An Operational Semantics for Timed RAISE

Xia Yong and Chris George

United Nations University/International Institute for Software Technology,
P.O.Box 3058, Macau
{xy,cwg}@iist.unu.edu
http://www.iist.unu.edu/{∼xy,∼cwg}

Abstract. The reliability of software is an increasingly important demand, especially for safety critical systems. RAISE is a mathematically based method which has been shown to be useful in the development of many kinds of software systems. However, RAISE has no particular features for specifying real-time requirements, which often occur in safety critical systems. Adding timing features to RAISE makes a new specification language, Timed RAISE Specification Language (TRSL), and gives it the power of specifying real-time applications. We then have to find a theoretical foundation for TRSL. In this paper, an operational semantics of TRSL is first introduced. Then we define a pre-order and test equivalence relation for TRSL. Some proof rules for TRSL are listed, and their soundness corresponding to our operational model is also explained.

1 Introduction

The reliability of software is an increasingly important demand, especially for critical systems like train control systems or banking systems, for which failures may have very severe consequences. Mathematically based "formal" methods for specification and stepwise development of software have been invented in order to increase the reliability of software. Some of these languages provide facilities to specify concurrent systems, and therefore, they can capture various *qualitative* aspects of system behaviour, such as deadlock, synchronisation and safety. However, in a real-time system we may be concerned with the timing of events. We might want not merely to say that an event occurs, but to say that it occurs within a particular time interval.

RAISE is a mathematically based method which has been shown to be useful in the development of many kinds of software systems. However, RAISE has no particular features for specifying such real-time requirements. Adding real-time features to RAISE Specification Language (RSL) is not only an interesting topic for theoretical computer science research, but also a requirement of some RAISE users.

Integrating RSL with a real-time logic, the Duration Calculus (DC) [ZHR91], seems a good choice to achieve the above aim. RAISE has good features (in particular modularity) for describing large systems, while DC is concerned only with timing properties. The degree of overlap between the two languages is therefore very small.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1008–1027, 1999.

We do not wish to perform a syntactic integration of RSL and DC. This would create a large language and probably cause the added complications of time to permeate much of RSL. Instead we note that adding time to a description can be seen as a design step. For example, going from "B must follow A" to "B must follow A within 3 time units" adds an extra constraint, a design decision. It therefore seems reasonable to add time within the part of RSL that is used later in design. The idea is then to be able to (partially) *interpret* TRSL descriptions in terms of DC formulae, and show that these formulae satisfy the timing requirements, also written in DC.

So we have two tasks. The first is extending original RSL to Timed RSL (TRSL) by introducing some real-time constructs. The second step is relating TRSL to DC. This paper concentrates on the first of these.

The proposed TRSL, the syntactic extension to RSL, can be found in [GX98]. Section 2 summarises the proposed extension and discusses its effect on the existing language and its proof system.

After syntactically proposing TRSL, we should establish a theoretical foundation for this new specification language. The theoretical foundation we need is the proof system, the collection of rules that enable us to reason about specifications. In this paper we propose an operational semantics and show how it can be used to establish validity of proof rules. We give an operational semantics of TRSL in Section 3, define an equivalence relation among TRSL expressions in Section 4, and apply it to the proof of soundness of TRSL proof rules in Section 5. Section 6 considers future and related work.

2 Adding Time to RSL

We would like the addition of time to RSL to be the smallest extension that gives us a useful language, and if possible for it to be a conservative extension, i.e. for it to leave the existing proof rules unchanged. By "useful" we mean expressive and convenient for establishing properties. The latter implies an intuitive and simple proof system, which in turn suggests a simple semantics.

The simplest extension to RSL to include time would seem to be to add a **wait** expression. Since we want eventually to relate timed RSL (TRSL) to DC we will make the parameter type of **wait** non-negative reals, which we will define as the type **Time**. For convenience, we allow natural numbers as arguments of **wait** by overloading it. A **Nat** argument is converted to **Time** by the existing RSL prefix operator **real**. For example, **wait** 1 is equivalent to **wait** 1.0.

If we need a parallel expansion rule, it seems necessary also to add a new construct, "time dependence", to input and output. An input, as well as returning the value input, will also return a time value representing the time elapsed between the input being ready and the the communication taking place. Similarly, an output will return the time elapsed between the output being ready and the the communication taking place.

The extension defined here owes much to the work of Wang Yi [Wang91]. He in particular introduced time dependence. We also follow him in making only

wait expressions, and input and output, cause or allow time to elapse. All other evolutions of expressions are regarded as instantaneous.

We also follow Wang Yi in adopting the *maximal progress* assumption. This means that the time between an input or output being ready and the communication taking place is minimised. In other words, when an expression can evolve without waiting for the environment, it will not wait.

This raises a question of what we mean by an internal (non-deterministic) choice like

e1 ⌈⌉ **wait** 1 ; e2

where *e1* and *e2* do not initially wait. Blindly applying the maximum progress assumption leads to this expression evolving only to *e1*. But this would remove the possibility of specifying an expression that might immediately perform *e1*, or might (non-deterministically) wait for one time unit and then perform *e2*. We want to allow this possibility in specification. This leads to the need for a new operator to replace internal choice in the parallel and interlock expansion rules, where we need the "maximal progress" version of internal choice. But this is no more than the addition of another special function internal to the proof rules: it is not needed in the language.

To see how **wait** can be used in parallel or interlocked composition, consider

c? ; normal() [] **wait** 1 ; time_out()

The intention is that this expression initially waits for its environment to offer an output on channel *c*. If this output on channel *c* is available within 1 time unit then the communication should be accepted and *normal()* is executed. If the output is not available within 1 time unit then it should instead execute *time_out()*. We can specify these behaviours using the RSL interlock operator ⫴. Interlocked composition is like parallel composition in that the expressions evolve concurrently, but more aggressive: it forces them to communicate only with each other until one of them terminates. We expect the following equivalences to hold for any strictly positive *k*, assuming that *time_out()* can not itself initially wait:

(c? ; normal() [] **wait** 1 ; time_out()) ⫴ (**wait**(1 − k) ; c!()) ≡
wait(1 − k) ; normal()

(c? ; normal() [] **wait** 1 ; time_out()) ⫴ **wait**(1 + k) ≡
wait 1 ; (time_out() ⫴ **wait** k)

2.1 Conservative Extension

Conservative extension of RSL to make TRSL, i.e. all existing RSL proof rules being unchanged, would be ideal but does not seem to be achievable. There are two problems.

First, introducing time can reduce non-determinacy. For example, specifying an expression like the one we considered earlier, that will take a special action (time-out) if some event does not occur within a specified period, can only be specified without time as a non-deterministic choice between the normal and time-out behaviour. When time is included we may be able to calculate which behaviour will be taken; the non-determinacy may be reduced.

Secondly, there are are some rules in RSL that we expect not to hold because of the kind of properties we are interested in when we want to relate TRSL to DC. DC is concerned with the duration of states, i.e. for how long properties hold. We expect properties to be reflected in the values of imperative variables in RSL. Now consider the following equivalence that is valid in RSL, provided the expression e does not involve input or output and is convergent:

c? ; v := e ≡ v := e ; c?

The assignment and the input can be commuted. In TRSL in general we have to introduce a let expression for the time dependence. We would expect from the RSL proof rules, provided again e does not involve input or output and is convergent, and provided also that t is not free in e, to be able to derive the following:

let t = c? **in** v := e **end**
≡
let t = v := e ; c? **in skip end**
≡
v := e ; **let** t = c? **in skip end**

It is not immediately clear what the meaning of the second expression should be, but it is clear that the last would differ from the first in changing the duration of the state in which v has the value e; the possible wait for the communication on c shifts from before the assignment to after it. So this derivation cannot be allowed in TRSL.

These two examples, of reduced non-determinism and restrictions on commuting expressions, do seem, however, to encompass the problems. It also seems likely (though this is the subject of further work) that there is a reduction from TRSL to RSL (just throwing away the timing information) that is consistent with a "more deterministic" ordering: the ordering derived later in in Section 4.2. That is, any behaviour of a timed specification will be a possible behaviour of its reduction to an untimed one. The second problem involves the strengthening of applicability conditions for commuting sequential expressions.

3 Operational Semantics

For the sake of clarity, we follow the approach of [HI93, BD93, Deb94]: the operational semantics in this paper for untimed part of TRSL is closely based on them, and we only consider a core syntax of TRSL. Our operational semantics can be viewed as a version of Timed CCS [Wang91] without τs.

3.1 The Core Syntax

For simplicity we restrict the types of expressions to be **Unit**, **Bool** and **Real**. The set of allowed expressions includes:

- As constants the reals, the booleans **true** and **false**, the **Unit** value (). The basic expression **skip** is an expression that immediately terminates successfully. We consider also the basic expression **stop** which represents deadlock and the basic expression **chaos** which stands for the divergent process.
- Three binding operators that are the abstraction, the recursion and the let definition (λ, **rec**, **let**). The reader should notice that the **rec** is not an RSL binding operator: RSL does not syntactically distinguish recursion. In the core syntax, it is convenient to indicate where recursion may occur.
- Imperative aspects are supported through the notion of variables and assignment.
- We have the following combinators:
  - _⌈⌉_ : Nondeterministic choice between two expressions (also called internal choice). One of the two expressions is selected nondeterministically for evaluation.
  - _[]_ : External choice between two expressions. The choice is context dependent, i.e. the environment influences the choice between the two expressions.
  - _‖_ : Parallel composition of two expressions.
  - _#_ : The interlock operator. It is similar to the parallel operator, but more aggressive. In other words, two interlocked expressions will communicate if they are able to communicate with one another. If they are able to communicate with other concurrently executing value expressions but not with each other, they deadlock unless one of them can terminate. The interlock operator is the main novelty in the RSL process algebra. It has been devised mainly to allow implicit specification of concurrency.
  - _;_ : Sequencing operator.

The above operators in TRSL have the same meanings as those in RSL. We also have the extensions to be included:

- TRSL is essentially independent of the time domain. For simplicity, in our core syntax of TRSL, we just assume the time Domain to be Real^{+0}.
- The expression **wait** E means we first evaluate the expression E, get the result d, then delay exactly d units of time.
- Expressions may communicate through unidirectional channels. The expression **let** t = c!E1 **in** E2 means: evaluate E1, send the result (when possible) on the channel c, and then evaluate E2. t records the time between the communication on c being ready and it occurring. The expression **let** t = c?x **in** E means: assign any value received on the channel c to variable x, and then evaluate E. Again, t records the time between the communication on c being ready and it occurring.

More formally the BNF syntax of our language is:

Syntactic Categories:

- E in Expressions
- x in Variables
- t, id in Identifiers
- c in Channels
- r in Reals
- T in Time
- τ in Types
- V in ValueDefinitions

Expression The BNF grammar of expressions is:

V ::= id : τ | id : τ, V
E ::= () | **true** | **false** | r | T | id | x | **skip** | **stop** | **chaos** |
x := E | **if** E **then** E **else** E | **let** id = E **in** E |
wait E | **let** t = c?x **in** E | **let** t = c!E **in** E |
E $\lceil\rceil$ E | E $[]$ E | E $\|$ E | E $\|\|$ E | E ; E |
λ id : τ • E | E E | **rec** id : τ • E |

In fact E ; E′ is equivalent to **let** id = E **in** E′ provided id is chosen so as not to be free in E′. We include E ; E to give a conventional presentation.

3.2 Definition

Store A store s is a finite map from variables (noted x) to values (noted v):
$s = [x \mapsto v, ...]$

Environment An environment ρ is a finite map from identifiers (noted id) to values (noted v):
ρ = [id $\mapsto$ v, ...]

Closures A closure, $[\![\lambda$ id : τ • E, $\rho]\!]$, is a pair made of

- a lambda expression : λ id : τ • E
- an environment : ρ

Computable Values $\mathcal{V}$ is the least set which satisfies:

- $\mathcal{V}$ contains values from our types: (), **true**, **false**, ... , -1, ..., 0, ..., 1, ...
- if ρ is an environment, then $\mathcal{V}$ contains $[\![\lambda$ id : τ • E, $\rho]\!]$

Expressions and Computable Values The set $\mathcal{EV}$ of expressions and computable values is defined as
$\mathcal{EV} = \mathcal{E} \cup \mathcal{V}$

Events "$\diamond$" denotes any event;
"$\triangle$" denotes visible events and silent events.

Visible events
Visible events a consist of :
- input events : $c?v$
- output events : $c!v$

where c is a channel and v is a value in $\mathcal{V}$.
$\bar{a}$ denotes the complement action of a (e.g. : $\overline{c?v} = c!v$).

Time-measurable events
$\varepsilon(d)$ denotes waiting d unit of time, where d is a value from the time domain and $d > 0$.

Silent events
ε denotes internal moves, including internal behaviours of communication (which is denoted as "τ" in CCS).

Time Model We assume that all silent events can perform instantaneously and will never wait unnecessarily. Once both sides of a channel are ready for communication, the communication will happen without any delay (unless some other visible event or silent event happens instead) and the communication takes no time.

The above assumptions are conventional and the reason for adopting them is just to make proof theory easier.

Notations We introduce some notations that are used later.

1. v, v', ... represent values drawn from $\mathcal{V}$
2. d, d', ... represent values drawn from the **Time** domain.
3. ev, ev', ... represent values drawn from $\mathcal{EV}$,
4. $a, \bar{a}, \varepsilon(d), \varepsilon, \triangle, \diamond$ ... represent events,
5. E, E_i, ... represent expressions.
6. x, y, ... represent variables.
7. s, s', s'', ... represent stores.

Configurations Our operational semantics is based on the evolution of configurations.

The set of basic configurations $\mathcal{BC}$ is defined as:

$$\mathcal{BC} = \{< ev, s > \mid ev \ \in \mathcal{EV} \wedge s \in Store\}$$

The set of configurations, $\mathcal{C}$, is the least set which satisfies:

1. $\mathcal{BC} \subset \mathcal{C}$
2. $\alpha, \beta \in \mathcal{C}$ implies $\alpha \ op \ \beta \in \mathcal{C}$ where: $op = \lceil\!\rceil$, $[]$, $\|$, $\|\!\|$
3. $\alpha, \beta \in \mathcal{C}$ implies $\alpha \ op \ s \ op \ \beta \in \mathcal{C}$ where: $op = \|$, $\|\!\|$
4. $\alpha \in \mathcal{C}$ implies $\alpha \ ; \ E, wait \ \alpha, x := \alpha, (\alpha \ E) \in \mathcal{C}$

5. $\alpha \in \mathcal{C}$ implies $\alpha v \in \mathcal{C}$
6. $\alpha \in \mathcal{C}$ implies :
   (a) $[\![\lambda \text{ id} : \tau \bullet \alpha, \rho]\!]\ v \in \mathcal{C}$
   (b) $[\![\lambda \text{ id} : \tau \bullet E, \rho]\!]\ \alpha \in \mathcal{C}$
7. $\alpha \in \mathcal{C}$ implies :
   (a) **let** $id = \alpha$ **in** $E \in \mathcal{C}$
   (b) **if** α **then** E_1 **else** $E_2 \in \mathcal{C}$
   (c) **let** $t = c!\ \alpha$ **in** $E \in \mathcal{C}$

3.3 Operational Rules

The operational rules are given in Figure 1 and Figure 2. Each rule is divided into two parts: the lower part describes the possible evolution of the configurations, and the upper part presents the precondition of that evolution. □ indicates that there is no precondition.

We use the standard notation $E[v/t]$ to describe the substitution of v for all free occurrences of the identifier t in E.

3.4 Semantic Function : Merge

$\text{merge(s, s}', \text{ s}'') = \text{s}' \dagger [\,\text{x} \mapsto \text{s}''(\text{x}) \mid \text{x} \in \mathbf{dom}(\text{s}'') \cap \mathbf{dom}(\text{s}) \bullet \text{s(x)} \neq \text{s}''(\text{x})\,]$

3.5 Meaning of "Sort$_d$" and "SORT$_d$"

Sort$_d$ $\text{Sort}_d(\alpha)$ is a set of ports (channel names tagged as input or output), whose intuitive meaning is the possible (input or output) visible events that α can evolve to within the next d units of time. We define "Sort$_d$" inductively according to configuration structures.

We find that there are three kinds of configuration that can evolve with $\varepsilon(d)$: **wait**, input and output. So, they are named "Basic Forms". There are some other kinds of configurations that can evolve with $\varepsilon(d)$, if their components are in Basic Forms. They are named "Extended Forms".

BASIC FORMS:

- $\text{Sort}_0(\alpha) = \emptyset$ for $\alpha \in \mathcal{C}$
- $Sort_d(c?) = \overline{Sort_d(c!)}$ and $Sort_d(c!) = \overline{Sort_d(c?)}$ for any channel c
- $\text{Sort}_d(\ \mathbf{wait} < (d + d'),\ s >) = \emptyset$
- $\text{Sort}_d(\ < \mathbf{let}\ t = c?x\ \mathbf{in}\ E\ , s >) = \{c?\}$
- $\text{Sort}_d(\ \mathbf{let}\ t = c! < v, s > \ \mathbf{in}\ E = \{c!\}$

EXTENDED FORMS:
Assume that α and β are one of the Basic Forms.

- $\text{Sort}_{d+d'}(\mathbf{wait}d; E) = \text{Sort}_{d'}(E)$
- $\text{Sort}_d(\alpha\ ;\ \text{E}) = \text{Sort}_d(\alpha)$

Basic Expressions

$$\frac{\square}{\rho \vdash < \mathbf{skip}, s > \xrightarrow{\varepsilon} < (), s >}$$

$$\frac{\square}{\rho \vdash < \mathbf{stop}, s > \xrightarrow{\varepsilon(d)} < \mathbf{stop}, s >}$$

$$\frac{\square}{\rho \vdash < \mathbf{chaos}, s > \xrightarrow{\varepsilon} < \mathbf{chaos}, s >}$$

Configuration Fork

$$\frac{\square}{\rho \vdash < E_1 \ op \ E_2, s > \xrightarrow{\varepsilon} < E_1, s > \ op \ < E_2, s >}$$

where $op = \sqcap,\ \mathbin{[]}$

Look Up

$$\frac{\square}{\rho \dagger [id \mapsto v] \vdash < id, s > \xrightarrow{\varepsilon} < v, s >}$$

$$\frac{\square}{\rho \vdash < id, s \dagger [id \mapsto v] > \xrightarrow{\varepsilon} < v, s \dagger [id \mapsto v] >}$$

Sequencing

$$\frac{\square}{\rho \vdash < E_1 \ ; \ E_2, s > \xrightarrow{\varepsilon} < E_1, s > \ ; \ E_2}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash \alpha \ ; \ E \xrightarrow{\diamond} \alpha' \ ; \ E}$$

$$\frac{\square}{\rho \vdash < v, s > ; E \xrightarrow{\varepsilon} < E, s >}$$

Assignment

$$\frac{\square}{\rho \vdash < x := E, s > \xrightarrow{\varepsilon} x :=< E, s >}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash x \ := \ \alpha \xrightarrow{\diamond} x \ := \ \alpha'}$$

$$\frac{\square}{\rho \vdash x :=< v, s > \xrightarrow{\varepsilon} < (), s \dagger [x \mapsto v] >}$$

Waiting

$$\frac{\square}{\rho \vdash < \mathbf{wait} \ E, \ s > \xrightarrow{\varepsilon} \mathbf{wait} \ < E, \ s >}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash \mathbf{wait} \ \alpha \xrightarrow{\diamond} \mathbf{wait} \ \alpha'}$$

$$\frac{\square}{\rho \vdash \mathbf{wait} < (d + d'), \ s > \xrightarrow{\varepsilon(d)} \mathbf{wait} \ < d', \ s >}$$

when $\{ \ d \ > \ 0 \ \}$

$$\frac{\square}{\rho \vdash \mathbf{wait} \ < (0), \ s > \xrightarrow{\varepsilon} < (), \ s >}$$

Input

$$\frac{\square}{\rho \vdash < \mathbf{let} \ t \ = c?x \ \mathbf{in} \ E, s > \xrightarrow{c?v} < E[0/t], s \dagger [x \mapsto v] >}$$

$$\frac{\square}{\rho \vdash < \mathbf{let} \ t \ = c?x \ \mathbf{in} \ E, s > \xrightarrow{\varepsilon(d)} < \mathbf{let} \ t \ = c?x \ \mathbf{in} \ E[t + d/t], s >}$$

Output

$$\frac{\square}{\rho \vdash < \mathbf{let} \ t \ = \ c \ ! \ E \ \mathbf{in} \ E, \ s > \xrightarrow{\varepsilon} \ \mathbf{let} \ t \ = c \ ! < E, \ s > \ \mathbf{in} \ E}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash \mathbf{let} \ t \ = \ c \ ! \ \alpha \ \mathbf{in} \ E \xrightarrow{\diamond} \mathbf{let} \ t \ = \ c \ ! \ \alpha' \ \mathbf{in} \ E}$$

$$\frac{\square}{\rho \vdash \mathbf{let} \ t = c \ ! < v, s > \ \mathbf{in} \ E \xrightarrow{c!v} < E[0/t], s >}$$

$$\frac{\square}{\rho \vdash \mathbf{let} \ t = c \ ! \ < v, s > \ \mathbf{in} \ E \xrightarrow{\varepsilon(d)} \mathbf{let} \ t \ = \ c \ ! < v, s > \ \mathbf{in} \ E[t + d/t]}$$

Internal Choice

$$\frac{\square}{\rho \vdash \alpha \ \sqcap \ \beta \xrightarrow{\varepsilon} \alpha \quad \xrightarrow{\varepsilon} \beta}$$

External Choice

$$\frac{\rho \vdash \alpha \xrightarrow{a} \alpha'}{\begin{array}{r}\rho \vdash \alpha \mathbin{[]} \beta \xrightarrow{a} \alpha' \\ \beta \mathbin{[]} \alpha \xrightarrow{a} \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\varepsilon(d)} \alpha' \ , \ \rho \vdash \beta \xrightarrow{\varepsilon(d)} \beta'}{\begin{array}{r}\rho \vdash \alpha \mathbin{[]} \beta \xrightarrow{\varepsilon(d)} \alpha' \mathbin{[]} \beta' \\ \beta \mathbin{[]} \alpha \xrightarrow{\varepsilon(d)} \beta' \mathbin{[]} \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\varepsilon} \alpha'}{\begin{array}{r}\rho \vdash \alpha \mathbin{[]} \beta \xrightarrow{\varepsilon} \alpha' \mathbin{[]} \beta \\ \beta \mathbin{[]} \alpha \xrightarrow{\varepsilon} \beta \mathbin{[]} \alpha'\end{array}}$$

$$\frac{\square}{\begin{array}{r}\rho \vdash < v, s > \mathbin{[]} \alpha \xrightarrow{\varepsilon} < v, s > \\ \alpha \mathbin{[]} < v, s > \xrightarrow{\varepsilon} < v, s >\end{array}}$$

Fig. 1. Operational Rules for TRSL : Part 1

Parallel Combinator

$$\frac{}{\rho \vdash < E_1 \parallel E_2,\ s > \xrightarrow{\varepsilon} < E_1,\ s > \parallel s \parallel < E_2,\ s >}$$

$$\frac{\rho \vdash \alpha \xrightarrow{a} \alpha' \ , \rho \vdash \beta \xrightarrow{\bar{a}} \beta'}{\begin{array}{l}\rho \vdash \alpha \parallel s \parallel \beta \xrightarrow{\varepsilon} \alpha' \parallel s \parallel \beta' \\ \beta \parallel s \parallel \alpha \xrightarrow{\varepsilon} \beta' \parallel s \parallel \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\triangle} \alpha'}{\begin{array}{l}\rho \vdash \alpha \parallel s \parallel \beta \xrightarrow{\triangle} \alpha' \parallel s \parallel \beta \\ \beta \parallel s \parallel \alpha \xrightarrow{\triangle} \beta \parallel s \parallel \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\varepsilon(d)} \alpha' \ , \rho \vdash \beta \xrightarrow{\varepsilon(d)} \beta'}{\begin{array}{l}\rho \vdash \alpha \parallel s \parallel \beta \xrightarrow{\varepsilon(d)} \alpha' \parallel s \parallel \beta' \\ \beta \parallel s \parallel \alpha \xrightarrow{\varepsilon(d)} \beta' \parallel s \parallel \alpha'\end{array}}$$

$$when \left\{ \begin{array}{l} [\ Sort_d(\alpha) \cap \overline{Sort_d(\beta)} = \emptyset\ ; \\ \ \ Sort_d(\alpha) \cap \overline{SORT_d} = \emptyset\ ; \\ \ \ Sort_d(\beta) \cap \overline{SORT_d} = \emptyset\ ] \end{array} \right\}$$

$$\frac{}{\begin{array}{l}\rho \vdash \alpha \parallel s \parallel < v,\ s' > \xrightarrow{\varepsilon} \alpha \parallel s \parallel s' \\ < v,\ s' > \parallel s \parallel \alpha \xrightarrow{\varepsilon} s' \parallel s \parallel \alpha\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\begin{array}{l}\rho \vdash \alpha \parallel s \parallel s' \xrightarrow{\diamond} \alpha' \parallel s \parallel s' \\ s' \parallel s \parallel \alpha \xrightarrow{\diamond} s' \parallel s \parallel \alpha'\end{array}}$$

$$\frac{}{\begin{array}{l}\rho \vdash < v, s'' > \parallel s \parallel s' \xrightarrow{\varepsilon} < v, merge(s, s', s'') > \\ s' \parallel s \parallel < v, s'' > \xrightarrow{\varepsilon} < v, merge(s, s', s'') >\end{array}}$$

Interlocking

$$\frac{}{\rho \vdash < E_1 \ \sharp\ E_2,\ s > \xrightarrow{\varepsilon} < E_1,\ s > \sharp\ s \ \sharp\ < E_2,\ s >}$$

$$\frac{\rho \vdash \alpha \xrightarrow{a} \alpha' \ , \rho \vdash \beta \xrightarrow{\bar{a}} \beta'}{\begin{array}{l}\rho \vdash \alpha \ \sharp\ s \ \sharp\ \beta \xrightarrow{\varepsilon} \alpha' \ \sharp\ s \ \sharp\ \beta' \\ \beta \ \sharp\ s \ \sharp\ \alpha \xrightarrow{\varepsilon} \beta' \ \sharp\ s \ \sharp\ \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\varepsilon} \alpha'}{\begin{array}{l}\rho \vdash \alpha \ \sharp\ s \ \sharp\ \beta \xrightarrow{\varepsilon} \alpha' \ \sharp\ s \ \sharp\ \beta \\ \beta \ \sharp\ s \ \sharp\ \alpha \xrightarrow{\varepsilon} \beta \ \sharp\ s \ \sharp\ \alpha'\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\varepsilon(d)} \alpha' \ , \rho \vdash \beta \xrightarrow{\varepsilon(d)} \beta'}{\begin{array}{l}\rho \vdash \alpha \ \sharp\ s \ \sharp\ \beta \xrightarrow{\varepsilon(d)} \alpha' \ \sharp\ s \ \sharp\ \beta' \\ \beta \ \sharp\ s \ \sharp\ \alpha \xrightarrow{\varepsilon(d)} \beta' \ \sharp\ s \ \sharp\ \alpha'\end{array}}$$

$$when\ \{\ Sort_d(\alpha) \cap \overline{Sort_d(\beta)} = \emptyset\}$$

$$\frac{}{\begin{array}{l}\rho \vdash \alpha \ \sharp\ s \ \sharp\ < v,\ s' > \xrightarrow{\varepsilon} \alpha \ \sharp\ s \ \sharp\ s' \\ < v,\ s' > \sharp\ s \ \sharp\ \alpha \xrightarrow{\varepsilon} s' \ \sharp\ s \ \sharp\ \alpha\end{array}}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\begin{array}{l}\rho \vdash \alpha \ \sharp\ s \ \sharp\ s' \xrightarrow{\diamond} \alpha' \ \sharp\ s \ \sharp\ s' \\ s' \ \sharp\ s \ \sharp\ \alpha \xrightarrow{\diamond} s' \ \sharp\ s \ \sharp\ \alpha'\end{array}}$$

$$\frac{}{\begin{array}{l}\rho \vdash < v, s'' > \sharp\ s \ \sharp\ s' \xrightarrow{\varepsilon} < v, merge(s, s', s'') > \\ s' \ \sharp\ s \ \sharp < v,\ s'' > \xrightarrow{\varepsilon} < v, merge(s,\ s',\ s'') >\end{array}}$$

Function

$$\frac{}{\rho \vdash < E_1\ E_2, s > \xrightarrow{\varepsilon} < E_1,\ s >\ E_2}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash \alpha\ E \xrightarrow{\diamond} \alpha'\ E}$$

$$\frac{}{\rho \vdash < \lambda\ id : \tau \bullet E, s > \xrightarrow{\varepsilon} < [\lambda\ id : \tau \bullet E, \rho\ ], s >}$$

$$\frac{}{\rho \vdash < [\lambda\ id : \tau \bullet E_1, \rho_1\ ], s >\ E_2 \xrightarrow{\varepsilon} [\lambda\ id : \tau \bullet E_1, \rho_1]\ < E_2,\ s >}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash [\lambda\ id : \tau \bullet E, \rho_1\ ]\ \alpha \xrightarrow{\diamond} [\lambda\ id : \tau \bullet E, \rho_1\ ]\ \alpha'}$$

$$\frac{}{\rho \vdash [\lambda\ id : \tau \bullet E, \rho_1\ ]\ < v,\ s > \xrightarrow{\diamond} [\lambda\ id : \tau \bullet < E,\ s >, \rho_1\ ]\ v}$$

$$\frac{\rho_1 \dagger [id \mapsto v] \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash [\lambda\ id : \tau \bullet \alpha, \rho_1\ ]\ v \xrightarrow{\diamond} [\lambda\ id : \tau \bullet \alpha', \rho_1\ ]\ v}$$

$$\frac{\rho_1 \dagger [id \mapsto v] \vdash \alpha \xrightarrow{\diamond} \ < v',\ s >}{\rho \vdash [\lambda\ id : \tau \bullet \alpha, \rho_1\ ]\ v \xrightarrow{\diamond} < v',\ s >}$$

Let Expression

$$\frac{}{\rho \vdash < \textbf{let}\ id\ =\ E_1\ \textbf{in}\ E_2,\ s > \xrightarrow{\varepsilon} \textbf{let}\ id\ = < E_1,\ s >\ \textbf{in}\ E_2}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash\ \textbf{let}\ id\ =\ \alpha\ \textbf{in}\ E \xrightarrow{\diamond} \textbf{let}\ id\ =\ \alpha'\ \textbf{in}\ E}$$

$$\frac{}{\rho \vdash \textbf{let}\ id\ =\ < v,\ s >\ \textbf{in}\ E \xrightarrow{\varepsilon}\ < E[v/id],\ s >}$$

If Expression

$$\frac{}{\rho \vdash < \textbf{if}\ E\ \textbf{then}\ E_1\ \ \textbf{else}\ E_2,\ s > \xrightarrow{\varepsilon}\ \textbf{if}\ < E,\ s >\ \textbf{then}\ E_1 \textbf{else}\ E_2}$$

$$\frac{\rho \vdash \alpha \xrightarrow{\diamond} \alpha'}{\rho \vdash\ \textbf{if}\ \alpha\ \textbf{then}\ E_1\ \ \textbf{else}\ E_2 \xrightarrow{\diamond}\ \textbf{if}\ \alpha'\ \textbf{then}\ E_1\ \ \textbf{else}\ E_2}$$

$$\frac{}{\rho \vdash \textbf{if}\ \ < \textbf{true},\ s >\ \textbf{then}\ E_1\ \ \textbf{else}\ E_2\ \xrightarrow{\varepsilon}\ \ < E_1, s >}$$

$$\frac{}{\rho \vdash \textbf{if}\ \ < \textbf{false},\ s >\ \textbf{then}\ E_1\ \ \textbf{else}\ E_2\ \xrightarrow{\varepsilon}\ \ < E_2, s >}$$

Recursion

$$\frac{}{\rho \vdash < \textbf{rec}\ id :\ \tau \bullet E,\ s > \xrightarrow{\varepsilon} < E[\textbf{rec}\ id :\ \tau \bullet E/id],\ s >}$$

Fig. 2. Operational Rules for TRSL : Part 2

- $\mathrm{Sort}_d(x := \alpha) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\mathbf{wait}\ \alpha) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\mathbf{let}\ t = c!\ \alpha\ \mathbf{in}\ \mathrm{E}) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\alpha \parallel s \parallel s') = \mathrm{Sort}_d(s' \parallel s \parallel \alpha) = Sort_d(\alpha)$
- $\mathrm{Sort}_d(\alpha \between s \between s') = \mathrm{Sort}_d(s' \between s \between \alpha) = Sort_d(\alpha)$
- $\mathrm{Sort}_d(\alpha\ \mathrm{E}) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\alpha\ \mathrm{v}) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d([\![\lambda\ id : \tau \bullet \alpha,\ \rho]\!]\ v) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\mathbf{let}\ id = \alpha\ \mathbf{in}\ \mathrm{E}) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\mathbf{if}\ \alpha\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2) = \mathrm{Sort}_d(\alpha)$
- $\mathrm{Sort}_d(\alpha\ op\ \beta) = \mathrm{Sort}_d(\alpha) \cup \mathrm{Sort}_d(\beta)$ where op = "[]", "||"

$\mathbf{SORT_d}$ SORT_d is a set of ports. Its definition is just same as Sort_d, but can only be calculated if we know what the environment expressions are. I.e. port c? (c!) $\in \mathrm{SORT}_d$ means that within d units of time, there are some other processes that will be ready for complementary communication, c! (c?), on channel c.

3.6 Commentary on Operational Rules

The transition relation is defined as the smallest relation satisfying the axioms and rules given in our operational rules. We note in particular:

Time-measurable event A configuration can evolve with a time-measurable event only if all its sub-configurations on both sides of combinators [], || and ⫴, can evolve with this same time-measurable event.

Maximal progress Maximal progress in RSL means that once a communication on a channel is ready, it will never wait. In the rules for interlocking, the semantic function, Sort_d, is used to specify that only if no pair of complementary actions, one from each side of the combinator, is ready for communication, can this configuration evolve with a time-measurable event. In the rules for parallel combinator, the condition is stronger: a configuration can evolve with a time-measurable event only when no communication is possible, either internal (between the parallel expressions) or external (between one of them and the environment). (c.f. Section 2). Using "Sort (SORT)" to guarantee that the composite processes satisfy maximal progress was first proposed by Wang Yi in his work on Timed CCS [Wang91].

4 Time Test Equivalence

4.1 Definitions

- Let l be a sequence of events, α, β two configurations in $\mathcal{C}$, d $\in$ **Time** and d > 0. We define $\alpha \overset{l}{\Rightarrow} \beta$ by:
  1. $\alpha \overset{<>}{\Longrightarrow} \beta$ if $\alpha\ (\overset{\varepsilon}{\rightarrow})^*\ \beta$.

2. $\alpha \stackrel{al'}{\Longrightarrow} \beta$ if for some α, α', α'' we have : $\alpha \stackrel{<>}{\Longrightarrow} \alpha'$, $\alpha' \stackrel{a}{\rightarrow} \alpha''$, and $\alpha'' \stackrel{l'}{\Longrightarrow} \beta$.
3. $\alpha \stackrel{\varepsilon(d)l'}{\Longrightarrow} \beta$ if for some α, α', α'' we have : $\alpha \stackrel{<>}{\Longrightarrow} \alpha'$, $\alpha' \stackrel{\varepsilon(d)}{\longrightarrow} \alpha''$, and $\alpha'' \stackrel{l'}{\Longrightarrow} \beta$.

where <> stands for the empty sequence. Moreover, we merge successive time-measurable events by treating the sequence $\varepsilon(d_1)\varepsilon(d_2)...\varepsilon(d_n)$ as the event $\varepsilon(d_1 + d_2 + ... + d_n)$.

– Let L be set of traces of a configuration, defined as :

$$L(\alpha) = \{l \mid for\ some\ \beta\ ,\ \alpha \stackrel{l}{\Longrightarrow} \beta\}$$

– We define the following convergence predicates:
1. We write $\alpha \downarrow$ if there is no infinite sequence of internal moves:

$$\alpha = \alpha_0 \stackrel{\varepsilon}{\rightarrow} \alpha_1 \stackrel{\varepsilon}{\rightarrow} ...$$

2. $\alpha \downarrow <>$ if $\alpha \downarrow$
3. $\alpha \downarrow\ al'$ if $\alpha \downarrow$ and for all α' if $\alpha \stackrel{a}{\Rightarrow} \alpha'$ then $\alpha' \downarrow\ l'$
4. $\alpha \downarrow\ \varepsilon(d)l'$ if $\alpha \downarrow$ and for all α' if $\alpha \stackrel{\varepsilon(d)}{\Longrightarrow} \alpha'$ then $\alpha' \downarrow\ l'$
5. $\alpha \uparrow$ if $\alpha \downarrow$ is false and $\alpha \uparrow\ l$ if $\alpha \downarrow\ l$ is false.

– We define the set S(α) of the next possible moves of the configuration α by:

$$S(\alpha) = \{c? \mid for\ some\ v\ and\ \beta,\ \alpha \stackrel{c?v}{\Longrightarrow} \beta\} \cup \{c! \mid for\ some\ v\ and\ \beta,\ \alpha \stackrel{c!v}{\Longrightarrow} \beta\}$$

– We define A(α, l), the acceptance set of events of α after performing the events in the sequence l by :

$$A(\alpha,\ l)\ =\ \{S(\alpha') \mid \alpha \stackrel{l}{\Longrightarrow} \alpha'\}$$

– We define : T(α) = $\pi_2(\alpha)$, if for some d > 0 and $\alpha \stackrel{\varepsilon(d)}{\longrightarrow}$
(i.e. α can evolve an event of $\varepsilon(d)$ in the next step).
Otherwise T(α) is defined as $\emptyset$.
π_2 is a "projection" function, which returns the set of stores in a configuration that can perform a time-measurable event:
- For basic configurations: $\pi_2(<\text{ev, s}>) = \{\text{s}\}$
- For configurations, $\alpha\ op\ \beta$ where $op = \|, \|\!\|$: $\pi_2(\alpha\ op\ \beta) = \pi_2(\alpha) \bigtriangledown \pi_2(\beta)$
- For configurations: $\pi_2(\alpha\ [\!]\ \beta) = \pi_2(\alpha) \cup \pi_2(\beta)$
- For other configurations, e.g. $\pi_2(\alpha\ ;\ \text{E}) = \pi_2(\alpha)$

The function "$\bigtriangledown$" is defined by

$$\{s_1, ..., s_{n1}\} \bigtriangledown \{t_1, ..., t_{n2}\} = \bigcup_{\substack{i = 1...n1 \\ j = 1...n2}} \{s_i \cup t_j\}$$

– We define W(α, l), the store set of events of α after performing the events in the sequence l by :

$$W(\alpha,\ l)\ =\ \{T(\alpha') \mid \alpha \stackrel{l}{\Longrightarrow} \alpha'\}$$

– We define also R(α, l), the set of possible returned pairs (of values and stores) after l:

$$R(\alpha,\ l)\ =\ \{(v,s)\ \mid\ \alpha \stackrel{l}{\Longrightarrow} < v, s >\}$$

4.2 Equivalence of TRSL Expressions

We first define a pre-order between TRSL configurations.

Definition. For α, β in $\mathcal{C}$, $\alpha\ \ll_{SOS}\ \beta$ if for every l and for any given ρ:

$\alpha\ \downarrow\ l \Rightarrow$ *a*) $\beta\ \downarrow\ l$
b) $A(\beta,l)\ \subset\subset\ A(\alpha,l)$
c) $W(\beta,l)\ \subset\subset\ W(\alpha,l)$
d) $R(\beta,l)\ \subseteq\ R(\alpha,l)$

where:

$\mathcal{A}\ \subset\subset\ \mathcal{B}$ is defined by: $\forall\ X\ \in \mathcal{A} \bullet \exists\ Y\ \in \mathcal{B} \bullet\ Y\ \subseteq\ X$

Now, we begin to define the equivalence between TRSL expressions through their operational semantics.

Actually, the equivalence between TRSL configurations: α , β, can be defined as : $\alpha\ \ll_{SOS}\ \beta$ and $\beta\ \ll_{SOS}\ \alpha$. For simplicity of future proof, we rewrite that equivalence definition as follows.

– $\alpha \uparrow l$ iff $\beta \uparrow l$
– if $\alpha \downarrow l$ and $\beta \downarrow l$ then
  1. A(α, l) $\subset\subset$ A(β, l) and A(β, l) $\subset\subset$ A(α, l)
  2. W(α, l) $\subset\subset$ W(β, l) and W(β, l) $\subset\subset$ W(α, l)
  3. R(α, l) = R(β, l)

Definition. For any TRSL expressions: P and Q, P = Q iff for any s and for any given ρ, <P, s> = <Q, s>

4.3 Commentary and Examples

Pre-order Our definition of the pre-order relation on two configuration : $\alpha\ \ll_{SOS}\ \beta$ stands for

1. α is more general than β, or
2. α is more nondeterministic than β, or
3. α is implemented by β, or
4. α is more unstable than β, ...

Therefore, in order to guarantee the condition 2, we ask A(β, l) $\subset\subset$ A(α, l) to hold; and to guarantee the condition 4, we ask W(β, l) $\subset\subset$ W(α, l) to hold.

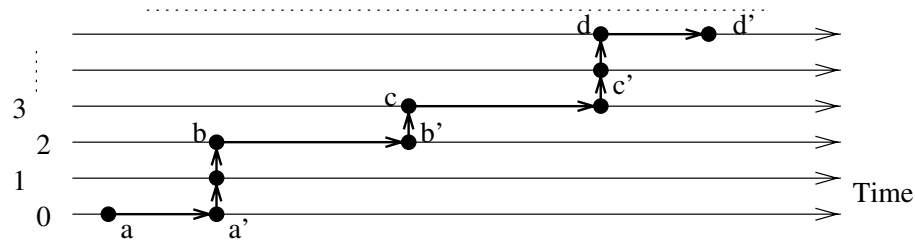

Fig. 3. A Trajectory in Two-dimension Time Space

Time Model We view processes under a super-dense model [MP93] as a trajectory in a two dimensional time space [ZH96, PD97, QZ97]. We suppose there are countably infinite time axes, indexed by natural numbers. Events and processes happen and evolve in this space. A process starts at some time on a time axis. When the process executes a time-measurable event, time progresses horizontally, and the process stays on the same time axis. When the process executes visible and silent events, it jumps vertically up to another time axis, and may have a new state there. A trajectory of a super-dense behaviour is shown in Figure 3.

There are two types of turning point. One is called a start_turning_point (points a, b, c, d in Figure 3), from which the process will execute a time-measurable event. The other is called an end_turning_point (points a', b', c', d' in Figure 3), from which the process will execute a visible or silent event.

The super-dense model distinguishes clearly between time measurable events like delays and waits for synchronisation, and visible and silent events like synchronisation and assignments. It allows arbitrary numbers of the latter to occur instantaneously but in some order, which matches well with the interleaving semantics of concurrency in (T)RSL.

In our time test equivalence definition, for two equivalent processes (expressions), α and β, demanding $\mathrm{A}(\alpha, l) = \mathrm{A}(\beta, l)$ guarantees the same possible temporal order of visible events and time-measurable events of the two processes.

Demanding $\mathrm{W}(\alpha, l) = \mathrm{W}(\beta, l)$ guarantees that the stores (variable states) of two processes (expressions) on every start_turning_point are the same.

Demanding $\mathrm{R}(\alpha, l) = \mathrm{R}(\beta, l)$ guarantees that two expressions, if they terminate, can return the same sets of possible values and final stores.

5 Soundness of Proof Rules

5.1 Proof Rules of TRSL

One of the major reasons for expressing specifications in a formal language like (T)RSL is to prove properties of specification. Therefore, a proof system for

TRSL should be set up. We list some of the proof rules involving newly added time constructs.

[wait_annihilation]
wait 0.0 $\simeq$ **skip**

[wait_plus]
wait er ; **wait** er$'$ $\simeq$ **wait**(er $\dot{+}$ er$'$)

[wait_introduction]
e $\simeq$ **wait** er ; **shift**(e, er)
when pure(er) $\wedge$ **convergent**(er) $\wedge$ er $\geq$ 0.0 $\wedge$ **must_wait**(e, er)

The complete set of proof rules can be found in [GX98]. The original "special functions" **convergent**, **pure**, **express**,etc. are defined in [RMG95]. New special functions **must_wait**, **shift**, etc. are defined in [GX98]. The parallel expansion rule is changed to:

eu $\parallel$ eu$'$ $\simeq$
if parallel_ints(eu, eu$'$) $\equiv$ **swap**
then parallel_exts(eu, eu$'$) [] **parallel_exts**(eu$'$, eu)
else
(**parallel_exts**(eu, eu$'$)[]**parallel_exts**(eu$'$, eu)[]**parallel_ints**(eu, eu$'$))$\dot{\sqcap}$
parallel_ints(eu, eu$'$)
end
when isin_standard_form(eu) $\wedge$ **isin_standard_form**(eu$'$) $\wedge$
($\Box$ **assignment_disjoint**(eu, eu$'$))

where the operator "$\dot{\sqcap}$" is the "maximal progress" version of the internal choice operator mentioned in Section 2 and defined in [GX98]. The other "dotted" operators like "$\dot{+}$" are simple extensions of the standard arithmetic operators, returning zero if the result would otherwise be negative.

The revised definitions of **parallel_exts**, **parallel_ints**, and **interlock_ints** are (showing just one case of each):

parallel_exts(**wait** er ; **let** (b,t) = c? **in** eu **end**, eu$'$) $\simeq$
wait er ; **let** (b,t) = c? **in** eu $\parallel$ **shift**(eu$'$, er $\dot{+}$ t) **end**
when no_capture(b, eu$'$) $\wedge$ **no_capture**(t, eu$'$) $\wedge$
no_capture(b, er) $\wedge$ **no_capture**(t, er)

parallel_ints(**wait** er ; **let** (b,t) = c? **in** eu **end**,
wait er$'$; **let** t$'$ = c!e **in** eu$'$ **end**) $\simeq$
wait max(er,er$'$) ;
let b = e **in subst_expr**(er$'$ $\dot{-}$ er,t,eu) $\parallel$ **subst_expr**(er $\dot{-}$ er$'$,t$'$,eu$'$) **end**
when no_capture(b, eu$'$) $\wedge$ **no_capture**(b, er) $\wedge$ **no_capture**(b, er$'$)

interlock_ints(**wait** er ; **let** (b,t) = c? **in** eu **end**,
wait er′ ; **let** t′ = c!e **in** eu′ **end**) ≃
wait max(er,er′) ;
let b = e **in subst_expr**(er′ $\dot{-}$ er,t,eu) ∥ **subst_expr**(er $\dot{-}$ er′,t′,eu′) **end**
when no_capture(b, eu′) ∧ **no_capture**(b, er) ∧ **no_capture**(b, er′)

5.2 Soundness

We would like to show that

- The original RSL Proof Rules for the TRSL expressions not involving time (e.g. *simple* assignment expressions) still hold in our semantic model.
- Most of the original RSL Proof Rules for TRSL expressions involving time (e.g. input expressions, output expressions) with newly added side conditions hold in our semantic model.
- New rules applied to extended operators are sound with respect to our operational semantics
- In our semantic model, no new rules for the original RSL syntax are generated.

As mentioned in Section 2.1, not all the original RSL proof rules are sound with respect to our semantic model.

However, it is trivial to prove that all the original proof rules for TRSL expressions not involving time-measurable events still hold in our semantic model, because our semantics and the definition of equivalence are just the same as the original one, if we ignore the "$\varepsilon(d)$" transitions.

For the same reason, it is clear that no new rules for the original RSL syntax are generated in our semantic model.

We need to add side conditions to some of the proof rules for TRSL expressions involving time-measurable events. We are interested in proving the soundness of these rules with respect to our semantic model. Most of the rules that we need to study are listed on page 457 of [RMG95].

Of course we should also prove the soundness of rules for the extended operators too. above recommendations.

Proof
Here we just show one example. Other detailed proofs can be seen in [GX98]

[ext_choice_replacement]
e [] e′ ≃ e″ [] e‴
when (e ≡ e″) ∧ (e′ ≡ e‴)

Proof for any s, for any given ρ,

- For Divergence: if one of the configuration is divergent, w.l.g. suppose <e,s>$\uparrow$ l, because e $\equiv$ e$''$, we have <e$''$,s>$\uparrow$ l too. then from the 3rd rule in External Choice (c.f. Section **??**), we know <e [] e$'$, s>$\uparrow$ l and <e$''$ [] e$'''$, s>$\uparrow$ l
- if none of configurations are divergent, we would like to prove
  1. for any l, we have A(<e [] e$'$, s>, l) = A(<e$''$ [] e$'''$, s>, l) :

     For visible action, one branch will be selected. For silent action either e or e$'$ will evolve to next configuration. For time-measurable action, both of them will evolve. So for any possible sequence of action, A(<e [] e$'$, s >, l) $\subseteq$ A(<e, s>, l) $\cup$ A(<e$'$, s>, l). On the other hand, for any possible sequence, from semantics, it is clear A(<e [] e$'$, s >, l) $\supseteq$ A(<e, s>, l) and A(<e [] e$'$, s >, l) $\supseteq$ A(<e$'$, s>, l). So A(<e [] e$'$>, l) = A(<e, s>, l) $\cup$ A(<e$'$, s>, l). For the same reason, we know A(<e$''$ [] e$'''$, s>, l) = A(<e$''$, s>, l) $\cup$ A(<e$'''$, s>, l). Because e $\equiv$ e$''$ and e$'$ $\equiv$ e$'''$, A(<e, s>, l) = A(<e$''$,s>, l) and A(<e$'$, s>, l) = A(<e$'''$, s>, l).
     So A(<e [] e$'$, s>, l) = A(<e$''$ [] e$'''$, s>, l).
  2. for any l, we have W(<e [] e$'$, s>, l) = W(<e$''$ [] e$'''$, s>, l) :

     From the definition of "π_2" function : $\pi_2(\alpha$ [] $\beta) = \pi_2(\alpha) \cup \pi_2(\beta)$, we can conclude trivially that W(<e [] e$'$>, l) = W(<e, s>, l) $\cup$ W(<e$'$, s>, l) and W(<e$''$ [] e$'''$, s>, l) = W(<e$''$, s>, l) $\cup$ W(<e$'''$, s>, l). Because e $\equiv$ e$''$ and e$'$ $\equiv$ e$'''$, W(<e, s>, l) = W(<e$''$,s>, l) and W(<e$'$, s>, l) = W(<e$'''$, s>, l).
     So, we get W(<e [] e$'$, s>, l) = W(<e$''$ [] e$'''$, s>, l).
  3. for any l, we have R(<e [] e$'$, s>, l) = R(<e$''$ [] e$'''$, s>, l) :

     From semantics, we know only one branch of the choice can be selected and evolve to its end. So R(<e [] e$'$, s>, l) = R(<e, s>, l) $\cup$ R(<e$'$, s>, l) and R(<e$''$ [] e$'''$, s>, l) = R(<e$''$, s>, l) $\cup$ R(<e$'''$, s>, l). because e $\equiv$ e$''$ and e$'$ $\equiv$ e$'''$, R(<e, s>, l) = R(<e$''$,s>, l) and R(<e$'$, s>, l) = R(<e$'''$, s>, l).
     We get R(<e [] e$'$, s>, l) = R(<e$''$ [] e$'''$, s>, l) at last.

This completes the proof.

6 Discussion

6.1 Future Work

This paper gives a set of proof rules and an operational semantics for TRSL. A denotational semantics and its formal interrelations with proof rules (axiomatic semantics) and operational semantics needs to be further investigated. What is more, a formal relation between an event-based process algebra and a state-based logic like the Duration Calculus is a non-trivial research topic [Rav94,

PG96]. Actually, [LH99] gives a denotational DC semantics of TRSL, and an "operational semantics with behaviour", which relates TRSL with DC, has been proposed in [HX99]. We need more time to give further results.

The method for developing timed RSL specifications is also an important research direction for TRSL. Some initial results can be seen in [LH99].

6.2 Related Work

Over the past decade, a number of formal calculi (also called process algebras) for real-time, concurrent systems have been developed; examples are TCCS [Wang91] and TCSP [Dav93]. These calculi are suitable specification languages to describe real-time system requirements. They give us ideas for our construction of Timed RSL and its operational semantics.

However, if one uses those specification languages, the design part of the program has to be given in another language. Using TRSL, we can stay with the same language in all steps of development. This is a major motivation for us to add real-time features to RSL.

There are other approaches to adding real time features to a specification language. [F92] represents RTL formulae in Z and [FHM98] directly introduces the differential and integral calculus operators into the Z notation. They are essentially encodings of time using facilities already in Z. As such they add no power to the language. In addition they allow all variables to be functions of time and so permeate the language. For example, notions of refinement become more complicated. [HX98] embeds DC into RSL using high order logic and also proposes an extension of RSL syntax with DC constructs. But again this is an encoding and the power of the language is not changed.

These notational extensions are also at the abstract specification level. They provide no explicit assistance with implementation.

Our aim is rather different. The addition of the **wait** construct adds to the power of RSL. Further, it allows both the abstract specification of timing features in a DC notation and also the concrete specification of particular timed algorithms that can be readily expressed in suitable programming languages.

The *super-dense computation* model is an important abstract model of real-time systems [MP93]. Some industrially applicable programming languages, such as Esterel, adopt similar models.

[ZH96, PD97, QZ97] use (Extended) Duration Calculus to give a denotational semantics to an OCCAM-like programming language under the super-dense computation model.

Acknowledgements

The authors thank Zhou Chaochen for his advice and guidance while doing this research work, Anne Haxthausen for her ideas and comments on Timed RAISE, and He Jifeng for his comments on a draft of this paper. Anonymous reviewers also provided useful comments.

References

[BD93] D.Bolignano, and M.Debabi. *RSL: An Integration of Concurrent, Functional and Imperative Paradigms.* Technical Report LACOS/BULL/MD/3/V12.48, 1993.

[Dav93] Jim Davies. *Specification and Proof in Real-Time CSP.* Distinguished Dissertation Series. Cambridge University Press, 1993.

[Deb94] M.Debabi. *Intégration des paradigmes de programmation parallèle, fonctionnelle et impérative : fondements sémantiques.* Ph.D. Thesis (Thèse de Doctorat en Informatique), Université Paris XI, Centre d'Orsay, July 1994.

[F92] C. J. Fidge Specification and verification of Real-Time Behaviour Using Z and RTL in J. Vytopil (ed.), Proc FME'92, LNCS571 (Springer), 1992.

[FHM98] C. J. Fidge, I. J. Hayes and B. P. Mahony, *Defining Differentiation and Integration in Z*, Technical report 98-09, Software Verification Research Centre, School of Information Technology, The University of Queensland, September 1998.

[GX98] Chris George and Xia Yong *An Operational Semantics for Timed RAISE* Technical Report No. 149, United Nations University/International Institute for Software Technology, November 1998.

[HI93] M. Hennessy and A. Ingólfsdóttir. *Communicating Process with Value-passing and Assignments.* In *Formal Aspects of Computing*, 1993.

[HX98] Anne Haxthausen and Xia Yong *A RAISE Specification Framework and Justification Assistant for the Duration Calculus.* In *ESSLLI-98 Workshop on Duration Calculus*, August 1998.

[HX99] Anne Haxthausen and Xia Yong. *Linking DC together with TRSL.* Research Report, Department of Information Technology, Technical University of Denmark, April 1999.

[LH99] Li Li and He Jifeng *Towards a Denotational Semantics of Timed RSL using Duration Calculus* Technical Report No. 161, United Nations University/International Institute for Software Technology, April 1999.

[MP93] Z. Manna and A. Pnueli. *Models of reactivity.* In *Acta Informatica.* 30(7), 609–678, Springer-Verlag, 1993.

[Rav94] Anders P. Ravn. *Design of Embedded Real Time Computing Systems.* PhD thesis, Department of Computer Science, Technical University of Denmark, Denmark, September 1994.

[RLG92] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice Hall Int., 1992.

[RMG95] The RAISE Method Group. *The RAISE Development Method.* The BCS Practitioners Series. Prentice Hall Int., 1995.

[PD97] Paritosh K. Pandya and Dang Van Hung. *Duration Calculus of weakly monotonic time.* Technical Report No. 122, United Nations University/International Institute for Software Technology, September 1997.

[PG96] Jifeng He, C.A.R. Hoare, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. The ProCoS Approach to the Design of Real-Time Systems: Linking Different Formalisms. In *Formal Methods Europe 96*, Oxford, UK, March 1996. Tutorial Material.

[QZ97] Qiu Zhongyan and Zhou Chaochen *A Combination of Interval Logic and Linear Temporal Logic* Technical Report No. 123, United Nations University/International Institute for Software Technology, September 1997.

[Wang91] Wang Yi. *A Calculus of Real Time Systems.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göterborg, Sweden, 1991

[ZH96] Zhou Chaochen and Michael R. Hansen. *Chopping a point.* In J. F. He *et al* (Eds.), *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing, Springer-Verlag, 1996.

[ZHR91] Zhou Chaochen, C.A.R. Hoare and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991. Revised June 3, 1992.