A Proposal for Real-time Interfaces in SPEEDS

Purandar Bhaduri
Indian Institute of Technology Guwahati, India
Email: pbhaduri@iitg.ernet.in

Ingo Stierand
University of Oldenburg, Germany
Email: stierand@informatik.uni-oldenburg.de

Abstract—The SPEEDS project is aimed at making rich components models (RCM) into a mature framework in all phases of the design of complex distributed embedded systems. The RCM model is required to be expressive enough to cover the entire development process from requirements to code through design, and also capture both functional and non-functional aspects. In this paper we propose a language-based framework for real-time component interfaces in SPEEDS that is suitable at the ECU layer when a target processor has been identified, and WCET analysis done. We assume a discrete time model.

I. INTRODUCTION

While formal models for component interfaces have been investigated in recent years ([1], [2]), such models for real-time embedded systems are still in their infancy. As an example, there is a critical need for real-time interface models in the automotive industry, where original equipment manufacturers (OEMs) define the system requirements and architecture, while the actual development of various components are contracted to suppliers. Since the components provided by suppliers are black boxes, it is essential that the OEMs can integrate the components from a knowledge of their interfaces. However, for real-time embedded applications, such as in the automotive domain, it is not enough to model just the external interactions of components in order to compose them. Timing issues, such as periods, deadlines, jitter and end-to-end latencies must be modelled, along with other non-functional aspects of the behaviour of the component (e.g., reliability, fault-tolerance and power consumption).

The SPEEDS project [3], [4] is an attempt at making rich components models (RCM) [5] into a mature framework in all phases of the design of complex distributed embedded systems, such as those used in avionics and automotive systems. The RCM is required to be expressive enough to cover the entire development process from requirements to code through design, and also capture both functional and non-functional aspects. In this paper we propose a real-time component interface in SPEEDS that is suitable at the electronic control unit (ECU) layer when a target processor has been identified, and worst case execution time (WCET) analysis done. We assume a discrete time model.

Our interface model has two operations. First, two interfaces can be composed to form a compound interface. This allows the system integrator to derive the interface of a composed system from the interfaces of its constituent components. Second, an interface can be refined by comparing a more detailed specification against a more abstract one. An interface refines another, when it can safely be substituted for the latter in any context. For example, the abstract specification may require “task 2 must be started within 10ms after task 1 finishes”, and the detailed specification may refine this to “task 2 starts 5ms after task 1 finishes”.

The properties of incremental design and independent implementability are desirable features of an interface-based design method. The property of incremental design ensures that component interfaces can be composed into a subsystem in any order. Independent implementability is supported by preservation of parallel composition by interface refinement. This means that if the interfaces of the constituent components in a system are refined independently, the resultant system will refine the original one. The two operations of our real-time interface algebra support incremental design and independent implementability by satisfying the usual algebraic laws.

Our real-time interface model is a language based formalism, based on the control interfaces proposed by Weiss and Alur [6]–[8]. We assume a discrete time model, where time is divided into slots of pre-defined equal length. All scheduling related events, such as task arrivals, completions and preemptions, take place at these discrete time points. A schedule on a processor is described by an ω-word that describes the sequence of tasks that run in the discrete time slots. An ω-regular language describes the set of all legal schedules. The operation of composition of interfaces corresponds to substitution followed by intersection, and refinement is language inclusion. All these operations are regular and are decidable, albeit with high complexity [9]. Moreover, many real-time task models (such as periodic, periodic with jitter etc.) and scheduling strategies (fixed-priority, earliest deadline first etc.) can be described in this framework.

The main contribution of this paper is a proposal for enhancing the SPEEDS framework with the notion of a real-time interface. Currently in SPEEDS real-time requirements of components can be specified as timed automata. But unlike the modelling and analysis of behavioural properties, the current approach to analysing timing properties in SPEEDS is through global methods at system level, rather than at the component level. These approaches are based on holistic scheduling methods [10] for analysing timing properties rather than compositional assume-guarantee reasoning.

We provide a notion of contracts which are consistent with the hybrid automata based contracts in SPEEDS, and show how to combine them using an operation of contract composition. We abstract from functional aspects and restrict
expressiveness to partial ordering of events (such as task release and completion) and timing.

We consider the language-based interfaces described above as implementations of a component which take into account the resource constraints of a given platform, such as the execution time of a task. We define what it means for such an implementation to satisfy a contract by translating the assumptions and guarantees in a contract into the interface language. Roughly speaking, the satisfaction relation asserts that the implementation is a subset of the guarantees, when restricted to the set of assumptions. We show that the satisfaction relation (between implementations and contracts) is preserved by parallel composition, and this enables compositional reasoning about real-time properties.

II. REAL-TIME INTERFACES

A. Introduction

We assume that a set of real-time components are to be executed on a single processor. Each component has a number of tasks. The traditional way to specify a component interface would be to specify the timing characteristics of each task in the set (e.g., its period, deadline, execution time, etc.). However, this notion of interface is not compositional. Given two interfaces, each of which is individually schedulable on the processor, it’s not clear how to combine this information to deduce whether two components together are schedulable. This is because designers are free to use any scheduling algorithm inside their components to schedule the tasks, and these are not necessarily known to the system integrator.

Our component interfaces are based on the automata based interfaces of Weiss and Alur [6]–[8]. Similar ideas on automata based scheduling frameworks are also presented in [11]. The idea can be explained as follows. Consider a real-time component with two tasks 1 and 2, which are scheduled on a single processor in discrete slots of some fixed duration. A schedule for this component can be described by an infinite word over the alphabet \( \{0, 1, 2\} \), where 0 indicates the processor is idle, and 1 or 2 signifies the corresponding task is running during the slot. The interface of a component is then an \( \omega \)-language (see [9] for an introduction to the theory of \( \omega \)-languages) containing all legal schedules. It is the job of the component developer to ensure that the internal scheduling algorithm produces a schedule belonging to the set of legal schedules in the interface. This notion of component interface is compositional: composing two interfaces corresponds to language theoretic operations of substitution followed by intersection. Schedulability of a set of components on a single processor corresponds to checking the emptiness of their intersection.

Our interfaces can easily express commonly occurring timing requirements such as periodic tasks with or without jitter, etc. Modelling sporadic tasks requires some more machinery, as task release times have to be somehow recorded. Task dependencies and mutual exclusion constraints can be taken care of in the language theoretic setting by just specifying what the legal schedules are. We will describe abstraction techniques to deal with the issue of efficiency of representation of real-time interfaces in a separate paper. Ease of specification is also an issue, but tool support for translating from user-centric notations to the language of \( \omega \)-automata can alleviate the problem.

B. Related Work

Assume-guarantee interfaces for real-time systems have been investigated in [12]–[14] among others. The interface theory in [12] is based on the formalism of timed games, and extends the interface automata formalism of [1] to the timed case. The work in [13], [14] are based on Real-Time Calculus (see [15]), a framework based on a general event and resource model that can be used to derive hard upper and lower bounds of various performance criteria.

The paper [16] introduced an open environment for scheduling independently developed real-time applications. It is a hierarchical scheduling framework where each application can use any scheduling algorithm to schedule its tasks assuming a virtual processor of a certain speed. At a lower level the system uses the earliest deadline first (EDF) policy to schedule the applications. Exact schedulability conditions for such a scheduling framework was explored in [17], assuming the system scheduler knows the deadlines of each individual task in each of the applications. Note that such a hierarchical framework is not compositional in our sense, where the composition of a number of components can be treated as a single component.

Another hierarchical approach to scheduling with compositional analysis using interfaces has been proposed in [18], [19]. In these papers, an interface abstracts the collective real-time requirements of a set of periodic tasks into a single real-time periodic requirement. It is based on necessary and sufficient conditions for the schedulability of periodic tasks according to rate monotonic (using a response time analysis) and earliest deadline scheduling policies. Note that this line of work assumes very specific task and resource models, such as periodic and bounded delay. Moreover, the abstraction of a set of periodic tasks as a single periodic task naturally entails loss of information and leads to schedulability results that are too pessimistic.

C. Formalisation of Real-Time Interfaces

We formalise the definitions motivated above by defining an algebra of component interfaces below. Let \( \mathcal{T} \) be a global set of tasks containing the special symbol 0 denoting the empty task.

**Definition II.1.** A real-time interface (or simply an interface) \( I \) is a pair \( (L, T) \) where \( T \subseteq \mathcal{T} \) is the set of tasks in the interface (the alphabet of \( I \)) and \( L \subseteq T^0 \) is an \( \omega \)-regular language denoting the set of acceptable or legal schedules of \( I \) (the behaviour of \( I \)). We require that \( 0 \in T \), i.e., the empty task 0 belongs to every alphabet. \( \Diamond \)

From now on, we refer to a real-time interface simply as an interface. The intuition behind the definition is that an
interface is the set of schedules that satisfy the component’s requirements.

**Definition II.2.** Let $S$ and $T$ be alphabets with $S \subseteq T$. Define the function $\text{proj}(T, S) : T^\omega \rightarrow S^\omega$ by the unique extension of the function $T \rightarrow S$ that is identity on the elements of $S$ and maps every element of $T \setminus S$ to 0.

In other words, if $S \subseteq T$ then projecting a word over the larger alphabet $T$ into a word over the smaller alphabet $S$ will map any symbol from $T$ not belonging to $S$ to 0; symbols that belong to $S$ are mapped to themselves. Taking the inverse projection of a word over $S$ will result in a set of words where any 0 in the word will be replaced by all the letters in $T$ which are not in $S$.

**Notation:** For $f : X \rightarrow Y$, $A \subseteq X$ and $B \subseteq Y$, we write $f(A)$ for the direct image $\{ f(a) \mid a \in A \}$ and $f^{-1}(B)$ for the inverse image $\{ x \in X \mid f(x) \in B \}$.

**Definition II.3.** Given any two alphabets $\Sigma, \Delta$, a substitution is a function $\sigma : \Sigma \rightarrow 2^\Delta$ assigning some language $\sigma(a) \subseteq \Delta^*$ to every symbol $a \in \Sigma$. A substitution $\sigma$ is extended to a map $\sigma : 2^\Sigma \rightarrow 2^\Delta$ by first extending $\sigma$ to $\omega$-words using concatenation, and then to $\omega$-languages by letting

\[
\sigma(L) = \bigcup_{w \in L} \sigma(w),
\]

for every language $L \subseteq 2^\Sigma$.

In general, a substitution allows replacing a symbol by a language. In our setting, we will replace a symbol by an alphabet. This will capture the fact that an idle slot (corresponding to the symbol 0) can be allotted to a task.

**Definition II.4.** The parallel composition $I_1 \parallel I_2$ of two real-time interfaces $I_1 = (L_1, T_1)$ and $I_2 = (L_2, T_2)$ is the interface $(L, T)$ defined by $L = L_1' \cap L_2'$ and $T = T_1 \cup T_2$, where $L_1' = \text{proj}(T, T_1)^{-1}(L_1)$ and $L_2' = \text{proj}(T, T_2)^{-1}(L_2)$.

Note that $L_1' = \sigma_1(I_1)$, where $\sigma_1$ is the substitution defined by $\sigma_1(t) = \{ t \}$ for $t \in T_1 \setminus \{ 0 \}$ and $\sigma_1(0) = T_2$. In other words, $L_1'$ is like $L_1$, except that it allows tasks in $T_2$ to run when the processor is idle. Likewise for $L_2'$. So $L = L_1' \cap L_2'$ is the set of schedules over $T$, whose ‘projections’ are in $L_1$ and $L_2$.

The idea behind the definition is that a schedule is legal in $I_1 \parallel I_2$, if and only if its restriction to $T_1$ is legal in $I_1$ and its restriction to $T_2$ is legal in $I_2$. It is a kind of intersection of sets of schedules of $I_1$ and $I_2$, except we allow tasks from the other set to run when the processor is idle. The parallel composition is the largest set of schedules that satisfies the requirement of both the interfaces.

Since $\omega$-regular languages are closed under regular substitution and intersection, parallel composition of interfaces is well defined. Moreover, the constructions involved are effectively computable [9], so parallel composition is computable.

**Lemma II.5.** When interfaces $I_1 = (L_1, T_1)$ and $I_2 = (L_2, T_2)$ have the same alphabet $T_1 = T_2 = T$, their parallel composition is given by intersection of behaviours: $I_1 \parallel I_2 = (L_1 \cap L_2, T)$.

**Lemma II.6.** Parallel composition is associative and commutative:

1) $(I_1 \parallel I_2) \parallel I_3 = I_1 \parallel (I_2 \parallel I_3)$, and
2) $I_1 \parallel I_2 = I_2 \parallel I_1$.

We say that an interface $I_1$ refines $I_2$ when $I_1$ can safely be substituted for $I_2$ in all contexts. Put another way, $I_1$ is more detailed than $I_2$, and offers fewer design choices.

**Definition II.7.** The interface $I_1 = (L_1, T_1)$ refines $I_2 = (L_2, T_2)$, written $I_1 \preceq I_2$, if and only if $T_2 \subseteq T_1$ and $\text{proj}(T_2, T_1)(L_1) \subseteq L_2$.

In other words, $I_1$ refines $I_2$ when the legal schedules of $I_1$, on restriction to the alphabet $T_2$, are contained in the legal schedules of $I_2$. In addition, $I_1$ is able to schedule some tasks from the set $T_1 \setminus T_2$ in the gaps left by schedules in $I_2$. Note that when $T_1 = T_2$, $I_1 \preceq I_2$ if and only if $L_1 \subseteq L_2$.

Since the inclusion of $\omega$-regular languages is decidable, checking refinement of interfaces is decidable. The following results are immediate.

**Lemma II.8.** Refinement is a partial order.

**Lemma II.9.** (Compositionality of refinement) $I \preceq J \preceq K$ implies $I \parallel K \leq J \parallel K$ for all interfaces $I, J, K$.

**Proof:** In the following, we write $(L_I, T_I)$ for the components of interface $I$. Suppose $I \preceq J$. This implies $T_J \subseteq T_I$ and $\text{proj}(T_I, T_J)(L_I) \subseteq L_J$. Now, $T_J \subseteq T_I$ implies $T_J \cup T_K \subseteq T_I \cup T_K$, so the first condition in Definition II.7 required to show $I \parallel K \leq J \parallel K$ is met. To show that the other condition also holds, we show that $\text{proj}(T_I, T_J)(L_I) \subseteq L_J$ implies $\text{proj}(T_I[K], T_J[K])(L_I[K]) \subseteq L_J[K]$. Suppose $w \in \text{proj}(T_I[K], T_J[K])(L_I[K])$. This means there exists a $v \in L_I[K]$ such that $w = \text{proj}(T_I[K], T_J[K])(v)$, i.e., $w$ is obtained from $v$ by replacing letters in the set $I \setminus J$ by 0. Since $v \in L_I[K]$, we have the following two conditions by the definition of parallel composition:

\[
\text{proj}(T_I[K], T_J)(v) \in L_I \quad (1)
\]
\[
\text{proj}(T_I[K], T_J)(v) \in L_K \quad (2)
\]

Now, since $T_J \subseteq T_I$, it follows from (1) that $\text{proj}(T_I[K], T_J)(v) = \text{proj}(T_I[T_J[K], T_J[T_J[K]])(v)) \in \text{proj}(T_I[T_J[K], T_J[T_J[K]])(L_I) \subseteq L_J$ by the hypothesis. From the way $w$ is obtained from $v$, it follows that $\text{proj}(T_I[T_J[K], T_J[T_J[K]])(w) \in L_J$. From (2) it is immediate that $\text{proj}(T_I[T_J[K], T_J[T_J[K]])(w) \in L_K$. Hence, $w \in \text{proj}(T_I[T_J[K], T_J[T_J[K]])^{-1}(L_I) \cap \text{proj}(T_I[T_J[K], T_J[T_J[K]])^{-1}(L_K) = L_I[K]$. □

Thus, our interfaces satisfy the usual laws for incremental design and independent implementability.

**D. An Example: Scheduling of Periodic Tasks**

This section is adapted from [11], which uses ordinary automata on finite strings to represent and solve scheduling problems. Consider a set of periodic tasks $T = \{ \tau_1, \ldots, \tau_n \}$, where each task $\tau_i$ is characterised by a period $p_i$, an execution time $c_i$, relative deadline $d_i$ and phasing $\phi_i$, with $d_i \leq p_i$. For
simplicity, we assume that the unit of time is the smallest
time slot that can be scheduled atomically. Assume that the
tasks run on a single processor, and are preemptible, with
no preemption overheads. Consider task $\tau_i$ in isolation, and
assume that it is the only task running. Then the $k^{th}$ instance of
task $\tau_i$ is released at $q_i + (k-1)p_i$, and to meet its deadline
it must finish execution by $q_i + (k-1)p_i + d_i$. Then the
processor is idle for $p_i - d_i$ time units, following which the
$(k+1)^{th}$ instance of the task is released. The computation
requirement of the task $\tau_i$ can be expressed in our framework
as the real-time interface $I_i = (L_i,T_i)$ where $T_i = \{0,1\}$ and
$L_i = 0\omega_1[0p_1^1 \parallel v_1^1 0p_1^2 \parallel v_1^2 \parallel \ldots \parallel v_1^{n-1} 0p_1^n \parallel v_1^n ]\omega_0$, where $u \parallel v$ is the shuffle or
interleaving of finite words $u$ and $v$. The set of tasks $T$ is
schedulable if and only if the language $L_\pi$ in the composed
interface $I_\pi = (L_\pi,T) = I_1 \| I_2 \| \ldots \| I_n$ is nonempty. Note that $L_\pi$
is the set of all legal schedules of the task set $T$, i.e., schedules
where all the tasks in $T$ meet their deadlines.

Now consider a fixed priority scheduling (FPS) algorithm,
such as rate monotonic scheduling [20]. Suppose the tasks
$\tau_1, \ldots, \tau_n$ are ordered in non-increasing order of priority. We
show how to capture the timing requirements of such a task set $T$ in terms of an interface $I_{fps}$ in the next paragraph. The
task set $T$ is schedulable using the fixed priority scheme if
and only if $I_{fps} \subseteq L_\pi$, where the interface $I_\pi$ is the composition
$I_1 \| I_2 \| \ldots \| I_n$, as defined above. Since the tasks sets of $I_{fps}$
and $L_\pi$ are the same, namely $T$, this amounts to checking the
inclusion of two $\omega$-regular languages.

To derive the $\omega$-regular expression for the interface $I_{fps}$, let
us make the simplifying assumption that the phasings of all
tasks are zero, i.e., the first instances of all tasks are released
simultaneously. Let $p_{lcm}$ be the least common multiple of
all the periods $p_1, \ldots, p_n$. It is clear that using fixed priority
scheduling, the schedule will repeat after an interval of $p_{lcm}$
time units. Now let $u$ be the finite word that describes the schedule in the initial time interval of length $p_{lcm}$ using fixed
priority scheduling. Then $I_{fps} = (u^\omega, T \cup \{0\})$.

A similar analysis can be done for scheduling with the
earliest deadline first (EDF) [20] algorithm, using the fact that
the schedule will repeat every $p_{lcm}$ time units. We remark on
how our framework can handle more general task models and
scheduling algorithms.

1) Handling of sporadic tasks will require recording the
task release events in addition to the time slices during
which the task runs. Otherwise it will not be possible to
see whether sporadic tasks meet their deadlines. We will
see how to do this with contracts in the next section.

2) Handling a periodic task with deadline $d$ greater than
its period $p$ can be achieved by taking the parallel
composition of $m$ copies of the task with phasings $0,p,2p,\ldots, (m-1)p$ where $m = \lceil d/p \rceil$, and then repeating
this pattern forever.

### III. Adding Contracts

One of the shortcomings of our definition of a real-time
interface is that it lacks the notion of a contract. In the
SPEEDS framework components are characterised by formal
contracts, i.e., pairs $(A,G)$ where $A$ is an assumption about
the environment of the component, and $G$ is the guarantee
that the component offers to its environment [4].

For real-time interfaces, both assumptions and guarantees
will talk about bounds on the frequency of task arrivals
and time to completions. In addition, they can capture the
dependencies between tasks, for example, by stating that “task
2 is triggered whenever task 1 completes”.

We equip our component interfaces with contracts as fol-
lows. Both the assumptions $A$ and the guarantees $G$ consist of
task release (or arrival) times as well as task finishing
(or completion) times. These are again modelled using $\omega$
-regular languages, but now the semantics is different from
the real-time interface we discussed in earlier sections. The
alphabet for a given task $i$ for an $\omega$-word is $\Sigma_i = \{0,a_i,f_i\}$. An $\omega$-word corresponds to time points (instants) when either
nothing happens (modelled by 0), a task arrives (modelled
by $a_i$) or finishes execution (modelled by $f_i$). The contract
$(A,G)$, where $A = L_1 \times L_2 \times \ldots \times L_n$ and $G = L'_1 \times L'_2 \times \ldots \times L'_n$
with $L_i, L'_i$ being $\omega$-regular words over $\Sigma_i$, specifies promises
on the arrival and finishing times of tasks 1 to $n$, given the
assumptions on the arrival and finishing times of the same set
of tasks. A dependency between tasks, such as task $i$ triggers
task $j$, is captured by the occurrence of $f_i$ in position $k$ of any
word in $L_i$ implying the occurrence of $a_j$ in position $k+1$ of the
corresponding word in $L_j$ in the product $L_1 \times L_2 \times \ldots \times L_n$.

We require that both the assumptions $A$ and the guarantees
$G$ in a contract are subsets of the $\omega$-language $(0^*a_i0^* f_i)^\omega$, so
that the arrival-completion intervals are disjoint in time. This
means that two instances of a task cannot be active at the same
time.

We now have to reconcile the two views of a real-time
interface – the set of legal schedules in an interface $I = (L,T)$
and the contract $C$ specified by the pair $(A,G)$. This is captured
by the fact that a task can only execute after it is released, and
it completes execution at the end of the last slot in its
execution. These two constraints can be captured easily by
enforcing certain relations involving the $(A,G)$-pair and the
interface $I$. We say that the interface $I$ is an implementation that
satisfies the contract $C$ in case the constraints hold.

The relation between contracts and interfaces is provided by
a map $\alpha$ that translates the assumptions and guarantees
into interface languages. We define $\alpha$ on individual $\omega$-words,
and then extend the definition to $\omega$-languages by pointwise
union. Note that $\alpha$ will send a single word to a set of
words. Suppose task $i$ has computation time $c_i$. For a word
$w = 0^*a_i0^* f_i0^*a_i0^* f_i \ldots$ in $A$ or $G$, its translation $\alpha(w)$, an
interface language over the alphabet $\{0, i\}$, is the set of words
$0^*i(0^* 1^* \parallel 0^*a_i0^* f_i0^*a_i0^* f_i \ldots)$ which the translation of $w$ involves the execution time $c_i$ for task $i$, an
implementation level concept. While $\alpha$ can be defined for
arbitrary task arrival and completion times as shown above, we
illustrate the definition on a periodic task model.

**Example III.1.** Suppose we want to model a periodic task $i$
with period $p_i$, zero phasing, and relative deadline $d_i$. Assume

that $d_i \leq p_i$. This is captured by the contract $(A, G)$ where $A = (\bigcup_{i \geq 1} a_i (0^i) f_i (0^i))^0$ and $G = (0^a a_i 0^{d_i - 1} f_i )^0$. Suppose task $i$ has execution time of $c_i$ on the processor. The translation of $A$ is given by $\alpha(A) = (f_i \mid (0^{d_i - c_i}))^0$ and that of $G$ by $\alpha(G) = (0^a (f_i \mid (0^{d_i - c_i}))^0$.

The language of the real-time interface of the periodic task in Example III.1 is given by $L_4 = [(f_i \mid (0^{d_i - c_i})0^{p_i - d_i})]^0$. The interface $L_4$, seen as an implementation, satisfies the contract $C = (A, G)$ if and only if $\alpha(A) \cap L_4 \subseteq \alpha(G)$. This is consistent with the following definition of an implementation $M$ satisfying a contract $C = (A, G)$ in [4], modulo the translation $\alpha$.

Definition III.2. [4] Let $C = (A, G)$ be a component. An implementation $M$ of the component satisfies $(A, G)$, written $M \models (A, G)$, if and only if $M \cap A \subseteq G$. Here $M$, $A$ and $G$ are all sets of traces (sequences).

Note that if $A = L_1 \times L_2 \ldots \times L_n$ then the translation of $A$ is given by $\alpha(A) = \{1 \leq i \leq n \} \text{proj}(T_i, T_i)^{-1}(L_i)$, where $T = \{0, \ldots, n\}$ and $T_i = \{0, i\}$, in keeping with the spirit of our definition of parallel composition of interfaces.

The parallel composition of contracts $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ with the same set of tasks, numbered 1 to $n$, can be defined as follows. If $A_i = L_i^1 \times L_i^2 \ldots \times L_i^n$ and $G_i = L_i^1 \times L_i^2 \ldots \times L_i^n$ for $i = 1, 2$, then the composition $C = C_1 \parallel C_2$ is the pair $(A, G)$ given by $A = L_1 \times L_2 \ldots \times L_n$ where $L_k = (L_1^k \cap L_2^k) \cup (L_2^k \cap L_3^k) \ldots (L_n^k \cap L_1^k)$ for $1 \leq k \leq n$, and $G = L_1^1 \times L_2^2 \ldots \times L_n^n$ where $L_k = L_1^k \cap L_2^k \ldots \cap L_n^k$ for $1 \leq k \leq n$. This definition of contract composition is consistent with the definition in [4]:

Definition III.3. Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. The parallel composition $C = (A, G) = C_1 \parallel C_2$ is given by

$$A = (A_1 \cap A_2) \cup -(G_1 \cap G_2),$$

$$G = G_1 \cap G_2$$

Lemma III.5. If the interfaces $I_1$ and $I_2$ satisfy contracts $C_1$ and $C_2$ respectively, then the interface $I_1 \parallel I_2$ satisfies $C_1 \parallel C_2$.  

IV. AN EXAMPLE

To show an application of the approach, two simple uppaal\footnote{Although UPPAAL is a tool for dense time models, it is simple to use and efficient enough for the case studies.} models have been constructed manually that utilise the formalism of timed automata with additional counters to represent the intended interface languages. The models “discretise” time by introducing a global periodic tick event that the automata of the model synchronise with.

We model the activation and the execution of a periodic task (with period $P$, relative deadline $D$ and computation time $C$) by the synchronous product of two automata (cf. Figure 1). We use a total of three counters to represent the scheduling problem for a single task. The input to the task automaton is the arrival $a$.

The counters are used as follows. The first counter $z_e$ keeps track of the number of time slices the current instance of the task has executed. Whenever the counter reaches the execution time $c$ of the task, it is reset. The second counter $z_d$ counts the total time elapsed since the current instance of the task was released. Whenever its value reaches the deadline $d$ of the task, and the first counter has not reached the execution time requirement $c$, an error state is entered. Counter $z_d$ is reset whenever $z_e$ is reset. In order to perform its time keeping function, the second counter is incremented whenever it is non-zero (and below the deadline $d$), or when the task is released. For convenience, the counters $z_e$ and $z_d$ actually store one plus the actual value, in order to distinguish epochs when the task is active (i.e., has been released and not completed) from the ones when the task is inactive; in the latter case the two counters should not change their values. We need just two states (i.e., locations) in the automaton on the right, one for normal operation and the other an error state. In addition to the two counters described above, a counter $z_p$ is needed to mark the periods of the task. The task is schedulable if and only if there exists an infinite path along which the error state is not reachable, i.e., the CTL formula $\text{EF}(-\text{error})$ holds.

For the example, an additional automaton has been constructed representing a contract. To keep things simple, the
assumption is set to true, and the guarantee specifies a maximum deadline for a task (cf. Figure 2). For simplicity the contract automaton re-uses the counter of the task automaton. For the first example, we verified (the obvious fact) that a single task with period 10 and execution time 5 is schedulable, and that it satisfies the given contract. For the second example, we constructed a model with a second task with the same execution characteristics, and repeated the schedulability and satisfaction verification against the contract.

As expected, verification showed that both properties are satisfied in the second case as well. We could also verify that neither schedulability nor contract satisfaction holds when either the execution time of the task is increased, or the deadline defined by the contract is reduced.

TABLE I

<table>
<thead>
<tr>
<th>Schedulability of task 1.</th>
<th>Satisfaction of task 1 against contract.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedulability of (task 1</td>
<td></td>
</tr>
</tbody>
</table>

V. CONCLUSION AND FUTURE WORK

In this paper we have proposed a real-time interface for the SPEEDS framework. In future work, we plan to address several limitations of the interface language and the language of contracts. For instance, the interface language cannot specify dependencies between tasks when only upper and lower bounds and not exact values for their execution times are known. Moreover, there is a loss of information in the translation between contracts and the interface language concerning the visibility of arrivals and completion times of tasks. This loss of expressive power leads to an approximate analysis of schedulability. These are issues for future investigation.

ACKNOWLEDGEMENTS

We thank Werner Damm for sharing his insights and providing helpful suggestions and encouragement. Discussions with S. Ramesh, Prahlad Sampath, Alexander Metzner and Matthias Büker have clarified many issues. The first author gratefully acknowledges the support from the University of Oldenburg and AVACS, and OFFIS for hosting him during the work.

REFERENCES