring to it.

Informally speaking, a real-time constraint is any condition on the timing of events, including event enabling, firing, initiation, resource usage, synchronization, and termination. A real-time constraint may be as simple as the specification of a deadline for a particular task to complete execution. A periodic real-time task may be further associated with a period constraint. For example, task *A* must execute once every 50 ms period with a deadline of 60 ms. Further details on the specification of real-time constraints are given in Section 3.

Constraints may be classified in several ways [1]. As far as strictness of timing is concerned, constraints may be classified as *hard* or *soft*. Hard constraints *must* be satisfied, the failure of which results in system crashes or serious consequences. Soft constraints *may* be satisfied and have tolerance ranges associated, the violation of which merely degrades a system behavior, without endangering it or the environment. Hard real-time constraints are found in high-assurance systems, such as nuclear reactors, avionics, power systems, medical emergency equipments, and space navigation systems. Soft real-time constraints are found in low-assurance systems, such as telecommunication systems, network systems, electric home appliances, flexible manufacturing systems, and office automation systems.

Constraints can also be classified based on the type of specification. There are two types: (1) *abstract* specifications in the form of an assertion language that is independent from the design or implementation language, and (2) *integrated* specifications that are inseparable from the implementation language and make use of language variables like actual time and resource status.

Timing constraints can be evaluated in different ways: (1) *static* evaluation (i.e. pre-runtime), which implies that the timing specifications are implementation independent, (2) *dynamic* evaluation (i.e. at runtime), which is necessary if priorities are specified, and (3) *hybrid* evaluation, which is a combination of the above two and is necessary in off-line schedulability analysis with exact runtimes.

The time domain in a real-time system can be *discrete* or *dense*. Discrete time allows simpler analysis and instrumentation procedures because the tasks in a real-time system may then be simply taking turns as in a card game of poker. Examples of such systems are telephone networks, communication protocols, and manufacturing systems, where automatic control can be applied by enabling and disabling of system events. Here, the domain of integers is used to model time [2], [3]. A smallest measurable time unit is specified *a priori* in the discrete time model. The *fictitious clock* approach includes an explicit tick transition making time a global state variable [4], [5]. Each tick increments time by some predetermined time quantum. In this model, events between the *i*-th and (*i*+1)-th clock ticks are assumed to occur at some unspecified time between times *i* and *i*+1. Thus, it is impossible to know the exact time delay between any two events. The model can be interpreted as an approximation to real-time, where events between time *i* and *i*+1 have their occurrence times truncated to *i*.

Dense time domain makes analysis and instrumentation procedures much more complex due to requirement of exact timeliness. Some systems must be analyzed and implemented using the dense time model for it to correctly satisfy given real-time constraints. Examples of such systems include automation of transport systems, such as railway and flight control, which depends critically on reaction times. Computer networks demand a maximal response time [6]. There are four strong reasons why a dense model of time is necessary [7]. A dense model of time is needed for *correctness*. It is more *expressive* than the others. *Composition* of processes is straightforward in the dense-time model. Finally, some important problems for finite-state systems have the same *complexity* using a dense-time model as for the other models.

This article is organized as follows. Section 2 introduces differences between un-timed and real-time systems along with system models. Section 3 presents real-time constraints, specification methods, language constructs, and constraint checker. Section 4 deals with real-time system design, including hardware, software, and their models. Section 5 covers real-time system verification, including model-checking, verification tools, and verification techniques. Section 6 concludes this article with pointers to future research directions and technology improvements.

2. Un-timed and Real-Time Systems

Control systems such as certain flexible manufacturing systems perform a sequence of tasks based on external events, such as the push of a switch or a lever. The correctness of such systems depends on the execution sequence and not on time. Thus, they are called *un-timed systems*. Although un-timed systems do not depend on time for task execution, their overall performance may still be related to time. For example, a scheduling criterion for an un-timed flexible manufacturing system may be the minimization of total execution time or the maximization of total throughput. But, often such types of timing constraints affect neither the correctness nor the stability of a system. These timing constraints are thus not real-time constraints and are out of scope of our discussion in this article.

From the above discussion, it can be stated that real-time constraints include only those constraints that actually affect a system's correctness, feasibility, or stability. Real-time systems are control systems that have constraints on the exact timing of task executions, which are expressed as real-time constraints. For example as shown in **Fig. 1**, an *Autonomous Intelligent Cruise Controller* (AICC) developed by Swedish Road Transport Informatics Programme and installed in a Saab automobile [8] requires traffic light and speed information to be polled every 200 ms, information processing to be performed every 100 ms, and final coordination control to be performed every 50 ms. In the figure, SRC stands for *Short Range Communication* and EST stands for *Electronic Servo Throttle*. This is an example of a typical embedded real-time system.

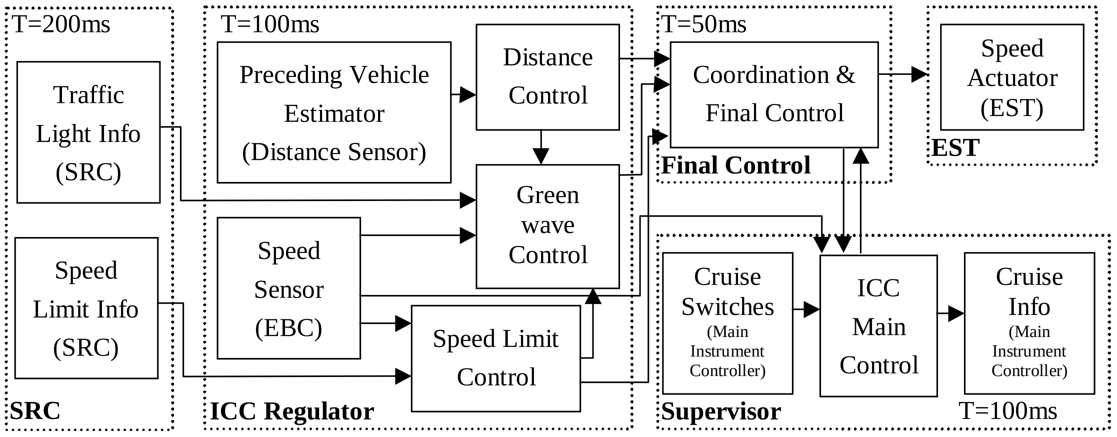

Fig. 1 Autonomous Intelligent Cruise Controller [8]

In the rest of this section, different models of real-time systems will be introduced and compared, including timed variants of Petri Nets, timed automata, process algebra, and object-oriented models.

2.1 Petri Nets

Petri nets are a graphical form of formal system model, which can be used to efficiently model transition systems characterized by concurrency, non-determinism or conflicts, synchronization, merging, confusion, mutual exclusion, and priority [9]. We first define a standard Petri net and then introduce its timed versions, which can be used to model real-time systems.

In the following, the set of integers and non-negative real numbers are denoted by $\boldsymbol{N}$ and $\boldsymbol{R}_{\geq 0}$, respectively A standard Petri net can be defined as follows:

Definition 1: **Petri Net**
A *Petri net* is a 5-tuple (P, T, I, O, M_0), where

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*,
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*, $P \cup T \neq \varnothing$, and $P \cap T = \varnothing$,
- I: $(P \times T) \rightarrow \boldsymbol{N}$ is an *input function* that defines directed *arcs* from places to transitions,
- O: $(P \times T) \rightarrow \boldsymbol{N}$ is an *output function* that defines directed arcs from transitions to places, and
- M_0: $P \rightarrow \boldsymbol{N}$ is the initial *marking*, where a marking is an assignment of *tokens* to the places of a Petri net.

A token is a primitive concept for Petri nets (like places and transitions). Tokens are assigned to, and can be thought to reside in, the places of a Petri net. The number and position of tokens may change during the execution of a Petri net. The tokens are used to define the execution of a Petri net.

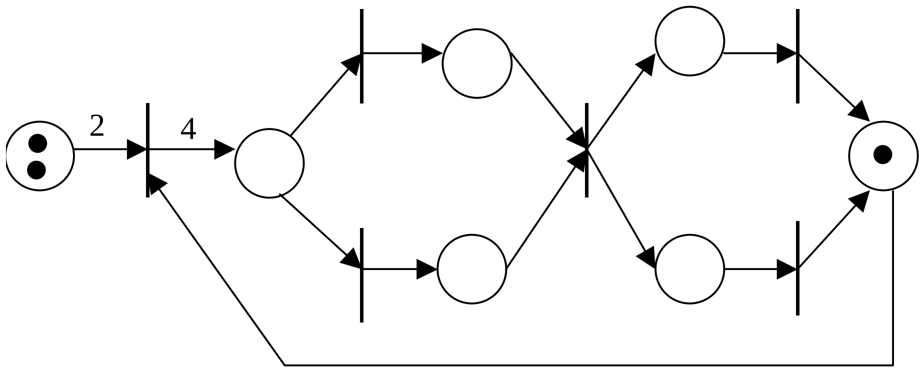

Fig. 2 A standard Petri net

A Petri net can be graphically represented as shown in **Fig. 2**, where a circle represents a place, a bar or a box represents a transition, an arrow represents an arc connecting a place and a transition, and a black dot represents a token. If $I(p_j, t_i) = k$ (or $O(p_j, t_i) = k$), then there exist k arrows connecting place p_j to transition t_i (or connecting transition t_i to place p_j). An arc may be labeled by an integer, which represents its multiplicity or weight.

A transition t is said to be enabled if each input place p of t contains at least the number of tokens equal to the weight of the directed arc connecting p to t, i.e., $M(p) \geq I(p, t)$ for any p in P, where M is the current marking. An enabled transition t may or may not fire depending on additional interpretation. A firing of an enabled transition t removes from each input place p the number of tokens equal to the weight of the directed arc connecting p to t. It also deposits in each output place p the number of tokens equal to the weight of the directed arc connecting t to p.

Petri nets can be used to analyze system properties such as reachability, boundedness, conservativeness, and liveness. Analysis methods of Petri nets include the coverability tree, incidence matrix and state equation, invariant analysis, and reduction rules.

The standard Petri net has been extended into high-level Petri nets by several domain experts, including extensions such as fuzzy, object-oriented, stochastic, generalized, colored, and timed. Three timed versions of Petri nets are introduced here, namely *Deterministic Timed Petri Nets* (DTPN), *Time Petri Nets* (TPN), and *Timing Constraint Petri Nets* (TCPN).

There are three types of DTPN, depending on where deterministic time labels (representing time delays) are placed. If time labels are associated with transitions, then it is called *Deterministic Timed Transitions Petri Nets* (DTTPNs) [10]. If time labels are associated with places, then it is called *Deterministic Timed Places Petri Nets* (DTPPN). If time labels are associated with arcs, then it is called *Deterministic Timed Arcs Petri Nets* (DTAPN). Only DTTPNs are defined here, because the other DTPNs can be defined similarly.

Definition 2: **Deterministic Timed Transitions Petri Net**
A *deterministic timed transitions Petri net* (DTTPN) is a 6-tuple (P, T, I, O, M_0, D), where (P, T, I, O, M_0) is a standard Petri net, and D: $T \rightarrow \boldsymbol{R}_{\geq 0}$ is a function that associates transitions with deterministic time delays. A transition t_i in a DTTPN can fire at time d if and only if

- in any input place p of t_i, $w(p, t)$ tokens have resided for the time interval $[d{-}d_i, d]$, where $w(p, t)$ is the weight associated with arc connecting p to t and d_i is the associated firing time of t_i, and

- after a transition fires, tokens are produced at output places at time d.

Depending on a given real-time system, different DTPNs may be used to model it. DTAPN is more general and can thus be used to model complex real-time systems.

Merlin and Farber [11] proposed *Time Petri Nets* (TPN), in which two time values are associated with each transition. The values constitute a time interval within which a transition is enabled and may fire. A formal definition is as follows.

Definition 3: **Time Petri Net**
A *Time Petri net* (TPN) is a 6-tuple (P, T, I, O, M_0, S), where (P, T, I, O, M_0) is a standard Petri net, and S: $T \rightarrow Q^+ \times (Q^+ \cup \infty)$ is a mapping called static interval, where Q^+ is the set of positive rational numbers.

If a transition t_i has an interval (a, b) associated with it, then a is the minimum time t_i must wait for after it is enabled and before it is fired, and b is the maximum time the transition can wait for before firing if it is still enabled. Transition firing is instantaneous, that is, firing a transition takes no time to complete. When a pair (a, b) is not defined, then it is implicitly assumed that the corresponding transition is a classical Petri net transition and ($a = 0$, $b = \infty$).

Sloan and Buy [12] developed a set of reduction rules for TPNs, such as serial fusion, pre-fusion, post-fusion, and lateral fusion. These reduction rules can reduce the size of the exponentially large state-spaces and thus help in analyzing TPNs. Further compositional TPNs are also defined for augmenting Petri nets with module constructs. A compositional TPN consists of two basic elements: component TPN models and inter-component connections.

One more type of timed Petri nets is called *Timing Constraint Petri Nets* (TCPN) [13], which was inspired from DTPN and TPN. The major difference is that TCPN assume a *weak firing mode*, in contrast to the *strong firing mode* of the other two types of timed Petri nets. The weak firing mode does not force any enabled transition to fire. The strong firing mode forces an enabled transition to fire immediately. The strong firing mode is not suitable for some nets with conflict structures, which results in contradictions.

Definition 4: **Timing Constraint Petri Net**
A *Timing Constraint Petri net* (TCPN) is a 7-tuple (P, T, I, O, M_0, C, D), where (P, T, I, O, M_0) is a standard Petri net, C is a set of integer pairs, ($TC_{min}(pt_j)$, $TC_{max}(pt_j)$), where $TC_{min}(pt_j) \leq TC_{max}(pt_j)$ and pt_j is either a place or a transition, and D is a set of firing durations, $\{FIRE_{dur}(pt_j)\}$.

A transition t_j with a time pair, ($\underline{TC_{min}}(t_j)$, $TC_{max}(t_j)$), is said to be enabled if each of its input places has at least one token. A transition t_j, which is enabled at time t, is said to be firable during the time period $t + TC_{min}(t_j)$ to $t + TC_{max}(t_j)$. A firable transition can fire but there is no guarantee that the firing will complete successfully because the firing of a transition takes a period of time $FIRE_{dur}(t_j)$.

2.2 Timed Automata

When real-time systems are more control-oriented, they can be modeled by *Timed Automata* (TA), which are a timed extension of finite state machines. Before defining TA, some necessary terms are defined as follows, where the set of integers and non-negative real numbers are denoted by $\boldsymbol{N}$ and $\boldsymbol{R}_{\geq 0}$, respectively.

Definition 5: **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := false \mid x{\sim}c \mid x-y{\sim}c \mid d{\sim}c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim\, \in \{\leq, <, =, \geq, >\}$, $c \in \boldsymbol{N}$, $d \in D$, and η_1, η_2 are mode predicates.

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

Definition 6: **Timed Automaton**
A *Timed Automaton* (TA) is an 8-tuple $A = (M, m^0, C, D, X, E, T, R)$ such that: M is a finite set of modes, $m^0 \in M$ is the initial mode, C is a set of clock variables, D is a set of discrete variables, X: $M \rightarrow B(C, D)$ is an invariance function that labels each mode with a condition true in that mode, $E \subseteq M \times M$ is a set of transitions, T: $E \rightarrow B(C, D)$ defines the transition triggering conditions, and R: $E \rightarrow 2^{C \cup (D \times \boldsymbol{N})}$ is an assignment function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values.

We further define the semantics of a TA by defining its state, mode transition, and a feasible computation run as follows.

Definition 7: **State**
Given a TA $A = (M, m^0, C, D, X, E, T, R)$, a *state* s of A is defined as a mapping from $C \cup D$ to $\boldsymbol{R}_{\geq 0} \cup \boldsymbol{N}$ such that for all x in C, $s(x) \in \boldsymbol{R}_{\geq 0}$ is the reading of clock x in s, and for all d in D, $s(d) \in \boldsymbol{N}$ is the value of d in s.

Definition 8: **Mode Transition**

Given two states s_1, s_2, there is a mode transition from s_1 to s_2, in symbols $s_1 \rightarrow s_2$, iff both s_1 and s_2 belong to some defined modes, mode invariants are satisfied by the states, there is a transition between the two modes, the triggering condition of the transition is satisfied by s_1 and for all clocks x in C, $s_2(x) = 0$ when x is in the reset assignment of the transition and all other clocks are unchanged.

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

2.3 Process Algebra

Process Algebra [14], [15] is a term-based formal specification language for system design and analysis. *Calculus of Communicating Systems* (CCS) [14] was extended in several works to model real-time systems, resulting in real-time process algebras [15], [16]. Another recent work tackles state-space explosions by using dynamic priorities, called CCS with dynamic priority, which extends CCS by assigning priority values to actions. Unlike other real-time process algebras, CCS with dynamic priority avoids the unfolding of delay values into sequences of elementary steps, each consuming one time unit, thereby providing a formal foundation for efficiently implementing real-time semantics. CCS with dynamic priority has been proved to be bisimilar to CCS with real-time.

The syntax of CCS with real-time and with dynamic priority can be defined as follows.

Definition 9: CCS with real-time and with dynamic priority
The syntax of CCS with real-time and dynamic priority is defined as follows:

$$P \quad ::= \quad \mathbf{0} \mid x \mid \alpha{:}k.P \mid P+P \mid P \Diamond P \mid P|P \mid P[f] \mid P \backslash L \mid \mu x.P$$

where x is a variable taken from some countable domain V, α is an action, $k \in \boldsymbol{N}$, the mapping f: $A \rightarrow A$ is a relabeling, $L \subseteq A \setminus \{\tau\}$ is a restriction set, A is the set of all actions, and τ is an internal action. As far as the binary operators are concerned, + is a non-deterministic choice, $\Diamond$ is a disabling operator, | is a parallel operator, and \ is the set subtraction operator.

The semantics can be defined by action transitions and clock transitions. Here, in CCS with real-time, $\alpha{:}k.P$ means the action α has a delay equal to k time units associated with it and the resulting process is P. In CCS with dynamic priority, $\alpha{:}k.P$ means the action has a priority k associated with it and the priority can be changed dynamically. The labeled transition system for a process P is a 4-tuple ($\mathbf{P}$, $A\cup\{1\}$,$\rightarrow$ P), where **P** is the set of states, $A\cup\{1\}$ is the alphabet, $\rightarrow$ is the transition relation, and P represents the start state. Maximal progress is assumed here, that is, no idling is allowed when a communication can take place. Further in CCS with dynamic priority, higher priority processes can pre-empt lower priority tasks.

2.4 Object-Oriented Techniques

Real-time systems can be modeled by object-oriented techniques [17], [18], which have been widely accepted in the software as well as the hardware design community. Object-orientation has many benefits not found in traditional structured design. In a real-time system, each process thread can be modeled by a class and each resource can also be modeled by a class. Encapsulations of data and time in a class result in safer systems, which can be upgraded more easily than conventional systems.

A standard for object-oriented modeling language, called *Unified Modeling Language* (UML) [19] has been extended with a real-time profile. Real-time UML [20] is currently a well-received design modeling paradigm in the real-time community [21]. Further, programming languages like Java has also been recently extended for real-time application design, with a *Real-Time Specification for Java* [22], by the Real-Time for Java Experts Group. CORBA is another standard in designing distributed object-oriented real-time applications [23].

In real-time UML modeling resources [24], QoS (quality of service) characteristics are taken as the basis for quantitative analysis. These characteristics are given as constraints to model elements that specify behavior at runtime, including use cases, interactions, operations, state machine transitions, activities, and individual actions. A *realization mapping* is used to compare QoS characteristics. This mapping is a syntactical declaration that a particular resource supports a particular logical element in some unspecified way. The user must determine the realization’s semantics and validity. More formal and more sophisticated forms such as standard stereotypes, involve semantic knowledge of the nature of the logical and engineering model elements being bound. For example, a CORBA or a COM channel can realize a communication link between two objects in a logical model. A *realization package* is modeled as a UML package and represents a consistent set of mappings that are mutually compatible and nonexclusive. A given logical model can have any number of realization packages, each of which represents one distinct mapping of the logical model to exactly one engineering model.

The *real-time specification for Java* (RTSJ) was recently proposed by the

Real-Time for Java Experts Group (RTJEG), which begin development in March 1999 under the Java Community Process. The programming language Java itself was left untouched. The specification merely enhances Java by defining new classes that provide real-time behavior. Seven areas were proposed that required new specification, namely *scheduling*, *memory management*, *synchronization*, *asynchronous event handling*, *asynchronous transfer of control*, *asynchronous thread termination*, and *physical memory access*. Each area considered the state-of-art technology in real-time system and application development.

For scheduling, the only specification was that a *fixed-priority preemptive scheduler* with no fewer than 28 priorities was required. For memory management, several areas of memory were newly defined, namely scoped memory, immortal memory, and Java heap. For synchronization, the *priority inheritance protocol* was required to be implemented by default and wait queues were defined for communication among regular Java threads, *Real-Time* (RT) threads, and *No-Heap Real-Time* (NHRT) threads. For asynchronous event handling, two new classes were defined: *AsyncEvent* and *AsyncEventHandler*, where the former represents something that can happen, e.g., a *Posix* signal, and the latter is a schedulable object that handles an asynchronous event. A *Clock* class is also specified for modeling time. More than one clock may also be implemented. For asynchronous control transfer, *AsynchronouslyInterruptedException* (AIE) is specified asynchronously transfer control. For asynchronous thread termination, safe stopping of threads is implemented. For physical memory access, RTSJ defines two classes: *RawMemoryAccess*, which allows memory access in terms of byte, word, long, and multiple-byte granularity, and *PhysicalMemory*, in which Java objects can be located.

3. Real-Time Constraints

A *real-time constraint* is defined as a Boolean condition on the values of clock variables. Clock variables are variables whose value increases with time. Clock variables, or clocks in short, may be either *global* or *local*. The values of global clocks are visible to all processes of a system and those of local clocks are visible only to their owner processes. Clocks may be either *absolute* or *relative*. Absolute clocks take values from a global timer, which is never reset after initialization. Relative clocks take values from a timer, which could be a difference of two other clock values. Clocks may be *discrete* or *dense*. Discrete clocks increase value by integral increments, while dense clocks increase value by real-time quantums. In a single real-time system, all clocks are discrete or all clocks are dense, but absolute and relative clocks, just as global and local clocks, may co-exist a system. Since different models propose different syntaxes for real-time constraint specification, the exact syntax of a Boolean condition on clock variables is dependent on the system model used. In general, the condition evaluates to either true or false in a system state. For example, in a timed automaton, $x < 3 \wedge y \geq 8$ is a Boolean condition on clock variables x and y, such that it evaluates to true in a particular system state only when both the clocks have values satisfying the two predicates, respectively, in the condition.

In the following subsection, real-time extensions to existing programming and modeling languages are described such that real-time constraints can be specified. Real-time languages often add statements about the temporal constraints of computations to the syntax of the language. However, most current real-time languages using the process model for programming also assume the conventional run-time model which manages a set of processes preempting one another according to their execution priority, competing for resources, and blocking when resources are already in use, which tends to limit their ability to predict execution behavior. In the final subsection, constraints checking for temporal correctness is described and an associated *Real-Time Logic* (RTL) introduced.

3.1 Real-Time Languages

Real-time Euclid [25] was one of the earliest real-time languages, which is an extension of Euclid. It restricts language constructs such as recursion and dynamic memory allocation. Concurrency can also be controlled through **signal** and **wait** constructs. Real-time Euclid was designed mainly for schedulability analysis under a number of assumptions on the system and process behavior.

Real-time Mentat [26] is an object-oriented real-time language, which is an extension of C++. In real-time Mentat, a programmer may specify timing constraints in statement level. Both soft and hard deadlines can be specified. A block with soft deadline may be skipped if the hard real-time tasks cannot meet their deadlines. It does not support preemption of objects.

RTC++ [17] is an object-oriented real-time language, which is also an extension of C++. It supports preemption, but not soft deadlines. Active objects are introduced and active objects with timing constraints are called real-time objects. It supports inheritance in active objects, synchronous communications among active objects, and exception handling. Time is encapsulated in an active object so as to specify timings for an operation. A critical region is realized in RTC++ by implementing an object with a guard expression. Rate monotonic scheduling is assumed.

The programming language for the Maruti system, MPL [27], also extends C++. MPL provides several ways to specify temporal constraints on blocks of code within an object. Loop bounds are specified and recursion forbidden to increase predictability.

Kenny and Lin describe the *Flex* language [28], another extension of C++, which includes a number of timing constraint expressions and exception handling clauses. A polymorphism analogous to operator overloading is adopted for approximate processing [29]. Here, polymorphism refers to providing several routines implementing the same function, which have different properties in space and/or time. Flex also supports monotonic algorithms, which support computations with unpredictable behavior by establishing an initial result early and then iteratively refining it until the deadline is reached.

Real-Time Concurrent C [30] extends Concurrent C by providing facilities for building systems with strict timing constraints. Real-Time Concurrent C allows processes to execute activities with specified periodicity or deadline constraints, to seek dynamic guarantees that timing constraints will be met, and perform alternative actions when either the timing constraints cannot be met or the guarantees are not available.

Spring system’s real-time system description language, *SDL* [31], explicitly supports specifying a computation's real time behavioral constraints, end-to-end constraints, concurrency, and details of the hardware-software platform that are required to accurately analyze the system and achieve predictability. The programming language, *Spring-C* [32], works in concert with the specification language. Its structure constrains the programmer in ways, which ensure that worst-case execution behavior, including execution times, can be automatically predicted for the particular hardware platform being used. Of course, such platforms should have predictable instruction execution times. A key aspect of the Spring-C compiler is that it automatically identifies all of a computation's potential blocking points, i.e., points during execution when it can block for resources or wait for synchronous communication to occur [33].

3.2 Constraints Checking and Real-Time Logic

A *reactive real-time system* has to compute a result to an external trigger event within certain timing constraints even in the presence of faults. It has to initiate some adaptive reactions when an assumption is violated [34]. Possible reactions include the activation of stand-by resources, a rescheduling of the remaining resources, or the execution of alternative algorithms for solving the problem under certain emergency conditions. For a reaction to be made, a system has to be informed of the occurrence of a violation. Run-time monitoring and checking of constraints are thus a part of such reactive systems. A *constraint checker* is the required facility.

Major work in the area of checking timing constraints in a real-time system were done by Jahanian and Mok, which was initiated by proposing *Real-Time Logic* (RTL) language [35] for the specification of real-time systems. The semantics of RTL is based on the occurrence of events, which result on the execution of a real-time system, such as the start and the end of code blocks or the assignment of values to state variables. Algorithms for checking safety assertions [36] and for partial event-traces [37] against RTL-specifications were developed. A distributed on-line monitoring and checking tool, which allows to specify timing assertions in a subset of RTL and to check whether these assertions are violated or not, was later developed [38], [39]. Events storing, definition of timing constraints, and evaluation of constraints in a distributed environment are all handled by the monitoring tool. Later, the work was extended to object-oriented models, integrated into standard programming languages like C++, and code instrumented with event-triggers [34].

As far as constraints checking is concerned, RTC++ and Flex provide schedulability analysis, but do not provide static worst-case execution time analysis. Real-time Euclid provides static worst-case execution time analysis. Using a real-time language cannot guarantee that timing violations will not happen. To cope with such violations, most real-time languages contain a checker for deadline violations and an exception-handling mechanism. In [34], a new component called “constraint” section is added to a class description. This section contains a list of named RTL-like formulas, which are composed out of basic events such as “start” and “end” of code sections and changes of state variables. When a constraint is violated, the object produces an event with the static name of the constraint, the dynamic context of the object that violated the constraint and a time-stamp that is the earliest point in time when the checker could evaluate that the constraint will be violated. This independence of functional and timing specifications avoids inheritance anomalies and it allows the construction of two separate systems: the object-oriented real-time system and its constraint checker.

As shown in Fig. 3, there are two constraints in the constraints section of a class description. Constraint 1, named “`max_time`” states that an execution of the member function `compute()` must not take longer than 8 ms. The expression `@(compute.start, -1)` denotes the start time of the most recent execution of the

```
class sensor {
public:
    int compute();
    :
    [[ // The Constraint Section
    // Constraint 1:
    // compute() must not take longer than 8 ms
        max_time: @(compute.start, -1) >= @(compute.end, -1) – 4ms;
    // Constraint 2:
    // compute() must not be called more than once per second
        recovery: @(compute.start, -2) <= @(compute.end, -1) – 2s;
    ]]
}
```

Fig. 3 A C++ Class with Constraints Section

member function `compute()` and `@(compute.end, -1)` evaluates to the end time of the same execution. Constraint 2, named "`recovery`", expresses that two successive calls to `compute()` must have a distance of at least 2 seconds.

For constraints checking, a compile-time and run-time support is required. A compiler for a proposed language extension has to do two additional tasks besides the production of the object code. It has to translate the timing constraints into a) an instrumentation of the object-oriented program in order to produce the required events, and b) a representation of the constraints that can be evaluated by the constraint checker. An eventing system is the run-time support. It has to receive event records from code instrumentations, it has to provide time-stamps from a global clock, and it has to filter out irrelevant events. It then collects the events from the different nodes of a distributed system and merges them into a global event stream according to the total order imposed by the time-stamps. The constraint checker has to receive static information about the structure of the timing constraints from the compiler. During run-time, it has to react to incoming events. It maintains a global event dispatch table that maps other incoming events to the objects and the constraints that might be affected. A constraint violation is detected by constructing a current instance of a graph out of the graph-templates. Upon detection of a constraint violation, the checker itself produces a corresponding event, which will immediately be checked (since it is the next event in the total order of events).

There are three possible modes in which a checker can be used: *off-line*, *on-line*, and *real-time*. In the off-line mode, performance of the checker has not effect on the system being checked. An on-line checker has to cope with the average event rate, so that it can keep track of a running system. If it is a real-time checker, then all of its parts must have known worst case execution times and they must be scheduled with the application itself.

4. Real-Time System Design

Real-time system design deals with how real-time constraints may be feasibly implemented in working systems that might contain pure hardware, pure software, or both hardware and software. In general, a real-time system is designed as follows. A set of system specifications, including real-time constraints, is specified by a designer. A synthesis methodology uses some kind of system models, performance models, estimation models, and exploration models, to design a system that satisfies all the system specifications. The final design is then validated through simulation, testing, or rapid prototyping. A target design could be a sequential system, consisting of either one CPU or one ASIC, or a parallel system, consisting of multiple CPUs or multiple ASICs. The latter is much more complex to design than the former, as described in Section 4.2. Irrespective of hardware or software implementations, real-time scheduling of multiple tasks on a single CPU or ASIC is essentially the most validated and theoretically proven. Real-time scheduling will be discussed briefly in Section 4.1. Hardware design is introduced in Section 4.2. Different paradigms for software design are discussed in Section 4.3. Hardware-software co-design methodologies are presented in Section 4.4.

Definition 10: Synthesis of multiple tasks hard-real time multiprocessor systems
Given a set of tasks with hard real-time constraints such as period, start time, and finish time or deadline, design a system consisting of multiple CPUs or multiple ASICs such that the set of tasks is partitioned into several subsets, each subset is implemented on one dedicated CPU or ASIC, all the given real-time constraints are satisfied, and the overall system cost is minimal.

The above defined optimization design problem is NP-complete [40]. Even several sub-problems of the above problem are NP-complete, such as scheduling of a single task on minimal resources [41], or minimization of only one type of resource [42], or register minimization [42]. This layering of computationally intractable sub-problems does not affect overall worst-case asymptotic computational complexity, but it makes the synthesis problem exceptionally challenging in practice because numerous contradictory effects along several hardware dimensions, at both process and task granularity levels, must be taken into account.

4.1 Real-Time Scheduling

A real-time system generally needs to process various concurrent tasks. *Real-time scheduling* is defined as assigning the exact execution times for a set of real-time tasks such that all temporal constraints including period, phase, deadline, priority, and resource requirements are satisfied.

A *task* is a finite sequence of computation steps that collectively perform some required action of a real-time system and may be characterized by its execution time, deadline, etc. *Periodic* tasks are tasks that are repeatedly executed once per period of time. Each execution instance of a periodic task is called a *job* of that task.

In a processor-controlled system, when a processor is shared between time-critical tasks and non-time-critical ones, efficient use of the processor can only be achieved by careful scheduling of the tasks. Here, time-critical tasks are assumed to be preemptive, independent, periodic, and having constant execution times with hard, critical deadlines.

Scheduling may be *time-driven* or *priority-driven*. A time-driven scheduling algorithm determines the exact execution time of all tasks. A priority-driven scheduling algorithm assigns priorities to tasks and determines which task is to be executed at a particular moment.

In the following, we mainly discuss time-critical periodic tasks with the above assumptions and scheduled using priority-driven scheduling algorithms. Depending on the type of priority assignments, there are three classes of scheduling algorithms: *fixed priority*, *dynamic priority*, and *mixed priority* scheduling algorithms. When the priorities assigned to tasks are fixed and do not change between job executions, the algorithm is called fixed priority scheduling algorithm. When priorities change dynamically between job executions, it is called dynamic priority scheduling. When a subset of tasks is scheduled using fixed priority assignment and the rest using dynamic priority assignment, it is called mixed priority scheduling.

Before going into the details of scheduling algorithms, we define the task set to be scheduled as a set of n tasks $\{\phi_1, \phi_2, \ldots, \phi_n\}$ with computation times $c_1, c_2, \ldots, c_\mathrm{n}$, request periods $p_1, p_2, \ldots, p_\mathrm{n}$, and phasings $h_1, h_2, \ldots, h_\mathrm{n}$. A task ϕ_i is to be periodically executed for c_i time units once every p_i time units. The first job of task ϕ_i starts execution at a time h_i. The worst-case phasing called a *critical instant* occurs when $h_i = 0$, for all i, $1 \le i \le n$.

Liu and Layland [43] proposed an optimal fixed priority scheduling algorithm called the *rate-monotonic* (RM) scheduling algorithm and an optimal dynamic priority scheduling algorithm called *earliest-deadline first* (EDF) scheduling.

The RM scheduling algorithm assigns higher priorities to tasks with higher request rates, that is, smaller request periods. Liu and Layland proved that the worst-case utilization bound of RM was $n \times (2^{1/n} - 1)$ for a set of n tasks. This bound decreases monotonically from 0.83 when $n = 2$ to $\mathbf{log}_e 2 = 0.693$ as $n \to \infty$. This result shows that any periodic task set of any size will be able to meet all deadlines all of the time if RM scheduling algorithm is used and the total utilization is not greater than 0.693.

The exact characterization for RM was given by Lehoczky, Sha, and Ding [44]. They proved that given periodic tasks $\phi_1, \phi_2, \ldots, \phi_n$ with request periods $p_1 \le p_2 \le \ldots \le p_n$, computation requirements $c_1, c_2, \ldots, c_n$, and phasings $h_1, h_2, \ldots, h_n$, ϕ_i is schedulable using RM iff

$$\mathbf{Min}_{\{t \in G_i\}} W_i(t)/t \quad \le \quad 1 \qquad (1)$$

where $W_i(t)=\Sigma^i_{j=1} c_j\lceil t/p_j \rceil$, the cumulative demands on the processor by tasks over $[0, t]$, 0 is a critical instant (i.e., $h_i = 0$ for all i), and $G_i = \{k \times p_j \mid j = 1, \ldots, i, k = 1, \ldots, \lfloor p_i/p_j \rfloor\}$. Liu and Layland discussed the case when task deadlines coincide with request periods, whereas Lehoczky [45] considered the fixed priority scheduling of periodic tasks with *arbitrary* deadlines and gave a feasibility characterization of RM in this case: given a task set with arbitrary deadlines $d_1 \le d_2 \le \ldots \le d_n$, ϕ_i is RM schedulable iff $\mathbf{Max}_{k \le N_i} W_i(k, (k-1)p_i+d_i) \le 1$, where $W_i(k, x) = \mathbf{min}_{t \le x} ((\Sigma_{j=1\ldots i-1} c_j\lceil t/p_j \rceil + k \times c_i)/t)$ and $N_i = \mathbf{min}\{k \mid W_i(k, k \times p_i) \le 1\}$.

The worst case utilization bound of RM with arbitrary deadlines was also derived in [45]. This bound (U_∞) depends on the common deadline postponement factor $\mathit{\Delta}$, i.e., $d_i = \mathit{\Delta}\, p_i$, $1 \le i \le n$.

$$U_\infty(\mathit{\Delta}) = \mathit{\Delta}\, \mathbf{log}_e((\Delta+1)/\mathit{\Delta}), \quad \mathit{\Delta} = 1, 2, \ldots \qquad (2)$$

For $\mathit{\Delta} = 2$, the worst-case utilization increases from 0.693 to 0.811 and for $\mathit{\Delta} = 3$ it is 0.863. Recently, the timing analysis for a more general hard real-time periodic task set on a uni-processor using fixed-priority methods was proposed by Härbour et al [46].

Considering the earliest deadline first dynamic priority scheduling, Liu and Layland [43] proved that given a task set, it is EDF schedulable iff

$$\Sigma_{i=1\ldots n} c_i/p_i \le 1 \qquad (3)$$

and showed that the processor utilization can be as high as 100%.

Liu and Layland also discussed the case of *Mixed Priority* (MP) scheduling, where given a task set $\phi_1, \phi_2, \ldots, \phi_n$, the first k tasks $\phi_1, \ldots, \phi_k$, $k < n$, are scheduled using fixed priority assignments and the rest $n-k$ tasks $\phi_{k+1}, \ldots, \phi_n$ are scheduled using dynamic priority assignments. It was shown that considering the accumulated processor time from 0 to t available to the task set ($a_k(t)$), the task set is mixed priority schedulable iff

$$\Sigma_{i\,=\,1\ldots n-k} \lfloor t/p_{k+i} \rfloor\, c_{k+i} \leq a_k(t) \qquad (4)$$

for all t which are multiples of p_{k+1} or … or p_n. Here, $a_k(t)$ can be computed as follows.

$$a_k(t) = t - \Sigma_{j=1\ldots k}\, c_j \lceil t/p_j \rceil$$

Although the EDF dynamic priority scheduling has a high processor utilization, in recent years fixed priority scheduling has received great interests from both academy and industry [44], [45] ,[46], [47], [48], [49], [50], [51].

Summarizing the above scheduling algorithms, we have five different cases of schedulability considerations:

- RM-safe: all task sets are schedulable as long as the server utilization is below $\mathbf{log}_e\, 2 = 0.693$,
- RM-exact: all task sets satisfying Equation (1) are schedulable,
- RM-arbitrary: all task sets are schedulable as long as the server utilization is below $\Delta\mathbf{log}_e((\Delta+1)/\Delta)$ (Equation (2)),
- EDF: all task sets satisfying Equation (3) are schedulable, and
- MP: all task sets satisfying Equation (4) are schedulable,

4.2 Hardware System Design

As far as hardware system design is concerned, Potkonjak and Wolf [40] recently developed a new two-domain iterative refinement multi-resolution synthesis strategy to help manage the complexity of the above defined synthesis problem (Definition 10). The final solution implements the set of processes into a partitioned system of multiple ASICs.

Each process is initially considered in a single process domain. Estimations are made using the Hyper high-level synthesis system [52] through area-time trade-off curves for three types of hardware resources: execution units, interconnect, and registers. Estimations are then made for each partition with respect to the required hardware resources and feasibility of timing constraints. In the single process domain, augmented Hyper-LP estimations are made for all hardware components and complete implementations obtained. Inferior and non-feasible solutions are discarded. Finally, the complete single process and task-level schedules are obtained using the Hyper scheduler and a task-level scheduler. The proposed design methodology is a basis for an optimal worst-case exponential time branch and bound synthesis algorithm as well as fast heuristic synthesis algorithm.

4.3 Software System Design

Designing a software system to solve the real-time synthesis problem (Definition 10) is a scheduling problem, such that a given set of tasks is to be scheduled on a set of processors while simultaneously satisfying real time constraints and using the processor and memory resources as efficiently as possible [53]. As mentioned at the beginning of Section 4, this problem itself is NP-complete. Hence, many software scheduling strategies have been proposed.

4.3.1 Formal Software Synthesis

Scheduling of software can be accomplished based on data computations and control structures in a system specification. Three types of scheduling can be combined to obtain an ideal scheduling technique. Firstly, *static* scheduling can be used to exploit fixed dependencies between blocks of operation. Secondly, *quasi-static* scheduling can be used to identify data-dependent operations with the same rate and schedule them. Thirdly, *dynamic* scheduling can be used to determine which tasks should be executed.

For the synthesis of software executing on a single processor, several researches are still ongoing. Buck [54] proposed a quasi-static schedule computation algorithm based on *Boolean Data Flow* (BDF) network model. Theon et al. [55] proposed a technique to exploit static information in the specification and extract from a constraint graph description of the system statically schedulable clusters of threads. Lin [56], [57] used intermediate Petri net models to generate a software program from a concurrent process specification. Here, it is assumed that the Petri nets are *safe*, i.e. buffers can store at most one data unit, and hence cannot handle multi-rate specifications, like FFT computations and down sampling. Zhu and Lin [58] then proposed a compositional approach to software synthesis such that the size of the resulting C program was directly proportional to the size of the original specification. Later, Sgroi et al [53] proposed a software synthesis method based on quasi-static scheduling (QSS) of *Free Choice Petri Nets* (FCPN). The proposed algorithm is complete, in that it can solve QSS for any FCPN that is quasi-statically schedulable. Recently, an approach that maximizes the amount of static scheduling to reduce the need for context switching and operating system intervention was proposed by Cortadella et al [59].

Formal *real-time* software synthesis based on Petri nets is still at a premature stage and research work is ongoing in this area. Some work on using timed Petri nets to schedule flexible manufacturing systems have been proposed. Onaga et al [60] proposed a linear programming based heuristic approach for generating minimal time strict periodic schedules. Qadri and Robbi [61] uses a *Timed Petri Net Simulation* tool, TPNS [62], to model the performance of a flexible manufacturing cell arrangement with different scheduling approaches. Later, Zuberek [63] used invariant analysis of timed Petri nets to provide performance characteristics of manufacturing cells with composite schedules. Recently, Di Natale et al [64] proposed an iterative solution to schedule reactive real-time transactions modeled by a network of *Codesign Finite State Machines* (CFSM). It offers a priority assignment scheme together with a tight worst-case analysis.

4.3.2 Object-Oriented Application Frameworks

Another paradigm of software development for real-time systems is *Object-Oriented Application Frameworks* (OOAFs). An OOAF is a reusable, “semi-complete” application that can be specialized to produce custom applications [65]. Examples include MacApp, ET++, Interviews, ACE, Microsoft’s MFC and DCOM, Javasoft’s RMI and implementation of OMG’s CORBA. Compared to other application domains, *real-time* OOAFs are limited in number. Currently, there are *Real-Time Framework* (RTFrame), which is also called SESAG [66] and *Object-Oriented Real-Time System Framework* (OORTSF) [67].

SESAG is modularized into five components, namely *Specifier*, *Extractor*, *Scheduler*, *Allocator*, and *Generator*. Two different views of SESAG were presented: a *Components-Patterns* view and a *Class* view. Application domain objects are specified using the Specifier. Real-time constraints are either specified separately or coupled with the application domain objects. In the latter case, Extractor is used for extracting constraints. Extractor is also used to extract tasks from the given domain objects. Scheduler schedules the tasks using some scheduling algorithm and Allocator allocates resources among the tasks that are running concurrently. Finally, Generator is used to generate the application code based on the decisions made in the other components. Through applications on avionics and cruiser controls, SESAG has been shown to decrease design efforts to less than 5% of that required without using SESAG. The evaluation was made based on a *relative design effort* metric.

OORTSF emphasizes on high-level design reuse. Several design patterns and schedulers have been implemented into OORTSF. A five-step process is defined for developing real-time applications using OORTSF. First, domain task objects are identified and defined. Second, real-time requirements for each domain task object are generated. Third, schedulability check is performed on the set of tasks. Fourth, OORTSF is used to generate the target system code. Fifth, the generated target system is validated and verified. Currently, it has also been extended into a framework for developing distributed real-time applications. Three components called *AppNode*, *AppControl*, and *RemotePipeDirector* have been defined for a distributed application environment. Its integration with CORBA [68] and with Java RMI [69] has also been discussed. OORTSF has been used to design an airborne vehicle flight path control real-time application.

4.3.3 Real-Time Operating Systems

Last but not least in real-time software development is *Real-Time Operating Systems* (RTOS), which are stripped down and optimized versions of timesharing operating systems. Some features of RTOS include: fast context switch, small size, quick response to external interrupts, minimal interrupt-disable intervals, no virtual memory, code and data locking in memory, and fast accumulation of data through special sequential files [33]. RTOS kernels maintain a real-time clock, provide priority-scheduling mechanisms, provide for special alarms and timeouts, and permit tasks to pause/resume execution. In general RTOS kernels are multi-tasking and inter-task communication and synchronization are achieved via standard primitives such as mailboxes, events, signals, and semaphores. Many real-time UNIX OS [70] and a standard for RTOS, called RT POSIX [71], have been developed. There are also over 70 commercial proprietary RTOS including: QNX, LynxOS, OS-9, VxWorks, and VRTXsa. Real-Time Mach [18] is a RTOS developed in academia.

4.4 Hardware-Software Co-design

An embedded system often contains both *hardware* in the form of one or more ASICs or ASIPs and *software* executable on one or more microprocessors. Several works have been done on synthesizing a hardware-software system [72], but there are relatively fewer results targeted at hardware-software *real-time* systems. In the following, a recently proposed methodology, called *Distributed Embedded System Codesign* (DESC) methodology [73] is briefly presented.

DESC methodology uses three types of semantically equivalent models, namely, *Object Modeling Technique* (OMT) [74] models for system description and input, *Linear Hybrid Automata* (LHA) [75] models for system evaluation during partitioning and for formal verification, and SES/*workbench simulation* [76] models for performance evaluation after partitioning. A *hierarchical partitioning* algorithm [77] is proposed specifically for distributed systems. Software is synthesized by task scheduling and hardware is synthesized by *object-oriented* design techniques [78], [79], [80]. Design alternatives for synthesized hardware-software systems are then checked for design feasibility through rapid prototyping using hardware-software emulators. Timing coverification of real-time constraints is performed using LHA models [81], [82]. DESC methodology has been applied to a case study on a *Vehicle Parking Management System* (VPMS) [73], which shows the benefits of OO codesign, and the benefits of considering physical restrictions.

5. Real-Time System Verification

Since the correctness of real-time systems depends on whether the specified real-time constraints are satisfied or not, the validation or verification of such systems are all the more crucial. Validation of real-time systems can be done in the following ways: *simulation*, *testing*, *emulation*, *rapid prototyping*, and *worst-case execution time analysis*. Validation is not a *complete* or *full* technique, in the sense that after validation, a system designer still cannot guarantee 100% system correctness. Often statistical or probabilistic figures are cited after a real-time system is validated. For example, one can say after validation, that a real-time system is 99.99% correct with a 95% confidence range, or that it is correct for 99.5% of execution time.

In contrast, *formal verification* or *analysis* is complete, that is, a real-time system is verified to be 100% correct, with respect to some kind of temporal specification. In the recent few years, *model-checking* [83] has gained wide recognition due to its algorithmic approach at verifying real-time systems. In the following, model-checking is presented based on the *timed automata* (TA) system model and *timed computation tree logic* (TCTL) specification, as defined in Definition 6 and Definition 11, respectively.

Definition 11: Timed Computation Tree Logic (TCTL)
A timed computation tree logic formula has the following syntax.

$$\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi'\mathbf{U}_{\sim c}\phi'' \mid \neg\phi' \mid \phi'\vee\phi'' \qquad (5)$$

Here, η is a mode predicate (Definition 5), ϕ', ϕ'' are TCTL formulae, $\sim\ \in \{<, \leq, =, \geq, >\}$, and $c \in \boldsymbol{N}$. $\exists\Box\phi'$ means there exists a computation, from the current state, along which ϕ' is always true. $\exists\phi'\mathbf{U}_{\sim c}\phi''$ means there exists a computation, from the current state, along which ϕ' is true until ϕ'' becomes true, within the time constraint of $\sim$c. Traditional shorthands like $\exists\Diamond$, $\forall\Box$, $\forall\Diamond$, $\forall\mathbf{U}$, $\wedge$, and $\rightarrow$ can all be defined [84].

Model checking is an automatic procedure to verify is a given system satisfies a given temporal property. A dense real-time system can be described using a set of timed automata and a property specified in TCTL. In the following, a brief introduction to the intrinsic of model checking is given.

```
Symbolic_Mcheck($B$, $\phi$)
Set of TA $B$;
TCTL formula $\phi$;
{
    Let $Reach = Unvisited = \{R_{init}\}$;
    While ($Unvisited \neq$ NULL) {
        $R'$ = Dequeue($Unvisited$);
        For all out-going transition $e$ of $R'$ {
            $R''$ = Successor_Region($R'$, $e$);
            If $R''$ is consistent and $R'' \notin Reach$ {
            $Reach = Reach \cup \{R''\}$;
            Queue($R''$, $Unvisited$); }}}
    Label_Region($Reach$, $\phi$);
    Return L($R_{init}$);
}
```

Fig. 4 Symbolic Model Checking Procedure

A symbolic model checking procedure is given in Fig. 4, where two data-structures are maintained: a queue of regions (*Unvisited*) and a set of reachable regions (*Reach*). The former keeps a record of which regions are yet to be explored, while the latter keeps a record of all the regions reached. The procedure starts from an initial region, R_{init}, which is a Cartesian product of the initial modes of all the TA in the input set of TA, *B*. Initially, the initial region is queued in *Unvisited* and recorded in *Reach*. A region, R', is dequeued from *Unvisited* and corresponding to each out-going transition, *e*, of R' a successor region, R'', is constructed by the function **Successor_Region**(R', *e*) (see Fig. 5). If R'' is consistent and is not already in *Reach*, then it is recorded in *Reach* and queued in *Unvisited* for further exploration of its successors. The procedure loops until all regions in the queue have been explored. Finally, the regions in *Reach* are labeled according to the labeling algorithm **Label_Reach**(*Reach*, ϕ) (see Fig. 6), where ϕ is a TCTL formula, such that all regions in *Reach* satisfy ϕ. The procedure finally outputs the label that has been assigned to the initial region, R_{init}.

As detailed in Fig. 5 (**Successor_Region**()), the successor region is constructed as follows. Given a region *R* and an out-going transition *e*, the successor region R' is constructed by first advancing (**Advance**()) all clock values till it satisfies the triggering condition (*e.Trigger*) of *e*, while at the same time still satisfying the clock condition *R*, *R.ClockCond*. This first step gives an intermediate symbolic condition *R′.ClockCond* for the successor region R'. Second, the clock resets in *e.Assign* are applied to *R′.ClockCond* by **Assign**(). Third, the clock conditions of all sub-regions of

R′ have also to be satisfied by *R′.ClockCond*. Finally, the discrete variable values are assigned to *R.DvarCond* to obtain the new symbolic condition *R′.DvarCond*. In this way, both the clock and discrete variable symbolic conditions of the successor region *R′* are thus computed.

```
Successor_Region($R$, $e$)
region $R$;
transition $e$;
{
    $R'$ = New_Region();
    $R'.ClockCond$ = Advance($R.ClockCond$) $\wedge$ $e.Trigger \wedge R.ClockCond$;
    $R'.ClockCond$ = Assign($R'.ClockCond$, $e.Assign$);
    $R'.ClockCond = R'.ClockCond \wedge (\Lambda_i R'.SubRegion_i.ClockCond)$;
    $R'.DvarCond$ = Assign($R.DvarCond$, $e.Assign$);
    Return $R'$;
}
```

Fig. 5 Successor Region Function

The labeling algorithm, **Label_Region**(), is presented in Fig. 6. This algorithm assigns a label, $L(R, \phi)$, to each region, *R*, in the set of regions *RSet*. The label indicates if the region *R* satisfies ϕ. This labeling is computed as follows. For a mode predicate (see Definition 5), the label is *true* if the region satisfies the mode predicate and it is *false* otherwise. For a TCTL path formula, ϕ, the label is computed recursively according to the semantics of the formula.

```
Label_Region($RSet$, $\phi$)
set of region $RSet$;
TCTL formula $\phi$;
{
    For each $R \in RSet$, calculate recursively the label of $R$, $L(R)$, as follows.
    case $\phi = x \sim c$: $L(R, \phi)$:= true, if $x \sim c$ is true in $R$; false otherwise;
    case $\phi = x - y \sim c$: $L(R, \phi)$:= true, if $x - y \sim c$ is true in $R$; false otherwise;
    case $\phi = d \sim c$: $L(R, \phi)$:= true, if $d \sim c$ is true in $R$; false otherwise;
    case $\phi = \eta_1 \wedge \eta_2$: $L(R, \phi)$:= true, if both $\eta_1$, $\eta_2$ are true in $R$; false otherwise;
    case $\phi = \eta_1$: $L(R, \phi)$:= true, if $\eta_1$ is false in $R$; false otherwise;
    case $\phi = \exists\Diamond\phi' \mathbf{U}_{\sim c} \phi''$: $L(R, \phi)$:= true, if there is a successor $R'$ of $R$ such that
        $L(R', \phi'')$ is true, there is a path, $\pi$, from $R$ to $R'$ such that for all regions
        $R''$ along $\pi$, $L(R'', \phi')$ is true, and $time_\pi(R, R') \sim c$ is true; false otherwise;
    Similarly, for the other cases:
        $\phi = \exists\Box\phi' \mathbf{U}_{\sim c}\phi''$, $\phi = \forall\Diamond\phi' \mathbf{U}_{\sim c}\phi''$, and $\phi = \forall\Box\phi' \mathbf{U}_{\sim c}\phi''$.
}
```

Fig. 6 Label Region Function

Model checking based tools can be mostly found in academia. Verification tools that can be used to specify and verify real-time systems include UPPAAL [85], Kronos [86], SGM [87], [88], NuSMV [89], RED [90], XTL [91], and several others. Besides the above-presented symbolic model checking procedure, there are also process algebra based [92] and logic-based [93] verification for real-time systems. As for hardware-software coverification, there are also some recently proposed works on it [81], [82], [94].

6. Conclusion and Future Work

At the turn of a new century, computer technology is no long confined in the laboratories of academia and research institutes. In the last few years, the world has experienced a burgeoning widespread increase of embedded systems in intelligent appliances and high-assurance systems, which are mostly *real-time*. Software and hardware standards in modeling and programming languages are all being extended to cover the realm of *real-time* domain. Some examples include real-time UML, real-time Java, and real-time CORBA. Real-time constraints have permeated from highly advanced systems, such as nuclear reactors and spacecrafts, to everyday-life systems such as telecommunications, transportation systems, and home appliances. Real-time constraints have even entered the wireless technology such as the *Bluetooth technology*, which allows users to make effortless, wireless and instant connections between various communication devices, such as mobile phones and desktop and notebook computers. Since it uses radio transmission, transfer of both voice and data is in real-time. This article comes at a time where real-time constraints are here to stay, both in academy and in industry, for a very long period into the future of computer science history. Real-time constraints have been specified, modeled, designed, and verified in this article. This introductory material did not intend to be exhaustive and the technology is still developing! A major future work is the integration of the Internet with real-time constraints. *Real-time Internet* is still a dream, though real-time networking has already matured to some stage today with the use of *Video-On-Demand* systems and other real-time multimedia applications and systems. Another major breakthrough that most information technologists are awaiting for is *gigabit real-time wireless*. This is currently a dream, too. Nevertheless, progressive works are being carried out with an ambitious goal. Although the verification of real-time systems has seen some breakthroughs through the automatic model-checking procedure, some more efficient methods, either improved model-checking or other formal methods, are required to really attack the large exponential state-spaces of complex real-time systems.

References

[1] D. K. Hammer, L. R. Welch, and O.S. van Roosmalen, "A taxonomy for distributed object-oriented real-time systems," *ACM OOPS Messenger*, Vol. 7, No. 1, January 1996.

[2] Y. Brave and M. Heymann, "Formulation and control of real-time discrete event processes," In *Proceedings of 27th Conference on Decision and Control*, December 1988.

[3] C. H. Golaszewski and P. J. Ramadge, "On the control of real-time discrete event systems," In *Proceedings of 23rd Conference on Information Sciences and Systems*, pp. 98 – 102, March 1989.

[4] J. S. Ostroff, "Synthesis of controllers for real-time discrete event systems," In *Proceedings of 28th Conference on Decision and Control*, pp. 138-144, December 1989.

[5] J. S. Ostroff and W. M. Wonham, "A framework for real-time discrete event control," *IEEE Transactions on Automatic Control*, Vol. 35, No. 4, pp. 386-397, April 1990.

[6] H. Wong-Toi and G. Hoffmann, "The control of dense real-time discrete event systems," *Technical Report* STAN-CS-92-1411, Stanford University, 1992.

[7] R. Alur, "Techniques for automatic verification of real-time systems," *Technical report* STAN-CS-91-1378, Department of Computer Science, Stanford University, CA, August 1991, Ph.D. Thesis.

[8] H. A. Hansson, H. W. Lawson, M. Stromberg, and S. Larsson, "BASEMENT: A distributed real-time architecture for vehicle applications," *Real-Time Systems*, Vol. 11, No. 3, pp. 223-244, Kluwer Academic Publishers, 1996.

[9] J. Wang, *Timed Petri Nets — Theory and Application*, Kluwer Academic Publishers, USA, 1998.

[10] C. Ramamoorthy and G. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, pp. 440-449, 1980.

[11] P. Merlin and D. Farber, "Recoverability of communication protocols – Implication of a theoretical study," *IEEE Transactions on Communications*, pp. 1063-1043, September 1976.

[12] R. Sloan and U. Buy, "Reduction rules for time Petri nets," *Acta Informatica*, Vol. 33, pp. 687-706, 1996.

[13] J. J. P. Tsai, S. J. Yang, and Y.-H. Chang, "Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications," *IEEE Transactions on Software Engineering*, Vol. 21, No. 1, pp. 32-49, January 1995.

[14] R. Milner, *Communication and Concurrency*, Prentice-Hall, London, UK, 1989.

[15] F. Moller and C. Tofts, "A temporal calculus of communication systems," in

Proceedings of CONCUR'90 (*Concurrency Theory*), J. Baeten and J. Klop, eds., Vol. 458 of Lecture Notes in Computer Science, pp. 401-415, August 1990, Springer Verlag.

[16] W. Yi, "CCS + time = an interleaving model for real-time systems," in Automata, Languages, and Programming (ICALP'91), J. L. Albert, B. Monien, and M.R. Artalejo, eds., Vol. 510 of Lecture Notes in Computer Science, pp. 217-228, July 1991, Springer Verlag.

[17] Y. Ishikawa, H. Tokuda, and C. Mercer, "Object-oriented real-time language design: constructs for timing constraints," *Proceedings of OOPSLA/ECOOP*, ACM, October 1990.

[18] H. Tokuda, T. Nakajima, and P. Rao, "Real-time MACH: towards a predictable real-time system," *Proceedings of the USENIX MACH Workshop*, 1990.

[19] Object Management Group, *The Unified Modeling Language Specification*, Nov. 1999, http://www.omg.org.

[20] B. P. Douglass, *Real-Time UML*, Addison Wesley Publishing Company, Nov. 1999.

[21] B. P. Douglass, *Doing Hard Time*, Addison Wesley Publishing Company, May 1999.

[22] Bollella et al. (Real-Time for Java Experts Group), *The Real-Time Specification for Java*, Addison Wesley Publishing Company, June 2000.

[23] Object Management Group, *The Common Object Request Broker*: *Architecture and Specification*, 2.3 ed., June 1999.

[24] B. Selic, "A generic framework for modeling resources with UML," *IEEE Computer*, pp. 64-69, June 2000.

[25] E. Kligerman and A.D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 9, pp. 941-949, September 1986.

[26] A. S. Grimshaw, A. Silberman, and J.W.S. Liu, "Real-Time Mentat: A data-driven, object-oriented system," *Proceedings of IEEE Globecom*, Dallas, Texas, pp. 141-147, November 1989.

[27] V. Nirkhe, S. Tripathi, and A. Agrawala, "Language support for the Maruti real-time system," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1990.

[28] K. Kenney and K. Lin, "Building flexible real-time systems using the Flex language," *IEEE Computer*, Vol. 24, No. 5, pp. 70-78, May 1991.

[29] J. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, "Algorithms for scheduling imprecise calculations," *IEEE Computer*, Vol. 24, No. 5, pp. 58-68, May 1991.

[30] N. Gehani and K. Ramamritham, "Real-Time Concurrent C (C++): A language

for programming dynamic real-time systems," *Real-Time Systems*, Vol. 3, No. 4, pp. 377-405, December 1991.

[31] D. Niehaus, J.A. Stankovic, and K. Ramamritham, "A real-time system description language," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 1995.

[32] D. Niehaus, *Program Representation and Execution in Real-Time Multiprocessor Systems*, PhD dissertation, University of Massachusetts, Amherst MA, 1994.

[33] J.A. Stankovic, K. Ramamritham, D. Niehaus, M. Humphrey, G. Wallace, "The Spring System: Integrated support for complex real-time systems," *Technical Report* CS-98-18, University of Virginia, August 1998.

[34] M. Gergeleit, J. Kaiser, and H. Streich, "Checking timing constraints in distributed object-oriented programs," *ACM OOPS Messenger*, Vol. 7, No. 1, Special Issue on Object-Oriented Real-Time Systems, January 1996.

[35] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, pp. 890-904, September 1986.

[36] F. Jahanian and A. Mok, "A graph-theoretic approach for timing analysis and implementation," *IEEE Transactions on Computers*, Vol. C-36, No. 8, August 1987.

[37] F. Jahanian and A. Goyal, "A formalism for monitoring real-time constraints at run-time," *Proceedings of IEEE Fault-Tolerant Computing Symposium*, pp. 148-155, June 1990.

[38] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," *Proceedings of Real-Time Systems Symposium* (RTSS), pp. 74-83, December 1991.

[39] F. Jahanian, R. Rajkumar, and S. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems*, Vol. 7, No. 3, pp. 247-274, November 1994.

[40] M. Potkonjak and W. Wolf, "A methodology and algorithms for the design of hard real-time multi-tasking ASICs," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, No. 4, pp. 430-459, October 1999.

[41] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A.H.G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of Discrete Mathematics*, Vol. 5, pp. 287-326, 1979.

[42] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and company, New York, 1979.

[43] C.-L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real time environment," *Journal of the Association for Computing Machinery*, Vol. 20, pp. 41-61, January 1973.

[44] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm:

exact characterization and average case behavior," *Proceedings of the Real-Time Systems Symposium*, pp. 166-171, December 1989.

[45] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," *Proceedings of the Real-Time Systems Symposium*, pp. 201-209, December 1990.

[46] M. Harbour, M. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Transactions on Software Engineering*, Vol. 20, January 1994.

[47] L. Sha and J. Goodenough, "Real-time scheduling theory and Ada," *IEEE Computer*, Vol. 23, pp. 53-62, April 1990.

[48] M. Harbour, M. Klein, and J. Lehoczky, "Fixed priority scheduling of periodic tasks with varying execution priority," Proceedings of the Real-Time Systems Symposium, pp. 116-128, 1991.

[49] L. Sha, M. Klein, and J. Goodenough, "Rate monotonic analysis for real-time systems," in Foundations of Real-Time Computing: Scheduling and Resource Management, pp. 129-155, Kluwer Academy Publishers, New York, 1991.

[50] K. Tindell, A. Burns, and A. Wellings, "Mode changes in priority pre-emptively scheduled systems," *Proceedings of the Real-Time Systems Symposium*, pp. 100-109, 1992.

[51] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and analysis of fixed priority scheduler," *IEEE Transactions on Software Engineering*, Vol. 19, pp. 920-934, September 1993.

[52] J. Rabaey, C. Chu, P. Hoang, and P. Potkonjak, "Fast prototyping of datapath-intensive architectures," IEEE Design and Test of Computers, Vol. 8, No. 2, pp. 40-51, 1991.

[53] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," *Proceedings of the Design Automation Conference*, ACM Press, pp. 805-810, 1999.

[54] J. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*, Ph.D. dissertation, UC Berkeley, 1993.

[55] F. Theon et al, "Intellectual property re-use in embedded system codesign: an industrial case study," Proceedings of the *International System Synthesis Symposium*, 1995.

[56] B. Lin, "Software synthesis of process-based concurrent programs," *Proceedings of Design Automation Conference*, pp. 502-505, ACM Press, June 1998.

[57] B. Lin, "Efficient compilation of process-based concurrent programs without run-time scheduling," *Proceedings of the Conference on Design And Test in Europe* (DATE), pp. 211-217, February 1998.

[58] X. Zhu and B. Lin, "Compositional software synthesis of communicating processes," *Proceedings of the International Conference on Computer Design*, pp.

646-651, October 1999.

[59] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, A. Sangiovanni-Vincentelli, “Task generation and compile-time scheduling for mixed data-control embedded software,” Proceedings of Design Automation Conference, pp. 489-494, ACM Press, 2000.

[60] K. Onaga, M. Silva, T. Watanabe, “On periodic schedules for deterministically timed Petri net systems,” *Proceedings of the 4th International Workshop on Petri Nets and Performance Models* (PNPM’91), pp. 210-215, 1991.

[61] F. Qadri and A. Robbi, “Timed Petri nets for flexible manufacturing cell design,” *Proceedings of the IEEE International Conference on Humans, Information, and Technology*, Vol. 2, pp. 1695-1699, 1994.

[62] J. Siddiqi, Y. Chen, and A. Robbi, “A timed Petri net simulation tool,” *Proceedings of the 9th International Conference on CAD/CAM, Robotics, and Factories of the Future*, August 1993.

[63] W. M. Zuberek, “Composite schedules of manufacturing cells and their timed Petri net models,” *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 4, pp. 2990-2995, 1996.

[64] M. Di Natale, A. Sangiovanni-Vincentelli, and F. Balarin, “Task scheduling with RT constraints,” *Proceedings of Design Automation Conference*, pp. 483-488, ACM Press, 2000.

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

[66] P.-A. Hsiung, “RTFrame: An object-oriented application framework for real-time applications,” *Proceedings of the 27th International Conference on Technology of Object-Oriented Languages and Systems* (TOOLS'98), pp. 138-147, IEEE Computer Society Press, September 1998.

[67] W.-B. See and S.-J. Chen, “Object-oriented real-time system framework,” in *Domain-Specific Application Frameworks*, M.E. Fayad and R.E. Johnson, eds., Chapter 16, pp. 327-370, John Wiley, 2000.

[68] D. C. Schmidt and F. Kuhns, “An overview of the real-time CORBA specification,” *IEEE Computer*, Vol. 33, No. 6, pp. 56-63, June 2000.

[69] Sun Microsystems, “RMI specifications and tutorials,” http://java.sun.com/products/jdk/1.2/docs/guide/rmi/.

[70] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. McRoberts, *Real-Time Unix Systems, Design and Application Guide*, Kluwer Academic Publishers, Boton, M.A., 1991.

[71] B. Gallmeister, *POSIX.4: Programming for the Real World*, O’Reilly and Associates, 1995.

[72] P.-A. Hsiung, “CMAPS: A cosynthesis methodology for application-oriented parallel systems,” *ACM Transactions on Design Automation of Electronic*

Systems, Vol. 5, No. 1, pp. 51-81, January 2000.

[73] T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen, “A case study in hardware-software codesign of distributed systems — vehicle parking management system,” *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA’99), Vol. 6, pp. 2982-2987, June 1999.

[74] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[75] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, “HyTech: the next generation,” *Proceedings of the 16th Real-Time Systems Symposium*, pp. 56-65, IEEE Computer Society Press, 1995.

[76] Scientific and Engineering Software, Inc., *SES/Workbench User’s Manual*, Release 2.0, January 1991.

[77] T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen, “Hardware-software multi-level partitioning for distributed embedded multiprocessor systems,” to appear in *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 2001.

[78] P.-A. Hsiung, S.-J. Chen, T.-C. Hu, and S.-C. Wang, “PSM: An object-oriented synthesis approach to multiprocessor system design,” *IEEE Transactions on VLSI Systems*, Vol. 4, No. 1, pp. 83-97, March 1996.

[79] P.-A. Hsiung, C.-H. Chen, T.-Y. Lee, and S.-J. Chen, “ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems,” *ACM Transactions on Design Automation of Electronic Systems*, Vol. 3, No. 2, pp. 109-135, April 1998.

[80] P.-A. Hsiung, “POSE: A parallel object-oriented synthesis environment,” to appear in *ACM Transactions on Design Automation of Electronic Systems*, Vol. 6, No. 1, January 2001.

[81] P.-A. Hsiung, “Timing coverification of concurrent embedded real-time systems,” *Proceedings of the 7th IEEE/ACM International Workshop on Hardware/Software Codesign* (CODES’99), pp. 100-114, ACM Press, May 1999.

[82] P.-A. Hsiung, “Hardware-software timing coverification of concurrent embedded real-time systems,” IEE Proceedings on Computers and Digital Techniques, Vol. 147, No. 2, pp. 81-90, March 2000.

[83] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill, “Model checking for real-time systems,” *Proceedings of IEEE International Conference on Logics in Computer Science* (LICS), 1990.

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

[85] J. Bengtsson, F. Larsen, K. Larsson, P. Petterson, Y. Wang, and C. Weise, “New

generation of UPPAAL," *Proceedings of the International Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

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

[87] P.-A. Hsiung and F. Wang, "A state-graph manipulator tool for real-time system specification and verification," *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications* (RTCSA'98), October 1998.

[88] P.-A. Hsiung and F. Wang, "User-friendly verification," *Proceedings of the International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols & Protocol Specification, Testing, and Verification* (FORTE/PSTV'99), October 1999.

[89] A. Cimatti, F. Clarke, E. Giunchiglia, and M. Roveri, "NuSMV: A reimplementation of SMV," *Proceedings of the International Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

[90] W. Farn, "Region encoding diagram for fully symbolic verification of real-time systems," *Proceedings of IEEE Computer Software and Applications Conference* (COMPSAC'2000), IEEE CS Press, October 2000.

[91] R. Mateescu and H. Garavel, "XTL: A meta-language and tool for temporal logic model checking," *Proceedings of the International Workshop on Software Tools for Technology Transfer* (STTT'98), July 1998.

[92] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems," *ACM Transactions on Programming Languages and Systems*, Vol. 15, pp. 36 – 72, 1993.

[93] J. Gulmann, J. Jensen, M. Jorgensen, N. Klarlund, T. Rauhe, and A. Sandholm, "Mona: Monadic second-order logic in practice," *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'95), Vol. 1019 of Lecture Notes in Computer Science, Springer-Verlag, May 1995.

[94] J.-M. Fu, T.-Y. Lee, P-A. Hsiung, and S.-J. Chen, "Hardware-software timing coverification of distributed embedded systems," to appear in *IEICE Transactions on Information and Systems*, 2001.