Ciobanu G, Koutny M, Steggles J. [Strategy based semantics for mobility with time and access permissions](#). *Formal Aspects of Computing* 2015, 27(3), 525-549.

Copyright:

The final publication is available at Springer via <http://dx.doi.org/10.1007/s00165-014-0324-9>

DOI link to article:

<http://dx.doi.org/10.1007/s00165-014-0324-9>

Date deposited:

20/05/2015

Embargo release date:

09 December 2015

This work is licensed under a [Creative Commons Attribution-NonCommercial 3.0 Unported License](#)

Strategy Based Semantics for Mobility with Time and Access Permissions

Gabriel Ciobanu¹, Maciej Koutny² and Jason Steggles²

¹Romanian Academy, Institute of Computer Science, Iași, Romania.
gabriel@info.uaic.ro

²School of Computing Science, University of Newcastle, UK.
{maciej.koutny,jason.steggles}@ncl.ac.uk

Abstract. The process algebras T_IMo (Timed Mobility) and its extension PERT_IMo (Permissions, Timers and Mobility) were recently proposed to support engineering applications in distributed system design. T_IMo provides a formal framework in which process migration between distinct locations and timing constraints linked to local clocks can be modelled and analysed. This is extended in PERT_IMo by associating access permissions to communication to model security aspects of a distributed system. In this paper we develop a new semantic model for T_IMo using Rewriting Logic (RL) and strategies, with the aim of providing a foundation for tool support; in particular, strategies are used to capture the locally maximal concurrent step of a T_IMo specification which previously required the use of action rules based on negative premises. This RL model is then extended with access permissions in order to develop a new semantic model for PERT_IMo. These RL semantical models are formally proved to be sound and complete with respect to the original operational semantics on which they were based. We present examples of how the developed RL models for T_IMo and PERT_IMo can be implemented within the strategy-based rewriting system ELAN and illustrate the range of (behavioural) properties that can be analysed using such a tool.

Keywords: process algebra, mobility, time, access permissions, rewriting logic, strategies

1. Introduction

The process algebras T_IMo (Timed Mobility) [CiK08] and its extension PERT_IMo (Permissions, Timers and Mobility) [CiK11b, CiK14] were recently proposed to support engineering applications in distributed system design. In T_IMo processes can migrate between distinct locations and timers linked to local clocks are used to control mobility and communication actions. The resulting semantic approach based on local maximal concurrency is motivated by the assumptions made by the GALS approach [Das06], which specifies that systems operate under a ‘globally asynchronous/locally synchronous’ execution strategy. The behaviour of T_IMo specifications can be captured using a set of SOS rules or suitable Petri nets [CiK11a], both based

Correspondence and offprint requests to: Jason Steggles, School of Computing Science, University of Newcastle, UK. email: jason.steggles@ncl.ac.uk

on executing time actions with negative premises (i.e. premises which ensure that no other alternative actions are applicable). PERTiMO is an extension of TiMO which associates access permissions with communication actions in order to model security aspects of a distributed system [CiK11b, CiK14]. Moreover, processes are able to acquire new access permissions, or lose some of their current access permissions while moving between locations, modelling an important security feature.

In this paper, we present new semantic models for TiMO and PERTiMO with the aim of providing a basis for much needed tool support and a formal framework for investigating the different semantic options available. We use Rewriting Logic (RL) [Mes92], an algebraic formalism for modelling dynamic systems which uses equational specifications to define the states of a system, and rewrite rules to capture the dynamic state transitions. Strategies [BKKR01, BKKM02] are an integral part of RL and provide control over the rewriting process, allowing important dynamic properties to be modelled.

We develop an RL model for TiMO which involves formulating a strategy that captures the maximal parallel computational step of a TiMO specification, including its time rule based on negative premises. The resulting RL model is formally validated by showing that it is both *sound* and *complete* with respect to the original operational semantics of TiMO. To illustrate using the RL semantics as a basis for tool support we use the strategy-based rewrite system ELAN [BKKMR98, BKKM02] to simulate and analyse TiMO specifications. The simple example we provide gives a useful insight into the proposed RL modelling approach and illustrates the type of (behavioural) properties that can be analysed. This work extends the initial research on modelling TiMO presented in [CKS12].

In order to specify and reason about access permissions for mobility and communication in a distributed environment the TiMO framework has been extended with access permissions. PERTiMO [CiK11b, CiK14] is a formal framework for specifying interprocess communication controlled by access permissions that processes must possess in order to send and receive information. Access permissions are dynamic, and processes acquire and lose access permissions when migrating between locations. We consider how to modify our RL model for TiMO to develop a new semantic model for PERTiMO. This provides a basis for tool support for PERTiMO and also illustrates the flexibility and extendibility provided by using the RL framework. The resulting RL model is prototyped again using ELAN and formally validated against the original PERTiMO semantics.

The paper is structured as follows. Section 2 describes the syntax and semantics of TiMO and introduces a small illustrative example. Section 3 briefly introduces RL and the idea of rewriting strategies. In Section 4, we develop an RL model of TiMO and provide a formal argument for its correctness. A simple example of the RL model is then implemented using the support tool ELAN and the type of properties that can be analysed using our approach are illustrated. In Section 5 we introduce PERTiMO and extend our RL model with access permissions to provide a new semantic model for PERTiMO. The resulting RL model is formally shown to be correct and ELAN is again used to implement a small illustrative example. Finally, in Section 6 we conclude by discussing related work and by considering areas for future work.

2. TiMO (Timed Mobility Language)

TiMO (Timed Mobility) [CiK08, CiK11a, CiK11b, CiK14] is a process algebra for mobile systems where it is possible to add timers to the basic actions, and each location runs according to its own local clock which is invisible to processes. Processes have communication capabilities which are active up to a predefined time deadline. Other timing constraints specify the latest time for moving between locations.

We assume suitable data types together with associated operations, including a set *Loc* of *locations*, a set *Chan* of *communication channels*, and a set *Id* of process identifiers, where each *id* $\in$ *Id* has arity m_{id} . We use $\vec{x}$ to denote a finite tuple of elements $(x_1, \dots, x_k)$ whenever it does not lead to a confusion.

The syntax of TiMO is given in Table 1, where *P* represents *processes* and *N* represents *networks*. Moreover, for each *id* $\in$ *Id*, there is a unique process definition (DEF), where the u_i 's are distinct variables playing the role of parameters; the X_i^{id} 's are data types, and P_{id} is a process expression which may make use of the variables u_i . In Table 1, it is assumed that: (i) $a \in Chan$ is a channel, and $t \in \mathbb{N} \cup \{\infty\}$ represents a timeout; (ii) each v_i is an expression built from data values and variables; (iii) each u_i is a variable, and each X_i is a data type; (iv) l is a location or a location variable; and (v) $\textcircled{S}$ is a special symbol used to state that a process is temporarily 'stalled'.

The only variable binding construct is $a^{\Delta t} ? (\vec{u} : \vec{X}) \text{ then } P \text{ else } P'$ which binds the variables $\vec{u}$ within *P* (but *not* within *P'*). We use $fv(P)$ to denote the free variables of a process *P* (and similarly for networks).

<i>Processes</i>	$P ::= a^{\Delta t} ! \langle \vec{v} \rangle \text{ then } P \text{ else } P' \quad (\text{output})$ $a^{\Delta t} ? (\vec{u} : \vec{X}) \text{ then } P \text{ else } P' \quad (\text{input})$ $\text{go}^{\Delta t} l \text{ then } P \quad (\text{move})$ $P P' \quad (\text{parallel})$ $\text{id}(\vec{v}) \quad (\text{recursion})$ $\text{stop} \quad (\text{termination})$ $\textcircled{S} P \quad (\text{stalling})$
<i>Networks</i>	$N ::= l[P] \mid N N'$
<i>Definition</i>	$\text{id}(u_1, \dots, u_{m_{id}} : X_1^{id}, \dots, X_{m_{id}}^{id}) \stackrel{\text{df}}{=} P_{id} \quad (\text{DEF})$

Table 1. T1MO Syntax. Length of $\vec{u}$ is the same as $\vec{X}$, and length of $\vec{v}$ in $\text{id}(\vec{v})$ is m_{id} .

Fig. 1. A pictorial representation of a simple T1MO work flow example.

For a process definition as in (DEF), we assume that $fv(P_{id}) \subseteq \{u_1, \dots, u_{m_{id}}\}$, and so the free variables of P_{id} are parameter bound. Processes are defined up to the alpha-conversion, and $\{v/u, \dots\}P$ is obtained from P by replacing all free occurrences of a variable u by v , etc, possibly after alpha-converting P in order to avoid clashes. Moreover, if $\vec{v}$ and $\vec{u}$ are tuples of the same length then $\{\vec{v}/\vec{u}\}P$ denotes $\{v_1/u_1, v_2/u_2, \dots, v_k/u_k\}P$.

A process $a^{\Delta t} ! \langle \vec{v} \rangle \text{ then } P \text{ else } P'$ attempts to send a tuple of values $\vec{v}$ over the channel a for t time units. If successful, it continues as process P ; otherwise it continues as the alternative process P' . A process $a^{\Delta t} ? (\vec{u} : \vec{X}) \text{ then } P \text{ else } P'$ attempts for t time units to input a tuple of values of type $\vec{X}$ and substitute them for the variables $\vec{u}$. Mobility is implemented by a process $\text{go}^{\Delta t} l \text{ then } P$ which moves from the current location to the location l within t time units. Note that since l can be a variable, and so its value is assigned dynamically through communication with other processes, migration actions support a flexible scheme for moving processes around a network. Processes are further constructed from the (terminated) process stop and parallel composition $P | P'$. Finally, process expressions of the form $\textcircled{S} P$ are a purely technical device which is used in the subsequent formalisation of structural operational semantics of T1MO; intuitively, $\textcircled{S}$ specifies that a process P is temporarily (i.e., until a clock tick) *stalled* and so cannot execute any action. A located process $l[P]$ is a process running at location l , and a network is composed out of its components $N | N'$.

As an illustrative example, consider a simple workflow example in which a processing job moves from an initial location to a web service location and finally to a done location. If an error occurs with the web service then the job enters an error location. A pictorial representation of this example is given in Figure 1. The T1MO specification WF consists of four locations: *Init*; *Web*; *Done*; and *Err*. The following process identifier definitions are used:

$$\begin{aligned}
 \text{job} &\stackrel{\text{df}}{=} a^{\Delta 1} ? (l : \text{Loc}) \text{ then } \text{go}^{\Delta 1} l \text{ then } \text{job} \text{ else } \text{job} \\
 \text{serv}(l : \text{Loc}) &\stackrel{\text{df}}{=} a^{\Delta 2} ! \langle l \rangle \text{ then } \text{serv}(l) \text{ else } \text{serv}(l) \\
 \text{servErr}(l : \text{Loc}) &\stackrel{\text{df}}{=} a^{\Delta 2} ! \langle l \rangle \text{ then } \text{servErr}(l) \text{ else } \text{servErr}(\text{Err})
 \end{aligned}$$

For instance, $\text{Init} [\text{job} | \text{serv}(\text{Web})] \mid \text{Web} [\text{serv}(\text{Done})]$ could be an initial T1MO network for this example.

A network N is *well-formed* if: (i) there are no free variables in N ; (ii) there are no occurrences of the special symbol $\textcircled{S}$ in N ; (iii) assuming that id is as in the recursive equation (DEF), for every $\text{id}(\vec{v})$ occurring in N or on the right hand side of any recursive equation, the expression v_i is of type corresponding to X_i^{id} . We let $\text{Prs}(TM)$ and $\text{Net}(TM)$ represent the set of well-formed T1MO process and network terms respectively. The first component of the operational semantics of T1MO is the structural equivalence $\equiv$ on networks. It is the smallest congruence such that the equalities (EQ1–EQ3) in Table 2 hold. Using (EQ1–EQ3) one can always transform a given network N into a finite parallel composition of networks of the form

(EQ1)	$N N' \equiv N' N$
(EQ2)	$(N N') N'' \equiv N (N' N'')$
(EQ3)	$l \llbracket P P' \rrbracket \equiv l \llbracket P \rrbracket l \llbracket P' \rrbracket$
(CALL)	$l \llbracket id(\vec{v}) \rrbracket \xrightarrow{id@l} l \llbracket \textcircled{\$} \{\vec{v}/\vec{u}\} P_{id} \rrbracket$
(MOVE)	$l \llbracket go^{\Delta t} l' \text{ then } P \rrbracket \xrightarrow{l'@l} l' \llbracket \textcircled{\$} P \rrbracket$
(COM)	$\frac{v_1 \in X_1 \dots v_k \in X_k}{l \llbracket a^{\Delta t} !(\vec{v}) \text{ then } P \text{ else } Q \mid a^{\Delta t'} ?(\vec{u}; \vec{X}) \text{ then } P' \text{ else } Q' \rrbracket}$ $\xrightarrow{a(\vec{v})@l} l \llbracket \textcircled{\$} P \mid \textcircled{\$} \{\vec{v}/\vec{u}\} P' \rrbracket$
(PAR)	$\frac{N \xrightarrow{\psi} N'}{N N'' \xrightarrow{\psi} N' N''}$
(EQUIV)	$\frac{N \equiv N' \quad N' \xrightarrow{\psi} N'' \quad N'' \equiv N'''}{N \xrightarrow{\psi} N'''}$
(TIME)	$\frac{N \not\rightarrow_l}{N \xrightarrow{\check{l}} \phi_l(N)}$

Table 2. Three rules of the structural equivalence (EQ1-EQ3), and six action rules (CALL MOVE COM PAR EQUIV TIME) of the operational semantics. In (PAR) and (EQUIV) ψ is an action, and in (TIME) l is a location.

$l_1 \llbracket P_1 \rrbracket \mid \dots \mid l_n \llbracket P_n \rrbracket$ such that no process P_i has the parallel composition operator at its topmost level. Each subnetwork $l_i \llbracket P_i \rrbracket$ is called a *component* of N , the set of all components is denoted by $comp(N)$, and the parallel composition is called a *component decomposition* of the network N . Note that these notions are well defined since component decomposition is unique up to the permutation of the components. This follows from the rule (CALL) which treats recursive definitions as function calls which take a unit of time. Another consequence of such a treatment is that it is impossible to execute an infinite sequence of action steps without executing any local clock ticks.

Table 2 introduces two kinds of operational semantics rules: $N \xrightarrow{\psi} N'$ and $N \xrightarrow{\check{l}} N'$. The former is an execution of an action ψ by some process, and the latter a unit time progression at location l . In the rule (TIME), $N \not\rightarrow_l$ means that the rules (CALL) and (COM) as well as (MOVE) with $\Delta t = \Delta \theta$ cannot be applied to N for this particular location l . Moreover, $\phi_l(N)$ is obtained by taking the component decomposition of N and simultaneously replacing all the components of the form $l \llbracket go^{\Delta t} l' \text{ then } P \rrbracket$ by $l \llbracket go^{\Delta t-1} l' \text{ then } P \rrbracket$, and all components of the form $l \llbracket a^{\Delta t} \omega \text{ then } P \text{ else } Q \rrbracket$ (where ω stands for $!(\vec{v})$ or $?(\vec{u}; \vec{X})$) by $l \llbracket Q \rrbracket$ if $t = 0$, and $l \llbracket a^{\Delta t-1} \omega \text{ then } P \text{ else } Q \rrbracket$ otherwise. After that, all the occurrences of the symbol $\textcircled{\$}$ in N are erased.

The above defines executions of individual actions. A complete computational step is captured by a *derivation* of the form $N \xRightarrow{\Psi} N'$, where $\Psi = \{\psi_1, \dots, \psi_m\}$ ($m \geq 0$) is a finite multiset of l -actions for some location l (i.e., actions of the form $id@l$ or $l'@l$ or $a(\vec{v})@l$) such that $N \xrightarrow{\psi_1} N_1 \dots N_{m-1} \xrightarrow{\psi_m} N_m \xrightarrow{\check{l}} N'$. That is, a derivation is a condensed representation of a sequence of individual actions followed by a clock tick, all happening at the same location. In other words, for any multiset of actions Ψ , any permutation of the individual actions in Ψ corresponds to the same derivation (see Proposition 2 in [CiK11a]). Thus a derivation captures the cumulative effect of the concurrent (i.e. interleaved) execution of the multiset of actions Ψ at location l . It is important to note that the semantical treatment of TiMo goes beyond interleaving semantics by introducing an explicit representation of maximal concurrency in the execution of actions. We say that N' is *directly reachable* from N . Whenever there is only a time progression at a location, we have $N \xRightarrow{\emptyset} N'$.

As an example, consider three derivation steps in the workflow network:

$$\begin{array}{l}
\text{Init} \llbracket \text{job} \mid \text{serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \\
\hline \{ \text{job} @ \text{Init}, \text{serv} @ \text{Init} \} \\
\text{Init} \llbracket a^{\Delta^1} ? (l : \text{Loc}) \text{ then go}^{\Delta^1} l \text{ then job else job} \mid a^{\Delta^2} ! \langle \text{Web} \rangle \text{ then} \\
\text{serv}(\text{Web}) \text{ else serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \\
\hline \{ a(\text{Web}) @ \text{Init} \} \\
\text{Init} \llbracket \text{go}^{\Delta^1} \text{ Web then job} \mid \text{serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{serv}(\text{Done}) \rrbracket \\
\hline \{ \text{Web} @ \text{Init}, \text{serv} @ \text{Init} \} \\
\text{Init} \llbracket a^{\Delta^2} ! \langle \text{Web} \rangle \text{ then serv}(\text{Web}) \text{ else serv}(\text{Web}) \rrbracket \mid \text{Web} \llbracket \text{job} \mid \text{serv}(\text{Done}) \rrbracket
\end{array}$$

One can show that derivations are well defined as one cannot execute an unbounded sequence of action moves without time progressing, and the execution Ψ is made up of independent (or concurrent) individual executions (see explanation above). Moreover, derivations preserve well-formedness of networks (see [CiK08]).

3. Rewriting Logic and Strategies

Rewriting logic (RL) [Mes92] is an algebraic specification approach which is able to model dynamic system behaviour. In RL the static properties of a system are described by a standard algebraic specification (see [EhM85, MeT92]), whereas the dynamic behaviour of the system is modelled using rewrite rules. Rewrite strategies are then used to control the application of rewrite rules and allow an RL specification to capture subtle aspects of a dynamic system. A brief introduction to RL and rewriting strategies is presented below (for a more detailed introduction see [Mes92, BKKM02]).

An S -sorted signature Σ defines a collection of function symbols, where: $c : s \in \Sigma$ means c is a constant symbol of sort $s \in S$; and

$$f : s(1) \dots s(n) \rightarrow s \in \Sigma$$

means f is a function symbol in Σ of *domain type* $s(1) \dots s(n)$, *arity* n , and *codomain type* s . Let $X = \langle X_s \mid s \in S \rangle$ be a family of sets of variables. We let $T(\Sigma, X) = \langle T(\Sigma, X)_s \mid s \in S \rangle$ be the family of sets of all terms over Σ and X . For any term $t \in T(\Sigma, X)_s$, we let $\text{Var}(t) \subseteq \cup_{s \in S} X_s$ represent the set of variables used in t . We let $T(\Sigma, X)/E$ represent the free quotient algebra of terms with respect to a set of equations E over Σ and X . For any term $t \in T(\Sigma, X)_s$, we let $\langle t \rangle_E$ represent the equivalence class of term t with respect to the equations E (see [MeT92]).

In RL a specification (Σ, E) defines the states $\langle t \rangle_E$ of a system. The dynamic behaviour of the system is then specified by *rewrite rules* [Mes92, BKKM02]:

$$l \Longrightarrow r,$$

for terms $l, r \in T(\Sigma, X)_s$ and $s \in S$, where $\text{Var}(r) \subseteq \text{Var}(l)$. Such rules represent dynamic transitions between states $\langle l \rangle_E$ and $\langle r \rangle_E$. We also allow rules to be labelled and to contain conditions:

$$[lb] \ l \Longrightarrow r \quad \text{if } c,$$

where lb is a (not necessarily unique) label, $c \in T(\Sigma, X)_{\text{bool}}$ and $\text{Var}(c) \subseteq \text{Var}(l)$. Intuitively, the condition means that the rewrite rule can only be applied if term c rewrites to *true*. A *Rewriting logic specification* is therefore a triple $\text{Spec} = (\Sigma, E, R)$ consisting of an algebraic specification (Σ, E) and a set of (conditional) rewrite rules R over Σ and X .

As an example of an RL specification consider a model of a simple dynamic system in which states are multi-sets of symbols A , B , and C . The resulting RL specification $\text{Spec}(MS) = (\Sigma, E, R)$ is defined as follows. Let $S = \{\text{ent}, \text{ms}\}$ be a sort set and let Σ be an S -sorted signature which contains the following function symbols:

$$\begin{array}{l}
A, B, C : \text{ent} \in \Sigma, \\
\text{empty} : \text{ms} \in \Sigma, \\
@ : \text{ent} \rightarrow \text{ms} \in \Sigma, \\
@ \otimes @ : \text{ms ms} \rightarrow \text{ms} \in \Sigma,
\end{array}$$

(where @ is used to indicate the position of an argument in a function symbol to allow for an infix notation). Note that the signature contains an implicit type coercion operator $@ : ent \rightarrow ms$.) The set of equations E contains the equations which axiomatize the associative/commutative properties of a multi-set. Note that the rewrite rules defined below will be applied modulo these equations. Finally, we define R to contain the following three rewrite rules:

$$\begin{array}{ll} [Rule1] & A \otimes m1 \Longrightarrow B \otimes m1 \\ [Rule2] & B \otimes C \otimes m1 \Longrightarrow B \otimes A \otimes m1 \\ [Rule3] & B \otimes B \otimes m1 \Longrightarrow C \otimes m1. \end{array}$$

where $m1 \in X_{ms}$. Let $A \otimes C$ be a multi-set representing the initial state of the system. Then the trace

$$A \otimes C \Longrightarrow B \otimes C \Longrightarrow B \otimes A \Longrightarrow B \otimes B \Longrightarrow C$$

represents one possible evolution of the system.

Rewriting Logic provides the notion of a *strategy* for controlling the application of rewrite rules [BKRR01, BKMM02]. A strategy allows the user to specify the order in which rewrite rules are applied and the possible choices that can be made. The result of applying a strategy is the set of all possible terms that can be produced according to the strategy. A strategy is said to *fail* if it can not be applied (i.e. produces no results). The following is a brief overview of some *elementary strategies* (based on [BKRR01, BKMM02]):

- (i) **Basic strategy:** lb Any label used in a labelled rule $[lb] \quad t \Rightarrow t'$ is a strategy. The result of applying a basic strategy lb is the set of all terms that could result from one application of any rule labelled lb .
- (ii) **Concatenation strategy:** $s_1; s_2$ Allows strategies to be sequentially composed, i.e. s_2 is applied to the set of results from s_1 .
- (iii) **Don't know strategy:** $dk(s_1, \dots, s_n)$ Returns the union of all the sets of terms that result from applying each strategy $s_1, \dots, s_n$. This strategy fails if, and only if, all the strategies $s_1, \dots, s_n$ fail.
- (iv) **Don't care strategy:** $dc(s_1, \dots, s_n)$ Chooses nondeterministically to apply one of the strategies s_i which does not fail. Again, the strategy can only fail if all of the strategies $s_1, \dots, s_n$ fail. The strategy $dc \text{ one}(s_1, \dots, s_n)$ works in a similar way but chooses a *single* result term to return, where as $first(s_1, \dots, s_n)$ applies the first successful strategy in the sequence $s_1, \dots, s_n$.
- (v) **Iterative strategies:** $repeat^*(s)$ Repeatedly applies s , zero or more times, until the s fails. It returns the last set of results produced before s failed. The $repeat^+(s)$ version works in a similar way but insists that s must be successfully applied at least once.

As an example, the strategy

$$repeat^*(dk(Rule1, Rule2, Rule3))$$

for $Spec(MS)$ simply returns all terms reachable by repeatedly applying the three rules and so produces the same set of terms as in the original example. Alternatively, $repeat^*(first(Rule1, Rule2, Rule3))$ is a strategy for $Spec(MS)$ which prioritises the rules so that $Rule1$ is always applied first if it can be, $Rule2$ is applied only if the first rule cannot be applied and $Rule3$ is applied only if the previous two rules cannot be applied.

The above elementary strategy language can be extended to a *defined strategy language* [BKRR01, BKMM02] which allows recursive strategies to be defined. As an example, consider the simple recursive search strategy $search(i)$ defined below:

$$\begin{array}{l} doStep \Longrightarrow dk(Rule1, Rule2, Rule3) \\ search(i) \Longrightarrow fail \quad \text{if } i \leq 0 \\ search(i) \Longrightarrow first(found, doStep; search(i-1)) \quad \text{if } i > 0 \end{array}$$

The strategy $search(i)$ repeatedly applies the strategy $doStep$ looking for a multi-set term that satisfies the strategy $found$. It fails if the given maximum number of iterations i is reached. So to search for a multi-set term containing $A \otimes B \otimes C$ we would define the strategy $found$ by the following rewrite rule:

$$[found] \quad A \otimes B \otimes C \otimes m1 \Longrightarrow A \otimes B \otimes C \otimes m1$$

A range of tools have been developed for supporting rewriting logic and strategies, including: MAUDE [CDEL02]; ELAN [BKMR98, BKMM02]; STRATEGO [Vis01]; and TOM [BKMR07]. In this paper we have chosen to use ELAN to implement our examples given its simple strategy language and the authors' experience with this tool.

4. Modelling TiMO using Rewriting Logic and Strategies

In this section we develop a semantic model of TiMO using rewriting logic and strategies, and provide a formal argument of correctness.

4.1. Developing an RL Model for TiMO

Given a TiMO specification TM we consider how to develop a corresponding RL model $RL(TM)$ that correctly captures the meaning of TM . We begin by modelling the general concept of a process and network in RL.

Let S be the set of sorts in $RL(TM)$ containing: nat for time; $Chan$ for channels; $VLoc$, $ALoc$, and Loc for locations; Prs for processes, and $Nets$ for networks. Coping with the parameter passing that occurs in communication requires careful consideration and for this reason the sort Loc is defined as the union of two subsorts: $VLoc$ represents the input location variables; and $ALoc$ represents the actual locations used in TM . Note for simplicity, we have restricted the parameters used in communication between processes within TM to a single location parameter. Extending this would be relatively straightforward and would involve adding the appropriate sorts to the sort set S (as has been done for locations).

The S -sorted signature $\Sigma^{RL(TM)}$ for $RL(TM)$ contains the following function symbols to capture the syntax for processes given in Table 1:

$$\begin{aligned} stop &: Prs, & S(@) &: Prs \rightarrow Prs, & @ \mid @ &: Prs \ Prs \rightarrow Prs \\ go(@, @) & \text{ then } @ &: nat \ Loc \ Prs \rightarrow Prs \\ in(@, @)(@) & \text{ then } @ \text{ else } @ &: Chan \ nat \ VLoc \ Prs \ Prs \rightarrow Prs \\ out(@, @) < @ > & \text{ then } @ \text{ else } @ &: Chan \ nat \ Loc \ Prs \ Prs \rightarrow Prs. \end{aligned}$$

The function symbol $@ \mid @$ is defined equationally to be associative and commutative as per the definition of TiMO. To model process definitions we add a function symbol $id : s_1 \dots s_n \rightarrow Prs$ for each process identifier $id(u_1, \dots, u_n : s_1, \dots, s_n)$, where s_i is assumed to be a well-defined data type in our model. Note that given we have restricted parameters to being locations only, each s_i has to be one of the built in sorts nat , $Chan$ or $ALoc$.

We then define the following function symbols to represent networks:

$$@[@] : ALoc \ Prs \rightarrow Nets; \quad @ \mid @ : Nets \ Nets \rightarrow Nets;$$

where $@ \mid @$ is again defined to be associative and commutative.

We now need to formulate appropriate rewrite rules to begin to capture the intended semantics of TiMO. In the RL model developed here we choose the approach of forcing network components with the same location to merge (this turns out to be important since it simplifies the selection of a location to update). The above approach is realized using the rule $al[p1] \mid al[p2] \Rightarrow al[p1 \mid p2]$. Clearly, such a rule is compatible with Eq 3 from Table 2. Each network term will therefore have the form $at_1[pt_1] \mid \dots \mid at_n[pt_n]$, where each location at_i is unique and where each pt_i will represent a set of parallel processes. Any process term which does not contain the parallel operator at its topmost level is referred to as an *atomic process term*. Each individual network location term will have the form $at_i[pt_i^1 \mid \dots \mid pt_i^k]$, where each pt_i^j is an atomic process term.

Next we consider how to model the action rules given in Table 2 within our RL model. First, we define two labelled rules to model the action rule (MOVE):

$$\begin{aligned} [move] \quad al[go(t, al2) \text{ then } p1 \mid p2] &\Longrightarrow al2[S(p1)] \mid al[p2] \\ [move] \quad al[go(t, al2) \text{ then } p1 \mid p2] &\Longrightarrow \\ &\quad al[S(go(t-1, al2) \text{ then } p1) \mid p2] \quad \text{if } t > 0 \end{aligned}$$

The two rules can both be applied when $t > 0$ and this leads to a non-deterministic choice between moving location or allowing time to pass. Note that if $t = 0$ then only the rule that moves to a different location is applicable.

To model the synchronisation required for communication as defined by the action rule (COM) we have the following rule:

$$[com] \quad al[out(c, t1) < al1 > \text{ then } p1 \text{ else } p2 \mid in(c, t2)(vl) \text{ then } p3 \text{ else } p4 \mid p5] \\ \implies al[S(p1) \mid S(p3[vl/al1]) \mid p5]$$

This rule makes use of a substitution function $@[\@/\@] : Prs \ VLoc \ ALoc \rightarrow Prs$, where $pt[vt/at]$ represents the process term that results by substituting all free occurrences (not bound by an input action symbol) of $VLoc$ term vt by the $ALoc$ term at within the process term pt . This function is straightforward to define algebraically using recursion on process terms.

In any TiMo specification TM there will be process definitions of the form

$$id(u_1, \dots, u_n : s_1, \dots, s_n) \stackrel{df}{=} P_{id}$$

which allow each process identifier $id \in Id$ to be associated with a well-formed process expression P_{id} (see the action rule (CALL) in Table 2). In $RL(TM)$, for each $id \in Id$ we add a rewrite rule of the form:

$$[calls] \quad al[id(u_1, \dots, u_n) \mid p] \Rightarrow al[S(RL(P_{id})) \mid p]$$

where $RL(P_{id})$ is the process term that results from translating P_{id} into $RL(TM)$ and each u_i is a variable of sort s_i in $RL(TM)$. (Note that when this rule is applied the variables $u_1, \dots, u_n$ will be instantiated with values and therefore any instances of these variables in $RL(P_{id})$ will also be instantiated.)

The above labelled rules are collectively referred to as *process transition rules* and are used to define a strategy *step* that represents an update step as follows:

$$step \implies repeat^*(dc(calls, move, com))$$

The strategy repeatedly applies the three process transition rules and makes use of the *dc* built-in strategy as the order the rules are applied in is irrelevant given that they act on disjoint sets of terms.

In TiMo the last step of any derivation involves applying the (TIME) action rule which allows time to progress and removes all stall symbols. We model this by using a function $tick(@) : Prs \rightarrow Prs$ which is applied to the terms resulting from *step*. We define *tick* recursively as illustrated by the sample rules below:

$$\begin{aligned} tick(stop) &\implies stop \\ tick(S(p1)) &\implies p1 \\ tick(p1 \mid p2) &\implies tick(p1) \mid tick(p2) \\ tick(id(u_1, \dots, u_n)) &\implies id(u_1, \dots, u_n) \\ tick((out(a, t) < l > \text{ then } p1 \text{ else } p2)) &\implies \\ &\quad (out(a, t-1) < l > \text{ then } p1 \text{ else } p2) \quad \text{if } t > 0 \\ tick((out(a, 0) < l > \text{ then } p1 \text{ else } p2)) &\implies p2 \end{aligned}$$

To make the application of this function straightforward we overload *tick* so that it can be applied to networks by defining $tick(@) : Nets \rightarrow Nets$ by

$$tick(al[p]) \implies al[tick(p)], \quad tick(n1 \mid n2) \implies tick(n1) \mid tick(n2)$$

Note that when using *tick* we restrict its application to a single location at anyone time to ensure time progression confirms to the TiMo semantics.

We can now formulate a rewrite rule *oneStep* in $RL(TM)$ using the strategy *step* and function *tick* that models a derivation step in TM :

$$[oneStep] \quad al[p] \mid n1 \implies n3 \mid n1 \\ \text{where } n2 := (step) al[p], \\ n3 := () tick(n2)$$

The pattern $al[p] \mid n1$ is used to match non-deterministically with a collection of network components (due to the associative/commutative property of $@ \mid @$) and so chooses the next location to update.

It is interesting to note that different semantic choices can be considered for TiMo by appropriately updating the *oneStep* strategy. For example, we could straightforwardly consider a synchronous semantics, introduce priorities to locations or add fairness assumptions. This provides further motivation for developing our RL model.

4.2. Correctness of RL Model

Having developed an RL model for T1MO we now validate that it correctly captures the semantics of T1MO. We do this by showing that our model is *sound* (each step in our RL model represents a derivation step in T1MO) and *complete* (every derivation step possible in T1MO is represented in our RL model). In the sequel let TM be a T1MO specification and let $RL(TM)$ be the corresponding RL model as defined in Section 4.1.

Not all the terms of sort Prs in $RL(TM)$ represent valid processes in TM since they may contain the stall symbol S . Another problem can arise with the improper use of location variables, that is terms of sort $VLoc$, since all uses other than those in an input command need to be bound by an outer input command. We formalise what we mean by a valid process term by defining a function VP .

Definition 1 The function $VP : T(\Sigma^{RL(TM)})_{Prs} \times \mathcal{P}(T(\Sigma^{RL(TM)})_{VLoc}) \rightarrow \mathbf{B}$ is defined recursively over the structure of process terms as follows:

$$\begin{aligned}
VP(stop, VS) &= true \\
VP(S(pt), VS) &= false \\
VP(id(v_1, \dots, v_n), VS) &= \begin{cases} true & \text{if } v_i \in VS, \text{ for all } v_i \text{ of sort } VLoc \\ false & \text{otherwise} \end{cases} \\
VP(go(nt, at) \text{ then } pt, VS) &= VP(pt, VS) \\
VP(go(nt, vt) \text{ then } pt, VS) &= vt \in VS \wedge VP(pt, VS) \\
VP(in(ct, nt)(vt) \text{ then } pt_1 \text{ else } pt_2, VS) &= VP(pt_1, VS \cup \{vt\}) \wedge VP(pt_2, VS) \\
VP(out(ct, nt) < vt > \text{ then } pt_1 \text{ else } pt_2, VS) &= \\
&\quad VP(pt_1, VS) \wedge vt \in VS \wedge VP(pt_2, VS) \\
VP(out(ct, nt) < at > \text{ then } pt_1 \text{ else } pt_2, VS) &= VP(pt_1, VS) \wedge VP(pt_2, VS) \\
VP(pt_1 \mid pt_2, VS) &= VP(pt_1, VS) \wedge VP(pt_2, VS).
\end{aligned}$$

We define $valPrs(TM) = \{pt \mid pt \in T(\Sigma^{RL(TM)})_{Prs} \text{ and } VP(pt, \{\})\}$ to be the set of all *valid process terms* and define the set $valNet(TM)$ of *valid network terms* recursively by: (1) $at[pt] \in valNet(TM)$ if $pt \in valPrs(TM)$; and (2) $net_1 \mid net_2 \in valNet(TM)$, if $net_1, net_2 \in valNet(TM)$.

It can be shown that *oneStep* preserves valid network terms.

Theorem 2 The strategy *oneStep* is well-defined with respect to valid network terms, i.e. for any $net_1 \in valNet(TM)$, if $net_1 \Rightarrow net_2$ using *oneStep* then $net_2 \in valNet(TM)$.

Proof. By the definition of *oneStep* it suffices to consider a valid network location term of the form

$$at[pt_1 \mid \dots \mid pt_n] \in valNet(TM),$$

where $n > 0$ and each pt_i is an atomic process term. It can be seen that each process term pt_i is involved in at most one process transition rule application when *oneStep* is applied. This gives four possible cases to consider.

Case 1) Suppose pt_i is not involved in the application of any process transition rules. Then by the definition of *oneStep* and $valNet(TM)$ (Definition 1) we need to show that $tick(pt_i)$ results in a valid process term. There are three possible subcases to consider:

- i) Suppose pt_i is the process term *stop*. Then by definition of *tick* we know $tick(stop)$ rewrites to *stop* which is clearly a valid process term.
- ii) Suppose pt_i is the process term $tick(out(ct, nt) < vt > \text{ then } pt_i^1 \text{ else } pt_i^2)$. By the definition of *tick* there are two possibilities to consider. First, if $nt > 0$ holds then time is allowed to progress and the resulting process term $out(ct, nt - 1) < vt > \text{ then } pt_i^1 \text{ else } pt_i^2$ must be valid given that the original output term was. Secondly, we could have $nt = 0$ in which case the timer has expired and the resulting process term is pt_i^2 . Clearly pt_i^2 must be valid given that the original output term was.
- iii) Suppose pt_i is the process term $tick(in(ct, nt)(vt) \text{ then } pt_i^1 \text{ else } pt_i^2)$. Then a similar argument to ii) above can be used, noting that the input operator does not bind vt in pt_i^2 .

Case 2) Suppose pt_i has the form $id(v_1, \dots, v_n)$ and that a $[calls]$ rule is applied to it, i.e.

$$id(v_1, \dots, v_n) \Rightarrow S(pt)$$

where pt is the process term $RL(P_{id})$ with variables $u_1, \dots, u_n$ replaced by terms $v_1, \dots, v_n$. Clearly the stall symbol S will be removed by the application of the $tick$ function at the end of $oneStep$. Since P_{id} was a well-formed T1MO process expression it is straightforward to show that $pt \in valPrs(TM)$ as required.

Case 3) Suppose pt_i has the form $go(nt, at_2)$ then pt and that a $[move]$ rule is applied to it. Then we have two possible cases:

i) Suppose that $nt > 0$ holds and that applying the $[move]$ rule simply allows time to pass, i.e.

$$go(nt, at_2) \text{ then } pt \Rightarrow S(go(nt - 1, at_2) \text{ then } pt)$$

Since the stall symbol S will be removed by the application of the $tick$ function, it can be seen that $go(nt - 1, at_2) \text{ then } pt$ must be a valid process term given that the original process term was.

ii) Suppose that applying a $[move]$ rule results in the process moving locations, i.e. produces the network term $at_2[S(pt)]$, which becomes $at_2[pt]$ after applying $tick$. Since the original atomic process term was valid it follows that pt must be valid and so the resulting new network term must also be valid as required.

Case 4) Suppose that the $[com]$ rule has been applied to two process terms pt_i and pt_j , for $i \neq j$. That is, suppose

$$\begin{aligned} out(ct, nt_1) < at_2 > \text{ then } pt_i^1 \text{ else } pt_i^2 \mid in(ct, nt_2)(vt) \text{ then } pt_j^1 \text{ else } pt_j^2 \\ \Rightarrow S(pt_i^1) \mid S(pt_j^1[vt/at_2]) \end{aligned}$$

Then the stall symbol S will be removed by the application of the $tick$ function at the end of $oneStep$. It follows that pt_i^1 must be a valid process term since the original output term was valid. Also $pt_j^1[vt/at_2]$ must be valid since $VP(pt_j^1, \{vt\})$ must be true as the original input term was valid and as $pt_j^1[vt/at_2]$ is simply pt_j^1 with all unbounded occurrences of location variable term vt replaced by the actual location term at_2 . $\square$

We can define an interpretation mapping between T1MO terms in TM and terms in the corresponding RL model $RL(TM)$ as follows.

Definition 3 The process term mapping $\sigma_{Prs} : Prs(TM) \rightarrow valPrs(TM)$ is defined recursively by:

$$\begin{aligned} \sigma_{Prs}(stop) &= stop, & \sigma_{Prs}(id(v_1, \dots, v_n)) &= id(v_1, \dots, v_n), \\ \sigma_{Prs}(go^{\Delta t} l \text{ then } P) &= go(t, l) \text{ then } \sigma_{Prs}(P), \\ \sigma_{Prs}(a^{\Delta t} ? (vl : Loc) \text{ then } P \text{ else } P') &= \\ &in(a, t)(vl) \text{ then } \sigma_{Prs}(P) \text{ else } \sigma_{Prs}(P'), \\ \sigma_{Prs}(a^{\Delta t} ! \langle l \rangle \text{ then } P \text{ else } P') &= out(a, t) < l > \text{ then } \sigma_{Prs}(P) \text{ else } \sigma_{Prs}(P'), \\ \sigma_{Prs}(P \mid P') &= \sigma_{Prs}(P) \mid \sigma_{Prs}(P'). \end{aligned}$$

The network term mapping $\sigma_{Net} : Net(TM) \rightarrow valNet(TM)$ is defined using σ_{Prs} by $\sigma_{Net}(l \llbracket P \rrbracket) = l[\sigma_{Prs}(P)]$ and $\sigma_{Net}(N \mid N') = \sigma_{Net}(N) \mid \sigma_{Net}(N')$.

It is straightforward to show that σ_{Prs} and σ_{Net} are bijective mappings and thus have inverses. In order to show the correctness of the RL model we need to prove it is *sound* and *complete* with respect to T1MO (see Figure 2).

We now show that for any T1MO specification TM the RL model $RL(TM)$ defined in Section 4.1 is a sound and complete model of TM .

Theorem 4 (Soundness) Let $net_1, net_2 \in valNet(TM)$ be any valid network terms. Then if $net_1 \Rightarrow net_2$ using the strategy $oneStep$ then $\sigma_{Net}^{-1}(net_1) \xRightarrow{\Psi} \sigma_{Net}^{-1}(net_2)$ for some finite multiset $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions and some location l . In other words, the diagram for soundness in Figure 2 must commute.

Fig. 2. The properties of soundness and completeness required for $RL(TM)$ to be a correct model of the TiMo specification TM .

Proof. By the definition of *oneStep* and the notion of a derivation in TiMo it suffices to consider a valid network location term of the form

$$at[pt_1 \mid \dots \mid pt_n] \in valNet(TM),$$

where $n > 0$ and each pt_i is an atomic process term. It can be seen that each process term pt_i is involved in at most one process transition rule application when *oneStep* is applied. This gives us four possible cases to consider.

Case 1) Suppose pt_i is not involved in the application of any process transition rules during a derivation using *oneStep*. Then we need to show that applying a time step in TiMo to the process expression $\sigma_{P_{rs}}^{-1}(pt_i)$ results in the process $\sigma_{P_{rs}}^{-1}(tick(pt_i))$. There are three possible subcases to consider:

- i) Suppose pt_i is the process term *stop*. Then by definition of *tick* and $\sigma_{P_{rs}}^{-1}$ we have $\sigma_{P_{rs}}^{-1}(stop) = stop$ and $\sigma_{P_{rs}}^{-1}(tick(stop)) = stop$. Then by time progression in TiMo we have $at \llbracket stop \rrbracket \xrightarrow{\vee_{at}} at \llbracket stop \rrbracket$ as required.
- ii) Suppose pt_i is the process term $out(ct, nt) < at_2 > then pt_i^1 else pt_i^2$. By the definition of *tick* there are two possibilities to consider. First, if $nt > 0$ holds then time is allowed to progress and the resulting process term is $out(ct, nt - 1) < at_2 > then pt_i^1 else pt_i^2$. By definition of time progression in TiMo, if $nt > 0$ we know

$$at \llbracket ct^{\Delta nt} ! \langle at_2 \rangle then pt_i^1 else pt_i^2 \rrbracket \xrightarrow{\vee_{at}} at \llbracket ct^{\Delta nt - 1} ! \langle at_2 \rangle then pt_i^1 else pt_i^2 \rrbracket$$

as required. Secondly, we could have $nt = 0$ in which case the timer has expired and the resulting process term is pt_i^2 . Again, by definition of time progression we know

$$at \llbracket ct^{\Delta 0} ! \langle at_2 \rangle then pt_i^1 else pt_i^2 \rrbracket \xrightarrow{\vee_{at}} at \llbracket pt_i^2 \rrbracket$$

as required.

- iii) Suppose pt_i is the process term $in(ct, nt)(vt) then pt_i^1 else pt_i^2$. Then the result follows by a similar argument to ii) above.

Case 2) Suppose pt_i has the form $id(v_1, \dots, v_n)$ and that a *[calls]* rule is applied to it, i.e.

$$id(v_1, \dots, v_n) \Rightarrow S(pt)$$

where pt is the process term $RL(P_{id})$ with variables $u_1, \dots, u_n$ replaced by terms $v_1, \dots, v_n$. Note that *tick* will remove the stall symbol and so the final atomic process term will be pt . Then by the action rule (CALL) in Table 2 we have

$$at \llbracket id(v_1, \dots, v_n) \rrbracket \xrightarrow{id @ at} at \llbracket \textcircled{S} \sigma_{P_{rs}}^{-1}(pt) \rrbracket$$

The result follows since the stall symbol $\textcircled{S}$ will be removed by the time progression step in TiMo.

Case 3) Suppose pt_i has the form $go(nt, at_2) then pt$ and that a *[move]* rule is applied to it. Then we have two possible cases:

- i) Suppose that $nt > 0$ holds and that applying the *[move]* rule simply allows time to pass, i.e.

$$go(nt, at_2) then pt \Rightarrow S(go(nt - 1, at_2) then pt)$$

Clearly, the stall symbol S will be removed by the application of the *tick* function. In TiMo by the definition of time progression and the assumption $nt > 0$ we have

$$at \llbracket go^{\Delta nt} at_2 then \sigma_{P_{rs}}^{-1}(pt) \rrbracket \xrightarrow{\vee_{at}} at \llbracket go^{\Delta nt - 1} at_2 then \sigma_{P_{rs}}^{-1}(pt) \rrbracket$$

as required.

ii) Suppose that applying a $[move]$ rule results in the process moving locations, i.e. produces the network term $at_2[S(pt)]$. Clearly, the stall symbol S will be removed by the application of the $tick$ function. Then by the action rule (MOVE) in Table 2 we have

$$at \llbracket go^{\Delta nt} at_2 \text{ then } \sigma_{P_{rs}}^{-1}(pt) \rrbracket \xrightarrow{at_2 @ at} at_2 \llbracket \textcircled{S} \sigma_{P_{rs}}^{-1}(pt) \rrbracket$$

The results follows since the stall symbol $\textcircled{S}$ will be removed by the time progression step in TIMO.

Case 4) Suppose that the $[com]$ rule has been applied to two process terms pt_i and pt_j , for $i \neq j$. That is, suppose

$$\begin{aligned} & out(ct, nt_1) < at_2 > \text{ then } pt_i^1 \text{ else } pt_i^2 \mid in(ct, nt_2)(vt) \text{ then } pt_j^1 \text{ else } pt_j^2 \\ & \Rightarrow S(pt_i^1) \mid S(pt_j^1[vt/at_2]) \end{aligned}$$

The instances of the stall symbol S will be removed by the application of the $tick$ function at the end of $oneStep$. Then by the action rule (COM) in Table 2 we have

$$\begin{aligned} & at \llbracket ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } \sigma_{P_{rs}}^{-1}(pt_i^1) \text{ else } \sigma_{P_{rs}}^{-1}(pt_i^2) \mid ct^{\Delta nt_2} ? (vt : Loc) \\ & \text{ then } \sigma_{P_{rs}}^{-1}(pt_j^1) \text{ else } \sigma_{P_{rs}}^{-1}(pt_j^2) \rrbracket \xrightarrow{ct < at_2 > @ at} at \llbracket \textcircled{S} \sigma_{P_{rs}}^{-1}(pt_i^1) \mid \textcircled{S} \{at_2/vt\} \sigma_{P_{rs}}^{-1}(pt_j^1) \rrbracket \end{aligned}$$

The result follows since the stall symbol $\textcircled{S}$ will be removed by the time progression step in TIMO and since we can show $\sigma_{P_{rs}}^{-1}(pt_j^1[vt/at_2]) = \{at_2/vt\} \sigma_{P_{rs}}^{-1}(pt_j^1)$. $\square$

Theorem 5 (Completeness) Let $N_1, N_2 \in Net(TM)$ be any well-formed network terms in TM . Then, if $N_1 \xRightarrow{\Psi} N_2$, for some location l and some multi-set $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions, then $\sigma_{Net}(N_1) \Rightarrow \sigma_{Net}(N_2)$ using $oneStep$. In other words, the diagram for completeness in Figure 2 commutes.

Proof. By the definition of a derivation in TIMO and the strategy $oneStep$ it suffices to consider a well-formed network of the form

$$at \llbracket P_1 \mid \dots \mid P_n \rrbracket \equiv at \llbracket P_1 \rrbracket \mid \dots \mid at \llbracket P_n \rrbracket,$$

where $n > 0$ and each P_i is an atomic process. Suppose $at \llbracket P_1 \mid \dots \mid P_n \rrbracket \xRightarrow{\Psi} N'$, for some finite set of at -actions $\Psi = \{\psi_1, \dots, \psi_m\}$, $m \geq 0$. Then it can be seen that each atomic process P_i is involved in at most one at -action ψ_i . We show that the at -action applied to each process P_i is correctly captured by the $oneStep$ strategy in the RL model. We have four possible cases to consider.

Case 1): Suppose P_i is not involved in any of the action rules (CALL), (MOVE) and (COM) during the derivation step. That is $at \llbracket P_i \rrbracket \xrightarrow{\psi_i} at \llbracket P'_i \rrbracket$, where $at \llbracket P'_i \rrbracket = \phi_l(at \llbracket P_i \rrbracket)$. Then by definition of $oneStep$ we need to show that $tick(\sigma_{P_{rs}}(P_i))$ results in the process $\sigma_{P_{rs}}(P'_i)$. Considering the possible form of P_i gives us three subcases to consider:

i) Suppose P_i is the process **stop**. Then $\phi_l(at \llbracket \text{stop} \rrbracket) = at \llbracket \text{stop} \rrbracket$ by definition of ϕ_l and we have $\sigma_{P_{rs}}(\text{stop}) = stop$ and $\sigma_{P_{rs}}(\phi_l(\text{stop})) = stop$. It follows by the definition of $tick$ that $tick(stop) = stop$ as required.

ii) Suppose P_i is the process $ct^{\Delta nt} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2$. Then

$$\sigma_{P_{rs}}(P_i) = out(ct, nt) < at_2 > \text{ then } \sigma_{P_{rs}}(P_i^1) \text{ else } \sigma_{P_{rs}}(P_i^2).$$

By the definition of time progression in TIMO there are two possibilities to consider. First, if $nt > 0$ then time is allowed to progress resulting in the process

$$ct^{\Delta nt-1} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2$$

Applying $\sigma_{P_{rs}}$ to this process gives $out(ct, nt-1) < at_2 > \text{ then } \sigma_{P_{rs}}(P_i^1) \text{ else } \sigma_{P_{rs}}(P_i^2)$ and so by the definition of $tick$ the result follows. Secondly, we could have $nt = 0$ in which case the timer has expired and the resulting process is P_i^2 . Again, by definition of $tick$ it can be seen that the result follows.

iii) Suppose P_i is the process $ct^{\Delta nt} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2$. Then the result follows by a similar argument to ii) above.

Case 2) Suppose P_i is the process $id(v_1, \dots, v_n)$ and that the action rule (CALL) is applied, i.e.

$$at \llbracket id(v_1, \dots, v_n) \rrbracket \xrightarrow{id @ at} at \llbracket \textcircled{S} \{ \vec{v} / \vec{u} \} P_{id} \rrbracket$$

where the stall symbol $\textcircled{S}$ is removed by the final time progression step. We have $\sigma_{Prs}(id(v_1, \dots, v_n)) = id(v_1, \dots, v_n)$ and so applying the $[calls]$ rule to this term gives

$$id(v_1, \dots, v_n) \Rightarrow S(pt)$$

where pt is the process term $\sigma_{Prs}(P_{id})$ with variables $u_1, \dots, u_n$ replaced by terms $v_1, \dots, v_n$. Note that $tick$ will remove the stall symbol and so the final atomic process term will be pt . It is straightforward to see that $\sigma_{Prs}(\{ \vec{v} / \vec{u} \} P_{id}) = pt$ as required.

Case 3) Suppose P_i has the form $go^{\Delta nt} at_2 \text{ then } P'$. Then we have

$$\sigma_{Prs}(go^{\Delta nt} at_2 \text{ then } P') = go(nt, at_2) \text{ then } \sigma_{Prs}(P')$$

By the definition of a derivation in TIMO we have two possible cases:

i) Suppose that $nt > 0$ holds and that the action rule (MOVE) has *not* been applied and instead time has been allowed to progress, i.e.

$$at \llbracket go^{\Delta nt} at_2 \text{ then } P' \rrbracket \xrightarrow{\vee at} at \llbracket go^{\Delta nt-1} at_2 \text{ then } P' \rrbracket$$

This can be modelled by applying the appropriate $[move]$ rule

$$go(nt, at_2) \text{ then } \sigma_{Prs}(P') \Rightarrow S(go(nt-1, at_2) \text{ then } \sigma_{Prs}(P'))$$

where the stall symbol S will be removed by the application of the $tick$ function. It is straightforward to see that

$$\sigma_{Prs}(go^{\Delta nt-1} at_2 \text{ then } P') = go(nt-1, at_2) \text{ then } \sigma_{Prs}(P')$$

as required.

ii) Suppose the action rule (MOVE) was applied to P_i

$$at \llbracket go^{\Delta nt} at_2 \text{ then } P' \rrbracket \xrightarrow{at_2 @ at} at_2 \llbracket \textcircled{S} P' \rrbracket$$

where the stall symbol $\textcircled{S}$ is removed by the final time step. This can be copied in the RL model by applying the appropriate $[move]$ rule

$$at[go(nt, at_2) \text{ then } \sigma_{Prs}(P')] \Rightarrow at_2[S(\sigma_{Prs}(P'))]$$

Clearly, the stall symbol S will be removed by the application of the $tick$ function. It is then straightforward to see that

$$\sigma_{Net}(at_2 \llbracket P' \rrbracket) = at_2[\sigma_{Prs}(P')]$$

by the definition of σ_{Prs} .

Case 4) Suppose the action rule (COM) has been applied to two processes P_i and P_j , for $i \neq j$, i.e.

$$at \llbracket ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2 \mid ct^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2 \rrbracket$$

$$\xrightarrow{ct < at_2 > @ at} at \llbracket \textcircled{S} P_i^1 \mid \textcircled{S} \{ at_2 / vt \} P_j^1 \rrbracket$$

where the stall symbols $\textcircled{S}$ will be removed by the final time step. Then we have

$$\begin{aligned} & \sigma_{Prs}(ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } P_i^1 \text{ else } P_i^2 \mid ct^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2) \\ &= out(ct, nt_1) < at_2 > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) \mid in(ct, nt_2)(vt) \text{ then } \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) \end{aligned}$$

by definition of the term mapping. By applying the $[com]$ rule we have

$$\begin{aligned} & out(ct, nt_1) < at_2 > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) \mid in(ct, nt_2)(vt) \text{ then } \\ & \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) \Rightarrow S(\sigma_{Prs}(P_i^1)) \mid S(\sigma_{Prs}(P_j^1)[vt/at_2]) \end{aligned}$$

where all occurrences of the stall symbol S will be removed by the *tick* function. It is then straightforward to see that

$$\sigma_{P_{rs}}(P_i^1 \mid \{at_2/vt\}P_j^1) = \sigma_{P_{rs}}(P_i^1) \mid \sigma_{P_{rs}}(P_j^1)[vt/at_2]$$

by definition of $\sigma_{P_{rs}}$ and since we can show $\sigma_{P_{rs}}(\{at_2/vt\}P_j^1) = \sigma_{P_{rs}}(P_j^1)[vt/at_2]$. $\square$

4.3. An Illustrative Example

In this section we investigate using ELAN [BKKMR98, BKKM02], a strategy-based rewrite system, to implement a TiMo specification based on our RL model. We consider a small example which provides useful insight into the RL modelling approach used and illustrates the type of (behavioural) properties that can be analysed.

Recall the simple TiMo workflow example introduced in Section 2. The specification WF can be mapped into an RL model $RL(WF)$ as described in Section 4.1 and then investigated using ELAN to provide insight into the behaviour of the original TiMo specification. A range of (behavioural) properties can be analysed including time constraints, use of locations, and causality between actions. For example, consider the following initial TiMo network:

$$Init \llbracket job \mid serv(Web) \rrbracket \mid Web \llbracket serv(Done) \rrbracket$$

After translating this into $RL(WF)$ we can use ELAN to derive the following rewriting trace which shows how a processing job can reach the *Done* location:

$$\begin{aligned} & Init[job \mid serv(Web)] \mid Web[serv(Done)] \\ \Rightarrow & \\ & Init[in(a,1)(WL) \text{ then } go(1, WL) \text{ then } job \text{ else } job \mid out(a,2) < Web > \text{ then } \\ & serv(Web) \text{ else } serv(Web)] \mid Web[serv(Done)] \\ \Rightarrow & \\ & Init[go(1, Web) \text{ then } job \mid serv(Web)] \mid Web[serv(Done)] \\ \Rightarrow & \\ & Init[out(a,2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid Web[job \mid serv(Done)] \\ \Rightarrow & \\ & Init[out(a,2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid Web[in(a,1)(WL) \text{ then } \\ & go(1, WL) \text{ then } job \text{ else } job \mid out(a,2) < Done > \text{ then } serv(Done) \\ & \text{ else } serv(Done)] \\ \Rightarrow & \\ & Init[out(a,2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ & Web[go(1, Done) \text{ then } job \mid serv(Done)] \\ \Rightarrow & \\ & Init[out(a,2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ & Web[out(a,2) < Done > \text{ then } serv(Done) \text{ else } serv(Done)] \mid Done[job] \end{aligned}$$

The example trace contains six derivation steps and indeed it is easy to verify using ELAN that this is the smallest number of steps needed in order for a processing job starting at *Init* to reach the *Done* location. Next we consider what happens if we change our network so that it contains a faulty service process:

$$Init \llbracket job \mid serv(Web) \rrbracket \mid Web \llbracket servErr(Done) \rrbracket$$

Again, using ELAN and a simple search strategy we are able to confirm that it is still possible for a processing job to reach the *Done* location. Furthermore, we can show that it is now possible for a processing job to end up in the *Err* location as the following term derived using ELAN shows:

$$\begin{aligned} & Init[out(a,2) < Web > \text{ then } serv(Web) \text{ else } serv(Web)] \mid \\ & Web[out(a,2) < Err > \text{ then } servErr(Err) \text{ else } servErr(Err)] \mid Err[job] \end{aligned}$$

5. Extending the Semantics with Access Permissions

In a distributed environment, processes need to possess sufficient access rights to migrate and communicate; moreover, such access permissions can dynamically change. In this section, we outline **PERTiMo** [CiK11b, CiK14] which extends **TiMo** with interprocess communication controlled by access permissions that processes must possess in order to send and receive information. Access permissions are dynamic and processes can acquire new access permissions, or lose some of their current access permissions while moving between locations, modelling an important security feature.

To communicate over a channel at a given network location, the sender process should have a ‘put’ access permission, and the receiving process a ‘get’ access permission. In general, the set of access permissions Γ of a process is a subset of the overall set of access permissions:

$$AccPerm \stackrel{\text{df}}{=} \{put, get\} \times Chan \times Loc$$

We use the notation $get\langle a@l \rangle$ to denote an access permission $(get, a, l) \in AccPerm$ and $put\langle a@l \rangle$ to denote $(put, a, l) \in AccPerm$. Intuitively, we work with access permissions to sockets where l represents an IP address and a represents a communication port. Access permissions of a process change while moving from one location to another. To model this, we use the following four basic access permission modification operations:

$$put_{a@l}^+ \quad get_{a@l}^+ \quad put_{a@l}^- \quad get_{a@l}^-$$

where l is a location and a is a communication channel. The first two ($put_{a@l}^+$ and $get_{a@l}^+$) add access permissions, while the latter two ($put_{a@l}^-$ and $get_{a@l}^-$) remove access permissions. For instance,

$$put_{a@l}^+(\Gamma) = \Gamma \cup \{put\langle a@l \rangle\}$$

Then an *access permission modification* operation is either the identity on $AccPerm$, or a composition of some basic access permission modification operations such that if $put_{a@l}^+$ is used in the composition then $put_{a@l}^-$ is not used (giving and at the same time removing an access permission does not make sense). For instance,

$$get_{a@l}^+ \circ put_{b@l'}^-(\Gamma) = \Gamma \cup \{get\langle a@l \rangle\} \setminus \{put\langle b@l' \rangle\}$$

For a given network, we then specify what are the changes to the access permission sets of processes migrating from one location to another. This is specified as an *access permission modification* mapping $apmod$ which, for each pair of locations, returns a permission modification operation. Hence, if a process with the current access permissions Γ moves from location l to location l' , its new set of access permissions becomes $apmod(l, l')(\Gamma)$.

The syntax of **PERTiMo** extends that of **TiMo** with the notion of *processes with access permissions*

$$PP ::= P : \Gamma \mid PP \mid PP'$$

where P is a process defined exactly as in Table 1 and Γ is a set of access permissions. Networks are then re-defined in the following way:

$$N ::= l \llbracket PP \rrbracket \mid N \mid N'$$

Thus a network $l \llbracket P : \Gamma \rrbracket$ specifies a process P with the access permissions Γ running at the location l . Other syntactic notions and assumptions are as those defined for **TiMo**, with obvious modifications resulting from the additions of sets of access permissions.

The computational steps $N \xRightarrow{\Psi} N'$ of **PERTiMo** networks are defined following the same scheme as in the case of **TiMo** networks, except that some of the execution rules of Table 2 are modified to reflect the fact that now processes appearing in networks are equipped with sets of access permissions, and one new structural rule is added. Table 3 provides the necessary details. Two crucial rules there are (MOVEP) and (COMP). The former reflects the change of access permissions applied to a migrating process, and the latter expresses the assumption that communications (send or receive) can only happen if processes possess suitable access permissions.

Note that now in the rule (TIME), $N \not\rightarrow_l$ means that the rules (CALLP) and (COMP) as well as (MOVEP) with $\Delta t = \Delta 0$ cannot be applied to N for this particular location l . Moreover, $\phi_l(N)$ is obtained by taking the component decomposition of N and simultaneously replacing all the components of the form

$$\begin{array}{ll}
(\text{EQ4}) & l \llbracket P \mid Q : \Gamma \rrbracket \equiv l \llbracket P : \Gamma \mid Q : \Gamma \rrbracket \\
\\
(\text{CALLP}) & l \llbracket id(\vec{v}) : \Gamma \rrbracket \xrightarrow{id@l} l \llbracket \textcircled{\$} \{\vec{v}/\vec{u}\} P_{id} : \Gamma \rrbracket \\
(\text{MOVEP}) & l \llbracket go^{\Delta t} l' \text{ then } P : \Gamma \rrbracket \xrightarrow{l \triangleright l'} l' \llbracket \textcircled{\$} P : apmod(l, l')(\Gamma) \rrbracket \\
\\
(\text{COMP}) & \frac{\begin{array}{c} put\langle a@l \rangle \in \Gamma \quad get\langle a@l \rangle \in \Gamma' \quad \vec{v} \in \prod \vec{X} \\ l \llbracket a^{\Delta t} ! \langle \vec{v} \rangle \text{ then } P \text{ else } Q : \Gamma \mid a^{\Delta t'} ? (\vec{u} : \vec{X}) \text{ then } P' \text{ else } Q' : \Gamma' \rrbracket \end{array}}{a \langle \vec{v} \rangle @l \rightarrow l \llbracket \textcircled{\$} P : \Gamma \mid \textcircled{\$} \{\vec{v}/\vec{u}\} P' : \Gamma' \rrbracket}
\end{array}$$

Table 3. An additional structural equivalence rule of PERTiMO, and four modified action rules.

Fig. 3. A pictorial representation of a simple PERTiMO access permission example.

$l \llbracket go^{\Delta t} l' \text{ then } P : \Gamma \rrbracket$ by $l \llbracket go^{\Delta t-1} l' \text{ then } P : \Gamma \rrbracket$, and all components of the form $l \llbracket a^{\Delta t} \omega \text{ then } P \text{ else } Q : \Gamma \rrbracket$ (where ω stands for $! \langle \vec{v} \rangle$ or $? (\vec{u} : \vec{X})$) by $l \llbracket Q : \Gamma \rrbracket$ if $t = 0$, and $l \llbracket a^{\Delta t-1} \omega \text{ then } P \text{ else } Q : \Gamma \rrbracket$ otherwise. After that, all the occurrences of the symbol $\textcircled{\$}$ in N are erased.

Similarly, as for TiMO, all PERTiMO networks directly reachable from a well-formed network also are well-formed, and so computational steps $N \xRightarrow{\Psi} N'$ are well-defined evolutions of well-formed networks (see [CiK11b, CiK14]). Moreover, it is important to stress again that the semantical treatment of PERTiMO goes beyond interleaving semantics by introducing an explicit representation of maximal concurrency in the execution of actions.

We would like to stress that the above presentation of PERTiMO is semantically equivalent to that used in [CiK11b, CiK14], even though the formalisation of the operational rules is slightly different, as we decided to keep the (TIME) rule used in the formulation of TiMO.

As an illustrative example, consider a simple example in which processes attempt to gain access to a secure location. Only those processes that come initially from a trusted location *Safe* are allowed to obtain the address of the secure location *Area* with processes from other originating locations being blocked. A pictorial representation of this example is given in Figure 3. The PERTiMO specification *CheckAP* consists of four locations: *Safe*; *Other*; *Check*; and *Area*. It contains two process definitions:

$$\begin{aligned}
req &\stackrel{\text{df}}{=} go^{\Delta 1} \text{ Check then } a^{\Delta 1} ? (l : Loc) \text{ then } go^{\Delta 1} l \text{ then stop else stop} \\
chk &\stackrel{\text{df}}{=} a^{\Delta 2} ! \langle Area \rangle \text{ then } chk \text{ else } chk
\end{aligned}$$

Finally, the access permission modification operator is defined as follows:

$$\begin{aligned}
apmod(\text{Safe}, \text{Check}) &= get_{a@Check}^+, \\
apmod(\text{Other}, \text{Check}) &= get_{a@Check}^-, \\
apmod(\text{Check}, \text{Area}) &= get_{a@Check}^-,
\end{aligned}$$

with all other combinations of input locations returning the identity function. Then an example of a PERTiMO network would be

$$\text{Safe} \llbracket req : \odot \rrbracket \mid \text{Other} \llbracket req : \{get\langle a@Check \rangle\} \rrbracket \mid \text{Check} \llbracket chk : \{put\langle a@Check \rangle\} \rrbracket$$

(T _{MOVE})	$\frac{apmod(l, l')(\Gamma) \vdash_{l'} P}{\Gamma \vdash_l \text{go}^{\Delta t} l' \text{ then } P}$	(T _{STOP})	$\Gamma \vdash_l \text{stop}$
(T _{OUT})	$\frac{put\langle a@l \rangle \in \Gamma \quad \Gamma \vdash_l P \quad \Gamma \vdash_l Q}{\Gamma \vdash_l a^{\Delta t} ! \langle \vec{v} \rangle \text{ then } P \text{ else } Q}$	(T _{PAR})	$\frac{\Gamma \vdash_l P \quad \Gamma \vdash_l Q}{\Gamma \vdash_l P Q}$
(T _{IN})	$\frac{get\langle a@l \rangle \in \Gamma \quad \forall \vec{v} \in \prod \vec{X} : \Gamma \vdash_l \{\vec{v}/\vec{u}\}P \quad \Gamma \vdash_l Q}{\Gamma \vdash_l a^{\Delta t} ? (\vec{u}:\vec{X}) \text{ then } P \text{ else } Q}$	(T _{REC})	$\frac{\Gamma \vdash_l id(\vec{v})^{\uparrow H}}{\Gamma \vdash_l id(\vec{v})}$

Table 4. Derivation rules for processes with safe access permissions. In the rule (T_{STOP}), Γ is any set of access permissions. The constant H in the rule (T_{REC}) for recursive processes is equal to $2 \cdot |Loc| \cdot (1 + \sum_{id \in Id} |X_1^{id}| \dots |X_{m_{id}}^{id}|)$. For all $id \in Id$, $n \geq 0$ and $\vec{v} \in \prod \vec{X}^{id}$ (where $\prod \vec{X}^{id}$ stands for $X_1^{id} \times \dots \times X_{m_{id}}^{id}$), the n -th unfolding of $id(\vec{v})$ is given by $id(\vec{v})^{\uparrow n} \stackrel{\text{df}}{=} \text{stop}$ if $n = 0$, and $id(\vec{v})^{\uparrow n} \stackrel{\text{df}}{=} P$ if $n > 0$, where P is obtained from $\{\vec{v}/\vec{u}\}P_{id}$ by replacing each subexpression of the form $id'(\vec{w})$ with $id'(\vec{w})^{\uparrow n-1}$. Moreover, $\text{stop}^{\uparrow n} \stackrel{\text{df}}{=} \text{stop}$.

5.1. Safe Access Permissions

One of the problems which is central to designs expressed in PERTIMO is to ensure that a migrating process possesses a sufficiently rich set of initial access permissions so that whenever it attempts to communicate over a channel, it has the required access permission, irrespective of the other processes used in the construction of the system. (We then say that the process has *safe access permissions*.) This can be verified by taking into account that migrating processes have their access permission sets modified according to the mapping *apmod*. As a result, one can provide a solution to an important security problem related to migration and access permissions in the sense that one should rule out unauthorised attempts to communicate over the channels.

The solution provided in [CiK11b, CiK14], uses judgements of the form

$$\Gamma \vdash_l P \tag{1}$$

to mean that a single-component well-formed network $l[P:\Gamma]$ has *safe access permissions*. Note that it may be impossible to find Γ satisfying (1), for a given P and l , due to conflicts between access permission modifications resulting from migration and the subsequent communication. For example, if

$$P = \text{go}^{\Delta 0} l' \text{ then } a^{\Delta 1} ! \langle 1 \rangle \text{ then stop else stop}$$

and $apmod(l, l') = put_{a@l'}^-$, then there is no Γ such that $\Gamma \vdash_l P$.

Given a set of locations *Loc* together with the *apmod* mapping, as well as a process P and location l , one can devise typing rules for checking that a set of access permissions Γ satisfies (1). These rules are given in Table 4.

Returning to our PERTIMO example specification *CheckAP*, we can observe that the process contained within *Safe* $\llbracket req : \odot \rrbracket$ has safe access permissions, by applying the derivation rules of Table 4, in the following way. First we observe that for any PERTIMO system, and for any Γ, l and l' we have:

$$\Gamma \vdash_l \text{go}^{\Delta 1} l' \text{ then stop} \tag{2}$$

Indeed, by (T_{STOP}) we have

$$apmod(l, l')(\Gamma) \vdash_{l'} \text{stop}$$

Hence, by (T_{MOVE}), (2) holds. We then observe that by (2) and (T_{STOP}), we have in our case:

$$\begin{aligned} \{get\langle a@Check \rangle\} &\vdash_{Check} \text{go}^{\Delta 1} Check \text{ then stop} \\ \{get\langle a@Check \rangle\} &\vdash_{Check} \text{go}^{\Delta 1} Safe \text{ then stop} \\ \{get\langle a@Check \rangle\} &\vdash_{Check} \text{go}^{\Delta 1} Other \text{ then stop} \\ \{get\langle a@Check \rangle\} &\vdash_{Check} \text{go}^{\Delta 1} Area \text{ then stop} \\ \{get\langle a@Check \rangle\} &\vdash_{Check} \text{stop} \end{aligned}$$

Hence, by (T_{IN}), we obtain

$$\{get\langle a@Check \rangle\} \vdash_{Check} a^{\Delta 1} ? (l:Loc) \text{ then go}^{\Delta 1} l \text{ then stop else stop}$$

and so, by (TMOVE), we have

$$\odot \vdash_{\text{Safe}} \text{go}^{\Delta^1} \text{Check then } a^{\Delta^1} ? (l:\text{Loc}) \text{ then go}^{\Delta^1} l \text{ then stop else stop}$$

Finally, by (TREC),

$$\odot \vdash_{\text{Safe}} \text{req}$$

Note that it is not possible to establish that

$$\{\text{get}\langle a @ \text{Check} \rangle\} \vdash_{\text{Other}} \text{req}$$

Indeed, for this to hold we must have, by (TREC),

$$\{\text{get}\langle a @ \text{Check} \rangle\} \vdash_{\text{Other}} \text{go}^{\Delta^1} \text{Check then } a^{\Delta^1} ? (l:\text{Loc}) \text{ then go}^{\Delta^1} l \text{ then stop else stop}$$

In turn, for this to hold, we must have, by (TMOVE),

$$\odot \vdash_{\text{Check}} a^{\Delta^1} ? (l:\text{Loc}) \text{ then go}^{\Delta^1} l \text{ then stop else stop}$$

which clearly does not hold, by (TIN).

In the general case, a well-formed network N has *safe access permissions* if each of its parallel components has safe access permissions. Crucially, one can then show that networks with safe access permissions capture the desired notion of a guarantee of correct access to communication channels. More precisely, if N is a network reachable from a well-formed network with safe access permissions, then the following hold.

- If $l \llbracket a^{\Delta^1} ! \langle \vec{v} \rangle \text{ then } P \text{ else } Q : \Gamma \rrbracket$ is a parallel component of N , then $\text{put}\langle a @ l \rangle \in \Gamma$.
- If $l \llbracket a^{\Delta^1} ? (\vec{u}:\vec{X}) \text{ then } P \text{ else } Q : \Gamma \rrbracket$ is a parallel component of N , then $\text{get}\langle a @ l \rangle \in \Gamma$.

Moreover, one can also show that networks with safe access permissions are the only ones which enjoy the above properties regardless of an environment in which they are placed. We can therefore conclude that PERTiMO provides a sound and complete framework for ensuring safety of communication for networks of migrating processes. In particular, if safe access permissions for a part of a network can be established, one can simplify the checking of access permissions in the rule (COMP), simplifying the operational semantics rules for this part of the network and its descendants.

The notion of safe access permissions is powerful, but it may prove too restrictive for practical applications where safe access permissions are guaranteed thanks, for example, to the specific timings of the processes involved, even though there may be environments in which illegal access may occur. For this we need a finer behavioural analysis. In particular, it can be done by extending the technique developed earlier in this paper, as discussed in the next section.

5.2. Extending the RL Model to PERTiMO

In this section we consider extending our RL model of TiMO by incorporating access permissions. We show that the resulting RL model correctly captures the semantics of PERTiMO and thus provides an important basis for mechanising reasoning about PERTiMO models.

Given a PERTiMO specification TMP we consider how to develop a corresponding RL model $RL(TMP)$ that correctly captures the meaning of TMP . The idea is to extend our RL model of the TiMO aspects of TMP by defining an appropriate RL specification for sets of access permissions. This involves introducing a range of new sorts and operations; the sort $APType$ is introduced with associated constants $\text{put} : APType$ and $\text{get} : APType$ to represent the two permission types. Next the sort AP is added to represent access permissions. This sort has a constructor

$$AP(@, @, @) : APType \text{ Chan } ALoc \rightarrow AP.$$

So, for example, $AP(\text{put}, a, l)$ represent an access permission to output data on a channel a when in location l . Sets of access permissions are represented by sort $APSet$ which has constant $\odot : AP$ for the empty set, the embedding operation $@ : AP \rightarrow APSet$ to lift access permissions to singleton sets, a set union operation $\cup : APSet \ APSet \rightarrow APSet$, and an operation $(@ \text{ in } @) : AP \ APSet \rightarrow \text{bool}$ for checking if an access permission is in a set or not (all of above are equationally axiomatised in the standard way [EhM85, MeT92]).

Given the above abstract data type we can then associate a set of access permissions with a process and

update the definition of networks so that they contain this new type of process. We introduce a new sort $PrsAP$ for processes with access permissions and add a constructor

$$(@ : @) : Prs\ APSet \rightarrow PrsAP$$

which pairs up processes and sets of access permissions. A (commutative, associative) parallel composition operator $@|@ : PrsAP\ PrsAP \rightarrow PrsAP$ is added for processes with access permissions. The structural rules allow the conditional bi-directional transfer between the normal process parallel composition operator and this new operator (see Table 3). We take the approach of prioritizing the new composition operator for processes with access permissions to simplify the model construction and so introduce the rule:

$$(p1 \mid p2 : aps) \Longrightarrow (p1 : aps) \mid (p2 : aps)$$

The stall S and time progression $tick$ functions can be lifted to processes with access permissions in the obvious way.

Network terms in the RL model are now extended so that they are based on processes with access permissions:

$$@[@] : ALoc\ PrsAP \rightarrow Nets$$

In order to model the $apmod$ function, used in PERTIMO to record the permission updates that occur when a process migrates, we introduce a function $apm(@, @)(@) : ALoc\ ALoc\ APSet \rightarrow APSet$. For a given PERTIMO example, appropriate rules need to be given to specify the behaviour of this function and to facilitate this we introduce functions

$$putAdd, putSub, getAdd, getSub : APSet\ Chan\ ALoc \rightarrow APSet$$

to model the basic access permission modification operators $put_{a@l}^+$, $get_{a@l}^+$, $put_{a@l}^-$, and $get_{a@l}^-$ found in PERTIMO. The following rules for $putAdd$ and $putSub$ illustrate how these operators are axiomatised:

$$\begin{aligned} putAdd(\Gamma, a, l) &\Rightarrow \Gamma \cup \{AP(a, l, put)\}, \\ putSub(\emptyset, a, l) &\Rightarrow \emptyset, \\ putSub(AP(a, l, put), a, l) &\Rightarrow \emptyset, \\ putSub(APVar, a, l) &\Rightarrow APVar \quad \text{if } notEq(APVar, AP(a, l, put)) \\ putSub(AP(a, l, put) \cup \Gamma, a, l) &\Rightarrow putSub(\Gamma, a, l), \\ putSub(APVar \cup \Gamma, a, l) &\Rightarrow APVar \cup putSub(\Gamma, a, l) \quad \text{if } notEq(APVar, AP(a, l, put)) \end{aligned}$$

where $notEq(ap1, ap2)$ is a Boolean function which checks whether two access permissions are different.

We now need to consider how to modify the RL model to take account of the changes made to the action rules in PERTIMO (see Table 3). By carefully considering Table 3 we observe that only the (MOVE) and (COM) rules have to be substantially revised. We modify the rule labelled $[move]$ which models the action rule (MOVE) in TIMO so that it updates the access permission set associated with the process which is migrating:

$$[moveP] \quad al[(go(t, al2) \text{ then } p1 : aps) \mid pp] \Longrightarrow al2[S(p1 : apm(al, al2)(aps))] \mid al[pp]$$

where aps is a variable of sort $APSet$ representing a set of access permissions and pp is a variable of sort $PrsAP$ representing a process associated with an access permission set. The remaining rule that allows time to pass requires only a slight modification (shown to illustrate the minor changes required to most rules):

$$[moveP] \quad al[(go(t, al2) \text{ then } p1 : aps) \mid pp] \Longrightarrow al[S(go(t-1, al2) \text{ then } p1 : aps) \mid pp] \quad \text{if } t > 0$$

The rule labelled $[calls]$ for modelling the action rule (CALLS) on process definitions needs a slight modification and becomes:

$$[callsP] \quad al[(id(u_1, \dots, u_n) : aps) \mid pp] \Rightarrow al[S(RL(P_{id}) : aps) \mid pp]$$

where $id(u_1, \dots, u_n : s_1, \dots, s_n) \stackrel{df}{=} P_{id}$ and $RL(P_{id})$ is the process term that results from translating P_{id} into $RL(TMP)$.

The final action rule that needs significantly extending is the communication action rule (COM). The existing RL rule is extended to the following conditional rule:

$$\begin{aligned}
[comP] \quad & al[(out(c, t1) < al1 > \text{ then } p1 \text{ else } p2 : aps1) \mid (in(c, t2)(vl) \text{ then } p3 \text{ else } p4 : aps2) \mid pp] \\
& \implies al[S(p1 : aps1) \mid S(p3[vl/al1] : aps2) \mid pp] \\
& \quad \text{if } (AP(put, c, al) \text{ in } aps1) \text{ and } (AP(get, c, al) \text{ in } aps2)
\end{aligned}$$

The above updated process transition rules can then be combined as before into a strategy $stepP$ that captures the maximal concurrent update step that occurs in a derivation step in PERTiMO:

$$stepP \implies repeat*(dc(callsP, moveP, comP))$$

We can then combine the above parts into a rewrite rule $oneStepP$ which extends the previous rule $oneStep$ for TiMO to take account of access permissions:

$$\begin{aligned}
[oneStepP] \quad & al[pp] \mid n1 \implies n3 \mid n1 \\
& \text{where } n2 := (stepP) \ al[pp], \\
& \quad n3 := () \ tick(n2)
\end{aligned}$$

The above allows PERTiMO network specifications to be mapped to a corresponding RL model which can then be simulated and analysed using a rewriting tool such as ELAN [BKKMR98, BKKM02]. A simple illustrative example of this is presented in Section 5.3 below. What now remains is to show formally that the proposed mapping into RL correctly preserves the semantics of PERTiMO.

Recall that for any TiMO model TM we defined $valNet(TM)$ to denote the set of valid RL network terms (see Definition 1). We can extend this definition to the RL model of PERTiMO as follows: the set $valNet(TMP)$ of *valid network terms* for a PERTiMO specification TMP is defined recursively by: (1) $at[pt : aps] \in valNet(TMP)$ if $pt \in valPrs(TMP)$ (where $valPrs(TMP)$ is defined as in Definition 1) and $aps \in T(\Sigma^{RL(TMP)})_{APSet}$; and (2) $net_1 \mid net_2 \in valNet(TMP)$, if $net_1, net_2 \in valNet(TMP)$.

It can be shown that $oneStepP$ preserves valid network terms.

Theorem 6 The strategy $oneStepP$ is well-defined with respect to valid network terms, i.e. for any $net_1 \in valNet(TMP)$, if $net_1 \implies net_2$ using $oneStepP$ then $net_2 \in valNet(TMP)$.

Proof. The proof follows along similar lines to Theorem 2 and makes use of the fact that PERTiMO acts only to restrict the communication actions possible in a derivation step. $\square$

The term interpretation mapping $\sigma_{Net} : Net(TM) \rightarrow valNet(TM)$ introduced in Definition 3 for TiMO can be extended to PERTiMO as follows.

Definition 7 The access permission term mapping $\sigma_{AP} : AccPerm \rightarrow T(\Sigma^{RL(TMP)})_{AP}$ is defined by:

$$\sigma_{AP}(put(a@l)) = AP(a, l, put), \quad \sigma_{AP}(get(a@l)) = AP(a, l, get)$$

In a slight abuse of notation, we also let $\sigma_{AP}(\{ap_1, \dots, ap_n\})$ denote the lifting of σ_{AP} to sets of access permissions, defined by

$$\sigma_{AP}(\emptyset) = \emptyset, \quad \sigma_{AP}(\{ap_1, \dots, ap_n\}) = \sigma_{AP}(ap_1) \cup \dots \cup \sigma_{AP}(ap_n)$$

The network term mapping $\sigma_{Net} : Net(TMP) \rightarrow valNet(TMP)$ can then be defined by

$$\sigma_{Net}(l \llbracket P : \Gamma \rrbracket) = l[\sigma_{Prs}(P) : \sigma_{AP}(\Gamma)], \quad \sigma_{Net}(N \mid N') = \sigma_{Net}(N) \mid \sigma_{Net}(N')$$

To establish the correctness of the RL model we need to prove it is *sound* and *complete* with respect to PERTiMO (see Figure 4 for a pictorial representation of these correctness properties).

Theorem 8 (Soundness) Let $net_1, net_2 \in valNet(TMP)$ be any valid network terms. Then if $net_1 \implies net_2$ using the strategy $oneStepP$ then $\sigma_{Net}^{-1}(net_1) \xRightarrow{\Psi} \sigma_{Net}^{-1}(net_2)$ for some finite multiset $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions and some location l . In other words, the diagram for soundness in Figure 4 commutes.

Proof. The proof follows a similar structure to the proof of Theorem 4 and relies on the fact that for a valid network location term of the form

$$at[(pt_1 : aps_1) \mid \dots \mid (pt_n : aps_n)] \in valNet(TMP),$$

Fig. 4. The soundness and completeness properties required for $RL(TMP)$ to be a correct model of a $Pertimo$ specification TMP .

each atomic process term $(pt_i : aps_i)$ is involved in at most one process transition rule application when $oneStepP$ is applied. This gives us four possible cases to consider for each $(pt_i : aps_i)$: 1) no process transition rule is applied and time simply progresses; 2) $[callsP]$ rule is applied; 3) $[moveP]$ rule is applied; and 4) $[comsP]$ rule is applied. Cases 1) and 2) are straightforward adaptations of the cases in the proof of Theorem 4 and so we focus on the remaining two cases here.

For Case 3), suppose pt_i has the form $go(nt, at')$ then npt and that a $[moveP]$ rule is applied to migrate the process to a new location resulting in the network term $at'[S(npt : apm(at, at')(aps_i))]$. Clearly, the stall symbol S will be removed by the application of the *tick* function. Then by the action rule (MOVEP) in Table 3 we must have

$$at \llbracket go^{\Delta nt} at' \text{ then } \sigma_{Prs}^{-1}(npt) : \sigma_{AP}^{-1}(aps_i) \rrbracket \xrightarrow{at \triangleright at'} at' \llbracket \textcircled{S} \sigma_{Prs}^{-1}(npt) : apmod(at, at')(\sigma_{AP}^{-1}(aps_i)) \rrbracket$$

Since we can show that

$$\sigma_{AP}^{-1}(apm(at, at_2)(aps_i)) = apmod(at, at')(\sigma_{AP}^{-1}(aps_i))$$

it follows that

$$at \llbracket go^{\Delta nt} at' \text{ then } \sigma_{Prs}^{-1}(npt) : \sigma_{AP}^{-1}(aps_i) \rrbracket \xrightarrow{at \triangleright at'} at' \llbracket \textcircled{S} \sigma_{Prs}^{-1}(npt) : \sigma_{AP}^{-1}(apm(at, at_2)(aps_i)) \rrbracket$$

The result then follows since the stall symbol $\textcircled{S}$ will be removed by the time progression step in $Pertimo$.

For Case 4), suppose that the $[comP]$ rule has been applied to two process terms pt_i and pt_j , for $i \neq j$. That is, suppose

$$(out(ct, nt_1) < at_2 > \text{ then } pt_i^1 \text{ else } pt_i^2 : aps_i) \mid (in(ct, nt_2)(vt) \text{ then } pt_j^1 \text{ else } pt_j^2 : aps_j) \\ \Rightarrow S(pt_i^1 : aps_i) \mid S(pt_j^1[vt/at_2] : aps_j)$$

The instances of the stall symbol S will be removed by the application of the *tick* function at the end of $oneStep$. Clearly, for the above conditional rule $[comP]$ to be applicable the condition $(AP(put, ct, at) \text{ in } aps_i)$ and $(AP(get, ct, at) \text{ in } aps_j)$ must have evaluated to *true*. Then it follows by the definition of the term mapping and $RL(TMP)$ that $put\langle ct @ at \rangle \in \sigma_{AP}^{-1}(aps_i)$ and $get\langle ct @ at \rangle \in \sigma_{AP}^{-1}(aps_j)$ must hold. By the action rule (COMP) in Table 3 we then have

$$at \llbracket ct^{\Delta nt_1} ! \langle at_2 \rangle \text{ then } \sigma_{Prs}^{-1}(pt_i^1) \text{ else } \sigma_{Prs}^{-1}(pt_i^2) : \sigma_{AP}^{-1}(aps_i) \mid \\ ct^{\Delta nt_2} ? (vt : Loc) \text{ then } \sigma_{Prs}^{-1}(pt_j^1) \text{ else } \sigma_{Prs}^{-1}(pt_j^2) : \sigma_{AP}^{-1}(aps_j) \rrbracket \\ \xrightarrow{ct < at_2 > @ at} at \llbracket \textcircled{S} \sigma_{Prs}^{-1}(pt_i^1) : \sigma_{AP}^{-1}(aps_i) \mid \textcircled{S} \{at_2/vt\} \sigma_{Prs}^{-1}(pt_j^1) : \sigma_{AP}^{-1}(aps_j) \rrbracket$$

The result then follows since the stall symbol $\textcircled{S}$ will be removed by the time progression step in $Pertimo$ and since we can show $\sigma_{Prs}^{-1}(pt_j^1[vt/at_2]) = \{at_2/vt\} \sigma_{Prs}^{-1}(pt_j^1)$. $\square$

Theorem 9 (Completeness) Let $N_1, N_2 \in Net(TMP)$ be any well-formed network terms in TMP . Then, if $N_1 \xRightarrow{\Psi} N_2$, for some location l and some multi-set $\Psi = \{\psi_1, \dots, \psi_m\}$ of l -actions, then $\sigma_{Net}(N_1) \Rightarrow \sigma_{Net}(N_2)$ using $oneStepP$. In other words, the diagram for completeness in Figure 4 commutes.

Proof. The proof follows a similar structure to the proof of Theorem 5 and relies on a case analysis of the possible action taken to update each atomic process in a derivation step. It suffices to consider a well-formed $Pertimo$ network of the form $at \llbracket P_1 : \Gamma_1 \mid \dots \mid P_n : \Gamma_n \rrbracket$ where $n > 0$ and each P_i is an atomic process. Now suppose

$$at \llbracket P_1 : \Gamma_1 \mid \dots \mid P_n : \Gamma_n \rrbracket \xRightarrow{\Psi} N',$$

for some finite set of *at*-actions $\Psi = \{\psi_1, \dots, \psi_m\}$, $m \geq 0$. Then it can be seen that each atomic process P_i is involved in at most one *at*-action ψ_i . We need to show that the *at*-action applied to each process P_i is correctly captured by the *oneStepP* strategy in $RL(TMP)$. There are four possible cases to consider for each process P_i : 1) None of the action rules (CALLP), (MOVEP) and (COMP) are applied in the derivation step; 2) The action rule (CALLP) is applied; 3) The action rule (MOVEP) is applied; and 4) The action rule (COMP) is applied. Cases 1) and 2) are straightforward adaptations of the cases in the proof of Theorem 5 and so we focus on the interesting aspects of Cases 3) and 4) here.

In Case 3), suppose P_i has the form $\text{go}^{\Delta nt} \text{ at}' \text{ then } P'$. Then we have

$$\sigma_{Prs}(\text{go}^{\Delta nt} \text{ at}' \text{ then } P') = \text{go}(nt, \text{at}') \text{ then } \sigma_{Prs}(P')$$

Suppose the action rule (MOVEP) was applied to P_i

$$\text{at} \llbracket \text{go}^{\Delta nt} \text{ at}' \text{ then } P' : \Gamma_i \rrbracket \xrightarrow{\text{at} \triangleright \text{at}'} \text{at}' \llbracket \textcircled{S} P' : \text{apmod}(\text{at}, \text{at}')(\Gamma_i) \rrbracket$$

where the stall symbol $\textcircled{S}$ is removed by the final time step. Then this can be copied in the RL model by applying the appropriate *[moveP]* rule

$$\text{at}[(\text{go}(nt, \text{at}') \text{ then } \sigma_{Prs}(P') : \sigma_{AP}(\Gamma_i))] \Rightarrow \text{at}'[S(\sigma_{Prs}(P') : \text{apm}(\text{at}, \text{at}')(\sigma_{AP}(\Gamma_i)))]$$

Clearly, the stall symbol S will be removed by the application of the *tick* function. The result then follows since we can show $\sigma_{Net}(\text{at}' \llbracket P' : \text{apmod}(\text{at}, \text{at}')(\Gamma_i) \rrbracket) = \text{at}'[(\sigma_{Prs}(P') : \text{apm}(\text{at}, \text{at}')(\sigma_{AP}(\Gamma_i)))]$ using the definition of the term mapping and $RL(TMP)$.

In Case 4), suppose the action rule (COMP) has been applied to two processes P_i and P_j , for $i \neq j$, i.e.

$$\begin{aligned} & \text{at} \llbracket \text{ct}^{\Delta nt_1} ! \langle \text{at}' \rangle \text{ then } P_i^1 \text{ else } P_i^2 : \Gamma_i \mid \text{ct}^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2 : \Gamma_j \rrbracket \\ & \xrightarrow{\text{ct} < \text{at}' > @ \text{at}} \text{at} \llbracket \textcircled{S} P_i^1 : \Gamma_i \mid \textcircled{S} \{ \text{at}' / vt \} P_j^1 : \Gamma_j \rrbracket \end{aligned}$$

where the stall symbols $\textcircled{S}$ will be removed by the final time step. Now we have

$$\begin{aligned} & \sigma_{Net}(\text{at} \llbracket \text{ct}^{\Delta nt_1} ! \langle \text{at}' \rangle \text{ then } P_i^1 \text{ else } P_i^2 : \Gamma_i \mid \text{ct}^{\Delta nt_2} ? (vt : Loc) \text{ then } P_j^1 \text{ else } P_j^2 : \Gamma_j \rrbracket) \\ & = \text{at}[(\text{out}(\text{ct}, nt_1) < \text{at}' > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) : \sigma_{AP}(\Gamma_i)) \mid \\ & \quad (\text{in}(\text{ct}, nt_2)(vt) \text{ then } \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) : \sigma_{AP}(\Gamma_j))] \end{aligned}$$

by the definition of σ_{Net} . Clearly, we must have that $\text{put}(\text{ct}@at) \in \Gamma_i$ and $\text{get}(\text{ct}@at) \in \Gamma_j$ if the (COMP) action rule has been applied. It follows that $(AP(\text{put}, \text{ct}, \text{at}) \text{ in } \sigma_{AP}(\Gamma_i))$ and $(AP(\text{get}, \text{ct}, \text{at}) \text{ in } \sigma_{AP}(\Gamma_j))$ must also hold by definition of the term mapping and $RL(TMP)$. We can therefore apply the *[comP]* rule

$$\begin{aligned} & \text{at}[(\text{out}(\text{ct}, nt_1) < \text{at}' > \text{ then } \sigma_{Prs}(P_i^1) \text{ else } \sigma_{Prs}(P_i^2) : \sigma_{AP}(\Gamma_i)) \mid \\ & \quad (\text{in}(\text{ct}, nt_2)(vt) \text{ then } \sigma_{Prs}(P_j^1) \text{ else } \sigma_{Prs}(P_j^2) : \sigma_{AP}(\Gamma_j))] \\ & \Rightarrow \text{at}[S(\sigma_{Prs}(P_i^1) : \sigma_{AP}(\Gamma_i)) \mid S(\sigma_{Prs}(P_j^1)[vt/\text{at}'] : \sigma_{AP}(\Gamma_j))] \end{aligned}$$

where all occurrences of the stall symbol S will be removed by the *tick* function. It is then straightforward to see that

$$\sigma_{Net}(\text{at} \llbracket P_i^1 : \Gamma_i \mid \{ \text{at}' / vt \} P_j^1 : \Gamma_j \rrbracket) = \text{at}[(\sigma_{Prs}(P_i^1) : \sigma_{AP}(\Gamma_i)) \mid (\sigma_{Prs}(P_j^1)[vt/\text{at}'] : \sigma_{AP}(\Gamma_j))]$$

by definition of σ_{Net} and since we can show $\sigma_{Prs}(\{ \text{at}' / vt \} P_j^1) = \sigma_{Prs}(P_j^1)[vt/\text{at}']$. $\square$

5.3. An Illustrative Example for PERTiMO

In this section we illustrate the RL model we have developed for PERTiMO by using ELAN to simulate and analyse a simple network system with access permissions. Recall the simple PERTiMO specification *CheckAP* introduced at the beginning of Section 5. Then we can apply the approach described in Section 5.2 to map *CheckAP* into an RL model $RL(\text{CheckAP})$. To illustrate $RL(\text{CheckAP})$ and help clarify the definition of the *[callsP]* rewrite rules, consider the following two rewrite rules that capture the (CALLS) action rule for the two process definitions in *CheckAP*:

$$\begin{aligned}
& [callsP] \quad al[(req : aps) \mid pp] \Rightarrow \\
& \quad al[S(go(1, Check) \text{ then } in(a, 1)(SL) \text{ then } go(1, SL) \text{ then } stop \text{ else } stop : aps) \mid pp] \\
& [callsP] \quad al[(chk : aps) \mid pp] \Rightarrow al[S(out(a, 2) < Area > \text{ then } chk \text{ else } chk : aps) \mid pp]
\end{aligned}$$

We can now use the support tool ELAN to execute $RL(CheckAP)$ and to investigate the behaviour of various network systems based on the definitions in $CheckAP$. As an example, consider the following network:

$$Safe \llbracket req : \odot \rrbracket \mid Check \llbracket chk : \{put(a@Check)\} \rrbracket$$

After translating this into $RL(CheckAP)$ we can use ELAN to derive the following rewrite trace which shows how a process from location *Safe* is able to gain appropriate access permission to reach a secure location *Area*.

$$\begin{aligned}
& Safe[(req : \odot)] \mid Check[(chk : AP(put, a, Check))] \\
& \Rightarrow \\
& Safe[(go(1, Check) \text{ then } in(a, 1)(SL) \text{ then } go(1, SL) \text{ then } stop \text{ else } stop : \odot)] \mid \\
& \quad Check[(chk : AP(put, a, Check))] \\
& \Rightarrow \\
& Safe[(go(1, Check) \text{ then } in(a, 1)(SL) \text{ then } go(1, SL) \text{ then } stop \text{ else } stop : \odot)] \mid \\
& \quad Check[(out(a, 2) < Area > \text{ then } chk \text{ else } chk : AP(put, a, Check))] \\
& \Rightarrow \\
& Check[(out(a, 2) < Area > \text{ then } chk \text{ else } chk : AP(put, a, Check)) \mid \\
& \quad (in(a, 1)(SL) \text{ then } go(1, SL) \text{ then } stop \text{ else } stop : AP(get, a, Check))] \\
& \Rightarrow \\
& Check[(chk : AP(put, a, Check)) \mid (go(1, Area) \text{ then } stop : AP(get, a, Check))] \\
& \Rightarrow \\
& Check[(out(a, 2) < Area > \text{ then } chk \text{ else } chk : AP(put, a, Check))] \mid Area[(stop : \odot)]
\end{aligned}$$

The trace contains five derivation steps and this can be shown to be the minimum needed in order to allow a process *req* to migrate from location *Safe* to the secure location *Area*.

Next we consider whether a process starting from another initial location (represented by *Other* here) can gain access to *Area* using the following network:

$$Other \llbracket req : \{get(a@Check)\} \rrbracket \mid Check \llbracket chk : \{put(a@Check)\} \rrbracket$$

By applying a simple search strategy using ELAN to the translated network term we are able to confirm that it is not possible for the process starting from *Other* to reach *Area* even when initially given the required access permission. These results are in-line with those derived using the judgement rules in Section 5.1.

6. Conclusions

TiMo [CiK08] is an appealing process algebra proposed for prototyping software engineering applications where time and mobility are combined. PERTiMo [CiK11b, CiK14] extended TiMo with interprocess communication controlled by access permissions that processes must possess in order to send and receive information. Moreover, processes can acquire new access permissions, or lose some of their current access permissions while moving between locations, modelling an important security feature. Crucially, it is possible to verify that a migrating process possesses a sufficiently rich set of initial access permissions so that whenever it attempts to communicate over a channel at a certain location, it has the required access permission, irrespective of the other processes used in the overall system. Both TiMo and PERTiMo use local clocks and local maximal concurrency of actions, and such choice of semantics is motivated by the assumptions made by the GALS approach [Das06], which specifies that systems operate under a ‘globally asynchronous/locally synchronous’ execution strategy. Related models can be found in the literature, such as the timed π -calculus [Ber04], timed distributed π -calculus [CiP06], and timed mobile ambients [AmC07]. The distributed π -calculus also incorporates an explicit notion of location, and dealing with static resources access [Hen07] by using a type system. Several systems encompass various forms of access control policies

in distributed systems. Other related work on access control in distributed systems has been done in the context of the language KLAIM and its extensions [Bet03, Bet05]. Another example is [BuG07] which uses cryptographic operations and capability types to obtain a secure implementation of a typed π -calculus.

Rewriting Logic (RL) [Mes92] provides a well-supported logical framework for modelling concurrent systems and has been used to model a range of process algebra languages. In [MaM96, MaM02] a model of CCS is developed using RL, though this model is not fully executable by tools such as MAUDE [CDEL02]. This work is extended in [VeM05] by using the reflective properties of MAUDE to develop a fully executable model of CCS. This interesting paper also provides a comprehensive model for the the LOTOS [Lot89] specification language. A high-level discussion of the use of ELAN for prototyping π -calculus specifications is provided in [Vir96] but while the use of strategies is mentioned no specific details are provided. The RL model of TiMO presented here appears to be novel in its use of strategies to cope with maximal parallel computational steps.

In this paper we used RL to develop a model and implementation of both TiMO and PERTiMO. The RL model was based on developing a strategy which can capture a maximal parallel computational step of a TiMO specification, including its time rule based previously on negative premises. We have then provided an operational semantics for PERTiMO by modifying our RL semantics of TiMO, and investigated the safety of communication and migration in terms of access permissions. This illustrates the significant flexibility and extendibility provided by using the RL framework with strategies. We have also formally shown the correctness of the resulting semantics by proving it is both sound and complete. We illustrated how the ELAN tool and, in particular, its user defined strategies can be used to model and analyse TiMO and PERTiMO specifications. While the examples used are intentionally simple for brevity, they still provides an interesting first insight into the range of properties that can be investigated.

The development of analytical and verification techniques continues to be our main concern at the present moment — especially in order to deal with static timing analysis — initial results in this direction have been reported in [CKS12]. Developing efficient support tools for PERTiMO is part of the work, and we see this paper as key step forward.

In future work we intend to investigate extending our approach to handle other security related aspects of software engineering designs, such as allowing access permissions (at particular locations) to control process migration, adding security levels for migrating processes, allowing messages to contain access permissions, and analysing security policies for access and migration control. Interestingly, the RL model allows a range of semantic choices for PERTiMO to be considered by changing the derivation step strategy (e.g. adding priorities or fairness assumptions) and we are currently investigating these different semantic choices. We also intend to perform a variety of verification case studies to illustrate the practical application of our methods and investigate its limitations. Finally, we note that at present the analysis of TiMO and PERTiMO specifications is limited by the search capabilities and efficiency of ELAN. Work is now underway to develop MAUDE [CDEL02] implementations of the RL model presented here with the aim of improving both the range and efficiency of model analysis.

Acknowledgements We would like to thank the reviewers for several constructive comments and useful suggestions. The work of the first author was supported by a grant from the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919. We also gratefully acknowledge the financial support of the School of Computing Science, Newcastle University.

References

- [But81] Butcher, J.: *Copy-editing: The Cambridge Handbook*. Cambridge University Press, 1981.
- [PeS81] Peterson, G. E. and Stickel, M. E.: Complete Sets of Reductions for Some Equational Theories. *J. ACM*, **28**, 223–264 (1981).
- [JLR82] Jouannaud, J. P., Lescanne, P. and Reinig, F.: Recursive Decomposition Ordering, *Proc. Conf. on Formal Description of Programming Concepts II*, pp. 331–346, 1982.
- [AmC07] B.Aman and G.Ciobanu: Mobile Ambients with Timers and Types. *Proc. of ICTAC 2007*, Springer, LNCS 4711 (2007) 50–63.
- [BBKMR07] E.Balland, P.Brauner, R.Kopetz, P.-E.Moreau, and A.Reilles: Tom: Piggybacking rewriting on java. *Proc. of RTA'07*, Springer, LNCS 4533 (2007) 36–47.
- [Ber04] M.Berger: Basic Theory of Reduction Congruence for Two Timed Asynchronous Pi-Calculi. *Proc. of CONCUR'04*, Springer, LNCS 3170 (2004) 115–130.

- [Bet03] Bettini, L. et al. (2003) The KLAIM Project: Theory and Practice. Proceedings of GC'03, Rovereto, Italy, 9-14 February, pp. 88–150. Springer-Verlag, Berlin.
- [Bet05] Bettini, L. and De Nicola, R. (2005) Mobile distributed programming in X-Klaim. Proceedings of SFM-Moby'05, Bertinoro, Italy, 26-30 April, pp. 29–68. Springer-Verlag, Berlin.
- [BKMMR98] P.Borovanský, C.Kirchner, H.Kirchner, P.-E. Moreau and C.Ringeissen: An overview of ELAN. In: C. Kirchner and H. Kirchner (eds), Proc. of *WRLA '98, Electronic Notes in Theoretical Computer Science* 15 (1998).
- [BKRR01] P.Borovanský, C.Kirchner, H.Kirchner, and C.Ringeissen: Rewriting with Strategies in ELAN: A Functional Semantics. *International Journal of Foundations of Computer Science* 12(1) (2001) 69–95.
- [BKMM02] P.Borovanský, C.Kirchner, H.Kirchner, and P.-E.Moreau: Elan from a rewriting logic point of view. *Theoretical Computer Science* 285(2) (2002) 155–185.
- [BuG07] Bugliesi, M. and Giunti, M. (2007) Secure implementations of typed channel abstractions. Proceedings of POPL'07, Nice, France, 17-19 January, pp. 251–262. ACM, New York, NY.
- [CaG00] L.Cardelli and A.Gordon: Mobile Ambients. *Theoretical Computer Science* 240 (2000) 170–213.
- [CiK08] G.Ciobanu and M.Koutny: Modelling and Verification of Timed Interaction and Migration. Proc. of *FASE'08*, Springer, LNCS 4961 (2008) 215–229.
- [CiK11a] G.Ciobanu and M.Koutny: Timed Mobility in Process Algebra and Petri Nets. *Journal of Algebraic and Logic Programming* 80(7) (2011) 377–391.
- [CiK11b] G.Ciobanu and M.Koutny: Timed Migration and Interaction with Access Permissions. Proc. of *FM'11*, Springer, LNCS 6664 (2011) 293–307.
- [CiK14] G.Ciobanu and M.Koutny: PerTiMo: A Model of Spatial Migration with Safe Access Permissions. *The Computer Journal* (online) (2014) doi:10.1093/comjnl/bxu006.
- [CKS12] G.Ciobanu, M.Koutny, and L.J.Steggles: A Timed Mobility Semantics Based on Rewriting Strategies. In: G.Eleftherakis, M.Hinchey, and M.Holcombe (eds), Proc. of *SEFM'12*, Springer, LNCS 7504 (2012) 141–155.
- [CiP06] G.Ciobanu and C.Prisacariu: Timers for Distributed Systems. *Electronic Notes in Theoretical Computer Science* 164 (2006) 81–99.
- [CDEL02] M.Clavel, et al: Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science* 285(2) (2002) 187–243.
- [Cor00] F.Corradini: Absolute Versus Relative Time in Process Algebras. *Information and Computation* 156 (2000) 122–172.
- [Das06] S.Dasgupta, D.Potop-Butucaru, B.Caillaud and A.Yakovlev: Moving from weakly endochronous systems to delay-insensitive circuits. *Electronic Notes in Theoretical Computer Science* 146 (2006) 81–103.
- [EhM85] H. Ehrig and B. Mahr: *Fundamentals of Algebraic Specification 1 – Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science 6 (Springer-Verlag, Berlin). 1985
- [Hen07] M.Hennessy: *A Distributed π -Calculus*. Cambridge University Press (2007).
- [Lot89] ISO/IEC: *LOTOS-A formal description technique based on the temporal ordering of observational behaviour*. International Standard 8807, International Organization for standardization–Information Processing Systems–Open Systems Interconnection, Geneva (1989).
- [MaM96] N.Martí-Oliet and J.Meseguer: Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science* 4 (1996) 190–225.
- [MaM02] N.Martí-Oliet and J.Meseguer: Rewriting logic as a logical and semantic framework. In: D.M.Gabbay and F.Guenther (eds), *Handbook of Philosophical Logic, Second Edition, Vol. 9*, Kluwer Academic Publishers (2002) 1–87.
- [MeT92] K.Meinke and J.V.Tucker: Universal Algebra. In: S.Abramsky, D.Gabbay and T.Maibaum (eds), *Handbook of Logic in Computer Science, Volume I: Mathematical Structures*, Oxford University (1992) 189–411.
- [Mes92] J.Meseguer: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(2) (1992) 73–155.
- [Mil99] R.Milner: *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press (1999).
- [VeM05] A. Verdejo and N.Martí-Oliet: Two Case Studies of Semantics Execution in Maude: CCS and LOTOS. *Formal Methods in System Design* 27(1–2) (2005) 113–172.
- [Vir96] P.Viry: A rewriting implementation of pi-calculus. Technical Report TR-96-30, Dipartimento di Informatica, Università di Pisa, 26 (1996).
- [Vis01] E.Visser: Stratego: A language for program transformation based on rewriting strategies. Proc. of *RTA'04*, Springer, LNCS 2051 (2001) 357–361.