Verification of Concurrent Client-Server Real-Time Scheduling Systems

Pao-Ann Hsiung, Farn Wang, and Yue-Sun Kuo
Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC
{eric,farn,yskuo}@iis.sinica.edu.tw

Abstract

Formally verifying complex real-time systems is a formidable task due to state-space explosions. We propose a formal framework in which not only is system concurrency modeled, but scheduling policies are also taken into consideration for verifying temporal properties. We show how the verification of concurrent real-time systems, modeled as client-server systems, using the model-checking approach can benefit from taking advantage of scheduling policies. *Integration of these two concepts, namely scheduling and model-checking, provides a reduction of the state space when compared to pure model-checking strategies. Our implementation and experiments corroborate the feasibility of our approach. Wide-applicability, significant state-space reduction, and several scheduling semantics are important features of our framework.*

Keywords: concurrent real-time client-server systems, model-checking, automata, scheduling algorithms, state-space reduction, verification,

1. Introduction

Complex real-time systems and applications such as avionics, vehicle controllers, and multimedia systems often have several *concurrent* parts or components. Each component may function according to some kind of *scheduling policy* [8, 7, 6, 4]. The correctness of a real-time system depends on the satisfaction of stringent real-time temporal constraints and is often verified using *model-checking* [1]. Verifying the correctness of concurrent real-time systems with several scheduling policies is a formidable task due to high system complexity, degree of concurrency, and state-space explosions. Engineering paradigms such as *scheduling* could be taken advantage of for verification. We propose a formal framework where such complex real-time systems can be verified by taking advantage of both model-checking and scheduling. Our implementation and experiments show its benefit by comparing with a naive verification effort, that is, pure model-checking. Experiment data show that significant reductions in state-space sizes can be reached.

In our framework, a concurrent real-time system is modeled as a *client-server scheduling system*, which consists of a set of concurrent *servers*, with scheduling policies specified, and a set of concurrent *client automata* which are basically automata extended with scheduling tasks specified in different modes. Due to concurrency among servers and among clients, there arise several issues which eventually lead to a great variety of execution semantics. We will classify 64 semantics, how they are implemented, and finally how their verifications differ from each other.

One major issue in precisely modeling such systems is the difficulty of compromising between two time scales : the *job-computation time unit* Δ_J and the *schedulability-check time unit* Δ_S. Δ_J is the time unit in which client tasks are specified. Δ_S is the time unit in which task schedulability checks done by servers are specified. Usually Δ_J is several orders of magnitude larger than Δ_S. In real-time system model-checking, very often the time and space complexities are proportional to the timing constants used in the system description. With such a big disparity between Δ_J and Δ_S, the complexity of scheduling system model-checking can easily grow beyond manageable.

If the time spent for schedulability check is ignored, the above issue will not occur, but then the model would not be realistic. There are two ways to handle such a situation.

- **Schedulability-Check Time Approximation:** One way is to use Δ_J, the job computation time unit, as a common base time unit and then approximate all schedulability check times into the next larger integer (if it is itself not an integer), expressed in job computation time units.
- **Resolved Time Units:** Another way is to use Δ_S, the schedulability-check time unit, as a common base time unit such that jobs are terminated non-deterministically by the servers, which are now modeled by separate finite-state machines.

The first method was introduced and discussed in details in a related work of the authors [4]. The advantage of this method is that all automata models are uniform and verification straightforward. The disadvantage is that the model is not very accurate and not conforming to real systems.

For the second method of resolved time units, verification complexity is controlled by modeling each server by a separate automata. Each server keeps a record of all jobs that have been scheduled to run on the server and are currently running in the server. Jobs are then terminated non-deterministically by the servers and then clients are notified of job-termination by the servers. Clients may also move on to a sucessor mode (hence a different set of tasks) by first informing the servers to terminate their respective jobs. In this way, time units are resolved into the non-deterministic models and verification complexity decreased.

Another issue is who plays the active part in job termination. Either the clients or the servers can play the active part in terminating job executions. The passive part will have to be further characterized. All these will be explained in Section 3 with different semantics of a scheduling system.

A final issue is when exactly should the checking for schedulability of the tasks in a mode be performed. Two alternatives arise here, namely, (1) checking before an in-coming transition of the mode is taken, or (2) checking after an in-coming transition of the mode is taken. Several kinds of semantics related to schedulability checking are possible. These are discussed in Section 3. Following is a *Video-On-Demand* (VOD) system example of a concurrent client-server real-time system.

Example 1 : Video System
As illustrated in Fig. 1, there are two servers and two clients in a *Video-on-Demand* system. The two clients issue task service requests to both servers concurrently. The two servers check if requests are schedulable and then either acknowledge or reject the requests. One server for movies schedules with the *rate-monotonic* (RM) [8] scheduling policy while the other schedules with the *earliest deadline first* (EDF) scheduling policy [8].

Movie Server stores a set of movie files ready for access by clients under the rate-monotonic scheduling policy. *Commercials Server* stores a set of commercial files and work with the earliest-deadline first scheduling policy. As shown in Fig. 1, the clients are modeled by finite-state automata that are enhanced with scheduling tasks. In the figure, boxes represent different operational modes of the clients and the arrows represent transitions between modes.

Within each box, we specify tasks by a tuple (α, c, p, d, f) where α is the server identification, c is the computation time of the task within each period, p is the period for the task, d is the deadline for each instance of a task, and f specifies if task priorities are fixed ($f = 1$) or dynamic ($f = 0$). ||

In a practical client-server scheduling system, the transition-triggering conditions can be much more complex than in Example 1. In our implementation, we allow users

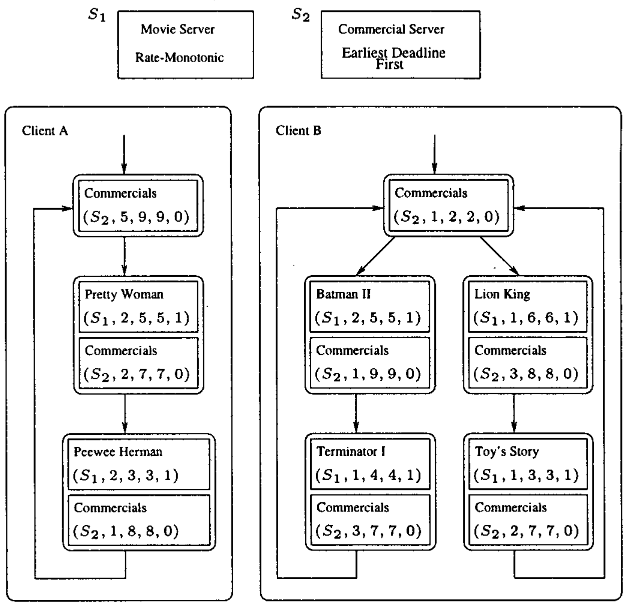

Figure 1. A video-on-demand system

to specify the following scheduling policies: EDF[8], RM-safe (RM policy based on Liu and Layland's number of $\log_e 2$ [8]), RM-arbitrary (RM policy based on Lehoczky's formula of $\Delta \log_e \left(\frac{\Delta+1}{\Delta}\right)$ [6] for arbitrary deadlines), RM-exact (RM policy based on Lehoczky, Sha, and Ding's schedulability test [7]), and MP (Liu and Layland's Mixed Priority scheduling in [8]).

The outline of this paper is as follows. Section 2 presents our formal client-server system model and describes how model-checking is used to verify a system. Section 3 presents all the different semantics possible in a client-server scheduling system. Section 4 describes our implementation of the model-checking approach using the popular HyTech tool. Section 5 shows the benefit of our approach using some application examples. Section 6 concludes the paper. In the following, we use $\mathcal{N}$ and $\mathcal{R}^+$ to denote the set of non-negative integers and the set of non-negative real numbers.

2. Client-Server Scheduling System Model

Modeling a real-time system as a client-server scheduling system, our target system of verification consists of a constant number m of *servers* that perform scheduling and a constant number n of *clients* that issue scheduling requests. A server adopts a scheduling policy. Each client is modeled with a client automaton such that the client issues different scheduling requests in various modes. On receiving a request for scheduling a set of tasks, a server decides whether the tasks are currently schedulable or not. In the following, we define a periodic task. It is assumed for uniformity and simplicity that all phasings are worst-case, that is $h_i = 0$ for all i as described in the previous section.

Definition 1 : A Periodic Task
A periodic task is a tuple $\phi = (\alpha, c, p, d, f)$, where α is the identification of the server on which the task is to be processed, c is the constant computation time of a job, p is the request period of a job, d is the deadline within which a job must be completed before the next job request occurs, and f specifies if the task must be scheduled using fixed priority [2, 5] or dynamic priority, that is, $f = 1$ for fixed priority and $f = 0$ for dynamic priority, $c \leq p$, $c \leq d$, and $c, p, d \in \mathcal{N}$, the set of nonnegative integers. ||

Notationally, we let $T_{\mathcal{H}}$ be the universal set containing all possible tasks in a system $\mathcal{H}$. We model the behavior of clients with finite-state machines or automata. It is assumed that the current mode of each client is broadcast to all the clients in the same system. The behavior of a client in each mode can be expressed through a *state predicate*, which is a boolean combination of propositions. Given a set of propositions P, a state predicate η of P has the following syntax.

$$\eta ::= false \mid r \mid \eta_1 \wedge \eta_2$$

where $r \in P$ and η_1, η_2 are state predicates. Let $B(P)$ be the set of all state predicates on P Given a set of propositions P, a client is modeled as follows.

Definition 2 : Client Automaton *(CA)*
A *Client Automaton* (CA) is a tuple $C = (M, m^0, P, \chi, \mu, E, \tau)$, where M is a finite set of modes. $m^0 \in M$ is the initial mode. P is a set of atomic propositions. $\chi : M \mapsto B(P)$ is a function that labels each mode with a condition true in that mode. $\mu : M \mapsto 2^{T_{\mathcal{H}}}$ maps each mode to a finite subset of tasks in $T_{\mathcal{H}}$. $E \subseteq M \times M$ is the set of transitions. $\tau : E \mapsto B(P)$ maps each transition to a triggering condition. ||

CA C starts execution at mode m^0. Transitions of the CA may be fired when triggering conditions are satisfied.

Definition 3 : Servers
A server is a tuple $\langle \alpha, \phi \rangle$ where α is the unique identification for the server and ϕ is the scheduling policy of the server. ||

Now with a set of servers and a set of client automata, we are ready to define a *scheduling system*.

Definition 4 : Scheduling systems
A *scheduling system* $\mathcal{H}$ is defined as a tuple $(\{S_1, S_2, \ldots, S_m\}, \{C_1, C_2, \ldots, C_n\}, P)$, where $\{S_1, S_2, \ldots, S_m\}$ is a set of servers, $\{C_1, C_2, \ldots, C_n\}$ is a set of client automata, and P is the set of atomic propositions used in $C_1, \ldots, C_n$. ||

Definition 5 : State
Given a system $\mathcal{H} = (\{S_1, \ldots, S_m\}, \{C_1, \ldots, C_n\}, P)$ with $C_i = (M_i, m_i^0, P, \chi_i, \mu_i, E_i, \tau_i)$, a state s of $\mathcal{H}$ is defined as a mapping from $\{1, ..., n\} \cup P$ to $\bigcup_{1 \leq i \leq n} M_i \cup \{true, false\}$ such that

- $\forall i \in \{1, ..., n\}$, $s(i) \in M_i$ is the mode of C_i in s;
- $\forall r \in P$, $s(r) \in \{true, false\}$ is the truth value of r in s; and ||

Definition 6 : Satisfaction of state predicate by a state
State predicate η is satisfied by a state s, written as $s \models \eta$ iff $s \not\models false$; $\forall r \in P, s \models r$ iff $s(r) = true$; and $s \models \eta_1 \wedge \eta_2$ iff $s \models \eta_1$ and $s \models \eta_2$ ||

Definition 7 : Mode Transition
Given a system $\mathcal{H} = (\{S_1, \ldots, S_m\}, \{C_1, \ldots, C_n\}, P)$ with $C_i = (M_i, m_i^0, P, \chi_i, \mu_i, E_i, \tau_i)$, and two states s and s', there is a *mode transition* from s to s' in $\mathcal{H}$, in symbols $s \rightarrow s'$, iff there is an $1 \leq i \leq n$ such that

- $(s(i), s'(i)) \in E_i$;
- $s(i) \models \tau_i(s(i), s'(i))$;
- for all $1 \leq j \leq n$ and $j \neq i$, $s(j) = s'(j)$; ||

3. Semantics of Scheduling Systems

In a client-server scheduling system, many different execution semantics are possible. Different semantics result in a difference in the verification results. The following are the main factors that result in different semantics.

- concurrency among clients and among servers,
- possible behaviors exhibited by clients and servers, e.g., passive and active, and
- possibility of performing schedulability check at different instants of time during execution.

In our verification framework, when the schedulability check time approximation model is used the servers are not modeled, only the client automata are translated into automata [4]. When the resolved time units model is used besides the client automata, each server is also modeled by an individual automaton.

When servers are also modeled, there are totally 64 different semantics modeled in our framework. We classify them into eight different factors, which are further grouped into two orthogonal classes, namely, *Type* and *Check* classes. There are eight semantics in the Type class, which differentiate between active and passive clients as well as servers. There are also eight semantics in the Check class, which differentiate between performing schedulability check before or after a transition. Since the two classes are orthogonal, we totally have $8 \times 8 = 64$ semantics possible. When servers are not modeled, the Type class of semantics do not exists, hence only 8 different semantics of the Check class are possible.

3.1. Type Class of Semantics

Recall that task scheduling requests are generated by clients, schedulability is checked by servers, and if schedulable, tasks are executed by servers. It remains to consider whether job executions are: (1) terminated by the servers and clients are notified, or (2) terminated by the clients and servers are notified. One side of the client-server system must play the active part (terminate jobs) and the other side must be passive (wait for termination notification). The following five factors are considered in this class of semantics.

- active and passive parts played by clients and servers,
- reliable and unreliable passive servers (see subsubsection 3.1.1),
- hard or soft passive clients (see subsubsection 3.1.2),
- time duration to wait for acknowledgments: zero, positive, forever, and
- entering error mode when no acknowledgment is received with a zero or a positive threshold time interval.

As shown in Fig. 2, eight different semantics can be distinguished by considering the above five factors.

3.1.1 Active-Clients & Passive-Servers

Clients actively terminate job executions by notifying servers. Servers are distinguished as *reliable* and *unreliable*. Reliable servers will eventually terminate jobs and send an acknowledgment to the clients. Unreliable servers may or may not send an acknowledgment to the clients. The following cases are possible.

- *Active-Clients & Passive Reliable Servers* (ACPRS): Clients here block-wait for acknowledgment of job termination from servers.
- *Active-Clients & Passive Unreliable Servers* (ACPUS): When job terminations are not guaranteed by unreliable servers, clients may either move on to its sucessor mode (ignoring the possibility that the jobs may have not terminated), or enter an error mode (indicating a dangerous state). Hence, we have the following cases: *Proceed Without Acknowledgment* (PWA), *Enter Error Mode* (EEM), and *Wait for Acknowledgment* (WFA). Each of the former two is further distinguished into Zero-Timeout and Positive-Timeout.

3.1.2 Passive-Clients & Active-Servers

Here, jobs are terminated actively by servers and clients are notified of job completion. Since clients are passive, they can be further distinguished into *hard* and *soft*. Hard clients move from one mode to another, that is perform a transition, when both the transition triggering conditions are satisfied and the currently executing jobs are complete or terminated. Soft clients may perform a transition without waiting for the current jobs to complete, only transition triggering conditions must be satisfied. Hence, we have two more semantics: *Passive Hard Client & Active Server* (PHCAS), and *Passive Soft Client & Active Server* (PSCAS).

3.2. Check Semantics

This class of semantics distinguish between when schedulability checks are performed, that is, either before a transition or after one. The following four factors affect execution semantics.

- Schedulability check of a set of tasks in a client mode is performed by the servers, but requests for such checks are generated by clients. Schedulability check could be done either *before* a mode is entered or *after*.
- Depending on whether these out-going transitions of a mode are considered (that is, triggering conditions tested) in *parallel* or *sequentially*, we can have different computation semantics.
- On the server side, where there are more than one servers schedulability check could be performed in parallel or sequentially.
- When a set of tasks in a mode are not schedulable, should the client have the choice of jumping to the next mode or must it keep trying in the current mode until the tasks are scheduled?

The first factor allows two different categories of semantics, namely, *Scheduling Check Before Transition* (SCBT) and *Scheduling Check After Transition* (SCAT). Within each category, four semantics are possible depending on the other factors listed above. Totally, eight different semantics are described in the following two subsubsections. In a previous related paper [4], the authors only proposed four different semantics and the implementations were also partial. Here, we present a more complete and organized classification of different semantics possible in a scheduling system and also implement all the semantics. The examples from [4] were rerun to obtain more complete reduction results.

3.2.1 Scheduling Check Before Transition (SCBT)

In this category of semantics, each client must be sure that all the tasks in a mode are schedulable by the servers before any transition that leads into that mode is taken, that is, triggering condition tested. In the following, we assume that a client has reached a mode m in some computation and there are at least two successor modes from m, labeled as m' and m'', with transitions e' and e'' leading from m to m' and m'', respectively. Based on second and third factors described above, a system has the following four semantics.

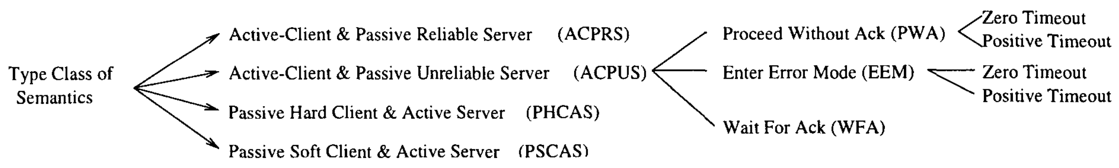
Type Class of Semantics
Active-Client & Passive Reliable Server (ACPRS)
Active-Client & Passive Unreliable Server (ACPUS)
Passive Hard Client & Active Server (PHCAS)
Passive Soft Client & Active Server (PSCAS)
Proceed Without Ack (PWA)
Enter Error Mode (EEM)
Wait For Ack (WFA)
Zero Timeout
Positive Timeout
Zero Timeout
Positive Timeout

Figure 2. Type Class of Semantics

(a) *Sequential-Test Sequential-Check* (STSC): A client checks the schedulability of the tasks in a successor mode m' by sequentially generating requests to the servers. After a positive schedulable response is obtained from each of the servers, indicating that the servers can concurrently schedule all the tasks in a successor mode m', then the client will test if the transition e' leading from m to m' is triggerable. Each out-going transition of mode m is test for triggerability in a sequential order.

(b) *Sequential-Test Parallel-Check* (STPC): In contrast to SCBT/STSC described above, schedulability check is performed concurrently by generating requests to all the servers simultaneously. The transitions are still tested sequentially for triggerability.

(c) *Parallel-Test Sequential-Check* (PTSC): Here, a client tests the triggering conditions of all out-going transitions of a mode m simultaneously, but schedulability check requests are generated to the servers sequentially.

(d) *Parallel-Test Parallel-Check* (PTPC): Here, both schedulability check requests generation to all the servers as well as testing of transition triggering conditions are performed in parallel.

3.2.2 Scheduling Check After Transition (SCAT)

Based on the third and fourth factors described at the start of Section 3, a scheduling system may have the following four semantics.

(a) *Strict-Scheduling Sequential-Check* (SSSC): Under this semantic, a client upon entering a mode after a transition, generates requests to the servers sequentially for schedulability check of the tasks in the mode. There are two cases:

- After each server has responded with a positive (schedulable) answer, the client starts the job instances associated with each task in the mode.
- If one server has returned a negative (unschedulable) answer, then the client re-starts the whole procedure of sequentially requesting each server for tasks schedulability check. A client *cannot* proceed on the next mode (along a triggerable transition) if the tasks in the current mode are not scheduled.

(b) *Strict-Scheduling Parallel-Check* (SSPC): Under SSPC semantic, a client upon entering a mode after a transition, generates request to all the servers simultaneously for schedulability check of the tasks associated with the mode. Strict scheduling is applied just as in SSSC (described above). Briefly, a client cannot proceed on to a next mode if the tasks in the current mode are not scheduled. It must keep trying by generating schedulability check requests to the servers.

(c) *Loose-Scheduling Sequential-Check* (LSSC): Under LSSC, a client upon entering a mode after a transition, generates requests sequentially to the servers for schedulability check of the tasks in the mode. If a server returns a negative (unschedulable) response, a client has the option of whether to repeat the whole procedure of schedulability check or proceed on to a next mode along some enable transition. This option if referred to as *loose scheduling*, that is, a client can non-deterministically choose whether to continue scheduling the tasks in a mode or proceed onward. Here, we restrict LSSC to at least checking for schedulability once for each mode.

(d) *Loose-Scheduling Parallel-Check* (LSPC): Under LSPC semantic, a client upon entering a mode after a transition, generates requests to all the servers simultaneously for schedulability check of the tasks specified in the mode. Loose scheduling is applied as in LSSC.

4. Implementation

The theoretical framework of a Client Server Scheduling System Model as presented in Section 2 has been implemented into a practical tool for verifying scheduling systems. The implementation mainly constitutes two parts: scheduling check time computation and translating a scheduling system description into a pure automaton.

Since schedulability check time computation had been dealt at length in [4], we will not go into the details here. Instead we will concentrate how the translation mechanism works. We developed a translator for translating the client-server scheduling system specification (in our own input language) to the HyTech specification. Although a scheduling system can be encoded using the HyTech input language, yet the specification would be very lengthy, tedious, and error-prone. Using our input language, the specification is short and compact and the translation is done systematically, thus avoiding any human-errors. For example, in the *real-time operating system* example (described in Section 5), using our input language the specification consisted of only 12 modes and 17 transitions, whereas the resulting translation into HyTech input language consisted of 58 modes and 416 transitions. Thus, the translator is a necessity for verifying scheduling systems.

HyTech [3] is a popular verification tool for systems modeled as linear hybrid automata. HyTech has been used to verify various different hybrid systems. Each client automaton is implemented as a linear hybrid automaton in HyTech and the analysis tool is used to verify our system. According to the different scheduling semantics, we have different types of implementation schemes.

4.1 Type Class of Semantics

Each client and each server is modeled by an automaton. A client automaton (Definition 2) is translated into a pure automaton (without task specifications in modes), while a server is modeled by a 3-modes automaton. Each client must lock the servers (either sequentially or in parallel), makes schedulability check requests to the servers, and then either enters an unschedulable error (retry) mode or goes into a job execution mode (RunJob).

- **ACPRS Implementation** Since the servers are reliable here, we simply model each server as in Fig. 3. A server synchronizes with a client on completion of each task. A mode transition in a CA is directly implemented as a location transition in HyTech.
- **ACPUS Implementation** Due to the servers being unreliable, each server automaton has one more location called *Ack*. In this way, the timeout (either zero or positive) for acknowledgment waiting could be modeled. An unreliable server automaton is shown in Fig. 4. To model the timeout for acknowledgment waiting in clients, one location and two transitions are used to implement each mode transition in a client automaton. As shown in Fig. 5 (Proceed Without Ack), a client waits in *Sync-Wait* either for zero time units or a prespecified time period. In Fig. 5 (Enter Error Mode), a client enters an error mode (on timeout). Further, in Fig. 5 (Wait for Ack), a client waits indefinitely for job termination acknowledgment.
- **PHCAS Implementation** An active server is modeled as shown in Fig. 6. It non-deterministically terminates a running ($R_i = 1$) task by setting R_i to 2. A client is thus notified that a task has completed ($R_i = 2$). On trigger satisfaction, a client then performs a mode transition, in synchrony with the servers. Here, a mode transition is again modeled as a single location transition except that there is a condition on task termination ($R_i = 2$) besides the original transition triggering condition. This is illustrated in Fig. 7
- **PSCAS Implementation** The active servers are modeled as in PHCAS. Passive soft clients perform a mode transition with or without mode task termination. Hence, each mode transition in a client automaton is modeled by two location transitions: one with only the original transition triggering condition and another with a boolean conjunction of the original transition triggering condition and job termination ($R_i = 2$). This is illustrated in Fig. 7.

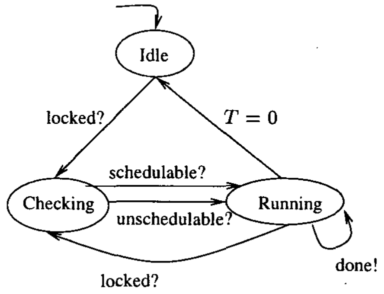

Figure 3. Passive Reliable Server

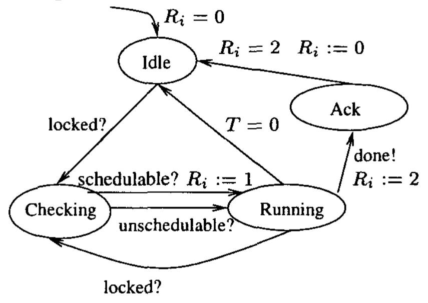

Figure 4. Passive Unreliable Server

4.2 Check Class of Semantics

For SCBT semantics, we have a transition-oriented implementation and for SCAT, we have a mode-oriented implementation. A *transition-oriented* implementation means that the implementation of a transition is complex (needs more than one HyTech location) while that of a mode is simple (implemented by a single HyTech location). Similarly, a *mode-oriented* implementation means that a mode implementation needs more than one HyTech location while that of a transition requires only a single HyTech transition. SCBT and SCAT implementations are almost the same as that given in [4] and hence omitted here.

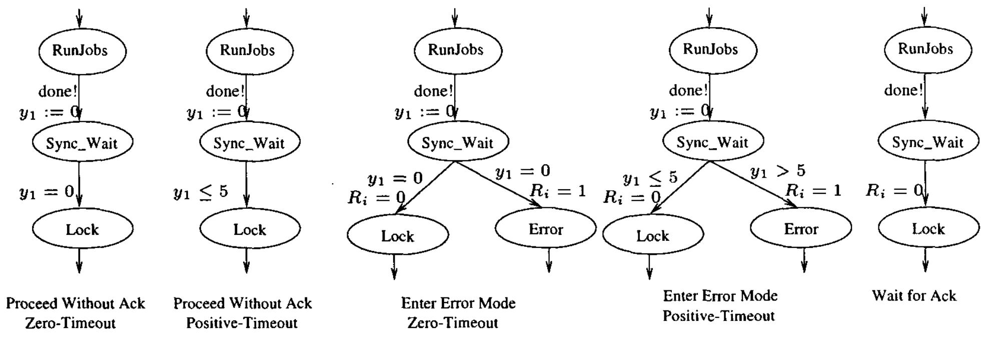
RunJobs
done!
$y_1 := 0$
Sync_Wait
$y_1 = 0$
$y_1 \leq 5$
$y_1 > 5$
$R_i = 0$
$R_i = 1$
Lock
Error
Proceed Without Ack Zero-Timeout
Proceed Without Ack Positive-Timeout
Enter Error Mode Zero-Timeout
Enter Error Mode Positive-Timeout
Wait for Ack

Figure 5. Active Clients

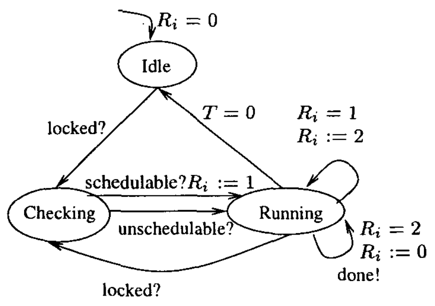

Figure 6. Active Server

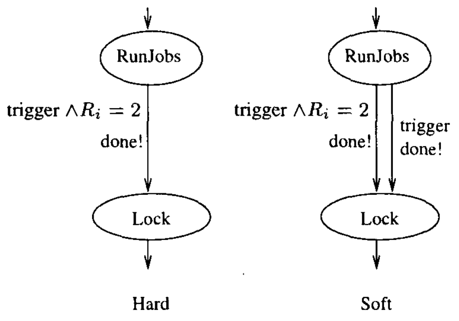

Figure 7. Passive Clients

5. Application Examples

To illustrate the generality of our approach, we demonstrate the benefits of three different types of systems: a hardware system such as a video-on-demand (VOD) system, a software system such as a real-time operating system (RTOS), and an agent system such as a package delivery system (PDS).

There are two servers in the video examples (just as in Fig. 1). The movie server schedules tasks with the rate-monotonic (safe) policy, while the commercial server does so with the earliest-deadline first policy. For the real-time OS example [4], there are four servers: OS kernel, display, memory, and printer, which use rate-monotonic (safe), earliest-deadline first, rate-monotonic (arb), rate-monotonic (exact) policies, respectively, for scheduling the tasks. For the delivery system example, it is assumed that there are three delivery agents and four clients. The delivery agents must deliver packages to the clients according to scheduling policies: rate-monotonic (exact), earliest-deadline first, and mixed scheduling.

Two versions are given for each of the three kinds of systems. All the six examples (see [4]) were specified in our input language which was then automatically translated by our translator into the HyTech input language. Two sets of results have been obtained.

One set of results is obtained by running all examples under the ACPRS type semantics with different check semantics (SCBT). These results are tabulated in Table 1. Another set of results is obtained by running the VOD (Fig. 1) example under different Type semantics (ACPRS, ACPUS, ...). These results are tabulated in Table 2. Both sets of results show that our approach indeed reduces the total size of the system state space for verification as compared to the pure model checking approach. Here, pure model checking means that we do not take advantage of the scheduling algorithms and directly verify the systems which might contain a lot of unschedulable states.

From the first set of results, we can make the following observations. Significant reductions can be achieved in systems that have a heavy workload. With each type of example, either VOD or RTOS, it is observed that with a high complexity in the client automata (i.e., the number of modes and transitions) the SCBT implementation shows a larger benefit (i.e., a smaller state space size) compared to all the semantics of the SCAT implementation (not shown here). This is due to the stronger semantics of a transition not occuring before the tasks schedulability of its destination mode is checked. Comparing the two semantics of SCAT: SSS and LSS, in all the examples it is observed that strict

Table 1. Comparison of Pure Model Checking and Our Approach (SCBT)

Example	Number of regions (convex predicates)					
	STSC			STPC		
	P_{MC}	S_{MC}	%	P_{MC}	S_{MC}	%
VOD	3226	2602	**80**	122	73	**60**
VOD1	3874	2284	**59**	284	171	**60**
VOD2	1167	378	**32**	96	28	**29**
RTOS1	O/M			2656	1941	**73**
RTOS2	O/M			739	215	**29**
PDS1	O/M			4149	3134	**76**
PDS2	O/M			O/M		

STSC: *Sequential Test Sequential Check*, **STPC**: *Sequential Test Parallel Check*, *%*: S_{MC} / P_{MC}, O/M: *Out of Memory*

Table 2. Comparison of Different Type Semantics (under SCAT/LSPC and SCBT/STPC)

Type	P_{MC}	S_{MC}	%
ACPRS	110	68	**62**
ACPUS/PWA	385	205	**53**
ACPUS/EEM	218	145	**67**
ACPUS/WFA	218	145	**67**
PHCAS	222	152	**68.6**
PSCAS	584	324	**55.5**

Type	P_{MC}	S_{MC}	%
ACPRS	122	73	**60**
ACPUS/PWA	428	226	**53**
ACPUS/EEM	239	157	**66**
ACPUS/WFA	239	157	**66**
PHCAS	245	162	**66**
PSCAS	671	366	**55**

ACPRS: *Active-Client & Passive Reliable Server*, **ACPUS**: *Active-Client & Passive Unreliable Server*, **PWA**: *Proceed Without Acknowledgment*, **EEM**: *Enter Error Mode*, **WFA**: *Wait For Acknowledgment*, **PHCAS**: *Passive Hard Client & Active Server*, **PSCAS**: *Passive Soft Client & Active Server*, *%*: S_{MC} / P_{MC}

semantics shows a larger benefit with our approach. This is due to the stronger restriction in SSS of tasks required to be scheduled before the client can progress on. Thus, we can conclude that both theoretically and experimentally we have shown that SCBT has the strongest notion of schedulability and LSS of SCAT has the weakest notion with SSS of SCAT in-between SCBT and LSS.

From the second set of results, we can see that ACPRS had the smallest state-space size before and after reductions. We also note that the greatest reduction was obtained in ACPUS/PWA, irrespective of the check semantics used. This is due to the unreliable behavior of the servers that results in a lot of states being unschedulable (thus unreachable) and were thus eliminated by our approach, while they were still considered by the pure model-checking approach. Further, we also noted that ACPUS/EEM and ACPUS/WFA gave the same state-space representation size and reduction. This is because entering an error mode is semantically the same as waiting forever for an acknowledgment.

6. Conclusion

Model-checking, though a popular verification method, has yet to be made more efficient for verifying the current highly complex systems. We have shown how complex real-time systems can be easily verified using the popular model-checking approach if we model the complex system as a client-server scheduling system and then verify it. This approach is meaningful when we observe that almost all complex systems need some sort of scheduling so that the tasks can be executed consistently and efficiently. Our framework has been shown feasible through the implementation using our translator and the HyTech verification tool. Different semantics have been implemented and compared using several examples. Future work will consists of further utilizing engineering paradigms for model-checking.

References

[1] R. Alur and D. Dill. Model checking for real-time systems. In *5th IEEE Conference on Logics In Computer Science*, 1990.

[2] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Procs. IEEE Real-Time System Symposium*, pages 116–128, 1991.

[3] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Procs. IEEE Real-Time Systems Symposium*, pages 56–65, 1995.

[4] P.-A. Hsiung, F. Wang, and Y.-S. Kuo. Scheduling system verification. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Lecture Notes in Computer Science (LNCS)*, volume 1579, pages 19–33, March 1999.

[5] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Software Engineering*, 19:920–934, September 1993.

[6] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Procs. IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[7] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Procs. IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[8] C. Liu and J. Laylang. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.