Scalable Timing Analysis with Refinement

Nan Guan[1], Yue Tang[1], Jakaria Abdullah[2], Martin Stigge[2], and Wang Yi[1,2]

[1] Northeastern University, China
[2] Uppsala University, Sweden

Abstract. Traditional timing analysis techniques rely on composing system-level worst-case behavior with local worst-case behaviors of individual components. In many complex real-time systems, no single local worst-case behavior exists for each component and it generally requires to enumerate all the combinations of individual local behaviors to find the global worst case. This paper presents a scalable timing analysis technique based on abstraction refinement, which provides effective guidance to significantly prune away state space and quickly verify the desired timing properties. We first establish the general framework of the method, and then apply it to solve the analysis problem for several different real-time task models.

Keywords: Real-time systems, timing analysis, scalability, digraph real-time task model.

1 Introduction

A real-time system is often described by a collection of recurring tasks, each of which repeatedly activates workload with fixed periods [10]. The analysis problem of this simple task model has been well-studied and efficient techniques exist. The key idea is to identify the worst-case behavior of each single task, and the system-level worst-case behavior is composed by the local worst-case behaviors of individual tasks.

To meet the increasing requirements on functionality and quality of service, real-time systems become more and more complex. For example, the workload activation pattern of a task may change from time to time depending on the system state. A major challenge is that there is no single local worst-case behavior of each task. Several candidate behaviors of a task may be incomparable, and it is generally necessary to enumerate and analyze all the combinations of the candidate behaviors of all tasks to figure out which particular combination is the worst. This leads to combinatorial state space explosion. It has been proved that the analysis problem of even very simple task models is strongly coNP-hard [14], as long as each task has multiple candidate behaviors that can potentially lead to the system-level worst case. Existing analysis techniques for such systems all suffer serious combinatorial state space explosion and are highly non-scalable.

This paper presents a timing analysis technique based on refinement to address the above challenge. For each task, we construct a tree-like structural state

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 3–18, 2015.
DOI: 10.1007/978-3-662-46681-0_1

space, where each leave corresponds to a concrete behavior of the task and each node in the tree over-approximates its children. Then the analysis is performed with the tree structures of all tasks in a top-down manner, starting with the combination of roots, i.e., the most coarse approximation, and being iteratively refined by moving down to the leaves. This provides us effective guidance to significantly prune away state space and quickly find the exact system-level worst-case behavior. This method is applicable to timing analysis problems for a wide range of real-time task models. In this paper, we first establish the general framework of refinement-based analysis, then apply it to three different real-time task models, namely the rate-adaptive real-time task model [4], the digraph real-time task model [12] and its extension with synchronization. We also present a tool suit currently under development, which is used for complex real-time systems modeling and efficient analysis based on techniques presented in this paper.

2 Behaviors, Abstractions and Refinement

A *system* Sys consists of a finite number of components, $\mathsf{Sys} = \langle \mathsf{C}_1, \cdots, \mathsf{C}_n \rangle$. Each component is defined as a finite set of *concrete behaviors* over domain $\mathcal{D}$. Semantically, a component C_i is a subset of $\mathcal{D}$. We use $\pi_i, \pi'_i, \cdots \in \mathsf{C}_i$ to represent the *concrete* behaviors of component C_i. A *concrete system behavior* $\Pi = \langle \pi_1, \cdots, \pi_n \rangle$ is a combination of concrete behaviors of individual components, and thus system Sys is defined by a subset of domain $\mathcal{D}^n$.

We analyze the *performance* of the system (or a particular component in the system), which is defined over a set of performance metrics $\mathcal{P}$ that forms a total order $\langle \mathcal{P}, \trianglerighteq \rangle$. For two elements $\omega, \omega' \in \mathcal{P}$, $\omega \trianglerighteq \omega'$ means that performance ω is at least as bad as ω'. For example, if we use "worst-case response time" as the performance metrics, then the performance is defined over the real number set, $\mathcal{P} = \mathbb{R}$, and the total order relation $\trianglerighteq$ is the numerical comparison "$\geq$". Moreover, we use $\omega \vartriangleright \omega'$ to denote that performance ω is strictly worse than ω' (e.g., the numerical comparison "$>$"). Given a concrete system behavior Π, the evaluation function $\mathsf{Evl}(\Pi)$ returns the performance of interest for Π.

We aim at hard real-time systems, for which we are interested in the *worst-case* performance $\overline{\omega}$ of the system:

$$\overline{\omega} = \max_{\forall \Pi \in \mathsf{Sys}} \{\mathsf{Evl}(\Pi)\} \tag{1}$$

where "max" denotes the maximum element of a set according to total order $\trianglerighteq$, i.e., $p = \max(p, p')$ if and only if $p \trianglerighteq p'$. If the worst-case performance meets the required timing properties, the system is guaranteed to honor the timing constraints under any circumstance at runtime.

Directly using Equation (1) to calculate the worst-case performance, we shall enumerate $\prod_{i=1}^{n} |\mathsf{C}_i|$ different system behaviors, which is highly intractable except for very small task systems.

2.1 Abstraction Tree

We define a superset of the concrete behaviors of each task and conduct the analysis with these supersets. The supersets, though lead to a even larger state space, have certain structure that helps us to effectively prune away a large portion of the state space and quickly come to the desired worst-case behavior.

Formally, for each component defined by C_i in the system $\langle \mathsf{C}_1, \cdots, \mathsf{C}_n \rangle$, we construct a join-semilattice $\langle \widetilde{\mathsf{C}}_i, \succcurlyeq, \sqcup \rangle$, where $\widetilde{\mathsf{C}}_i$ is a superset of C_i within domain $\mathcal{D}$, i.e., $\mathsf{C}_i \subseteq \widetilde{\mathsf{C}}_i \subseteq \mathcal{D}$. We call elements in $\widetilde{\mathsf{C}}_i \setminus \mathsf{C}_i$ the *abstract behaviors* of C_i. The partial order $\succcurlyeq$ is defined as follows: For any $\pi_i, \pi_i' \in \mathcal{D}$, $\pi_i \succcurlyeq \pi_i'$ (called π_i *dominates* π_i') if and only if

$$\forall \langle \pi_1, \cdots, \pi_{i-1}, \pi_{i+1}, \cdots, \pi_n \rangle \in \mathcal{D}^n : \\ \mathsf{Evl}(\langle \pi_1, \cdots, \pi_i, \cdots, \pi_n \rangle) \trianglerighteq \mathsf{Evl}(\langle \pi_1, \cdots, \pi_i', \cdots, \pi_n \rangle)$$

The $\sqcup$ operator gets an upper bound of the two operands according to $\succcurlyeq$, and thus $p \sqcup q \succcurlyeq p$ and $p \sqcup q \succcurlyeq q$.

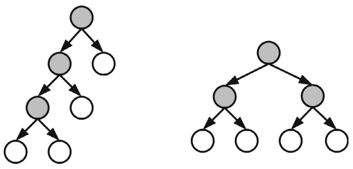

Fig. 1. Two possible abstraction trees of a component, where white nodes are concrete behaviors and grey nodes are abstract behaviors

Within the join-semilattice $\langle \widetilde{\mathsf{C}}_i, \succcurlyeq, \sqcup \rangle$, we can construct a binary *abstraction tree* according to the following rules:

– Each concrete behavior corresponds to one leave in the tree.
– The parent is the join (by $\sqcup$) of its two children.

Note that the abstraction tree of a component is in general *not* unique, i.e., the abstraction tree for a task can be constructed in different ways, as long as the above rules are satisfied. For example, Figure 1 shows two possible abstraction trees of a component with 4 concrete behaviors. Different abstraction trees may lead to different efficiency of the analysis procedure in the following.

2.2 Refinement-Based Analysis

The refinement-based analysis uses the abstraction tree of each component, and a prioritized working list $\mathcal{Q}$. Each element in the working list records a system behavior, i.e., a combination of behaviors from different abstraction trees. The priority is ordered according to the evaluation result of the system behavior: the worse performance the higher priority. The pseudo-code of the analysis algorithm is shown in Figure 2. The analysis procedure is performed in a top-down manner, starting with the combination of the root of each abstraction tree. Each time we take and remove the highest-priority system behavior, i.e., the first element, from $\mathcal{Q}$, and generate two new system behaviors by replacing one component's behavior by its children in the corresponding abstraction tree (the $refine(\Pi)$ routine). Then we evaluate these new system behaviors and add them to $\mathcal{Q}$, with proper order according to the evaluation results. This procedure iterates, until the highest priority element in $\mathcal{Q}$ is a combination of leaves. Then the evaluation result of this concrete system behavior is the desired worst-case performance.

```
1: $\mathcal{Q} \leftarrow \emptyset$
2: $\mathcal{Q}.add(\Pi, \mathsf{Evl}(\Pi))$
3: while $\Pi$ is abstract do
4:     $(\Pi', \Pi'') \leftarrow refine(\Pi)$
5:     $\mathcal{Q}.add(\Pi', \mathsf{Evl}(\Pi'))$
6:     $\mathcal{Q}.add(\Pi'', \mathsf{Evl}(\Pi''))$
7:     $\Pi \leftarrow \mathcal{Q}.pophead()$
8: end while
9: return $\mathsf{Evl}(\Pi)$
```

Fig. 2. Pseudo-code of the refinement-based analysis algorithm

In the refinement routine $refine(\Pi)$, Π is replaced by new system behaviors in which some component behavior is replaced by a node that is one step further from the root in the corresponding tree. So after a finite number steps of refinement, all elements in $\mathcal{Q}$ consist of only concrete behaviors (leaves), up on which the algorithm must terminate.

At any step of the algorithm, for a concrete system behavior Π that leads to the worst-case performance ($\mathsf{Evl}(\Pi) = \overline{\omega}$), there exists an element Π' in $\mathcal{Q}$ such that each behavior in Π' is an ascent of the corresponding concrete behavior in Π, so we have $\mathsf{Evl}(\Pi') \trianglerighteq \overline{\omega}$. On the other hand, due to the ordering rule of elements in $\mathcal{Q}$, the evaluation result ω of the head element in $\mathcal{Q}$ satisfies $\omega \trianglerighteq \mathsf{Evl}(\Pi')$, and thus $\omega \trianglerighteq \overline{\omega}$. When the algorithm terminates, the return value ω is the evaluation result of a concrete system behavior, which implies $\overline{\omega} \trianglerighteq \omega$. In summary, we have $\omega = \overline{\omega}$, i.e., the return value of the algorithm is the exact worst-case performance.

2.3 Early Termination

At any step during the execution of the algorithm in Figure 2, the evaluation result of the head of $\mathcal{Q}$ is an over-approximation of $\overline{\omega}$, and as the algorithm continues the result becomes more and more precise. In the design procedure, it is possible that the designer realized that the worst-case performance is guaranteed to satisfy the requirement even with an over-approximated estimation. For example, in schedulability analysis one can safely claim the system is valid when the over-approximate estimation of the worst-case response times are already smaller than the deadlines. In this case, we add the following codes

```
1: if $\mathsf{Evl}(\Pi)$ is satisfactory then
2:     return $\mathsf{Evl}(\Pi)$
3: end if
```

before line 4 in the algorithm of Figure 2 to let the algorithm terminate earlier.

3 Rate-Adaptive Tasks

In some real-time systems, task activation may depend on the system state. For example, in automotive applications, some tasks are linked to rotation, thus their activation rate is proportional to the angular velocity of a specific device [4,11,5]. To avoid overload, a common practice adopted in automotive applications is to let tasks execute less functionality with higher activation rates, which is formulated as the *rate-adaptive task system* in the following section.

3.1 Rate-Adaptive Task Model

We consider a task set τ of n independent rate-adaptive tasks (components) $\{T_1, T_2, \cdots, T_n\}$. Each task T_i has m_i different configurations, and it is characterized by a worst-case execution time (WCET) vector $e_i = \{e_i^1, \cdots, e_i^{m_i}\}$, a period vector $p_i = \{p_i^1, \cdots, p_i^{m_i}\}$, and a relative deadline vector $d_i = \{d_i^1, \cdots, d_i^{m_i}\}$. We assume tasks have constrained deadlines, i.e., $\forall a \in [1, m_i] : d_i^a \le p_i^a$. At runtime a task may use one of these m_i configurations, i.e., use $\langle e_i^a, p_i^a, d_i^a \rangle$, $a \in [1, m_i]$, and behaves like a regular periodic task with this particular parameter setting. Note that we do *not* consider dynamic transition among different configurations.

We use the static-priority scheduling algorithm to schedule jobs released by all tasks. Each task is assigned a static priority in a priori, and each of its released job inherits this priority. We assume tasks are ordered in decreasing priority order, i.e., T_i's priority is higher than T_j's iff $i < j$. At each time instant at runtime, the job with the highest priority among all the jobs that have been released but not finished yet is selected for execution. The performance metric we are interested in is the *worst-case response time*, i.e., the maximal delay between the release and the finishing time of a job, of each task with each configuration.

3.2 Analysis of Worst-Case Response Times

The worst-case response time of each task with each configuration can be analyzed independently. Therefore, without loss of generality, in the following we focus on the analysis of task T_i and only consider one configuration $\langle e_i, p_i, d_i \rangle$ (superscript omitted for simplicity).

Given a configuration $\langle e_j^a, p_j^a, d_i^a \rangle$, the maximal workload released by task T_j during time interval of size t can be precisely represented by function $rf_a(t)$ [8]:

$$rf_a(t) := \lceil t/p_j^a \rceil \times e_j^a$$

We use $rf_a(t)$ as a concrete behavior of task T_j (a leave in the abstraction tree) corresponding to configuration $\langle e^a_j, p^a_j, d^a_j \rangle$.

The partial order $\succcurlyeq$ among behaviors is defined as follows:

$$rf_a \succcurlyeq rf_{a'} \;\Leftrightarrow\; \forall t > 0, rf_a(t) \geq rf_{a'}(t)$$

The join operator $\sqcup$ is is defined as:

$$rf_a = rf_{a'} \sqcup rf_{a''} \;\Leftrightarrow\; \forall t > 0, rf_a(t) = \max(rf_{a'}(t), rf_{a''}(t))$$

We use $rf = \langle rf_{a_1} \cdots rf_{a_n} \rangle$ to denote a behavior of the system (only considering higher-priority tasks). The evaluation function $\mathsf{Evl}(rf)$ is defined as

$$\mathsf{Evl}(rf) = \min_{t>0} \left\{ t \;\middle|\; e_i + \sum_{rf_{a_j} \in rf} rf_{a_j}(t) \leq t \right\}$$

With the join operator and the evaluation function defined above, we can thus construct the abstraction tree of each task and perform refinement-based analysis by the algorithm in Figure 2, to calculate the worst-case response time of the task and the configuration under analysis.

4 Graph-Based Real-Time Tasks

Many real-time systems may have different workload activation states and switch among them at runtime. State transition systems can usually be modeled by graphs. In this section we consider a very general real-time workload representation, the Digraph Real-Time (DRT) task model [12], which models workload activation patterns by arbitrary directed graphs.

4.1 The DRT Task Model

A task system consists of n independent DRT tasks (components) $\{T_1, \cdots, T_n\}$. A task T is represented by a directed graph $G(T) = (V(T), E(T))$ with $V(T)$ denoting the set of vertices and $E(T)$ the set of edges of the graph. The vertices $V(T) = \{v_1, \cdots, v_n\}$ represent the types of all jobs that can be released by T. Each vertex v is labeled with a tuple $\langle e(v), d(v) \rangle$, where $e(v) \in \mathbb{N}$ denotes the worst-case execution time (WCET), $d(v) \in \mathbb{N}$ denotes the relative deadline. We implicitly assume the relation $e(v) \leq d(v)$ for all job types v. The edges of $G(T)$ represent the order in which jobs generated by T are released. Each edge $(u, v) \in E(T)$ is labeled with $p(u, v) \in \mathbb{N}$ denoting the minimum inter-release separation time between u and v. Deadlines are constrained, i.e., for each vertex u we have $d(u) \leq p(u, v)$ for all edges (u, v).

A job J is represented by a tuple (r, e) consisting of an absolute release time r and an execution time e. The semantics of a DRT task system is defined as the set of *job sequences* it may generate: $\sigma = [(r_0, e_0), (r_1, e_1), ...]$ is a job sequence if all jobs are monotonically ordered by release times, i.e., $r_i \leq r_j$ for $i \leq j$. A job sequence $\sigma = [(r_0, e_0), (r_1, e_1), ...]$ is generated by T if $\pi = (v_0, v_1, \cdots)$ is a path in $G(T)$ and for all $i \geq 0$:

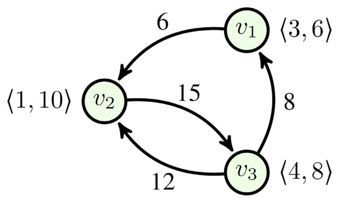

Fig. 3. A DRT task

1. $r_{i+1} - r_i \geq p(v_i, v_{i+1})$ and
2. $e_i \leq e(v_i)$

Combining the job sequences of individual tasks results in a job sequence of the task set.

Figure 3 shows an example to illustrate the semantics of DRT tasks. When the system starts, T releases its first run-time job by an arbitrary vertex. Then the released sequence corresponds to a particular direct path through $G(T)$. Consider the job sequence $\sigma = [(2,3),(10,1),(25,4),(37,1)]$ which corresponds to path $\pi = (v_1, v_2, v_3, v_2)$ in $G(T)$. Note that this example demonstrates the "sporadic" behavior allowed by the semantics of the DRT model. The first job in σ (corresponds to v_1) is released at time 2, and the second job in σ (v_2) is released 2 time units later than its earliest possible release time, while the job of v_3 and the second job of v_2 are released as early as possible.

We still use the static-priority scheduling algorithm to schedule jobs. The performance metric we are interested in is the *worst-case response time* of each vertex (job type) of each task.

4.2 Analysis of Worst-Case Response Time

Since the relative deadline of each vertex is no larger than the inter-release separation of all of its outgoing edges, in any feasible task system each vertex must be finished before the release of its successor vertices. Therefore, the analysis of each vertex within one task can be performed independently. In the following, we focus on the analysis of a particular vertex v of task T.

The response time of a vertex v is decided by the job sequences released by higher-priority tasks and the workload of itself. Therefore, to calculate the worst-case response time of v, conceptually, we should enumerate all the possible combinations of job sequences from each higher-priority task. Among the job sequences of a task, only the ones with minimal release separation and maximal execution demand (WCET) of vertices can possibly lead to the worst-case response time, each of which corresponds to a path in the task graph.

The workload of a path π can be abstracted with a *request function* [15], which for each t returns the maximal accumulated execution requirement of all jobs that π may release until time t (suppose the first job of π is released at time 0). For a path $\pi = (v_0, \cdots, v_l)$ through the graph $G(T)$ of a task T, we define its *request function* as

$$rf_\pi(t) := \max\{\, e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } p(\pi') < t \,\}$$

where $e(\pi) := \sum_{i=0}^{l} e(v_i)$ and $p(\pi) := \sum_{i=0}^{l-1} p(v_i, v_{i+1})$. We use $rf_\pi(t)$ as a concrete behavior of task T_j (a leave in the abstraction tree) corresponding to path π. Note that to analyze the worst-case response time of vertex v, we only need to look into time intervals of size up to $d(v)$, since otherwise v deems to be unschedulable. The number of different paths that can be generated by $G(T)$ in a bounded time interval is finite, so there are only finite number of concrete behaviors of a task.

The partial order $\succcurlyeq$ among behaviors is defined as follows:

$$rf_\pi \succcurlyeq rf_{\pi'} \;\Leftrightarrow\; \forall t \in (0, d(v)], rf_\pi(t) \geq rf_{\pi'}(t)$$

The join operator $\sqcup$ is is defined as:

$$rf_\pi = rf_{\pi'} \sqcup rf_{\pi''} \;\Leftrightarrow\; \forall t \in (0, d(v)], rf_\pi(t) = \max(rf_{\pi'}(t), rf_{\pi''}(t))$$

We use $rf = \langle rf_{\pi_1} \cdots rf_{\pi_n} \rangle$ to denote a behavior of the system (only considering higher-priority tasks). The evaluation function $\mathsf{Evl}(rf)$ is defined as

$$\mathsf{Evl}(rf) = \min_{t>0} \left\{ t \;\middle|\; e(v) + \sum_{rf_{\pi_i} \in rf} rf_{\pi_i}(t) \leq t \right\}$$

where $e(v)$ is the WCET of the analyzed vertex itself.

With the join operator and the evaluation function, we can thus construct the abstraction tree of each task and perform refinement-based analysis by the algorithm in Figure 2, to calculate the worst-case response time of v.

5 Digraph Tasks with Synchronization

In last two sections, tasks are assumed to be independent from each other, so it is easy to compose the system behaviors with individual component behaviors and can easily fit into the refinement-based analysis framework. In this section, we extend the DRT task model with synchronization. We show that the refinement-based analysis framework can also be applied to systems where strong inter-component dependency exists. The key is to construct proper behavior representation to capture interactions among components in the abstract domain.

5.1 DRT with Synchronization

Assume a finite number of communication channels $\{\mathsf{ch}_1, \cdots \mathsf{ch}_x\}$, through which tasks can send or receive signals and thus synchronize with each other. Sending a signal through channel ch is denoted by $\mathsf{ch}!$, while receiving a signal is denoted by $\mathsf{ch}?$. We call both sending and receiving operations *synchronization operations.*

We call ch! the *dual operation* of ch? and vice versa. We use $dual(a)$ to denote the dual operation of a, e.g. $dual(ch!) = ch?$. We use OP to denote the set of all synchronization operations in the system, which includes a pair of both sending and receiving operations for each channel and an null operation φ denoting that a vertex does not synchronize with others.

Each vertex v in a task graph is marked by a single synchronization operation (either sending or receiving), denoted by $\mathsf{op}(v)$. The release of a job by vertex v is synchronized with other jobs (from other tasks) marked with the dual operation of $\mathsf{op}(v)$, i.e., v can release a job only if another vertex marked with $dual(\mathsf{op}(v))$ is also eligible to release a job.

Figure 4 gives an example illustrating the semantics of the synchronization operations. The vertical dashed lines denote the time points at which the minimal inter-release separation constraints are satisfied. During the shadowed time intervals, a vertex waits for its dual operation to release a job. The vertical arrows denote actual job release times. Note that inter-release separation is counted relative to the actual release time of the previous job (the vertical arrows), rather than the time points when the previous inter-release separation constraint is satisfied (the dashed lines). It is allowed that at some time point a vertex v can synchronize with multiple vertices from different tasks. In this case, v non-deterministically chooses one of them for synchronization. In the example of Figure 4, at time 11 both task T_2 and T_3 can synchronize with T_1. The figure shows one possible running sequence where task T_1 chooses to synchronize with T_2, and the third vertex of T_3 continues to wait.

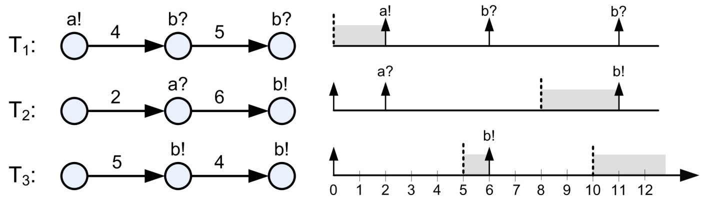

Fig. 4. An example illustrating the semantics of synchronization operations

5.2 Analysis of Worst-Case Response Time

Similar to Section 4, the analysis of each vertex within one task can be performed independently, so in the following we focus on the analysis of a particular vertex v of task T. Recall that in an independent DRT task system, the response time of a vertex only depends on higher-priority tasks and its own workload. However, when synchronization is added, the response time of a vertex also depends on lower-priority tasks, which can affect the execution of higher-priority tasks by synchronization operations.

We define the behavior corresponding to a path π in task graph $G(T)$ as a pair $bhv_\pi = \langle erf_\pi, pst_\pi \rangle$, where

$$erf_\pi(t,o) = \begin{cases} rf_\pi(t) & \text{if } o = \mathsf{op}(fv(\pi,t)) \\ 0 & \text{otherwise} \end{cases}, \quad pst_\pi(t,o) = \begin{cases} \{\pi\} & \text{if } o = \mathsf{op}(fv(\pi,t)) \\ \emptyset & \text{otherwise} \end{cases}$$

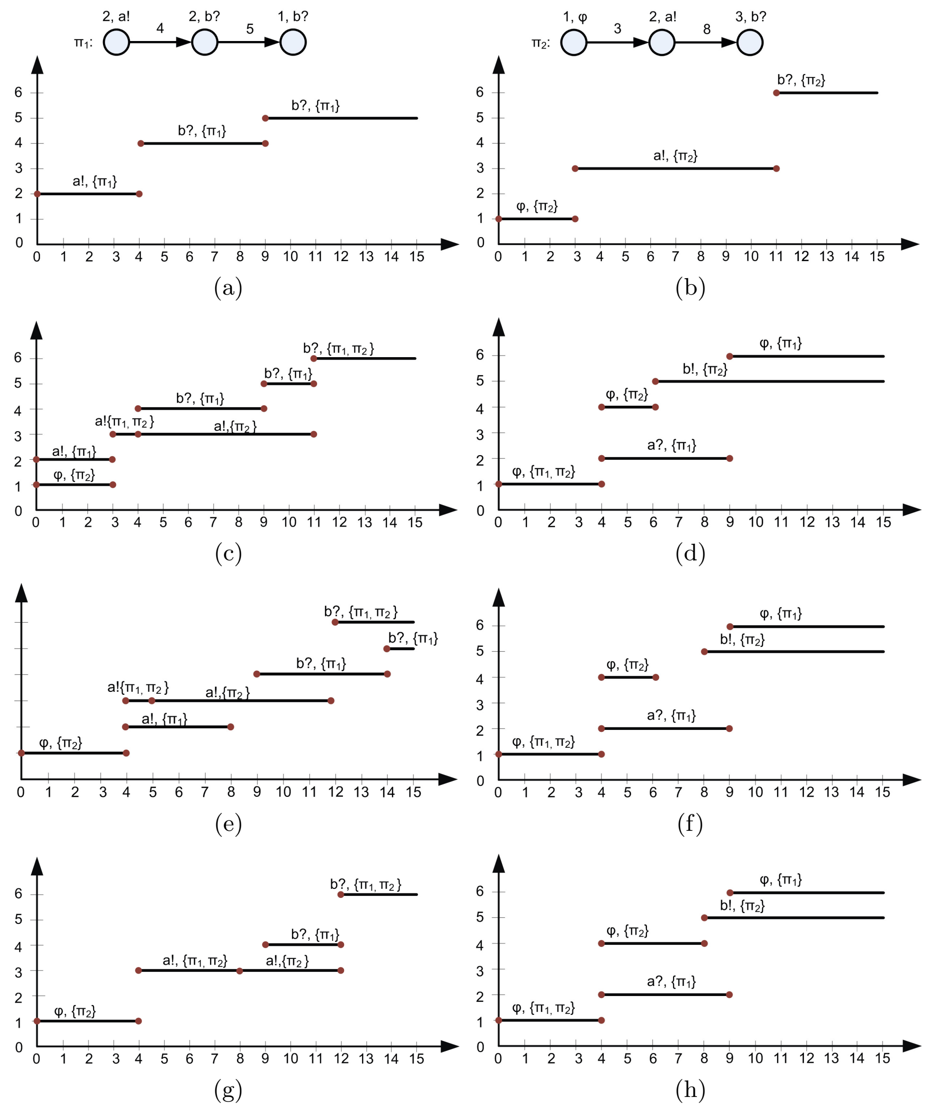

Fig. 5. An example illustrating the join of two behaviors and the update of system behaviors. (a) and (b) are two paths (of the same task) and the graphical representation of their behaviors; (c) shows the join of behaviors in (a) and (b); (d) is an (abstract) behavior of another task; (e) and (f) are the results of Update of (c) and (d); (g) and (h) are the results of fixing the inconsistency and redundancy in (e) and (f) by Set2Func.

$fv(\pi, t)$ returns the last vertex v_i along path $\pi = \{v_0, v_1, \cdots\}$ that is possible to release a job at time t (the path starts at time 0 and $t > 0$):

$$fv(\pi, t) = v_m \text{ s.t. } \left(\sum_{i=0}^{m} p(v_{i-1}, v_i) < t \wedge \sum_{i=0}^{m+1} p(v_{i-1}, v_i) \geq t \right)$$

where we let $p(v_{-1}, v_0) = 0$ for consistency. Figure 5-(a) and (b) show the graphical representation of the behaviors of two paths π_1 and π_2.

The partial order $\succcurlyeq$ among behaviors is defined as follows:

$$\langle erf_\pi, pst_\pi \rangle \succcurlyeq \langle erf_{\pi'}, pst_{\pi'} \rangle \Leftrightarrow$$
$$\forall t \in (0, d(v)], \forall o \in \mathsf{OP} : erf_\pi(t, o) \geq erf_{\pi'}(t, o) \wedge pst_\pi(t, o) \supseteq pst_{\pi'}(t, o)$$

The join operator $\sqcup$ is defined as:

$$\langle erf_\pi, pst_\pi \rangle = \langle erf_{\pi'}, pst_{\pi'} \rangle \sqcup \langle erf_{\pi''}, pst_{\pi''} \rangle \Leftrightarrow$$
$$\forall t \in (0, d(v)], \forall o \in \mathsf{OP} : \begin{cases} erf_\pi(t, o) = \max\left(erf_{\pi'}(t, o), erf_{\pi''}(t, o)\right) \\ pst_\pi(t, o) = pst_{\pi'}(t, o) \cup pst_{\pi''}(t, o) \end{cases}$$

Figure 5-(c) shows the resulting abstract behavior of joining the two concrete behaviors in Figure 5-(a) and (b). Graphically, a behavior $\langle erf_\pi, pst_\pi \rangle$ can also be represented by a set of segments, each segment $s = \langle start, end, o, pst \rangle$ having a start time *start*, an ending time *end*, a synchronization operation o and a path set *pst*. Let $bhv = \{bhv_1, \cdots, bhv_n\}$ be the function representation of a system behavior. Then *bhv* can be converted to the corresponding segment-set representation by $S = \mathsf{Func2Set}(bhv)$, and the inverse conversion is $bhv = \mathsf{Set2Func}(S)$.

```
1: $S \leftarrow \mathsf{Func2Set}(bhv)$
2: $\mathsf{Update}(S)$
3: $bhv \leftarrow \mathsf{Set2Func}(S)$
4: $\overline{erf} \leftarrow$ the subset of behaviors in $bhv.erf$ of higher priority tasks.
5: $R \leftarrow \min_{t>0}\{t | e(v) + sum(t)\}$, where
   $sum(t) \leftarrow \sum_{erf_T \in \overline{erf}} \left\{ \max_{\forall o \in \mathsf{OP}} \{erf_T(t, o)\} \right\}$
6: return $R$
```

Fig. 6. Pseudo-code of $\mathsf{Evl}(bhv)$

If *bhv* consists of only concrete behaviors, the evaluation with *bhv* can be performed by "simulating" the release and execution sequence of the corresponding paths, which will not be further discussed here. The interesting case is when *bhv* is an abstract system behavior, the evaluation function for which is defined by the algorithm in Figure 6. The release of vertices may wait for extra delay due to synchronization operations. If we use individual task behaviors to calculate the maximal possible workload of each task independently and sum them up as the total system workload (as in last section), the evaluation result will be very pessimistic. To improve evaluation precision with abstract behaviors, we use the Update function to also consider the extra release delay due to synchronization in abstract behaviors, the pseudo-code of which is shown in Figure 7.

```
1: $S' \leftarrow \emptyset$
2: **while** $S \neq S'$ **do**
3:     $S' \leftarrow S$
4:     **for all** $S_{T_i} \in S$ **do**
5:         **for all** $s \in S_{T_i} : s.o \neq \emptyset$ **do**
6:             $\Phi = \{ss \mid T_j \in \tau \setminus \{T_i\} \wedge ss \in S_{T_j} \wedge ss.o = dual(s.o)\}$
7:             **if** $\Phi = \emptyset$ **then**
8:                 $t' = +\infty$
9:             **else**
10:                $t' = \min_{ss \in \Phi}\{ss.start\}$
11:            **end if**
12:            $\triangle \leftarrow \max(t' - s.start, 0)$
13:            **for all** $s' \in S_{T_i} : s'.start \geq s.start \wedge s'.pst \subseteq s.pst$ **do**
14:                $s'.start = s'.start + \triangle$
15:                $s'.end = s'.end + \triangle$
16:            **end for**
17:        **end for**
18:    **end for**
19: **end while**
20: **return**
```

Fig. 7. Pseudo-code of Update(S)

Update(S) iteratively updates the abstract behavior of each task. At each step, the algorithm tries to shift a segment s to right by looking for the earliest eligible time point t' of $dual(s.o)$ in all other tasks. If t' is later than the start time of s, s should be shifted to start at t', and all later segments that depend on s in concrete behaviors also shift rightwards correspondingly. This procedure repeats until a global fixed point is reached. Figure 5-(e) and (f) show the resulting segment sets by applying the above procedure to the segment sets in Figure 5-(c) and (d).

Shifting some of the segments in S may lead to inconsistency and redundancy regarding the represented concrete behaviors. For example, in Figure 5-(e) there is no segment covering time interval $[3, 4)$. This is because segment $\langle 3, 4, a!, \{\pi_1, \pi_2\}\rangle$ is shifted to right. To resolve this, we shall extend segment $\langle 0, 4, \phi, \{\pi_2\}\rangle$ to cover $[3, 4)$. Also in Figure 5-(e), time interval $[13, 15)$ is covered by two segments both with $b?$ and path π_1. In this case, the lower segment is redundant and should be merged into the upper one. Function Set2Func addresses these inconsistencies and redundancies and transfer the segment sets to well-defined function representations of behaviors.

6 Experiments

In this section, we briefly report some experiment results with the DRT task model described in Section 4. Experiments are conducted with randomly generated DRT task sets. To generate a task, a random number of vertices is created with edges connecting them according to a specified branching degree. The following table gives details of the used parameter ranges, where p denotes inter-release separation between two vertices, e and d denotes the WCET and relative deadline of a vertex, respectively.

Table 1. Parameter ranges

vertices	out-degree	p	d/p	e/d
$[5, 10]$	$[1, 3]$	$[100, 300]$	$[0.5, 1]$	$[0, 0.07]$

Feasible task sets created by this method have sizes up to about 20 tasks with over 100 individual job types in total. For evaluating the effectiveness of the abstraction refinement scheme, we capture for each call to the refinement-base analysis how many system behaviors have been analyzed. We compare this number with the total number of system behaviors, i.e., all the combinations of individual task behaviors. This ratio indicates how much computational work the refinement scheme saves, compared to a naive brute-force style test.

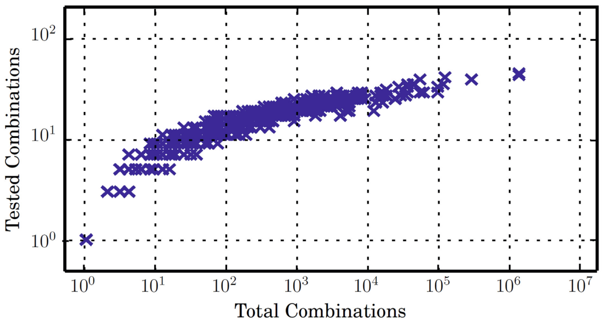

Fig. 8. Tested versus total number of system behaviors

We capture 105 samples and show our results in Figure 8. We see that the combinatorial abstraction refinement scheme saves work in the order of several magnitudes. More details of the experiments can be found in [13].

7 Tool

TIMES [1] is a tool suit for schedulability analysis of complex real-time systems, in which task systems are modeled with task automata [6] that are essentially timed automata [3], and the analysis problems are solved using the UPPAAL model checker [9]. The TIMES modeling language based on timed automata provides powerful expressiveness, but also limits the scalability of the tool due to the high complexity of verification problems for timed automata.

Currently we are developing a new version of TIMES based on the less expressive DRT task model and several scalable timing analysis techniques developed recently (including the refinement-based analysis in this paper). Note that a DRT task is in fact a task automaton with one clock, where only lower bounds on the clock are allowed to use in expressing timing constraints on job releases. The new TIMES is expected to have much higher analysis efficiency and can deal with large-scale realistic systems. The tool offers the following main features:

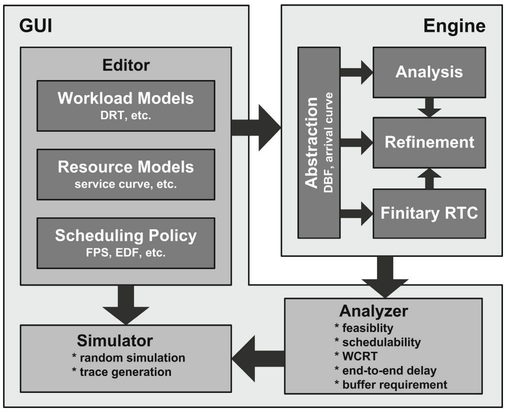

Fig. 9. Tool Architecture

- **Editor** to graphically model a system and the abstract behaviour of its environment. Workload is modeled as a set of DRT tasks, and different DRT tasks can synchronize with each other by communication channels and/or semaphores. System resource is modeled with a topology of processing and communication units, and each unit is associated with a service curve [16]. The users can choose the scheduling policy on each unit.
- **Simulator** to visualize the system behavior as Gant charts and message sequence charts. The simulator can be used to randomly generate possible execution traces, or alternatively the user can control the execution by selecting the transitions to be taken. The simulator can also be used to visualize error traces produced in the analysis phase.
- **Analyzer** to check various properties of the system model, including feasibility, scheduability, worst-case response time (WCRT), end-to-end delay and buffer requirement.

The tool architecture is depicted in Figure 9. The tool consists of two main parts, a Graphical User Interface (GUI) and an analysis engine. The GUI consists of editors, simulator and analyzer as described above, and uses XML to represent the system descriptions both internally and externally. The analysis engine consists of four parts. The *Abstraction* module transform the DRT workload models into abstract representations such as demand bound functions (DBF) [2] and request bound functions (RBF) [15]. The transformation is very efficient, based on the path abstraction technique proposed in [12]. The *Refinement* module is the core of the engine, which uses the framework in this paper to iteratively obtain tighter and tighter analysis results until the property of interest is proved/disproved. At each step of the analysis, it invokes either the *Analysis* module for traditional WCRT analysis and schedulability test, or invokes the *Finitary RTC* module for efficient system-wide performance analysis using Finitary Real-Time Calculus [7] in the presence of a distributed platform.

Acknowledgement. This work is partially supported by NSF of China (No. 61300022 and 61370076).

References

1. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES - A tool for modelling and implementation of embedded systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 460–464. Springer, Heidelberg (2002)
2. Baruah, S.K., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th Real-Time Systems Symposium (RTSS) (1990)
3. Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
4. Buttazzo, G.C., Bini, E., Buttle, D.: Rate-adaptive tasks: Model, analysis, and design issues. Technical Report (2013)
5. Davis, R.I., Feld, T., Pollex, V., Slomka, F.: Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In: the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2014)
6. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation (2007)
7. Guan, N., Yi, W.: Finitary real-time calculus: Efficient performance analysis of distributed embedded systems. Proceedings of the IEEE 34th Real-Time Systems Symposium (RTSS) (2013)
8. Joseph, M., Pandya, P.K.: Finding response times in a real-time system. The Computer Journal (1986)
9. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer, STTT (1997)
10. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (1973)
11. Pollex, V., Feld, T., Slomka, F., Margull, U., Mader, R., Wirrer, G.: Sufficient real-time analysis for an engine control unit with constant angular velocities. In: Design, Automation and Test Conference in Europe (DATE) (2013)

12. Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 71–80. IEEE (2011)
13. Martin, Stigge, N.G., Yi, W.: Refinement-based exact response-time analysis. In: the 26th EUROMICRO Conference on Real-Time Systems (ECRTS) (2014)
14. Martin Stigge and Wang Yi. Hardness results for static priority real-time scheduling. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS), pp. 189–198. IEEE (2012)
15. Stigge, M., Yi, W.: Combinatorial abstraction refinement for feasibility analysis. In: Procedings of the 34th IEEE Real-Time Systems Symposium (RTSS), pp. 340–349 (2013)
16. Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: Proc. Inti. Symposium on Circuits and Systems (2000)