Implementing a Real-Time Process Algebra in HOL

Richard Gerber* Elsa L. Gunter Insup Lee*

Dept. of Computer Science AT&T Bell Laboratories Dept. of Comp. and Info. Sci.
University of Maryland 600 Mountain Ave University of Pennsylvania
College Park, MD 20742 Murray Hill, NJ 07974 Philadelphia, PA 19104
rich@cs.umd.edu elsa@research.att.com lee@cis.upenn.edu

Abstract

In [6] a real-time process algebra was discussed, a model for the algebra was developed, and the axioma-
tization was shown to be sound and complete for rea-
soning about equality and containment of processes
for this model. In this work we show how to imple-
ment this theory in HOL and to develop a procedure
for proving the equality of finite processes.

1 Introduction

The correctness of a real-time system depends not
only on how concurrent processes interact, but also
the time at which these interactions occur. Real-time
systems are used in such environments as manufactur-
ing, robotics, avionics, medicine and communication.
The temporal correctness of a real-time system used
in such an environment can be important since a sys-
tem failure can result in a high financial loss, or even
the loss of life. Therefore, there is a high demand for
a rigorous framework in which real-time systems can
be specified and analyzed.

We have developed a real-time process algebra
which captures the temporal behavior of concurrent
systems. Our algebra consists of an abstract, CSP-
based language, a partially ordered semantic domain,
and a complete axiom system, which provides a ba-
sis for proving critical properties of real-time system.
Our experience of using the algebra has been limited
to only small toy examples due to the difficulty in car-
rying out a large and complex formal proof without
mechanical assistance.

In this paper, we discuss how to use abstract the-
ories in HOL to develop a real-time process algebra.
The axioms for the real-time process algebra are spec-
ified using the theory assumptions. Within the ab-
tract theory they are available for use as theorems.
In particular, they may be incorporated in tactics used
from proving theorems within the abstract theory.

Next we describe the development of a tactic for
solving (true) goals of equality of processes. Assuming
that processes are constructed solely from finite sets of
events and the processes constructors specified in the
formal parameter of the abstract theory, there exists a
canonical form for each process (assuming there exists
a canonical form for the sets of events). To be able
to implement a conversion which, when applied to a
process, will return a theorem stating that the process
is equal to its canonical form, we first require a similar
conversion for events and finite sets of events. Once we
have such normalization procedures, we can prove two
processes are equal by proving that each is equal to its
canonical form and then proving that the respective
canonical forms are equal.

Finally, we discuss the advantages of having em-
bedded the process algebra in a general-purpose theo-
rem prover. The semi-decision procedure for equality
of processes may be used as a tool for more general
reasoning about processes. For example, using the
fact that we are in a general theorem prover, we may
create a predicate that indicates the earliest possible
time that a process can commit an action. In reason-
ning about this predicate applied to processes, we have
available the semi-decision procedure as a tool.

In this paper we focus on how to derive proce-
dures for reasoning about real-time processes in gen-
eral, and particularly for being able to automatically
prove equality of two processes. We have not, as yet
implemented a model for the process algebra. For a

*This research was supported in part by ONR N00014-89-J-
1131
clear exposition of the general approach to implementing a process algebra in HOL, see [1].

2 The Language of Finite Real-Time Processes

The syntax of our language is similar to that of untimed CSP [4], with one major exception: The traditional prefix operator, \( a \rightarrow P \), has been replaced by the timed action operator \( A \overset{i}{\rightarrow} P \). In our model timed action incorporates the notion of true concurrency, and it allows the specification of pure time delay.

As in the untimed CSP, processes communicate by synchronously engaging in events, which are considered instantaneous. At any time during its execution, a process may simultaneously engage in a set of such events, after which it delays for some nonzero period of time. The terms of the language are defined as follows:

\[
P ::= \text{STOP} \mid A \overset{i}{\rightarrow} P \mid P \parallel P \mid P \cap P \mid P \parallel_{A} P
\]

Here, \( A \) is a subset of the finite alphabet \( \Sigma \); and \( i \) is an integer greater than 0.

While the syntax of the language is similar to its untimed counterpart, the semantics of each term incorporates an additional factor, the passage of time. The timed action operator, \( A \overset{i}{\rightarrow} P \), is defined for an event set \( A \) and a natural number \( i > 0 \). At time 0 the events in \( A \) are simultaneously executed, and after a delay of exactly \( i \) time units the process \( P \) is executed. If \( A = \emptyset \), the execution of \( P \) is delayed by exactly \( i \) time units. Choice, \( P \parallel Q \), differs from its untimed counterpart in that it also represents an external decision made on the occurrence time of an event. For example, let

\[
P = (\emptyset \overset{1}{\rightarrow} \{a\} \overset{1}{\rightarrow} \text{STOP}) \parallel (\emptyset \overset{2}{\rightarrow} \{b, c\} \overset{1}{\rightarrow} \text{STOP}).
\]

\( P \) must accept either \( a \) at time 1 if it is offered, or both \( b \) and \( c \) at time 2, if they are offered instead. If \( a \) is offered at time 2, \( P \) will not accept it. On the other hand, the nondeterministic process \( P \cap Q \), internally "decides" at time 0 to behave like \( P \) or \( Q \). Consider the process \( P \) above, with the "\( \parallel \)" operator replaced by "\( \cap \)". Here, if the process "decides" to behave like the left-hand alternative and an \( a \)-event is not offered at time 1, then when the events \( b \) and \( c \) are offered at time 2, they will not be accepted. Viewed externally to the process, the \( a \)-event may be accepted at time 1, or \( b \) and \( c \) may be accepted at time 2. However, the decision on which choices to accept is made internally by the process.

The parallel operator, \( P \parallel_{A} Q \), captures real-time concurrency by combining both the delay times and event executions of \( P \) and \( Q \). The set \( A \) denotes the events on which both \( P \) and \( Q \) must synchronize, while the processes may execute events in \( \Sigma - A \) independently. Consider the following processes:

\[
P = (\emptyset \overset{1}{\rightarrow} \{a\} \overset{1}{\rightarrow} \{c\} \overset{1}{\rightarrow} \text{STOP}) \\
\quad \parallel (\emptyset \overset{4}{\rightarrow} \{b\} \overset{5}{\rightarrow} \{d\} \overset{1}{\rightarrow} \text{STOP}),
\]

\[
Q = \emptyset \overset{1}{\rightarrow} \{a\} \overset{1}{\rightarrow} \text{STOP}.
\]

The process \( P \parallel_{\{a, d\}} Q \) is equivalent to

\[
(\emptyset \overset{1}{\rightarrow} \{a\} \overset{1}{\rightarrow} \{c\} \overset{1}{\rightarrow} \text{STOP}) \\
\quad \parallel (\emptyset \overset{4}{\rightarrow} \{b\} \overset{1}{\rightarrow} \text{STOP}).
\]

Examining the left-hand alternative, we see that both processes can synchronize on \( a \) at time 1. If this choice is taken, then \( P \) will execute its local event \( c \) one time unit later. Since in this case both processes successfully terminate, the combined termination time is the maximum of the two processes' individual termination times, being 2 time units. The right-hand alternative shows that \( P \) may execute \( b \) at time 4; here synchronization with \( Q \) is not necessary. However, since \( Q \) must synchronize on \( a \) at time 1, this step leaves \( Q \) deadlocked. The deadlock finally infects \( P \) at time 9, when synchronization on the event \( d \) becomes necessary. The parallel operator permits \( n \)-way synchronization between processes.

The semantic domain is based on two well-known untimed paradigms: Hoare’s Trace Algebra and Hennessy’s Acceptance Tree model. In our model, the trace algebra is temporally extended to depict the observable execution sequences of real-time processes. However, traces alone are unable to capture the meaning of nondeterminism; thus we include a set of states to accompany each trace. This state set denotes the array of choices – both internal and external – that a process may make after executing the trace. The representation for the state set is similar to the acceptance tree model of nondeterminism, although our state explicitly incorporates the notion of time. Here we depart from other CSP-based models, which capture nondeterminism through the use of refusals.
<table>
<thead>
<tr>
<th>Timed Action</th>
<th>Nondeterminism</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TA1) $A \xrightarrow{\text{SUC}} \frac{1}{i} P = A \xrightarrow{\frac{1}{i}} \emptyset$</td>
<td>(ND1) $P \cap P = P$</td>
</tr>
<tr>
<td>(TA2) $\emptyset \xrightarrow{\frac{1}{i}} \text{STOP} = \text{STOP}$</td>
<td>(ND2) $P \cap Q = Q \cap P$</td>
</tr>
<tr>
<td></td>
<td>(ND3) $(P \cap Q) \cap R = P \cap (Q \cap R)$</td>
</tr>
<tr>
<td>Choice</td>
<td>Distributivity</td>
</tr>
<tr>
<td>(CH1) $P \square P = P$</td>
<td>(D1) $A \xrightarrow{\frac{1}{i}} (P \cap Q) = (A \xrightarrow{\frac{1}{i}} P) \cap (A \xrightarrow{\frac{1}{i}} Q)$</td>
</tr>
<tr>
<td>(CH2) $P \square Q = Q \square P$</td>
<td>(D2) $(\emptyset \xrightarrow{\frac{1}{i}} P) \square (\emptyset \xrightarrow{\frac{1}{i}} Q) = \emptyset \xrightarrow{\frac{1}{i}} (P \square Q)$</td>
</tr>
<tr>
<td>(CH3) $(P \square Q) \square R = P \square (Q \square R)$</td>
<td>(D3) $P \cap (Q \square R) = (P \cap Q) \square (P \cap R)$</td>
</tr>
<tr>
<td>(CH4) $P \square \text{STOP} = P$</td>
<td>(D4) $P \square (Q \cap R) = (P \square Q) \cap (P \square R)$</td>
</tr>
<tr>
<td>(CH5) $A \neq \emptyset \Rightarrow (A \xrightarrow{\frac{1}{i}} P) \square (A \xrightarrow{\frac{1}{i}} Q)$</td>
<td>(D5) $P \parallel_c (Q \cap R) = (P \parallel_c Q) \cap (P \parallel_c R)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallelism</td>
<td></td>
</tr>
<tr>
<td>(PAR1) $\text{STOP} \parallel_c \text{STOP} = \text{STOP}$</td>
<td></td>
</tr>
<tr>
<td>(PAR2) $P \parallel_c Q = Q \parallel_c P$</td>
<td></td>
</tr>
<tr>
<td>(PAR3) $(\emptyset \xrightarrow{\frac{1}{i}} P) \parallel_c (\emptyset \xrightarrow{\frac{1}{i}} Q) = \emptyset \xrightarrow{\frac{1}{i}} (P \parallel_c Q)$</td>
<td></td>
</tr>
<tr>
<td>(PAR4) $A \cap C \neq \emptyset \Rightarrow ((A \xrightarrow{\frac{1}{i}} P) \square (\emptyset \xrightarrow{\frac{1}{i}} Q)) \parallel_c (\emptyset \xrightarrow{\frac{1}{i}} R) = (\emptyset \xrightarrow{\frac{1}{i}} Q) \parallel_c (\emptyset \xrightarrow{\frac{1}{i}} R)$</td>
<td></td>
</tr>
<tr>
<td>(PAR5) $A \neq \emptyset \cap A \cap C = \emptyset \Rightarrow$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Auxiliary</td>
<td></td>
</tr>
<tr>
<td>(NI1) $A \neq \emptyset \Rightarrow \text{noidle}(A \xrightarrow{\frac{1}{i}} P)$</td>
<td></td>
</tr>
<tr>
<td>(NI2) $\text{noidle} P \land \text{noidle} Q \Rightarrow \text{noidle}(P \square Q)$</td>
<td></td>
</tr>
<tr>
<td>(Det1) $\text{det} \ \text{STOP}$</td>
<td></td>
</tr>
<tr>
<td>(Det2) $A \neq \emptyset \Rightarrow \text{det}(A \xrightarrow{\frac{1}{i}} P)$</td>
<td></td>
</tr>
<tr>
<td>(Det3) $\text{det} P \Rightarrow \text{det}(\emptyset \xrightarrow{\frac{1}{i}} P)$</td>
<td></td>
</tr>
<tr>
<td>(Det4) $\text{det} P \land \text{det} Q \Rightarrow \text{det}(P \square Q)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A Proof System for Equality of Finite Processes.

3
have found it considerably more natural to specify those events a process can execute after a trace, instead of those that it cannot execute.

In our model a process is defined by all of its possible traces, and the state sets corresponding to those traces. Our semantic domain forms a complete partial order, where $P \sqsubseteq Q$ if $Q$ is a deterministic refinement of $P$. All of our operators are monotonic and continuous with respect to the partial order.

We also have a sound and complete axiomatization of process containment for finite processes described above. Table 1 gives a subset of these axioms that is in itself a sound and complete system for equality of finite processes.

In our studies of finite processes, it will be useful to consider a subset constructed by the grammar

$$P ::= \text{STOP} \mid A \sim P \mid P \sqcup P$$

with the following context-sensitive restriction. In all terms of the form $A \sim P \sqcup B \sim Q$, $A \neq B$. We shall refer to such finite processes as totally deterministic. The axioms (TA1), (TA2), (CH1) through (CH4) and (D2) form a sound and complete system for equality of totally deterministic processes. If we expand this class to include those processes that also involve $\sqcap$ (but not $\parallel A$), then the axioms (TA1), (TA2), (ND1) through (ND3), (CH1) through (CH4) and (D1) through (D4) form a sound and complete system for equality of these. Moreover, it turns out that every finite process is equal to such a process. These facts will allow us to develop procedures for proving the equality of finite processes in an incremental fashion.

3 The Real-Time Process Algebra in HOL

The language of real-time processes together with the axioms described above is an algebra in much the same sense as groups or rings are algebras. Therefore, it is appropriate to use the same methodology to implement it: namely that of abstract theories. Abstract theories are an extension to HOL88 that were discussed in the previous two meetings of the HOL Users Group [5]. Very briefly, an abstract theory is an HOL theory that is parametrized by a collection of assumed types, terms and theorem statements. Within an abstract theory, one has access to the elements of the formal parameter as type constants, term constants and axioms. To use the abstract theory within another theory, you must supply it with values for the types, terms and theorems matching the formal parameter. The reason for using an abstract theory is that it allows us to focus on the purely algebraic aspects of the system as it is described by the axioms in Table 1, without confusing issues with details of a particular model. We do have a model in mind, from which these axioms were derived. However, by using abstract theories we have separated the task of proving that the model satisfies these axioms from the task of developing the general theory that follows from them.

Below is how we introduce the abstract theory of real-time process algebras. We omit some of the theory assumptions because of their length.

```latex
new_abstract_theory "RTP"
(* Type Binder *)
[':process',':event']
(* Term Binder *)
['Proc:process -> bool',
'Stop:process',
'Timed_Action:( event -> bool) -> num -> process -> process',
'Choice:process -> process -> process',
'Or:process -> process -> process',
'Par:event -> process -> process -> process',
'noidle:process -> bool',
'det:process -> bool']

(* Theory Assumptions *)
[ ("StopProcess",
'(Proc:process -> bool) Stop'),
("TimedActionProcess",
'!(A:event -> bool) (P:process) (i:num).
Proc P ==> Proc(Timed_Action A i P)'),
("ChoiceProcess",
'!P Q:process. Proc P \ / Proc Q ==> Proc(Choice P Q)'),
("OrProcess",
'!P Q:process. Proc P \ / Proc Q ==> Proc(Or P Q)'),
("ParProcess",
'!(A:event -> bool) (P:process) Q.
Proc P \ / Proc Q =>
Proc(Par A P Q)'),
```
Table 2: Creating an HOL abstract theory for real-time process algebras.

Notice that, in addition to the type *process*, we assume we have a predicate on that type, and all the theory assumptions have been prefixed with the premis that all the processes being quantified over satisfy the predicate. This allows us to specify our finite real-time processes as elements of a subset of a larger universe (one containing infinite processes, for example). For a more extensive discussion of how to encode an algebra in HOL, see [3].

Within the abstract theory, the theory assumptions are available for use as theorems. In particular, they may be used in the proofs of other theorems and incorporated in tactics used for proving still other theorems within the abstract theory. An example of the theorems we need to prove is

\[
\text{INTERVAL\_SAT} = \text{``}\neg \neg \text{P Q R} \Rightarrow \text{Proc P} /\\text{Proc Q} /\text{Proc R} \Rightarrow \text{Or} (\text{Proc P} (\text{Choice Q R})) = \text{Or P} (\text{Or} (\text{Choice P Q}) (\text{Or} (\text{Choice P R}) (\text{Choice P} (\text{Choice Q R})))))``
\]

In order to facilitate proving theorems using theory assumptions and other theorems from the theory, we can use the closure assumptions StopProcess, TimedActionProcess, ChoiceProcess and OrProcess to create a tactic for solving goals of showing that processes constructed from the constants in the term binder of the abstract theory satisfy the predicate Proc. Such subgoals will arise every time we wish to use an instance of any theorem of the theory since they all have such premises.

While it is our position that abstract theories in HOL provide the most elegant means for implementing the theory of real-time process algebras as described above, it is not the only way to achieve essentially the same effect. Since abstract theories are not yet a standard part of the versions of HOL in common use, it is worth briefly mentioning how to implement real-time process algebras in HOL without the use of abstract theories. The general approach is the one that was taken in the implementation of group theory as is found in the distributed HOL Library directory, and as is discussed in [3]. To introduce the notion of a real-time process algebra, define a predicate RTPA on tuples (Proc, Stop, TimedAction, Choice, Or, Par, noidle, det) which is the conjunction of all the properties listed in the theory assumptions of the new abstract theory declaration above. With this definition it is trivial to prove that, under the assumption that the predicate RTPA holds of a tuple, each of the properties holds for that tuple. Subsequent theorems that we prove from these will also carry such premisses.

4 Determining Equality

The particular procedures we wish to focus on here are ones for determining the equality of two processes. The general approach we take to determining equality of two terms is to first find a canonical form that each term may be put in, and then test whether the two canonical forms are syntactically identical. If they are, then the terms are provably equal. If the two
canonical forms differ, we can not immediately conclude that they are provably unequal. However, we may be able to use this difference to prove that the two term are not equal in some particular model. Assuming that processes are constructed solely from finite sets of events and the process constructors specified in the formal parameter of the abstract theory, there exists a canonical form for each process (assuming there exists a canonical form for the sets of events). To be able to implement a conversion which, when applied to a process, will return a theorem stating that the process is equal to its canonical form, we first require a similar conversion for events and finite sets of events.

4.1 Equality of Finite Sets

Let us assume for the moment that we have a fixed alphabet of events, a procedure that either proves two events (from this fixed alphabet) are equal or are not equal, and a function (in ML) that totally orders them. Suppose in addition that we are given a set which is described simply as a composition of EMPTY, INSERT, DELETE, SINGLETON, UNION, and INTERSECT, together with atomic events. Then it should be possible, using the facts we know about these set constructors to rewrite such a set into a form that involves only the constructors EMPTY and INSERT. Moreover, if we take advantage of the ordering on events, we should be able to rewrite the term into a unique canonical form over these constructors. The facts that we need for accomplishing this rewriting are as follows:

\[
\begin{align*}
\text{INS1} & = |- !x S. \text{INSERT} x (\text{INSERT} x S) = \text{INSERT} x S \\
\text{INS2} & = |- !x S. \text{INSERT} x (\text{INSERT} y S) = \text{INSERT} y (\text{INSERT} x S) \\
\text{SING} & = |- !x. \text{SINGLETON} x = \text{INSERT} x \text{EMPTY} \\
\text{UN1} & = |- !S. \text{S UNION EMPTY} = S \\
\text{UN2} & = |- !S. \text{EMPTY UNION} S = S \\
\text{UN3} & = |- !x S U. (\text{INSERT} x S) \text{UNION} U = \text{INSERT} x (S \text{UNION} U) \\
\text{MEM1} & = |- !x. \neg(x \text{ MEMBER EMPTY}) \\
\text{MEM2} & = |- !x y S. x \text{ MEMBER} (\text{INSERT} y S) = ((x = y) \lor (x \text{ MEMBER} S)) \\
\text{INT1} & = |- !S. \text{S INTERSECT EMPTY} = \text{EMPTY} \\
\text{INT2} & = |- !S. \text{EMPTY INTERSECT} S = \text{EMPTY} \\
\text{INT3} & = |- !x S U. x \text{ MEMBER} U ==> ((\text{INSERT} x S) \text{INTERSECT} U = \\
\text{INT4} & = |- !x S U. \neg(x \text{ MEMBER} U) ==> ((\text{INSERT} x S) \text{INTERSECT} U = S \text{ INTERSECT} U) \\
\text{DEL1} & = |- !x. \text{DELETE} x \text{EMPTY} = \text{EMPTY} \\
\text{DEL2} & = |- !x S. \text{DELETE} x (\text{INSERT} x S) = \text{DELETE} x S \\
\text{DEL3} & = |- !x y. \neg(x = y) ==> (\text{DELETE} x (\text{INSERT} y S) = \\
\text{INSERT} y (\text{DELETE} x S))
\end{align*}
\]

Table 3: Theorems needed for rewriting sets.

We require facts about MEMBER even though it is not a constructor for sets because we need to be able to solve membership tests to satisfy the preconditions for INTERSECT. Using the above facts, we can write an ML function that takes an HOL term for a set, computes a canonical form for the term and returns a theorem stating that the term is equal to this canonical form. The function analyzes the syntactic structure of the term and proceeds by induction on that structure. We will discuss it in some detail here because it exemplifies the nature of the conversions we are creating.

Throughout, all the conversions we write need to be parameterized by one or more of the following functions: a conversion that takes two terms (of some specified type, such as ':event') and returns a theorem that states either the two terms are equal or that the two terms are not equal. Such a conversion could easily be written for types created using define type, for example, and a function that linearly orders terms of the type in question. We will use ELT_EQ_CONV to refer to the function that decides equality, and elt_ord for the functions that orders the elements. The ordering that elt_ord supplies is syntactic ordering in ML. That is, two terms which are provably equal in ML will nevertheless compare differently.

In order to write a conversion SET_CONV that will prove that a set is equal to a canonical form for that set, it is desirable to have some subsidiary conversions. Namely, we want to have a conversion MEMBER_CONV that returns a theorem stating whether an element is in a set, and a conversion SORT_SET_CONV that states that a set (involving only EMPTY and INSERT) is equal to the set with its elements sorted from greatest to least. An implementation (albeit a rather slow one) of is given as

\[
\text{fun MEMBER_CONV ELT_EQ_CONV x S =}
\]

6
if \( S = \text{EMPTY} \) then \( \text{AUTO\_SPEC} \ x \ \text{MEM1} \) else let

\[
\text{val th1} = \\
\text{REWRITE\_RULE} \\
[(\text{ELT\_EQ\_CONV} \ x \ y)] \\
(\text{AUTO\_SPECL} \ [x, y, U] \ \text{MEM2})
\]
in if \( \text{is\_eq} \ (\text{concl} \ \text{th1}) \) then

\[
\text{REWRITE\_RULE} \\
[(\text{MEMBER\_CONV} \ \text{ELT\_EQ\_CONV} \ x \ (\text{rhs} \ (\text{concl} \ \text{th1})))]
\]
th1
else th1 end

The functions \text{AUTO\_SPEC} and \text{AUTO\_SPECL} are the same as \text{SPEC} and \text{SPECL} except that they also specialize types appropriately.

The conversion \text{SORT\_SET\_CONV} will itself be most easily written in terms of two other conversions: \text{IDEM\_INSERT\_CONV} and \text{COMM\_INSERT\_CONV}. \text{IDEM\_INSERT\_CONV} when applied to a term \text{INSERT} \ x \ (\text{INSERT} \ y \ S) where \text{ELT\_EQ\_CONV} proves that \( \vdash x = y \), rewrites the term with this theorem and \text{INS1} to return the theorem

\[
\vdash \text{INSERT} \ x \ (\text{INSERT} \ y \ S) = \text{INSERT} \ x \ S
\]
and otherwise fails. \text{COMM\_INSERT\_CONV} when applied to a term \text{INSERT} \ x \ (\text{INSERT} \ y \ S) where \text{not(elt\_ord} \ y \ x) \ (\text{that is where} \ y \not\leq \ x) \text{, rewrites the term with} \text{AUTO\_SPECL} \ [x,y,S] \ \text{INS2} \text{to get the theorem}

\[
\vdash \text{INSERT} \ x \ (\text{INSERT} \ y \ S) = \text{INSERT} \ y \ (\text{INSERT} \ x \ S)
\]
and otherwise fails. With these two conversions, \text{SORT\_SET\_CONV} can now be given by

\[
\text{val SORT\_SET\_CONV ELT\_EQ\_CONV elt\_ord} \ S = \\
(\text{REDEPTH\_CONV} \\
(\text{IDEM\_INSERT\_CONV ELT\_EQ\_CONV} \ \text{ORELSEC} \\
(\text{COMM\_INSERT\_CONV elt\_ord}))) \ S
\]

In general, we can construct conversions that will rewrite a term to a sorted, contracted version of that term by first writing conversions for each of the ways of contracting or reordering, and then composing these with \text{REDEPTH\_CONV} and \text{ORELSEC}. For more on conversions in general, see [2].

We construct \text{SET\_CONV} in two phases. First, we create a conversion \text{WEAK\_SET\_CONV} that rewrites a set into a form involving only \text{EMPTY} and \text{INSERT}. Once we have \text{WEAK\_SET\_CONV}, we get

\[
\text{val SET\_CONV} = \\
\text{WEAK\_SET\_CONV THENC} \ \text{SORT\_SET\_CONV}
\]

The conversion \text{WEAK\_SET\_CONV} is constructed by induction on the structure of the set description. For each of the functions for constructing sets we write a conversion which assumes that \text{WEAK\_SET\_CONV} has already been applied to the arguments of the function. \text{WEAK\_SET\_CONV} determines the outermost constructor and its arguments, applies itself recursively to the arguments and then applies the appropriate conversion to the constructor applied to its rewritten arguments.

One of the more complex cases in \text{WEAK\_SET\_CONV} is the conversion \text{INTERSECT\_CONV}. When applied to a term \( S \ \text{INTERSECT} \ U \), if either \( S \) or \( U \) is \text{EMPTY} then it rewrites the term using either \text{INT1} or \text{INT2} as appropriate. If \( S \) is not empty, then it must be of the form \text{INSERT} \ x \ R. In this case, \text{MEMBER\_CONV ELT\_EQ\_CONV} \ x \ U \text{ is computed. If} \( \vdash x \ \text{MEMBER} \ U \) \text{is returned, then we use this with INST3 to rewrite} \ (\text{INSERT} \ x \ R) \ \text{INTERSECT} \ U \text{ to} \ \text{INSERT} \ x \ R \ \text{INTERSECT} \ U \), and then we rewrite \( R \ \text{INTERSECT} \ U \) with \text{INTERSECT\_CONV} to finish. If \( \vdash \mathrm{~L} \ (x \ \text{MEMBER} \ U) \) \text{is returned, then we use that with INST4 to rewrite} \ (\text{INSERT} \ x \ R) \ \text{INTERSECT} \ U \text{ to} \ R \ \text{INTERSECT} \ U \) \text{which is then rewritten with} \text{INTERSECT\_CONV} \text{ to finish.}

Once we have the conversion \text{SET\_CONV}, we can use it to create a procedure \text{SET\_EQ\_CONV} that will prove whether two sets are equal. This is done by first putting each of the two sets into canonical form. If the canonical forms are identical, then the two sets are provably equal. If the two canonical forms are not identical, then we may recurse down through the occurrences of \text{INSERT} until we find where the two canonical forms differ. At that point which ever of the two elements being inserted into the respective sets is the bigger is an element that is provably a member of one set and provably not a member of the other. Therefore, the two sets are provably not equal.

4.2 Equality of Finite Processes

In a fashion similar to what we have described with sets, but with greater complexity, we may use this
decision procedure for equality of sets of events, together with the axioms for the real-time process algebra to create a normalization procedure \text{FPFPROC\_CONV} for processes. Once we have such normalization procedures, we can prove two processes are equal by proving that each is equal to its canonical form and then proving that the respective canonical forms are equal. It should be pointed out that this gives us only a semi-decision procedure because it fails to provide us with any mechanism for proving that two processes are \textit{not} equal. It only allows us to prove equality when it does hold.

The conversion \text{FPFPROC\_CONV} is built up in stages, first with a conversion for totally deterministic processes, then with one for processes constructed from \text{Stop}, \text{Timed\_Action}, \text{Choice} and \text{Or}, and finally with one for all finite processes. We will also require, as we did for sets, a total ordering on finite processes and a conversion that will finish sorting a process into a canonical form.

Using the ordering on the elements in the event sets, we may order event sets that are in canonical form by the usual alphabetical ordering. Having that, we may write a function \text{proc\_ord} that orders finite processes (not necessarily in any special form) as below. (We leave out the cases for $\parallel_A$, since they will not be needed when the sorting is applied.)

- $\forall P.\text{STOP} \leq P$.  
- $\forall P. P \neq \text{STOP} \Rightarrow P \not\leq \text{STOP}$.  
- $A \triangleright i \Rightarrow P \leq B \triangleright j \Rightarrow Q$ if $A < B$ or $A = B$ and either $i < j$ or both $i = j$ and $P \leq Q$.  
- $A \triangleright i \Rightarrow P \leq Q \sqcap R$ if $A \triangleright i \Rightarrow P \leq Q$.  
- $A \triangleright i \Rightarrow P \leq Q \sqcap R$ if $A \triangleright i \Rightarrow P \leq Q$.  
- $P \sqcup Q \leq A \sqcup i \Rightarrow R$ if $P < A \triangleright i \Rightarrow R$.  
- $P \sqcup Q \leq R \sqcup S$ if either $P \leq R$ or both $P = R$ and $Q \leq S$.  
- $P \sqcup Q \leq R \sqcap S$ if $P \sqcup Q \leq R$.  
- $P \sqcap Q \leq R \sqcap S$ if either $P \leq R$ or both $P = R$ and $Q \leq S$.  
- $P \sqcap Q \leq R$, where $R$ is not outermost an $\sqcap$, if $P < R$.

When we were developing conversions for sets, we relied upon having a system of equations we could use for rewriting. In our present setting, we have a collection of conditional equations. Most of our theory assumptions have as preconditions that certain terms satisfy the predicate \text{Proc}. We need to have a conversion \text{IS\_PROC\_CONV} that, when applied to a term that is composed solely from \text{Stop, Timed\_Action, Choice, Or} and \text{Par}, returns a theorem stating that the term satisfies \text{Proc}. \text{IS\_PROC\_CONV} works by case analysis on the term, using \text{Stop\_Process, Timed\_Action\_Process, Choice\_Process, Or\_Process} and \text{Par\_Process} and calling itself recursively on the subterms.

In the presence of \text{IS\_PROC\_CONV}, for each of the theory assumptions \text{TA1} through \text{D5}, we may create a corresponding rewrite conversion which does top-level rewriting by the conditional equation. We shall refer to the conversion that corresponds to \text{TA1} as \text{e.g. TA1\_CONV}, and so forth.

As we did in the case of sets, we are now in a position to create a conversion \text{SORT\_CHOICE\_CONV} that when applied to totally deterministic process converts them to sorted order. To do so we create a conversion \text{COMM\_CHOICE\_CONV} using proc\_ord, \text{CH2} and \text{CH3} which when applied to (Choice P (Choice Q R)) with not(proc\_ord P Q) returns (Choice Q (Choice P R)). Also, when it is applied to (Choice P Q) with not(proc\_ord P Q), it returns (Choice Q P). Then to sort and compress a totally deterministic process, we repeatedly rewrite with \text{TA1\_CONV, TA2\_CONV, CH1\_CONV}, (which uses both \text{CH1} and a derived version rewriting Choice P (Choice P Q) to Choice P Q), \text{CH3\_CONV, D2\_CONV} and \text{COMM\_CHOICE\_CONV}, in the same manner as we did with \text{SORT\_SET\_CONV}. We can extend this to a conversion \text{SORT\_OR\_CONV} for sorting processes that may also involve \text{Or} by creating a \text{COMM\_OR\_CONV} and rewriting with it, \text{ND1\_CONV, ND2\_CONV} and \text{SORT\_OR\_CONV}.

Next we wish to have a conversion \text{OR\_CONV} that will put a finite process composed of \text{Stop, Timed\_Action, Choice} and \text{Or} into canonical form. This conversion proceeds in stages. First, we repeatedly rewrite with \text{D1\_CONV} and \text{D4\_CONV}. The result is a term in the form $\sqcap_i P_{ij}$ where each $P_{ij}$ is totally deterministic. Next, we rewrite each $\sqcap_i P_{ij}$ with \text{SORT\_CHOICE\_CONV}.

Each resulting $Q_i$ is now of the form $((A_{i1}\triangleright_{R_{i1}} R_{i1}) \sqcup ((A_{i2}\triangleright_{R_{i2}} R_{i2}) \sqcup \cdots))$ where for each $k$, $A_{ik}\triangleright_{R_{ik}} R_{ik} \leq A_{i(k+1)}\triangleright_{R_{i(k+1)}} R_{i(k+1)}$. In particular, all timed actions that begin with the same set of events will be amalgamated in our choice. The next step is to expand
out our choices by repeatedly rewriting with CH5_CONV (which also rewrites $(A \rightsquigarrow P) \sqcap ((A \rightsquigarrow Q) \sqcap R)$ to $((A \rightsquigarrow P) \sqcap (A \rightsquigarrow Q)) \sqcap R$ and with D1_CONV and D4_CONV. Following this we add to our $\sqcap$ the result of combining with each of the totally deterministic components of the $\sqcap$s. This is done by rewriting $i$ from the innermost $\sqcap$ onwards with CH6_CONV. Then we use SORT_CHOICE_CONV on the outcome. When this phase is completed, the result will be a term that is an $\sqcap$ of totally deterministic. Also the leftmost term in the $\sqcap$ is the result of combining all of the totally deterministic components with $\sqcap$. Let $M$ denote this maximal choice element. The next process to be performed is something we shall refer to as interval saturation. To begin, using ND1, CH2 and CH3, we rewrite $M$ to $\prod_{\sigma \in \Sigma}(\sigma M)$ where $\Sigma$ is the set of all permutations of $M$. Let us refer to this new term as $N$. Our whole term now looks $N \sqcap (\prod_{i=1}^{n} R_i)$. Using ND1 again, together with ND2 and ND3, we make $n$ copies of $N$ and rewrite the term to $\prod_{i=1}^{n}(N \sqcap R_i)$. Then we make as many copies of each $R_i$ as their are permutations of $M$ and regroup to get $\prod_{i=1}^{n}(\prod_{\sigma \in \Sigma}(\sigma M) \sqcap R_i)$. To finish the interval saturation we exhaustively rewrite with INTERVAL_SAT_CONV. Having finished this tortuous procedure, all that remains for OR_CONV to do is to resort and compress everything with SORT_OR_CONV.

Finally, to put an arbitrary process into canonical form FPROC_CONV proceeds by induction on the structure of the finite process. If the term is Stop then we are done. If the outermost constructor is Timed_Action, Choice or Or, then we recursively use FPROC_CONV on its subterms and then use OR_CONV on the composed result. Therefore, all that remains is to handle the case when the outermost constructor is Par. In this case, we first rewrite the subterms using FPROC_CONV as before. Following this we repeatedly rewrite with D5_CONV. The result is a term in the form $\prod_{i,j}(P_i \parallel_C Q_j)$ where each $P_i$ and each $Q_j$ is totally deterministic. Using PAR1 through PAR8 (together with a reversed version of TA2), we may rewrite each $P_i \parallel_C Q_j$ into the form $\prod_{j} Q_{ij}$. To finish, OR_CONV is used on the composed result, which now has no occurrence of Par in it.

As we did before with SET_CONV, we can now use FPROC_CONV to create a conversion FPROC_EQ_CONV that, when applied to two finite processes, converts each to its canonical form, tests whether the canonical forms are identical, and if they are returns the theorem stating that the original two terms are equal, and otherwise raises an exception. We can do no better than this within the abstract theory of real-time process algebras. This is because it is consistent with the axioms given that all processes are equal. Thus we can’t prove in general that any two are not equal.

5 Conclusions and Future Work

The combination of a conversion for putting processes in canonical form together with a tactic for determining the equality of two processes forms a powerful tool for reasoning about processes. In addition, because we are in an extensible environment such as HOL, it can also form the foundation of tools for determining other properties of processes, such as timing constraints and ordering by degree of nondeterminism.

We have omitted all proofs of correctness for the algorithms described above for converting finite processes into canonical form. However, it should be pointed out that, if the above conversions do return a theorem, then we know that in fact it is a theorem which follows exclusively from the theory assumptions and was not asserted by some miscalculation. For these proofs, it is necessary to have access to a specific model of this algebra (which is outlined briefly in Appendix A). Detailed discussion of that model and how it is used to prove correctness of the conversions described above is beyond the scope of this paper.

In future work on real-time process algebras in HOL, we anticipate developing a model (parametrized by events) in which two processes are equal if and only if they are provably equal by the axioms of the abstract theory (the one described in Appendix A). In the presence of such a model it would be possible to develop a “decision procedure” which given two processes (described in the restricted class described above) returns either a theorem stating that the two terms are equal in the algebra, or a theorem stating that they are not equal in the particular instantiation of the processes given by the model.

A A Model of the Real-Time Process Algebra

We give here a brief description of a model for the real-time process algebra that we have worked with in this paper. The axioms of the algebra are not only sound for this model, but also equationally complete for it.
Let $\Sigma$ be a fixed finite alphabet of events. A timed trace is a function $t : \mathcal{N} \rightarrow \mathcal{P}(\Sigma)$ where $\forall n \in \mathcal{N}$ such that $\forall m > n.t m = \emptyset$. Given a timed trace $t$, the duration of $t$ is

$$
\delta(t) = \bigcup_{n \in \mathcal{N}} \{ n \mid \forall m > n.t m = \emptyset \}.
$$

The delay of a timed trace $t$ by a natural number $i$

$$(t + i)n = \begin{cases} 
0 & \text{if } n - i < 0 \\
t(n - i) & \text{if } n - i \geq 0
\end{cases}.
$$

The merge of two timed traces $s$ and $t$ is

$$(s \parallel t)n = (s n) \cup (t n).$$

The concatenation of two timed traces $s$ and $t$ is $(s \cdot t) = s \parallel (t + \delta(s))$. Given $A \subseteq \Sigma$ and $i \in \mathcal{N}$, we may define the timed trace

$$(A, i)n = \begin{cases} 
\emptyset & \text{if } n \neq i \\
A & \text{if } n = i
\end{cases}.
$$

We define the prefix of $T$ as

$$(\text{pre } t)n = \begin{cases} 
t(n) & \text{if } n < \delta(t) \\
\emptyset & \text{otherwise}
\end{cases}.
$$

A state is a subset of $(\mathcal{P}(\Sigma) - \emptyset) \times \mathcal{N}$. The set of all states we shall refer to as STATES. Notice that STATES is closed under both union and intersection. The delay of a set of states $\sigma$ by a natural number $i \in \mathcal{N}$ is

$$\sigma + i = \{ S \mid \exists T \in \sigma \text{ such that } S = \{(A, j + i) \mid (A, j) \in T \} \}.$$

A subset of STATES is saturated if

1. $\forall \tau \subseteq \sigma, \tau \neq \emptyset \Rightarrow (\bigcup_{S \in \tau} S) \in \sigma$.
2. $S \in \text{STATES}.(\exists T \in \sigma. \exists R \in \sigma. R \subseteq S \subseteq T) \Rightarrow S \in \sigma$.
3. Given $S_i \in \sigma$ with $S_1 \supseteq \cdots \supseteq S_n \supseteq \cdots$, we have $\bigcap_{i=1}^{\infty} S_i$ is a subset of $\sigma$.

STATES is a saturated set, as is the empty set. Also, if $\{\sigma_i \mid i \in I\}$ is an indexed collection of saturated sets, then $\bigcap_{i \in I} \sigma_i$ is also a saturated set.

The saturation of a set $\sigma \subseteq \text{STATES}$ is

$$\text{sat}(\sigma) = \bigcap\{ \tau \subseteq \text{STATES} \mid \sigma \subseteq \tau, \tau \text{ saturated} \}.$$  

For all $\sigma \subseteq \text{STATES}$, $\text{sat}(\sigma)$ is saturated, and $\sigma$ is itself saturated iff $\sigma = \text{sat}(\sigma)$. If $\sigma$ is saturated then so $\sigma + i$.

An acceptance trace is a function from timed traces to saturated sets of states. Given an acceptance trace $A$, we may define

$$A + i(t) = \begin{cases} 
A(t) + i & \text{if } t = \lambda n.\emptyset \\
A(s) & \text{if } t \neq \lambda n.\emptyset \text{ and } t = s + i \\
\emptyset & \text{if } \forall s.t \neq s + i
\end{cases}.$$  

The timed traces of an acceptance trace is given by

$$\text{TRS}(A) = \{ t \mid A(t) \neq \emptyset \}.$$  

A real-time process $P$ is an acceptance trace satisfying the following properties:

1. $\lambda n.\emptyset \in \text{TRS}(P)$.
2. For all $t \in \text{TRS}(P)$, we have $\text{pre } (t) \in \text{TRS}(P)$.
3. For all $t \in \text{TRS}(P)$ with $t \neq \lambda n.\emptyset$, there exists $S \in P(\text{pre } t)) \text{ with } (t(\delta(t)), \delta(t) - \delta(\text{pre } t)) \in S$.
4. For all $t \in \text{TRS}(P)$, $S \in P(t)$, and $(A, i) \in S$, we have $(t' (A, i)) \in \text{TRS}(P)$.

Finally, we interpret the constants of our real-time process algebra as follows:

- $\langle \text{STOP} \rangle = \lambda t. \begin{cases} 
\emptyset & \text{if } t = \lambda n.\emptyset \\
\emptyset & \text{if } t \neq \lambda n.\emptyset
\end{cases}$
- $\langle A \rightsquigarrow P \rangle = \begin{cases} 
([P] + i)(t) & \text{if } t = \lambda n.\emptyset, \ A = \emptyset \\
\{ (A, 0) \} & \text{if } t = \lambda n.\emptyset, \ A \neq \emptyset \\
\emptyset & \text{if } t = \langle A, 0 \rangle s
\end{cases}$
- $\langle P \sqcup Q \rangle = \begin{cases} 
\{ \sigma \cup \tau \mid \sigma \in [P](t), \ \tau \in [Q](t) \} & \text{if } t = \lambda n.\emptyset \\
\{ (A, 0) \} & \text{if } t = \lambda n.\emptyset, \ t \in \text{TRS}(P) \cup \text{TRS}(Q) \\
\emptyset & \text{if } t \notin \text{TRS}(P) \cup \text{TRS}(Q)
\end{cases}$
- $\langle P \sqcap Q \rangle = \begin{cases} 
\text{sat}([P](t) \cup [Q](t)) & \text{if } t = \lambda n.\emptyset, \ t \in \text{TRS}(P) \cup \text{TRS}(Q) \\
\emptyset & \text{if } t = \lambda n.\emptyset, \ t \in \text{TRS}(P) \cup \text{TRS}(Q) \\
\emptyset & \text{if } t \notin \text{TRS}(P) \cup \text{TRS}(Q)
\end{cases}$
\[ \begin{align*}
\| P \|_A Q & = \\
\lambda t. \text{sat}\{ S \mid \exists t_P, t_Q, S_P, S_Q. t = t_P \| t_Q, \\
& \forall n. (t_P n \cap A) = (t_Q n \cap A), \\
& S_P \in [P](t_P), \\
& S_Q \in [Q](t_Q) \text{ and} \\
& S = \{(B, i) \mid \\
& (B, i + \delta(t_P) - \delta(t_Q)) \in S_P \\
& \text{and } B \cap A = \emptyset\} \\
& \cup \{(C, j) \mid \\
& (C, j + \delta(t_Q) - \delta(t_P)) \in S_Q \\
& \text{and } C \cap A = \emptyset\} \\
& \cup \{(B \cup C, k) \mid \\
& (B, k + \delta(t_P) - \delta(t_Q)) \in S_P, \\
& (C, k + \delta(t_Q) - \delta(t_P)) \in S_Q \\
& \text{and } (B \cup C) \cap A \subseteq B \cap C\} \}
\end{align*} \]

The set of finite real-time processes are those constructed from STOP, \( \sim \), \( \lor \), \( \cap \) and \( \|_A \). It is the set of all finite real-time processes that our axiom system is sound and equationally complete. Notice that we can model more than just finite real-time processes with the set of real-time processes. In particular, this setting is already rich enough to be able to model recursive processes. With a slight modification, we can also model graceful termination (i.e., SKIP). In each of these cases we will still be able to use the algebra dealt with in this paper to reason about equality of those processes of the larger model that are finite.

References


