Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 401–418
www.elsevier.com/locate/entcs

Integrating Time and Resource into *Circus* [1]

Geguang Pu, Qiu Zongyan[2]

LMAM and Department of Informatics, School of Math.
Peking University, Beijing 100871, China

He Jifeng[2]

International Institute for Software Technology,
United Nations University Macau, China

Abstract

In this paper, a formal model is introduced for reasoning about resource allocation and scheduling in real-time systems. We extend the concurrent refinement language *Circus* [19] through integrating continuous time and resource information. This model reflects resource issues when modelling the behavior of a system, and allows temporal properties to be accurately determined. We also apply the model to the problem of partitioning in co-design, and show how the partitioned programs preserve the behavior of the specification correctly.

Keywords: Timed *Circus*, UTP, resource reasoning, denotational semantics.

1 Introduction

The timing behavior of real-time systems depends on not only process synchronization, but also the availability of resources provided by systems. Many timed models are proposed for the design and verification of real-time systems, for example, TAM [15], Timed CSP [2], Timed CCS [17,9], TCOZ [12] and Real-Time Refinement [4]. These models have the implicit assumption that unbounded resources are available. However, in practice, real-time systems are

[1] Supported by NNSF of China (No.60173003).
[2] Email: ggpu@math.pku.edu.cn, zyqiu@pku.edu.cn, jifeng@iist.unu.edu

doi:10.1016/j.entcs.2005.03.020

always restricted by the resources. To deal with this problem, researchers proposed some methods to combine real-time programs with the scheduling and resource allocation, and thus, to facilitate the reasoning about systems that are sensitive to deadlines, process communication and resource availability.

Gerber and Lee [3] proposed a language *CCSR*, to reason about resource requirements in real-time systems. Their model is operational, based on CCS and extended to model priorities. Gavin Lowe [8] proposed a compositional denotational model based on TAM [15]. This model can reason about schedulers which allocate resources to tasks. The model is only suitable to analyze asynchronous systems, as TAM is an asynchronous language. Jin and He [7] proposed a hierarchy of resource models to handle the resource allocation and reclamation. However, their models are relatively abstract, and is not compositional.

In this paper we present a series of semantic models that can reason about resource allocation in real-time systems. The models are based upon *Circus*, a well-defined specification language combining the features of CSP and Z. *Circus* has a formal semantics based on the Unifying Theories of Programming(UTP) [5]. We extend the core languages of *Circus* by adding time information for reasoning about real-time programs. Moreover, we introduce the resource information into the model, and make it possible in analyzing the resource allocation of real-time systems, for example, the resource conflict, where two parallel programs access the same resource simultaneously.

Sherif and He [16] did the first step to add time into *Circus*, and they used discrete time in their model. Though discrete time models are easily implemented by computer systems, it is more natural to adopt the continuous time as the nature of time in the real world is continuous. Like other studies, such as TAM [15], Real-Time Refinement [4], Timed CSP [2], we use continuous time in our models and go further than [16]. When integrating time into *Circus*, a number of healthiness conditions are posed on the continuous model in order to specify its correct behavior.

This paper is organized as follows. Section 2 describes the semantic space adopted in our models. Continuous time is integrated into *Circus* in Section 3. Section 4 introduces two resource models based on the time model, and explores the relations among them. In Section 5, we present an example from co-design field to show how these models are applied to verify the correctness of program. We conclude in Section 6 with a discussion on future work.

2 Semantic Space for Timed *Circus*

To define a time model, we adopt a simple language CT, which is a subset of *Circus* and introduced in [16]. For simplicity, we only consider actions, guarded commands and assignment as the basic actions, other constructs of the language, such as abstractions and declarations, have little effect over the model. CT is also extended with timed structures, such as *idle* d, *deadline* d, *idle* etc. The informal description about CT will be given later with the semantics. The syntax of CT is as follows:

$$
\begin{array}{ll}
\mathsf{Action} & ::= \mathsf{Skip} \mid \mathsf{Stop} \mid \mathsf{Chaos} \mid \mathsf{idle}\ d \mid \mathsf{idle} \\
& \quad \mid \mathsf{deadline}\ d \mid \mathsf{widle}\ d \\
& \quad \mid \mathsf{Communication} \rightarrow \mathsf{Action} \mid \mathsf{b}\ \&\ \mathsf{Action} \\
& \quad \mid \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \,\Box\, \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \quad \mid \mathsf{Action} \,[\![\, \mathsf{CS} \,]\!]\, \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CS} \mid \mathsf{Command} \\
& \quad \mid [\mathsf{d}]\ \mathsf{Action} \mid \mu\, \mathsf{N} \bullet \mathsf{Action} \\
\mathsf{Communication} & ::= \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= ?\,\mathsf{N} \mid !\,\mathsf{e} \mid .\,\mathsf{e} \\
\mathsf{Command} & ::= \mathsf{N}^+ := \mathsf{e} \mid \mathsf{Action} \lhd b \rhd \mathsf{Action}
\end{array}
$$

The semantic space is introduced here. The continuous time is adopted to express the behaviors of the system at the design stage. Moreover, through the appropriate mapping from continuous model to discrete one, the system specification can be implemented by computers. We use the similar semantic space from [5]. The behaviors of Timed *Circus* are defined by means of predicative semantics. Here are some observational variables used in our models.

ok, ok': when boolean variable ok is true, the program has been properly started, $ok = \mathit{false}$ means that the program has never started and even the initial state is unobservable; $ok' = \mathit{true}$ means that the current state of program is stable. It can distinguish the stable case of program from the one with infinitely internal computation.

$wait, wait'$: boolean variable $wait$ can tell waiting state of program from its terminated state. If $wait'$ is true, the program stays in intermediate observations, and does not terminate, otherwise, it reaches in terminated state.

t: time is modelled by nonnegative real numbers:

$$Time \mathrel{\widehat=} \{r : \mathbb{R} \mid 0 \leq r < \infty\}$$

where $\mathbb{R}$ is the set of real numbers including infinities, and operators on the reals are extended to allow infinite arguments. We use t and t' to represent

the start time and current time respectively, and allow t' to take the value infinity.

$$t' \in Time_\infty \mathrel{\widehat{=}} Time \cup \{\infty\}$$

v: variable v is modelled as a function from time to value. With the adoption of continuous time, the super dense computation might be involved, which means at time point t, a number of computation steps may take place. To deal with this phenomenon, the value of v at t is designed artificially as a pair of observations, $v(t) = \langle e_1, e_2 \rangle$, where e_1 and e_2 are called the left value and right value of v at time t, and denoted as $v(t).l$ and $v(t).r$ respectively. The possibly internal values between e_1 and e_2 are invisible. The similar technique was used in [10]. For simplicity, we also use v to represent a list of program variables.

tr, ref: variable tr records the communications of a program with its environment, $tr \in TR = Time \to Seq(Time \times Event)$. $tr(t)$ records the historic observations happened so far, which is a sequence of timed events. A timed event (t, e) denotes the observation of event e at time t. For example, $tr(3.2) = \langle (1, a), (2, b), (3.2, c) \rangle$ means that there are 3 events already happened up to time point 3.2, where events a, b and c take place at time 1, 2 and 3.2 respectively. To track the set of events refused by a program, we introduce $ref \in Time \to \mathcal{P}(Event)$ to model the set of events refused at time t.

3 Denotational Semantics of Timed *Circus*

3.1 Healthiness Conditions

In this section, we introduce some healthiness conditions, which distinguish feasible descriptions of reality from infeasible ones. Healthiness conditions can reject a specifications if it makes implementation demonstrably impossible in the target programming languages. We list four healthiness conditions which are satisfied by all feasible Timed *Circus* programs. Some predicates which will be used in this paper are defined as follows:

$$\begin{aligned} \mathit{flow}(t, tr) \mathrel{\widehat{=}} & \; t \le t' \land \exists\, tr_0 \in TR \bullet tr(t') = tr(t) \frown tr_0 \land \\ & \forall\, t_1, t_2 \exists\, n \in N \bullet t \le \; t_2 \le t_1 \le t' \Rightarrow (\#tr(t_1) - \#tr(t_2)) \le n \end{aligned}$$

$\mathit{flow}(t, tr)$ asks that no program can ever make time go backwards or change what did happen before it starts. Condition $(\#tr(t_1) - \#tr(t_2)) \le n$ ensures that no infinitely many events can occur within a finite interval of time.

$\mathit{stable}(w) \mathrel{\widehat{=}} w(t') = w(t)$

$stable(w)$ denotes that timed variable w keeps the same from the at the start and current time. As our semantics is based on observations, the start and current states of program are interested and the middle states between them are not observable. w can be a list of variables.

Four healthiness conditions are introduced as follows.

H1. $P = P \wedge flow(t, tr)$

It states that any timed *Circus* program should satisfy predicate $flow(t, tr)$.

H2. $P = (\neg ok \wedge flow(t, tr)) \vee P$

If the program has never started, the initial values are even unobservable.

H3. $P = I\!I \lhd wait \rhd P$
where $I\!I \mathrel{\widehat{=}} (\neg ok \wedge flow(t, tr)) \vee (ok' \wedge stable(tr, v, ref) \wedge wait' = wait)$

This condition enables the sequential composition to work normally. As all intermediate observations of P are also the intermediate observations of $P;\ Q$, only when P terminates, the control can be passed to Q.

H4. $P = P \vee P[false/ok']$

This condition is designed for the upward closure of variable ok'. It formally encodes the fact that we cannot require a process to abort.

Property 1. $P = P \vee P[false/ok']$ iff $[P[false/ok'] \Rightarrow P[true/ok']]$

Proof: $P = P \vee P[false/ok']$
$\equiv P[false/ok'] = P[false/ok'] \vee P[false/ok'] \wedge$
$\quad P[true/ok'] = P[true/ok'] \vee P[false/ok']$
$\equiv P[true/ok'] = P[true/ok'] \vee P[false/ok']$
$\equiv [P[false/ok'] \Rightarrow P[true/ok']]$ □

The following property introduces another form for healthiness condition **H4**.

Property 2. $[P[false/ok'] \Rightarrow P[true/ok']]$ iff $P = P;\ J$,
where $J = (ok \Rightarrow ok') \wedge stable(tr, ref, v) \wedge wait' = wait$

We use $H(P)$ to denote that program P satisfies all the healthiness conditions. Notice that P satisfying a healthiness condition means that P is a fixed point of an equation corresponding to the condition. For example, If P satisfies **H1, H2** and **H3**, it is a fixed point of **R1, R2** and **R3** respectively

$\mathbf{R1}(X) \mathrel{\widehat{=}} X \wedge flow(t, tr)$
$\mathbf{R2}(X) \mathrel{\widehat{=}} (\neg ok \wedge flow(t, tr)) \vee X$

$\mathbf{R3}(X) \mathrel{\widehat{=}} \mathit{II} \lhd \mathit{wait} \rhd X$

From **Properties 1 and 2**, when P satisfies **H4**, it is a fixed point of **R4**

$\mathbf{R4}(X) \mathrel{\widehat{=}} X;\ J$

By means of these constructions of equation, we have the following theorem.

Theorem 3.1 *All predicates which satisfy the four healthiness conditions form a complete lattice.*

Proof: As all predicates form a complete lattice in the implication ordering, according to Tarski's fixed point theorem and the commutativity of the idempotents **Ri** $(i = 1, 2, 3, 4)$, the set of fixed points of monotonic functions **Ri** $(i = 1, 2, 3, 4)$ on a complete lattice is also a complete lattice. □

In the following subsections, the semantics of Timed *Circus* is given. We use $[\![P]\!]_t$ to denote the timed semantics of program P.

3.2 Basic Actions

Action *Skip* terminates immediately and does not change any timed variables.

$[\![\mathit{Skip}]\!]_t \mathrel{\widehat{=}} H(ok' \land \neg wait' \land \mathit{stable}(tr, v) \land t = t')$

Action *Stop* just waits for ever and does not communicate with its environment. It does not change any timed variables as well.

$[\![\mathit{Stop}]\!]_t \mathrel{\widehat{=}} H(ok' \land wait' \land \mathit{stable}(tr, v))$

The behavior of action *Chaos* is totally unpredictable. We denote it as *true*.

$[\![\mathit{Chaos}]\!]_t \mathrel{\widehat{=}} H(\mathit{true})$

Assignment $v := e$ does not consume time and updates the right value of v at time t simultaneously.

$[\![v := e]\!]_t \mathrel{\widehat{=}} H(ok' \land \neg wait' \land \mathit{stable}(tr) \land v(t').r = e[v/v(t).l] \land t = t')$

3.3 Timed Actions

Timed actions are introduced here. They are somewhat interesting. Action *idle* d just consumes d time units and has no effect on state variables.

$$[\![\mathit{idle}\ d]\!]_t \mathrel{\widehat{=}} H(((ok' \land wait' \land t' - t < d) \lor (ok' \land \neg wait' \land t' - t = d)) \land \mathit{stable}(tr, v)) \lhd d < \infty \rhd [\![\mathit{Stop}]\!]_t$$

Action *idle* may take time and the behavior is similar to that of action *idle* d.

$$[\![idle]\!]_t \mathrel{\widehat{=}} \exists\, d \bullet [\![idle\ d]\!]_t$$

If two programs have the same behaviors, but run in different speed, we prefer the fast program to the slow one on the performance. But for two simple programs, *idle* 3 and *idle* 4, we cannot conclude that *idle* 3 is better than *idle* 4 from the definition of $[\![idle\ d]\!]_t$. To solve this problem, we introduce a weak idle operation and denote it as *widle* d:

$$[\![widle\ d]\!]_t \mathrel{\widehat{=}} \exists\, e \leq d \bullet [\![idle\ e]\!]_t$$

From this definition, the following implication ordering is valid.

$$[\![widle\ 3]\!]_t \Rightarrow [\![widle\ 4]\!]_t$$

The deadline action allows a time deadline to be specified. The check of deadline can be embedded in program compiler to ensure whether the deadline is met by the generated code. This suggests that a timing path analysis is required to show the deadline action always finishes before its deadline. If t' is greater than d when executing the deadline action, the behavior of the program is like *Chaos*.

$$\begin{aligned}[\![deadline\ d]\!]_t \mathrel{\widehat{=}} & H((ok' \wedge \neg wait' \wedge stable(v, tr) \wedge t = t')) \\ & \lhd (t' \leq d) \rhd H(true)\end{aligned}$$

The action $[d]P$ executes P but gives it a deadline of d, this means that we are interested only in those executions of P that terminate before time d. In terms of the deadline action, we can easily define it:

$$[\![[d]P]\!]_t \mathrel{\widehat{=}} [\![P;\ deadline\ d]\!]_t$$

The definition of sequential composition will be given later.

3.4 Communication Actions

Here we model the behavior of communication actions. The state of $c!e$ has two possible behaviors: one is in waiting state (maybe in deadlock if event c is in *ref*), the other is to communicate with its ready partner and terminates.

$$\begin{aligned}[\![c!e]\!]_t \mathrel{\widehat{=}} & ([\![idle]\!]_t \wedge H(c \notin ref(t))) \vee \\ & H(ok' \wedge \neg wait' \wedge tr(t') = tr(t) \frown \langle t', c\rangle \wedge \\ & stable(v) \wedge t' = t + \varepsilon)\end{aligned}$$

The behavior of $c?v$ is similar to $c!v$, except that it changes the variable v on termination.

$$[\![c?v]\!]_t \mathrel{\widehat{=}} ([\![idle]\!]_t \wedge H(c \notin ref(t))) \vee$$
$$H(ok' \wedge \neg wait' \wedge tr(t') = tr(t) \frown \langle t', c\rangle \wedge$$
$$v(t').r = e[v/v(t).l] \wedge t' = t + \varepsilon)$$

We introduce symbol ε to denote a very short time period the communication actions consume, in order to separate two consecutive events. We assume $m\varepsilon < a$ for all reasonable $m \in \mathbb{N}$ (set of natural numbers) and $a \in \mathbb{R}^+$. Thus, ε is regarded as a kind of "infinitesimal". The similar view is taken implicitly in Timed CSP [2].

3.5 Composition

Conditional Choice. The conditional choice $P \lhd b \rhd Q$ executes P if b evaluates as *true*, otherwise executes Q instead. We assume that the evaluation of b takes no time. Actually, this assumption can be removed by adding *idle* action before P and Q respectively.

$$[\![P \lhd b \rhd Q]\!]_t \mathrel{\widehat{=}} (b \wedge [\![P]\!]_t) \vee (\neg b \wedge [\![Q]\!]_t)$$

Sequential Composition. As all predicates under consideration satisfy the healthiness condition H2, we can follow the standard definition of sequential composition in relational calculus, with some small modifications for the super dense computation. The sequential composition connects the right values of variables in P to the left values of them in Q.

$$[\![P;\ Q]\!]_t \mathrel{\widehat{=}} [\![P]\!]_t \circ [\![Q]\!]_t$$
$$[\![P]\!]_t \circ [\![Q]\!]_t \mathrel{\widehat{=}} \exists\, ok_0, wait_0, t_0, v_0 \bullet [\![P(ok',\ wait',\ t', v(t').r/ok_0, wait_0, t_0, v_0)]\!]_t \wedge [\![Q(ok,\ wait,\ t, v(t).l/ok_0, wait_0, t_0, v_0)]\!]_t$$

For example, from this definition, we can prove the equation like follows:

$$[\![v := v + 1;\ v := v + 2]\!]_t = [\![v := v + 3]\!]_t$$

Two assignments take place at the same time, and according to the definition of sequential composition, the middle state is hidden from observers.

Guarded Action. A guarded action is enabled only if the condition g holds. If g holds, the action behaves as P; otherwise it behaves like *Stop*. We also assume the evaluation of g does not consume time.

$$[\![g\&P]\!]_t \mathrel{\widehat{=}} [\![P \lhd g \rhd Stop]\!]_t$$

Internal Choice. Internal choice denotes the non-determinism of the observations. It is a very important concept in the design of parallel programs.

$$[\![P \sqcap Q]\!]_t \mathrel{\widehat{=}} [\![P]\!]_t \vee [\![Q]\!]_t$$

External Choice. The behavior of an external choice can be observed in two points: before the choice is made and after the choice is made. Before the choice, the system is in waiting state and only internal actions can take place. Moreover, an event is refused only if it is refused by both P and Q. After the choice is made, the rest of the behavior is described by either that of P or Q, depending on which action corresponds to the first event.

$$\begin{aligned}[\![P \,\Box\, Q]\!]_t \mathrel{\widehat{=}} &([\![P]\!]_t \wedge [\![Q]\!]_t \wedge [\![idle]\!]_t) \\ &\vee ([\![P]\!]_t \vee [\![Q]\!]_t \wedge H(\neg wait' \vee \neg stable(tr)))\end{aligned}$$

Recursion. From **Theorem 3.1**, all the Timed *Circus* actions form a complete lattice with respect to the implication relation. So we can define the recursion as a weakest fixed point of set $\{X \mid X \Rightarrow F(X)\}$, that is,

$$\mu\, X \bullet F(X) \mathrel{\widehat{=}} \sqcap \{X \mid X \Rightarrow F(X)\}$$

Hiding. When the set cs of events is hidden, any event in cs is hidden from the environment, then, events in cs take place automatically and instantaneously without any additional conditions. Hiding operator $\setminus$ defined as follows

$$\begin{aligned}[\![P \setminus cs]\!]_t \mathrel{\widehat{=}} &H(\exists\, tr_0, ref_0 \bullet [\![P]\!]_t[tr_0, ref_0/tr(t'), ref(t')] \wedge \\ &tr(t') - tr(t) = (tr_0 - tr(t)) \upharpoonright (Event - cs) \wedge ref(t') = ref_0 \cup cs)\end{aligned}$$

$\upharpoonright$ is the restriction operator defined in CSP, and *Event* is the set of all events.

3.6 Parallel Composition

The parallel composition is a little complicated. It is defined in terms of the merge parallel composition introduced in [5]. The parallel composition of two actions terminates whilst both components do. The action *idle* in the following definition just plays a role for this observation.

$$\begin{aligned}[\![P \,[\![\, cs \,]\!]\, Q]\!]_t \mathrel{\widehat{=}} \exists\, tr_0 \bullet H(tr_0 = tr(t)) \wedge (([\![P;\ idle]\!]_t \parallel_{M(cs)} [\![Q]\!]_t) \vee \\ ([\![P]\!]_t \parallel_{M(cs)} [\![Q;\ idle]\!]_t))\end{aligned}$$

Operation $M(cs)$ merge the separate copies of the shared variables in P and Q into the final values, where cs represents the set of events the two components should synchronize on. The merge operation is defined as follows,

$$
\begin{aligned}
M(cs) \mathrel{\widehat{=}} H(\ & ok' = (0.ok \wedge 1.ok) \\
\wedge\ & wait' = (0.wait \vee 1.wait) \\
\wedge\ & v(t') = 0.v(t) \oplus 1.v(t) \\
\wedge\ & ref(t') = ((0.ref(t) \cup 1.ref(t)) \cap cs) \vee ((0.ref(t) \cap 1.ref(t))) \\
\wedge\ & (tr(t') - tr_0) \in (0.tr(t) - tr_0 \parallel_m 1.tr(t) - tr_0) \\
\wedge\ & t' = t)
\end{aligned}
$$

Note that the merging style between variables is based on the definition of merging operator $\oplus$ for some specific application. Now we define the operator $\parallel_m$. We use tr_1 and tr_2 to stand for two traces. Let $x_i (i = 1, 2...)$ denote members of cs, and $y_i (i = 1, 2...)$ denote events that do not belong to cs. Operator $\parallel_m$ is defined by following rules:

$$
\begin{aligned}
& tr_1 \parallel_m tr_2 = tr_2 \parallel_m tr_1 \\
& \langle\rangle \parallel_m \langle\rangle = \{\langle\rangle\} \\
& \langle\rangle \parallel_m \langle(t_2, x)\rangle = \{\} \\
& \langle\rangle \parallel_m \langle(t_2, y)\rangle = \langle(t_2, y)\rangle \\
& \langle(t_1, x)\rangle \frown tr_1 \parallel_m \langle(t_2, y)\rangle \frown tr_2 = \{\langle(t_2, y)\rangle \frown tr_3 \mid tr_3 \in \langle(t_1, x)\rangle \frown tr_1 \\
& \qquad\qquad \parallel_m tr_2\} \quad \text{if } t_2 \geq t_1 \\
& \langle(t_1, x)\rangle \frown tr_1 \parallel_m \langle(t_2, y)\rangle \frown tr_2 = \{\} \quad \text{if } t_2 < t_1 \\
& \langle(t, x)\rangle \frown tr_1 \parallel_m \langle(t, x)\rangle \frown tr_2 = \{\langle(t, x)\rangle \frown tr_3 \mid tr_3 \in tr_1 \parallel_m tr_2\} \\
& \langle(t_1, x)\rangle \frown tr_1 \parallel_m \langle(t_2, x)\rangle \frown tr_2 = \{\} \quad \text{if } t_1 \neq t_2 \\
& \langle(t_1, x_1)\rangle \frown tr_1 \parallel_m \langle(t_2, x_2)\rangle \frown tr_2 = \{\} \quad \text{if } x_1 \neq x_2 \\
& \langle(t, y_1)\rangle \frown tr_1 \parallel_m \langle(t, y_2)\rangle \frown tr_2 = \{\langle(t, y_1)\rangle \frown tr_3 \mid tr_3 \in tr_1 \\
& \quad \parallel_m \langle(t, y_2)\rangle \frown tr_2\} \cup \{\langle(t, y_2)\rangle \frown tr_3 \mid tr_3 \in \langle(t, y_1)\rangle \frown tr_1 \parallel_m \langle(t, y_2)\rangle\ tr_2\} \\
& \langle(t_1, y_1)\rangle \frown tr_1 \parallel_m \langle(t_2, y_2)\rangle \frown tr_2 = \{\langle(t_1, y_1)\rangle \frown tr_3 \mid tr_3 \in tr_1 \\
& \quad \parallel_m \langle(t_2, y_2)\rangle \frown tr_2\} \lhd t_1 < t_2 \rhd \{\langle(t_2, y_2)\rangle \frown tr_3 \mid tr_3 \in \langle(t_1, y_1)\rangle \frown tr_1 \\
& \quad \parallel_m \langle(t_2, y_2)\rangle\ tr_2\}
\end{aligned}
$$

3.7 Properties of Timed Circus Actions

The following theorem shows that the healthy actions are closed under sequential composition, disjunction, conjunction, external choice, and parallel.

Theorem 3.2 *The set of Timed Circus actions is a* $\{\wedge, \vee, ;\ , \Box, [\![cs]\!]\}$*-closure*

for four healthiness conditions.

Proof: Here we prove only the closure property of conjunction.

• **H1**

$P \wedge Q \wedge \mathit{flow}(t, tr)$
$= P \wedge Q \wedge \mathit{flow}(t, tr) \wedge \mathit{flow}(t, tr)$
$= (P \wedge \mathit{flow}(t, tr)) \wedge (Q \wedge \mathit{flow}(t, tr)) = P \wedge Q$

• **H2**

$(P \wedge Q) \vee (\neg ok \wedge \mathit{flow}(t, tr))$
$= (P \vee (\neg ok \wedge \mathit{flow}(t, tr)) \wedge Q \vee (\neg ok \wedge \mathit{flow}(t, tr))) = P \wedge Q$

• **H3**

$\mathit{II} \lhd \mathit{wait} \rhd (P \wedge Q)$
$=(\mathit{II} \lhd \mathit{wait} \rhd P) \wedge (\mathit{II} \lhd \mathit{wait} \rhd Q) = P \wedge Q$

• **H4**

$(P \wedge Q) \vee (P \wedge Q)[\mathit{false}/ok']$
$= (P \wedge Q) \vee (P[\mathit{false}/ok'] \wedge Q[\mathit{false}/ok'])$
$= ((P \wedge Q) \vee P[\mathit{false}/ok']) \wedge ((P \wedge Q) \vee Q[\mathit{false}/ok'])$
$= ((P \vee P[\mathit{false}/ok']) \wedge (Q \vee P[\mathit{false}/ok'])) \wedge$
$((P \vee Q[\mathit{false}/ok']) \wedge Q \vee P[\mathit{false}/ok'])$
$= (P \wedge (Q \vee P[\mathit{false}/ok'])) \wedge (Q \wedge (P \vee Q[\mathit{false}/ok']))$
$= (P \wedge Q) \wedge (Q \vee P[\mathit{false}/ok']) \wedge (P \vee Q[\mathit{false}/ok']) \Rightarrow (P \wedge Q)$

Therefore

$P \wedge Q = (P \wedge Q) \vee (P \wedge Q)[\mathit{false}/ok']$ □

The next theorem shows that initial value of $tr(t)$ in the predicates may be replaced by an arbitrary one.

Theorem 3.3 *If P is an action in timed Circus, then P satisfies the condition* $P(tr(t), tr(t')) = \bigsqcap_{s \in TR} P(s, s \frown \langle tr(t') - tr(t) \rangle)$. □

4 Resource Model in Timed Circus

In this section, we extend the model of the previous section in order to represent the resources used by actions. Two resource models are proposed here, one is an unlimited resource model, and the other is a limited resource model. The latter is a natural extension of the former. In the unlimited model, we assume resources in the real-time environment are enough for the execution of any action.

The usage of resource r during time interval $[t, t']$ is denoted by $\langle [t, t'], r, n\rangle$, which means that n units of resource r is used in time interval $[t, t']$. Formally, we define the space *TResource* as

$$TResource \mathrel{\widehat{=}} Interval \times Rname \times \mathbb{N}$$

where *Interval* denotes time interval, *Rname* is a set of resource names.

Two observable variables R_{re} and R_{av} are used in our models, which stand for the resources required by an action, and the resources available, respectively. To compute R_{re}, a function ϕ is introduced to calculate the resources required by actions. It can be regarded as a global scheduler to monitor resource occupation. For example, if action P requires one unit of r_1 and two units of r_2 during execution in time interval $[t, t']$, we have $R_{re} = \phi(t) = \{\langle [t, t'], r_1, 1\rangle, \langle [t, t'], r_2, 2\rangle\}$. The definition of ϕ depends on not only the real environment, but also the command executed at the time. Fox example, $x := 1$ and $x := x + 1$ take different resource during their execution. The later requires an adder to perform the action, while the former only involves memory update. For simplicity, single parameter t is referred by function ϕ. When the resource models are used to analyze real-time systems, a concrete definition of ϕ should be given.

$R_{av} \in \mathcal{P}(Resource)$ where *Resource* is $Resource \mathrel{\widehat{=}} Rname \times \mathbb{N}$

4.1 Unlimited Resource Model

We use $[\![P]\!]_{ut}$ to denote the semantics of action P under the unlimited resource model. Most of the definitions here are simple extensions of ones in the previous section.

Basic Actions. Actions *Skip*, *Stop* do not consume resources:

$$[\![Skip]\!]_{ut} \mathrel{\widehat{=}} H(ok' \wedge \neg wait' \wedge stable(tr, v) \wedge t = t' \wedge R_{re} = \emptyset)$$
$$[\![Stop]\!]_{ut} \mathrel{\widehat{=}} H(ok' \wedge wait' \wedge stable(tr, v) \wedge R_{re} = \emptyset)$$

The behavior of *Chaos* is totally unpredictable, and also has no predication on resources. We denote it as *true*.

$$[\![Chaos]\!]_{ut} \mathrel{\widehat{=}} H(true)$$

Assignment uses resources $\phi(t)$ in the execution

$$[\![v := e]\!]_{ut} \mathrel{\widehat{=}} H(ok' \wedge \neg wait' \wedge stable(tr) \wedge v(t').r = e[v/v(t).l] \wedge t = t' \wedge R_{re} = \phi(t))$$

Timed Actions. The *deadline* timed actions do not consume resources, but other timed actions may need resources during execution. For example, *idle d* may consume CPU resources, even it does not make any progress.

$$[\![idle\ d]\!]_{ut} \mathrel{\widehat{=}} H(((ok' \wedge wait' \wedge t' - t < d) \vee (ok' \wedge \neg wait' \wedge t' - t = d)) \\ \wedge stable(tr, v) \wedge R_{re} = \phi(t)) \lhd d < \infty \rhd [\![Stop]\!]_t$$

$$[\![deadline\ d]\!]_{ut} \mathrel{\widehat{=}} H(ok' \wedge \neg wait' \wedge t = t' \wedge stable(tr, v) \wedge R_{re} = \emptyset) \\ \lhd (t' \leq d) \rhd H(true)$$

Communication Actions. The behaviors of communication actions are similar to those in the time model, except that they consume resources.

$$[\![c!e]\!]_{ut} \mathrel{\widehat{=}} ([\![idle]\!]_t \wedge H(c \notin ref(t'))) \vee \\ H(ok' \wedge \neg wait' \wedge tr(t') = tr(t) \frown \langle (t', c) \rangle \\ \wedge stable(v) \wedge t' = t + \varepsilon \wedge R_{re} = \phi(t))$$

$$[\![c?v]\!]_{ut} \mathrel{\widehat{=}} ([\![idle]\!]_t \wedge H(c \notin ref(t'))) \vee \\ H(ok' \wedge \neg wait' \wedge tr(t') = tr(t) \frown \langle (t', c) \rangle \wedge R_{re} = \phi(t) \\ \wedge t' = t + \varepsilon \wedge v(t').r = e[v/v(t).l])$$

Sequential Composition. The resource requirements of sequential composition are the union of the resource requirements of the two components.

$$[\![P;\ Q]\!]_{ut} \mathrel{\widehat{=}} \exists\, ok_0, wait_0, t_0, v_0\ R_{re1},\ R_{re2} \bullet \\ [\![P(ok',\ wait',\ t', v(t').r, R_{re}/ok_0,\ wait_0,\ t_0,\ v_0, R_{re_1})]\!]_t \wedge \\ [\![Q(ok,\ wait,\ t, v(t).l, R_{re}/ok_0,\ wait_0,\ t_0,\ v_0, R_{re_2})]\!]_t \wedge \\ H(R_{re} = R_{re1} \cup R_{re2})$$

Parallel Composition. We only need to modify the merge operation defined in the time model.

$$M(cs) \mathrel{\widehat{=}} H(ok' = (0.ok \wedge 1.ok) \wedge wait' = (0.wait \vee 1.wait) \\ \wedge v(t') = 0.v(t) \oplus 1.v(t) \\ \wedge ref(t') = ((0.ref(t) \cup 1.ref(t)) \cap cs) \vee ((0.ref(t) \cap 1.ref(t))) \\ \wedge (tr(t') - tr_0) \in (0.tr(t) - tr_0 \parallel_m 1.tr(t) - tr_0) \\ \wedge t' = t \wedge R_{re} = 0.R_{re} \cup 1.R_{re})$$

Other composition operators, such as *conditional, nondeterministic choice, recursion*, are the same with those defined in the time model.

4.2 Limited Resource Model

The assumption that the resources are unlimited is somehow unreasonable in reality, because it is not true in many cases. A limited resource model can be formalized as an extension of the unlimited model. We use $[\![P]\!]_{lt}$ to denote the semantics of P in limited resource model. We only need to modify the definitions of basic actions and some composition structures.

Operator $Lea(R_{re})$ returns the name and the amount of the resources required. For instance, $Lea(\{\langle[t,t'],r_1,1\rangle,\langle[t,t'],r_2,2\rangle\}) = \{\langle r_1,1\rangle,\langle r_2,2\rangle\}$. Let P_{atom} be those atomic actions, including basic and communication actions etc. We define:

$$\begin{aligned}[\![P_{atom}]\!]_{lt} \mathrel{\widehat{=}} &([\![P_{atom}]\!]_{ut} \wedge H(R'_{av} = R_{av} - Lea(R_{re}))) \\ &\lhd (Lea(R_{re}) \subseteq R_{av}) \rhd [\![Stop]\!]_t\end{aligned}$$

We just modify the definitions of sequential and parallel compositions by adding in the information of R_{av}. The final value of R_{av} in P is also passed on as the initial value of R_{av} in Q, which is only an intermediate state of $P;\ Q$ and unobservable by the environment. The following is the modified definition of sequential composition for limited resource model.

$$\begin{aligned}[\![P;\ Q]\!]_{lt} \mathrel{\widehat{=}} &\exists\, ok_0, wait_0, t_0, v_0\ R_{re1},\ R_{re2}\ R_{av0} \bullet \\ &[\![P(ok',\ wait',\ t', v(t').r, R_{re}, R'_{av}/ok_0,\ wait_0,\ t_0,\ v_0, R_{re_1}, R_{av0})]\!]_t \wedge \\ &[\![Q(ok,\ wait,\ t, v(t).l, R_{re}, R_{av}/ok_0,\ wait_0,\ t_0,\ v_0, R_{re_2}, R_{av0})]\!]_t \wedge \\ &H(R_{re} = R_{re1} \cup R_{re2})\end{aligned}$$

The merge operation in parallel composition needs to be modified as well:

$$\begin{aligned}M(cs) \mathrel{\widehat{=}} &H(ok' = (0.ok \wedge 1.ok) \wedge wait' = (0.wait \vee 1.wait) \\ &\wedge\ v(t') = 0.v(t) \oplus 1.v(t) \\ &\wedge\ ref(t') = ((0.ref(t) \cup 1.ref(t)) \cap cs) \vee ((0.ref(t) \cap 1.ref(t))) \\ &\wedge\ t' = t \wedge R_{re} = 0.R_{re} \cup 1.R_{re} \\ &\wedge\ (R'_{av} = 0.R_{av} \cup 1.R_{av} - R_{av}) \lhd (Lea(R_{re}) \subseteq R_{av}) \rhd [\![Stop]\!]_t)\end{aligned}$$

As the shared variable R_{av} is replaced by two variables $0.R_{av}$ and $1.R_{av}$, we need to subtract the initial value of R_{av} in the merge operation.

4.3 Refinement of Programs

The development of a program is often split into a series of steps, each of which focuses on different requirements. For example, the initial design may care about the functional requirements of the specification, and the time and resource requirements come afterwards. [16] discussed how to separate functional requirements of real-time program from the time model, and pointed out that the time model and untime model form a Galois Connection by constructing a pair of left and right adjoint functions. To analyze timed and resource model, we have the following theorem, which shows that program P in resource model preserves the behaviors of it in the corresponding time model.

Theorem 4.1 $[\![P]\!]_{ut} \Rightarrow [\![P]\!]_t$

Proof: By structural induction. □

This theorem shows that a resource model can be regarded as a refinement of the time model, if we neglect variable R_{re}. From the definition of the limited resource model, we have the following lemma:

Lemma 4.2 $[\![P]\!]_{lt} \Rightarrow (([\![P]\!]_{ut} \wedge R'_{av} = R_{av} - Lea(R_{re}))$
$\lhd (R_{av} \supseteq Lea(R_{re})) \rhd [\![Stop]\!]_t)$ □

From above lemma, we have the following theorem disclosing the relation between limited resource model and unlimited resource model.

Theorem 4.3 $(R_{av} \supseteq Lea(R_{re})) \Rightarrow ([\![P]\!]_{lt} \Rightarrow [\![P]\!]_{ut})$ □

That is, if the resources in the system are enough for a program P, the behavior of P is the same as that of P in the unlimited resource model.

5 Case Study

In this section, we will show how the semantic model with resources can be applied to verify the correctness of specifications in the field of co-design, specially the hardware/software partitioning stage. Co-design studies systematically design of systems containing both hardware/software components. Hardware/software partitioning is one of the key steps in co-design [18], which intends to partition a specification into hardware and software components. Most of the approaches proposed earlier are based on heuristic searching [14,11]. One of the key problem here is to preserve the behavior of the specification after partitioning. Some authors introduce algebraic methods to handle this issue [13,6], but they did not consider, for example, time and resources. We can use our resource models to verify the correctness of the partitioning process.

Here we adopt a simple model of partitioning. Suppose a specification contains a series of actions. The partitioning divides these actions into two sets, where actions belong to each set are put into software and hardware respectively. The actions in software and hardware need to communicate with each other.

To justify our method, we use a simple example. The initial specification is:

$IniSpe \mathrel{\widehat{=}} x := 1;\ x := x + 2;\ widle\ 3;\ widle\ 4;\ y := x + 1$

The partitioned program is as follows:

$Progra \mathrel{\widehat{=}} ((x := 1;\ x := x + 2;\ widle\ 3;\ c!x \rightarrow Skip) [\![c]\!]$
$(c?m \rightarrow Skip;\ widle\ 4;\ y := m + 1)) \setminus \{c\}$

Suppose the left side of the parallel structure is put into the software, and the right side into the hardware. As c is an internal event, it is hidden from the environment. Suppose the total resources in hardware is 10 units known in advance. The assignment and weak idle actions both consume one unit of control resources, and communication action consumes two units of control resources in hardware. If a program is put into software, the resources consumed are regarded as memory. Here we assume that the memory resources are enough for the program. Now we would like to identify the following implication:

$$[\![Progra]\!]_{lt} \Rightarrow [\![IniSpe]\!]_t$$

As $(Lea(R_{eq}) \subseteq R_{av})$ is satisfied, from **Theorem 4.1**, we need to prove:

$$[\![Progra]\!]_{ut} \Rightarrow [\![IniSpe]\!]_t$$

For simplicity, we omit the healthiness function H in the following proof. From the definition of composition operator, we have

$$\begin{aligned}
&[\![x := 1;\ x := x + 2;\ \mathit{widle}\ 3;\ \mathit{widle}\ 4;\ y := x + 1]\!]_t \\
&= [\![x := 1;\ x := x + 2]\!]_t \circ [\![\mathit{widle}\ 3;\ \mathit{widle}\ 4;\ y := x + 1]\!]_t \\
&= (\exists\, t_0, v_0 \bullet ok' \wedge \neg wait' \wedge v_0 = 1 \wedge t_0 = t \wedge x(t').r = v_0 + 2 \wedge t' = t_0) \circ \\
&\qquad\qquad [\![\mathit{widle}\ 3;\ \mathit{widle}\ 4;\ y := x + 1]\!]_t \\
&= (ok' \wedge \neg wait' \wedge x(t').r = 3) \circ [\![\mathit{widle}\ 3;\ \mathit{widle}\ 4;\ y := x + 1]\!]_t \\
&= ok' \wedge \neg wait' \wedge x(t').r = 3 \wedge y(t').r = 4 \wedge (t' - t) \leq 7 \wedge \mathit{stable}(tr)
\end{aligned}$$

As the initial specification does not involve communication, the refusal set need not to be considered. By the definition of the merging operator, we have

$$\begin{aligned}
&[\![Progra]\!]_{ut} \\
&= [\![(x := 3;\ \mathit{widle}\ 3;\ c!x \rightarrow \mathit{Skip})]\!]_{ut} \parallel_{M(cs)} \\
&\qquad\qquad [\![c?m \rightarrow \mathit{Skip};\ \mathit{widle}\ 4;\ y := m + 1]\!]_{ut} \setminus \{c\} \\
&\Rightarrow (ok' \wedge \neg wait' \wedge x(t').r = 3 \wedge y(t').r = 4 \wedge t' - t \leq 4 + \varepsilon \wedge \\
&\qquad tr(t') - tr(t) = \{(t_m, c)\} \wedge ref(t') = ref(t)) \setminus \{c\} \qquad \{\text{def. of hiding}\} \\
&\Rightarrow ok' \wedge \neg wait' \wedge x(t').r = 3 \wedge y(t').r = 4 \wedge \\
&\qquad t' - t \leq 4 + \varepsilon \wedge \mathit{stable}(tr) \qquad\qquad \{t' - t \leq 4 + \varepsilon \Rightarrow t' - t \leq 7\} \\
&\Rightarrow ok' \wedge \neg wait' \wedge x(t').r = 3 \wedge y(t').r = 4 \wedge t' - t \leq 7 \wedge \mathit{stable}(tr)
\end{aligned}$$

Thus, we have verified that the final partitioning result is a refinement of the initial specification, that is, $[\![Progra]\!]_{lt} \Rightarrow [\![IniSpe]\!]_t$. The behavior of *Progra* is similar to that of *IniSpe* except that it might run faster.

6 Discussion and Conclusion

In this paper, we proposed an approach to integrate time and resource information into the specification language *Circus*, which can help us to analyze and reason about resource in real-time systems. We also give an example from co-design field to show that the model can be used to prove the correctness of refinements. A proof-assistant tool is being developed based on our semantic model.

In these limited resource models, an implicit assumption is that the resources provided by environment are not reusable, as the required resources of process P are removed from the available resources. However, in the reality, the characters of resource and the policies of management do diverse. In many cases resources can be reused, or some kinds of resources can, others can't. How to integrate different kinds of resources into the timed model is an interesting topic as well.

Another issue we would like to do in the next step is to give a set of refinement laws for our models. *Circus* is a good language for refinements [19]. Woodcock and Cavalcanti [1] discussed the refinement strategy for *Circus*. The refinement laws enable us to develop the program more easily, and to make the procedure of verification of the real-time systems automatically. We hope to develop a set of refinement laws based upon our models as one part of the future work.

Acknowledgement: The authors wish to thank Dr.Tang Xinbei for her helpful comments on the draft.

References

[1] Cavalcanti, A. L. C., A. Sampaio and J. C. P. Woodcock, *A refinement strategy for Circus*, *Formal Aspects of Computing* **15** (2003), pp. 146–181.

[2] Davies, J. and S. Schneider, *A brief history of timed CSP*, *Theoretical Computer Science* **30** (1995), pp. 243–271.

[3] Gerber, R. and I. Lee, *Specification and analysis of resource-bound real-time systems*, in: Proceedings of Real-Time: Theory in Practice (1992), pp. 371–396.

[4] Hayes, I. J., *Reasoning about real-time repetitions: terminating and nonterminating* , *Science of Computer Programming* **43** (2002), pp. 161–192.

[5] Hoare, C. A. R. and J. He, "*Unifying Theories of Programming*," Prentice-Hall International, 1998, 1th edition.

[6] Iyoda, J., A. Sampaio and L. Silva, *ParTS: A partitioning transformation system*, in: World Congress on Formal Methods 1999 (1999), pp. 1400–1419.

[7] Jin, N. and J. He, *Resource Semantic Models for Programming Languages*, Technical Report 277, UNU-IIST (2003).

[8] Lowe, G., *Scheduling-oriented models for real-time systems.* , *The Computer Journal* **38** (1995), pp. 443–456.

[9] Moller, F. and C. Tofts, *A temporal calculus of communicating systems*, in: Proceedings of Concur '90*(LNCS 458)* (1990), pp. 401–415.

[10] Pandya, P. K., H. Wang and Q. Xu, *Toward a theory of sequential hybrid program*, in: Proceedings of PROCOMET'98, 1998, pp. 366–384.

[11] Pu, G., D. V. Hung, J. He and Y. Wang, *An optimal approach to hardware/software partitioning for synchronous model*, in: IFM 2004*(LNCS 2999)* (2004), pp. 363–381.

[12] Qin, S., J. S. Dong and W.-N. Chin, *A semantic foundation for TCOZ in unifying theories of programming*, in: FME 2003(LNCS 2805) (2003), pp. 321–340.

[13] Qin, S. and J. He, *An algebraic approach to hardware/software partitioning*, in: Proc. of the 7th IEEE International Conference on Electronics, Circuits and Systems (2000), pp. 273–276.

[14] Quan, G., X. Hu and G. W. Greenwood, *Preference-driven hierarchical hardware/software partitioning*, in: Internatitional conference on Computer Design (1999), pp. 652–657.

[15] Scholefield, D., H. Zedan and J. He, *A specification oriented semantics for the refinement of real-time systems* , *Theoretical Computer Science* **131** (1994), pp. 219–241.

[16] Sherif, A. and J. He, *Towards a time model for Circus*, in: Proceedings of the 4th International Conference on Formal Engineering Methods (LNCS 2495) (2002), pp. 613–624.

[17] Wang, Y., *Real-time behaviour of asynchronous agents* , in: Proceedings of Concur '90*(LNCS 458)* (1990), pp. 502–520.

[18] Wolf, W., *Hardware-Software co-design of embedded system* , *Proc. of the IEEE* **82** (1994), pp. 967–989.

[19] Woodcock, J. C. P. and A. L. C. Cavalcanti, *The semantics of Circus* , in: ZB2002*(LNCS 2272)* (2002), pp. 184–203.