Developing and Applying a Rewriting Framework for Timed Mobility

Gabriel Ciobanu[1] and Jason Steggles[2]

[1] Romanian Academy, Institute of Computer Science, Iaşi, Romania, `gabriel@info.uaic.ro`
[2] School of Computing Science, University of Newcastle, UK, `jason.steggles@ncl.ac.uk`

Abstract. TiMo (Timed Mobility) is a process algebra developed for modelling mobile based systems with timing constraints. We present a new semantic model for Timed Mobility (TiMo) by mapping its operational semantics rules to corresponding rewrite rules in Maude. We use the meta-programming capabilities of Maude to formulate a rewriting strategy that captures the maximal parallel computational steps of TiMo . We formally prove the correctness of the translation of TiMo into Maude by showing the soundness and completeness with respect to the operational semantics of TiMo . To illustrate the new rewriting framework, we formulate a TiMo specification of a simple robot swarm example, and use Maude to simulate and analyse it.

Keywords: Timed Mobility, Maximal Concurrency, Rewriting Logic.

1 Introduction

In this paper we highlight one of the many contribution made by Maciej Koutny to the field of modelling and reasoning about concurrent systems, namely the development of TiMo (TImed MObility). TiMo was introduced in [9] as a process calculus able to describe migrating agents using specific features such as explicit locations, timeouts, and timed migration and communication in distributed systems. After this first step, the same authors presented a structural translation of TiMo into behaviourally equivalent high level timed Petri nets in [10]. As a result, it is obtained a formal net semantics for timed interaction and migration which is both structural and allows to deal directly with concurrency and causality. In [11], TiMo is extended with (dynamic) access permissions; a more detailed version is presented in [18]. Overall, the approach is motivated by the 'globally asynchronous/locally synchronous' execution strategy (as found in GALS approach [22]) in a semantic framework based on local maximal concurrency. Processes are viewed as residing within distinct locations, where each location has its own local clock. Processes are allowed to migrate between locations, and this is controlled by timers linked to the local clock of the location containing the process. Timers are also used to control communication between co-located processes.

The operational semantics of TiMo is provided by a transition system labelled with multisets of actions executed in parallel in one step. A complete computational step is captured by a parallel execution of actions followed by a time step. Based on such an operational semantics, it is proved that the passage of time does not introduce any nondeterminism into the execution of a process, and the processes evolve properly in time. The standard notion of bisimilarity is extended to TiMo in [4] to deal with timed transitions and multisets of actions.

In this paper we use *Rewriting Logic* (RL), an algebraic formal modelling approach based on using equations to define static states and rewrite rules to define dynamic state transitions [30]. In previous work [13], a new semantic model for TiMo was developed by using RL and strategies with the aim of providing a foundation for tool support. In particular, rewriting strategies are used to capture the locally maximal concurrent step of a TiMo specification and the RL approach was realised using the support tool Elan [7]. This approach was then extended in [16] with access permissions in order to develop a new semantic model for PerTiMo [18]. These semantical models are formally proved to be sound and complete with respect to the original operational semantics on which they were based. We build on this work here, and develop an updated RL semantic model for TiMo based on using Maude [20], an adaptable formal tool framework for modelling and analysing RL models. Maude provides a range of interesting analysis tools (such as an LTL model checker [24]) and importantly, it has powerful meta–programming capabilities [19] which allow rewriting strategies [25] to be developed to refine an RL model.

We develop a new semantic model for TiMo by mapping its operational semantics rule set to a corresponding set of rewrite rules in Maude. An interesting aspect of this is how to cope with the maximal concurrency captured by the the time progression rule which is based on using negative premises. We make use of Maude's meta–programming capabilities to formulate a *rewriting strategy* [25] that captures this maximal parallel computational step. We formally discuss the correctness of the resulting Maude specification by proving it is both *sound* and *complete* with respect to the original operational semantics of TiMo.

To illustrate the new Maude framework, we consider a case study based on a *Complex Adaptive System (CAS)* [26, 5]. We formulate a new TiMo model of a simple robot swarm example based on robots collaborating to pull up sticks [27, 28], and then use Maude to simulate and analyse this model. This simple example gives useful insight into the flexibility of the proposed RL modelling approach and illustrates the type of interesting analysis that can be done.

The paper is structured as follows. Section 2 describes the syntax and semantics of TiMo. Section 3 briefly introduces RL and the Maude support tool. In Section 4, we develop an RL model of TiMo using Maude and its metaprogramming capabilities. In Section 5 we illustrate the framework we have developed using a robot swarm example based on collaborative stick pulling. Finally, in Section 6 we make some concluding remarks.

2 The Development of TiMo

TiMo (Timed Mobility) [9–11] is a process algebra for mobile based systems where it is possible to add timers to communication and mobility actions. Processes reside within distinct locations and each location runs according to its own local clock. Processes are allowed to migrate between locations and communication can occur between co–located processes; these actions are controlled by timers linked to the local clock of the location the process resides in.

To formalise TiMo we begin by defining 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 $\boldsymbol{x}$ to denote a finite tuple of elements $(x_1, \ldots, x_k)$ 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 P_{id} is a process expression, the u_i's are distinct variables playing the role of parameters, and the X_i^{id}'s are data types. 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) $\circledS$ is a special symbol used to state that a process is temporarily 'stalled'.

Processes	$P ::= a^{\Delta t}\,!\,\langle \boldsymbol{v} \rangle$ `then` P `else` P' $\mid$	(*output*)
	$a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' $\mid$	(*input*)
	`go`$^{\Delta t}$ l `then` P $\mid$	(*move*)
	$P \mid P'$ $\mid$	(*parallel*)
	$id(\boldsymbol{v})$ $\mid$	(*recursion*)
	stop $\mid$	(*termination*)
	$\circledS P$	(*stalling*)
Networks	$N ::= l\,[\![P]\!]$ $\mid$ $N \mid N'$	
Definition	$id(u_1, \ldots, u_{m_{id}} : X_1^{id}, \ldots, X_{m_{id}}^{id}) \overset{\text{df}}{=} P_{id}$	(Def)

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

The only variable binding construct is $a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' which binds the variables $\boldsymbol{u}$ within P (but *not* within P'). We use $fv(P)$ to denote the free variables of a process P (and similarly for networks). For a process definition as in (Def), we assume that $fv(P_{id}) \subseteq \{u_1, \ldots, u_{m_{id}}\}$, and so the free variables of P_{id} are parameter bound. Processes are defined up to the alpha-conversion, and $\{v/u, \ldots\}P$ 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 $\boldsymbol{v}$ and $\boldsymbol{u}$ are tuples of the same length then $\{\boldsymbol{v}/\boldsymbol{u}\}P$ denotes $\{v_1/u_1, v_2/u_2, \ldots, v_k/u_k\}P$.

A process $a^{\Delta t}\,!\,\langle \boldsymbol{v} \rangle$ `then` P `else` P' attempts to send a tuple of values $\boldsymbol{v}$ over the channel a for t time units. If successful, it continues as process P; otherwise

(Eq1-3) $N \mid N' \equiv N' \mid N \quad (N \mid N') \mid N'' \equiv N \mid (N' \mid N'') \quad l[\![\, P \mid P' \,]\!] \equiv l[\![\, P \,]\!] \mid l[\![\, P' \,]\!]$

(Call) $l[\![\, id(\boldsymbol{v}) \,]\!] \xrightarrow{id@l} l[\![\, Ⓢ\{\boldsymbol{v}/\boldsymbol{u}\}P_{id} \,]\!]$ (Move) $l[\![\, \mathtt{go}^{\Delta t}\ l'\ \mathtt{then}\ P \,]\!] \xrightarrow{l'@l} l'[\![\, Ⓢ P \,]\!]$

(Com) $$\frac{v_1 \in X_1\ \ldots\ v_k \in X_k}{l[\![\, a^{\Delta t}\,!\,\langle \boldsymbol{v} \rangle\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid a^{\Delta t'}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})\ \mathtt{then}\ P'\ \mathtt{else}\ Q' \,]\!] \xrightarrow{a\langle \boldsymbol{v} \rangle @l} l[\![\, Ⓢ P \mid Ⓢ\{\boldsymbol{v}/\boldsymbol{u}\}P' \,]\!]}$$

(Par) $$\frac{N \xrightarrow{\psi} N'}{N \mid N'' \xrightarrow{\psi} N' \mid N''}$$ (Time) $$\frac{N \not\longrightarrow_l}{N \xrightarrow{\surd_l} \phi_l(N)}$$

(Equiv) $$\frac{N \equiv N' \qquad N' \xrightarrow{\psi} N'' \qquad N'' \equiv N'''}{N \xrightarrow{\psi} N'''}$$

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.

it continues as the alternative process P'. A process $a^{\Delta t}\,?\,(\boldsymbol{u}{:}\boldsymbol{X})$ `then` P `else` P' attempts for t time units to input a tuple of values of type $\boldsymbol{X}$ and substitute them for the variables $\boldsymbol{u}$. Mobility is implemented by a process $\mathtt{go}^{\Delta t}\ l$ `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 Ⓢ P are a purely technical device which is used in the subsequent formalisation of structural operational semantics of TiMo; intuitively, Ⓢ 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 $N \mid N'$.

A network N is *well-formed* if: (i) there are no free variables in N; (ii) there are no occurrences of the special symbol Ⓢ in N; (iii) assuming that id is as in the recursive equation (Def), for every $id(\boldsymbol{v})$ 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 $Prs(TM)$ and $Net(TM)$ represent the set of well-formed TiMo process and network terms respectively. The first component of the operational semantics of TiMo 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 $l_1[\![\, P_1 \,]\!] \mid \ldots \mid l_n[\![\, P_n \,]\!]$ such that no process P_i has the parallel composition operator at its topmost level. Each subnetwork $l_i[\![\, P_i \,]\!]$ 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 $N \xrightarrow{\surd_l} N'$. 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 $\Delta t = \Delta 0$ 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 $l[\![\,\mathtt{go}^{\Delta t}\ l'\ \mathtt{then}\ P\,]\!]$ by $l[\![\,\mathtt{go}^{\Delta t-1}\ l'\ \mathtt{then}\ P\,]\!]$, and all components of the form $l[\![\,a^{\Delta t}\omega\ \mathtt{then}\ P\ \mathtt{else}\ Q\,]\!]$ (where ω stands for $\mathbf{!}\langle \boldsymbol{v}\rangle$ or $\mathbf{?}(\boldsymbol{u}{:}\boldsymbol{X})$) by $l[\![\,Q\,]\!]$ if $t = 0$, and $l[\![\,a^{\Delta t-1}\omega\ \mathtt{then}\ P\ \mathtt{else}\ Q\,]\!]$ otherwise. After that, all the occurrences of the symbol Ⓢ in N are erased.

The above defines executions of individual actions. A complete computational step is captured by a *derivation* of the form $N \overset{\Psi}{\Longrightarrow} N'$, where $\Psi = \{\psi_1, \ldots, \psi_m\}$ ($m \geq 0$) is a finite multiset of l-actions for some location l (i.e., actions of the form $id@l$ or $l'@l$ or $a\langle \boldsymbol{v}\rangle@l$) such that $N \xrightarrow{\psi_1} N_1 \cdots N_{m-1} \xrightarrow{\psi_m} N_m \xrightarrow{\surd_l} N'$. 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. Intuitively, we capture the cumulative effect of the concurrent execution of the multiset of actions Ψ at location l. We say that N' is *directly reachable* from N. Note that whenever there is only a time progression at a location, we have $N \overset{\varnothing}{\Longrightarrow} N'$.

One can show that derivations are well defined as one cannot execute an unbounded sequence of action moves without time progress, and the execution Ψ is made up of independent (or concurrent) individual executions. Moreover, derivations preserve well-formedness of networks (see [9]).

3 Rewriting Logic and Maude

Rewriting logic (RL) [30] is a formal modelling and analysis framework based on an algebraic specification approach. In order to model a dynamic system in RL there are two stages. First the static states of the system are specified using standard equational specification techniques. Secondly, rewrite rules are used to specify the non–deterministic state transitions that represent the dynamic behaviour of the system. The application of rewrite rules can be controlled using *rewriting strategies* [25] and the result is an expressive and versatile formal framework. RL has been applied to model a wide range of different formalisms and systems, including: biological systems [23, 31], Petri nets [34, 32], and process algebras [29, 16].

A range of different tools have been developed to support RL (see [20, 7, 6]). In this paper, we use Maude [20], an advanced support tool for RL that provides a range of interesting analysis tools (such as an LTL model checker

[24]) and meta–programming capabilities. MAUDE has been widely used to create executable environments for different languages and models of computation [21].

In order to illustrate using MAUDE consider the following simple RL specification written in MAUDE:

```
mod EXABCD is
  sorts Entity State .
  subsort Entity < State .

  ops A B C D : -> Entity .
  op __ : State State -> State [assoc comm].

  rl [rule1] : A A => B .
  rl [rule2] : A B => C .
  rl [rule3] : C => A A A .
  rl [rule4] : C B => D A .
endm
```

The system contains four entities `A`, `B`, `C`, and `D` which are declared as constants of sort `Entity`. The states of the system are represented as multi–sets of entities and these are modelled by introducing the sort `State`. The sort `Entity` is declared as a subsort of `State` which means that an entity can be viewed as a singleton state. We use an implicit multi-set union operator `__ : State State -> State` (where `_` is denotes the location of an infix argument) and define this to be associative and commutative by adding the flags `[assoc comm]` (this corresponds to adding the appropriate equations for these properties). The dynamic transitions allowed in the system are then defined using the four rewrite rules given in the specification. As an example, consider the following rewrite trace derived from the initial state `C C`:

```
C C => A A A C => B A C => D A A => D B
```

A range of analysis tools are provided by MAUDE, such as the built–in model checking command `search S =>+ P`, which allows us to check if a pattern term `P` can be reached by rewriting an initial ground term `S`. For example, we can use the `search` command to check if we can derive a state containing `D D` from an initial state `C C C`:

```
search C C C =>+ D D s:State .
```

This search returns true (with `s` instantiated with `A`) and we can view a corresponding witness rewrite trace.

One key motivation for using MAUDE is the meta–programming capabilities it offers. This meta–programming allows the definition of rewriting strategies which can be used to control the way in which the rewrite rules are applied. To illustrate this, consider the following meta–programming example which defines a rewrite strategy that prioritises `rule3` over the other rules.

```
ceq rwStrat(T) = if Step? :: Result4Tuple then getTerm(Step?)
    else (if Step2? :: ResultPair then getTerm(Step2?) else T  fi)
    fi
    if Step? := metaXapply(upModule('EXABCD,false),T,'rule3,
                 none,0,unbounded,0)
       /\  Step2? := metaRewrite(upModule('EXABCD,false),T,1) .
```

The rewriting strategy `rwStrat` is defined using a conditional equation which allows local definitions (based on `:=`) to be used to combine the various parts of the strategy. It makes use of two built–in meta–level functions: `metaXapply` which allows a given rule to be applied (in this case `rule1`) to a term `T` ; and `metaRewrite` which allows a term `T` to be rewritten. Type checking is used to ensure that the meta–level functions have been successfully applied, e.g. `Step? :: Result4Tuple` is used to check if `rule1` has been successfully applied. For full details of the notation used here and further rewriting strategy examples see the Maude manual [21].

4 Translating TiMo into Maude

In this section we consider how to translate a TiMo specification *TM* into a semantically equivalent Maude model $Md(TM)$. We begin by defining the sorts required in $Md(TM)$ to model the key TiMo concepts of channels, locations, processes and networks as follows:

```
sorts Chan VLoc ALoc Loc Prs Nets .
subsorts VLoc ALoc < Loc .
```

Note that in order to model the location parameter passing that can occur in communication we defined the sort `Loc` for locations to consist of two subsorts: `VLoc` which represents locations variables; and `ALoc` representing actual location names.

Next we define the function symbols needed in $Md(TM)$ to represent processes and networks.

```
op stop : -> Prs .
op S : Prs -> Prs .
op _|_ : Prs Prs -> Prs [assoc comm] .
op go : Time Loc Prs -> Prs .
op in : Chan Time VLoc Prs Prs -> Prs .
op out : Chan Time Loc Prs Prs -> Prs .
op _[_] : ALoc Prs -> Nets .
op _|_ : Nets Nets -> Nets [assoc comm] .
```

Note that the function symbol `_|_` is overloaded and is used to represent parallel composition of both processes and networks in TiMo. Both instances are defined (equationally) to be associative and commutative using the operator flags `[assoc comm]`. The function symbol `S` is used to represent the special stall symbol Ⓢ used in TiMo to control time progression.

To model process definitions we need to add a function symbol

```
op id : s1 ... sn -> Prs
```

for each process identifier $id(u_1, \ldots, u_n : s_1, \ldots, s_n)$ in our TiMo specification *TM*, where `si` is assumed to be a well–defined algebraic data type in $Md(TM)$ representing s_i.

To capture the dynamic semantics of TiMo processes and networks, as defined by the action rules in Table 2, we need to formulate appropriate rewrite rules to define the behaviour of the Maude model. In order to simplify working with network terms we make the simplifying assumption that network components with the same location are always merged into a single network structure (this assumption is clearly valid given Eq 3 from Table 2). We enforce this by adding the following equation to $Md(TM)$:

```
eq (AL[P1]) | (AL[P2]) = AL[P1 | P2] .
```

Each individual network location term `l[P1 | ... | Pn]` will therefore consist of a number of atomic processes `Pi` (where a process term is referred to as *atomic* if it does not have the parallel operator at its topmost level).

When considering mobility we have a non–deterministic choice between executing a `go` command or allowing time to pass. We incorporate this behaviour into $Md(TM)$ by adding the following pair of rules:

```
rl [move] : (AL[go(T,AL1,P1) | P2] | N) =>
              (AL[P2] | AL1[S(P1)] | N) .
rl [move] : (AL[go(T,AL1,P1) | P2] | N) =>
              (AL[S(go(T-1),AL1,P1)) | P2] | N)  if notExp(T) .
```

The idea is that while `T` > 0 (i.e. `notExp(T)` equates to true) either rule can be applied allowing the process to either wait or to execute the move. However, as soon as the timer `T` has expired (i.e. `T` $= 0$ and so `notExp(T)` equates to false) then only the first rule that moves `P1` to the new location `AL1` can be used.

Communication in TiMo is based on output and input processes synchronising on a common channel within a location. We model this synchronisation by using the following rule:

```
rl [com] : (AL[out(C,T1,AL1,P1,P2) | in(C,T2,VL,P3,P4)]) | N) =>
                (AL[S(P1) | S(sub(P3,VL,AL1))] | N) .
```

This rule makes use of a substitution function `sub : Prs VLoc ALoc -> Prs`, where the term `sub(P,VL,AL)` represents the process term resulting from substituting all free occurrences (not bound by an input action symbol) of the location variable `VL` in the process term `P` by the actual location term `AL`. It is straightforward to define `sub` equationally using recursion over process terms.

For each process definition

$$id(u_1, \ldots, u_n : s_1, \ldots, s_n) \overset{\mathrm{df}}{=} P_{id}$$

we add the following corresponding rule to $Md(TM)$:

```
rl [call] : (AL[id(u1,...,un) | P] | N) => (AL[S(Pid) | P] | N) .
```

where `Pid` is the process term that results from translating P_{id} into $Md(TM)$ and each `ui` is a variable of sort `si` in $Md(TM)$.

In the sequel we refer to rules `move`, `com` and `call` as *process transition rules*.

Every derivation step in TIMO finishes with the application of the (TIME) action rule (see Table 2). This allows time to progress by updating timers and removing all instances of the stall symbol. To model this in $Md(TM)$ we introduce a function `tick : Prs -> Prs` which we define equational using recursion over process terms. The following equations illustrate the approach taken:

```
eq tick(S(P)) = P .
eq tick(in(C,0,VL,P1,P2)) = P2 .
ceq tick(in(C,T,VL,P1,P2)) = in(C,T-1,VL,P1,P2) if notExp(T) .
eq tick(P1 | P2) = tick(P1) | tick(P2) .
```

Note that the `go` command does not feature explicitly in these equations since its timer is handled within the `move` process rules above. We then overload `tick` so that it can be applied to network terms in the obvious way.

We now have to compose the above components to correctly model within $Md(TM)$ a derivation step. This is done by defining a *rewriting strategy* using Maude's metalevel programming capabilities [20, 25]. First, we define a metalevel operation `update : Term -> Term` which allows processes in the chosen location to perform an action if allowable.

```
ceq update(T) =
        if Step? :: ResultPair then getTerm(Step?) else T  fi
    if Step? := metaRewrite(upModule('Md(TM), false), T, unbounded) .
```

We then build on this by defining a metalevel operation `next : Term -> Term` which applies `update` to a term and then allows time to progress by applying the `tick` function.

```
ceq next(T) =
        if Step? :: ResultPair then getTerm(Step?) else T1 fi
    if T1 := update(T) /\
       Step? := metaReduce(upModule('Md(TM), false), 'tick[T1]) .
```

Note that in the above we use the metalevel representation of `tick` as indicated by the backquote and that `metaReduce` is used to apply the defining equations in $Md(TM)$.

Finally, we introduce a conditional rule `step` which allows the rewriting strategy `next` to be automatically applied within MAUDE.

```
 crl [step] : AL[P] => downTerm(T1,NTerm)
                if T1 := nextState(upTerm(AL[P])) .
```

This conditional rule avoids the need to directly use MAUDE's metalevel notation by applying the built–in functions `upterm` and `downTerm` for moving between the module and metalevel representation of terms. Note the second parameter to `downTerm` is a term `NTerm` which sets the expected kind to be returned from the operation (see [21]).

We conclude this section by considering the important question of the correctness of the above semantic translation from TIMO to MAUDE. In order to formally show that the translation is correct we need to show: *soundness* – each step in our MAUDE model represents a derivation step in TIMO; and *completeness* – every derivation step possible in TIMO is represented in our MAUDE model. To formalise the above correctness properties we begin by defining a bijective mapping $\sigma : \mathit{Net}(\mathit{TM}) \rightarrow \mathit{valTerm}(\mathit{Md}(\mathit{TM}))$ from the set $\mathit{Net}(\mathit{TM})$ of TIMO network terms in TM to their corresponding terms in $\mathit{Md}(\mathit{TM})$ (this is straightforward to do using the natural translation given by the syntax). Note that not all terms in $\mathit{Md}(\mathit{TM})$ are well–defined since they may incorrectly contain the stall symbol or may contain unbounded location variables. We therefore let $\mathit{valTerm}(\mathit{Md}(\mathit{TM}))$ denote the set of all well–defined network terms in $\mathit{Md}(\mathit{TM})$.

The following result shows the `step` rule preserves $\mathit{valTerm}(\mathit{Md}(\mathit{TM}))$ terms.

Theorem 1 For any network term `net1` $\in \mathit{valTerm}(\mathit{Md}(\mathit{TM}))$, if `net1 => net2` by an application of the rule `step` then `net2` $\in \mathit{valTerm}(\mathit{Md}(\mathit{TM}))$.

Proof. Consider a network term `Loc[p1 | ... | pn]` $\in \mathit{valTerm}(\mathit{Md}(\mathit{TM}))$, where `n` > 0 and each process term `pi` is atomic. It can be shown that each process term `pi` can be updated by at most one of the process transition rules and this gives four cases to consider: 1) rule `move` was applied; 2) rule `com` was applied; 3) rule `call` was applied; or 4) no process transition rule was applied and time was simply allowed to progress. It can be shown that each of these cases results in a well-defined term that represents a corresponding TIMO process. (See [13] for an example of this type of proof.) □

We can now show that $\mathit{Md}(\mathit{TM})$ is a sound and complete model of TIMO specification TM (see Figure 1).

$$\begin{tikzcd}
N_1 \arrow[r, "\overset{\Psi}{\Longrightarrow}"] & N_2 \\
\texttt{net1} \arrow[u, "\sigma^{-1}"] \arrow[r, "\texttt{step}"'] \arrow[ur, phantom, "\text{Soundness}"] & \texttt{net2} \arrow[u, "\sigma^{-1}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{net1} \arrow[r, "\texttt{step}"] & \texttt{net2} \\
N_1 \arrow[u, "\sigma"] \arrow[r, "\overset{\Psi}{\Longrightarrow}"'] \arrow[ur, phantom, "\text{Completeness}"] & N_2 \arrow[u, "\sigma"']
\end{tikzcd}$$

Fig. 1. The properties of soundness and completeness required for $\mathit{Md}(\mathit{TM})$ to be a correct model of TM.

Theorem 2 (Soundness) Let `net1`, `net2` $\in valTerm(Md(TM))$ be valid network terms. Then if `net1 => net2` by an application of the rule `step` then $\sigma^{-1}(\texttt{net1}) \overset{\Psi}{\Longrightarrow} \sigma^{-1}(\texttt{net2})$ for some finite multiset $\Psi = \{\psi_1, \ldots, \psi_m\}$ of l-actions and some location l (i. e. the diagram for soundness in Figure 1 commutes).

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

$$\texttt{l[p1 | ... | pn]} \in valTerm(Md(TM)),$$

where $\texttt{n} > 0$ and each process term `pi` is atomic. It can be seen that each process term `pi` is involved in at most one process transition rule application when rule `step` is applied. This gives use four possible cases to consider: 1) rule `move` was applied; 2) rule `call` was applied; 3) rule `com` was applied; or 4) no process transition rule was applied and time was simply allowed to progress. For brevity, we consider only Case 1) in detail here (for a complete example of this type of proof see [13]).

Case 1) Suppose `pi` has the form `go(t,l2,p)` and that a `move` rule is applied. Then there are two possible cases to consider:
i) *Timer is reduced*: Suppose $\texttt{t} > 0$ and the `move` rule applied simply allowed time to progress

```
go(t,l2,p) => S(go(t-1,l2,p))
```

By the definition of strategy `next` and `tick` we know the stall symbol S will be removed resulting in the process term `go(t-1,l2,p)`. By the definition of time progression in TiMo and the assumption $\texttt{t} > 0$ we have

$$l\,[\![\, \texttt{go}^{\Delta t}\ l2\ \texttt{then}\ \sigma^{-1}(\texttt{p})\,]\!] \xrightarrow{\surd_l} at\,[\![\, \texttt{go}^{\Delta t-1}\ l2\ \texttt{then}\ \sigma^{-1}(\texttt{p})\,]\!]$$

as required.
ii) *Process moves*: Suppose applying the `move` rule resulted in the process moving to location `l2` producing the network term `l2[S(p)]`. By the definition of strategy `next` and `tick` we know the stall symbol S will be removed resulting in the network term `l2[p]`. By the action rule (Move) (Table 2) we have

$$l\,[\![\, \texttt{go}^{\Delta t}\ l2\ \texttt{then}\ \sigma^{-1}(\texttt{p})\,]\!] \xrightarrow{l2@l} l2\,[\![\, \circledS\sigma^{-1}(\texttt{p})\,]\!]$$

The result follows since the stall symbol $\circledS$ will be removed by the time progression step in TiMo. $\square$

Theorem 3 (Completeness) Let $N_1, N_2 \in Net(TM)$ be any well–formed network terms in TM. Then, if $N_1 \overset{\Psi}{\Longrightarrow} N_2$, for some location l and some multi-set $\Psi = \{\psi_1, \ldots, \psi_m\}$ of l-actions, then $\sigma(N_1)$ `=>` $\sigma(N_2)$ by applying the `step` rule. In other words, the diagram for completeness in Figure 1 commutes.

Proof. By the definition of a derivation in TiMo and the `step` rule it suffices to consider a well–formed network of the form

$$l\,[\![\, P_1 \;\mid\; \ldots \;\mid\; P_n \,]\!] \equiv l\,[\![\, P_1 \,]\!] \;\mid\; \ldots \;\mid\; l\,[\![\, P_n \,]\!],$$

where $n > 0$ and each P_i is an atomic process. Suppose

$$l\,[\![\, P_1 \;\mid\; \ldots \;\mid\; P_n \,]\!] \overset{\Psi}{\Longrightarrow} N',$$

for some finite set of l–actions $\Psi = \{\psi_1, \ldots, \psi_m\}$, $m \geq 0$. Then it can be seen that each atomic process P_i is involved in at most one l–action ψ_i. We show that the derivation applied to each process P_i is correctly captured by the `step` rule in $Md(TM)$. We have four possible cases to consider: 1) rule `move` was applied; 2) rule `call` was applied; 3) rule `com` was applied; or 4) no process transition rule was applied and time was simply allowed to progress. For brevity, we consider only Case 3) in detail here (for a complete example of this type of proof see [13]).

Case 3) Suppose the action rule (Com) has been applied to two processes P_i and P_j, for $i \neq j$, i.e.

$$l\,[\![\, c^{\Delta t_1}\,!\,\langle l_2\rangle \;\texttt{then}\; P_i^1 \;\texttt{else}\; P_i^2 \;\mid\; c^{\Delta t_2}\,?\,(vl : Loc) \;\texttt{then}\; P_j^1 \;\texttt{else}\; P_j^2)\,]\!]$$

$$\xrightarrow{c<l_2>@l} l\,[\![\, \circledS P_i^1 \;\mid\; \circledS\{l_2/vl\}P_j^1 \,]\!]$$

where the stall symbols $\circledS$ will be removed by the final time step. Then we have

$$\begin{aligned}&\sigma(c^{\Delta t_1}\,!\,\langle l_2\rangle \;\texttt{then}\; P_i^1 \;\texttt{else}\; P_i^2 \;\mid\; c^{\Delta t_2}\,?\,(vl : Loc) \;\texttt{then}\; P_j^1 \;\texttt{else}\; P_j^2))\\ &= \texttt{out(c,t1,l2,}\sigma(P_i^1)\texttt{,}\sigma(P_i^2)\texttt{)} \;\mid\; \texttt{in(c,t2,vl,}\sigma(P_j^1)\texttt{,}\sigma(P_j^2)\texttt{)}\end{aligned}$$

By applying the `com` rule we have

$$\begin{aligned}&\texttt{out(c,t1,l2,}\sigma(P_i^1)\texttt{,}\sigma(P_i^2)\texttt{)} \;\mid\; \texttt{in(c,t2,vl,}\sigma(P_j^1)\texttt{,}\sigma(P_j^2)\texttt{)}\\ &= \texttt{S(}\sigma(P_i^1)\texttt{)} \;\mid\; \texttt{S(sub(}\sigma(P_j^1)\texttt{,vl,l2))}\end{aligned}$$

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

$$\sigma(P_i^1 \;\mid\; \{l_2/vl\}P_j^1) = \sigma(P_i^1) \mid \texttt{sub}(\sigma(P_j^1), \texttt{vl}, \texttt{l2})$$

by definition of σ. □

5 Case Study: Robot Swarm

In this section we investigate applying the developed Maude framework to analyse a simple *Complex Adaptive Systems (CAS)* [26, 5]. We formulate a new TiMo model of a simple robot swarm example based on robots collaborating to pull up sticks [27, 28], and then use Maude to simulate and analyse this model. This simple example gives useful insight into the flexibility of the proposed Maude modelling approach and illustrates the type of interesting analysis possible.

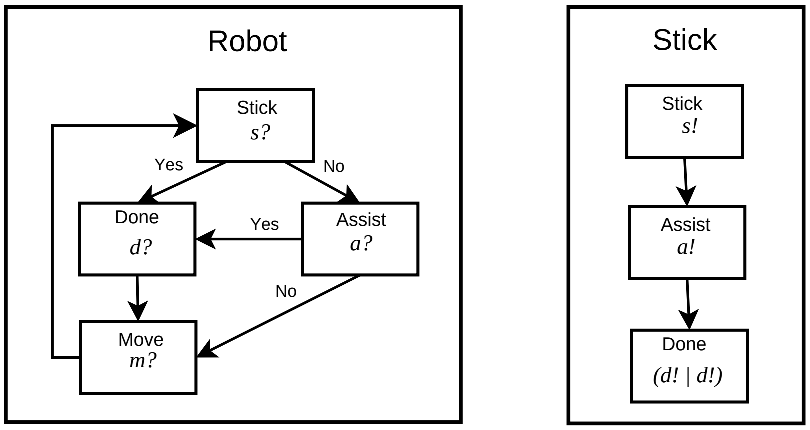

Fig. 2. A graphical representation of the two TiMo processes *stick* and *robot*

5.1 A TiMo Specification of the Stick Pulling Problem

The *stick pulling problem* [27, 28] is a robot swarm example in which a group of robots are given the task of locating and pulling up a set of sticks distributed in a given space. Importantly, no one robot is able to pull up a stick by themselves and so two robots need to collaborate to remove a stick.

To model the problem we define a TiMo specification *SP*. We begin by specifying a simple grid space which is made up of locations labelled $L(i, j)$, where i is the row and j is the column. To indicate the locations reachable from a given grid location we use an output process over a channel m defined as follows:

$$D(l) \overset{\text{df}}{=} m^{\Delta 2}\,!\,\langle l\rangle \;\; \texttt{then}\; D(l)\; \texttt{else}\; D(l)$$

As an example of how this can be used, consider defining a 3×3 grid in which robots are able to move vertically and horizontally:

$$\begin{aligned} grid \overset{\text{df}}{=} & \; L^{1,1}\,[\![\, D(L^{1,2}) \mid D(L^{2,1}) \,]\!] \;\mid\; L^{1,2}\,[\![\, D(L^{1,1}) \mid D(L^{1,3}) \mid D(L^{2,2}) \,]\!] \\ & \mid L^{1,3}\,[\![\, D(L^{1,2}) \mid D(L^{2,3}) \,]\!] \;\mid\; L^{2,1}\,[\![\, D(L^{1,1}) \mid D(L^{2,2}) \mid D(L^{3,1}) \,]\!] \\ & \mid L^{2,2}\,[\![\, D(L^{2,1}) \mid D(L^{1,2}) \mid D(L^{2,3}) \mid D(L^{3,2}) \,]\!] \\ & \mid L^{2,3}\,[\![\, D(L^{2,2}) \mid D(L^{1,3}) \mid D(L^{3,3}) \,]\!] \;\mid\; L^{3,1}\,[\![\, D(L^{2,1}) \mid D(L^{3,2}) \,]\!] \\ & \mid L^{3,2}\,[\![\, D(L^{3,1}) \mid D(L^{2,2}) \mid D(L^{3,3}) \,]\!] \;\mid\; L^{3,3}\,[\![\, D(L^{3,2}) \mid D(L^{2,3}) \,]\!] \end{aligned}$$

We model sticks and robots using the two processes given in Figure 2. The TiMo definitions for these two processes are given below. These definitions make use of three channels to synchronise the stick pulling operation (see Figure 2): channel s is used to check if a free stick needs pulling; channel a is used to check if a robot needs assistance to complete the pulling up of a stick; and channel d is used to confirm when a stick has been successfully pulled out.

$$\begin{aligned}
\mathit{stick} \overset{\mathrm{df}}{=}\ & s^{\Delta\infty}\,\texttt{!}\ \texttt{then}\ (a^{\Delta\infty}\,\texttt{!}\ \texttt{then}\ ((d^{\Delta\infty}\,\texttt{!}\ \texttt{then stop else stop}) \\
& \mid (d^{\Delta\infty}\,\texttt{!}\ \texttt{then stop else stop}))\ \texttt{else}\ \mathit{stick})\ \texttt{else}\ \mathit{stick}
\end{aligned}$$

$$\begin{aligned}
\mathit{robot} \overset{\mathrm{df}}{=}\ & s^{\Delta 0}\,\texttt{?}\ \texttt{then}\ (d^{\Delta\infty}\,\texttt{?}\ \texttt{then}\ (m^{\Delta 0}\,\texttt{?}\,(l)\ \texttt{then}\ (\texttt{go}^{\Delta 0}\ l\ \texttt{then}\ \mathit{robot}) \\
& \texttt{else}\ \mathit{robot})\ \texttt{else}\ \mathit{robot})\ \texttt{else}\ (a^{\Delta 0}\,\texttt{?}\ \texttt{then}\ (d^{\Delta\infty}\,\texttt{?}\ \texttt{then} \\
& (m^{\Delta 0}\,\texttt{?}\,(l)\ \texttt{then}\ (\texttt{go}^{\Delta 0}\ l\ \texttt{then}\ \mathit{robot})\ \texttt{else}\ \mathit{robot})\ \texttt{else}\ \mathit{robot})\ \texttt{else} \\
& (m^{\Delta 0}\,\texttt{?}\,(l)\ \texttt{then}\ (\texttt{go}^{\Delta 0}\ l\ \texttt{then}\ \mathit{robot})\ \texttt{else}\ \mathit{robot}))
\end{aligned}$$

Given the above TiMo specification *SP* we can define a range of different systems. As an example, consider the system given below which involves three robots and two sticks:

$$\mathit{grid} \mid L^{1,1,}\,[\![\,\mathit{robot}\,]\!] \mid L^{1,3,}\,[\![\,\mathit{robot}\,]\!] \mid L^{3,1,}\,[\![\,\mathit{robot}\,]\!] \mid L^{1,2}\,[\![\,\mathit{stick}\,]\!] \mid L^{2,2}\,[\![\,\mathit{stick}\,]\!]$$

5.2 Applying the Maude Framework

We now apply the techniques developed in Section 4 to translate the TiMo specification *SP* of the stick pulling problem into a Maude model $Md(SP)$. To begin we extend the Maude definitions to incorporate the locations and channels that are used in *SP*. We then need to translate the three process definitions for $D(l)$, *stick*, and *robot* into appropriate Maude rules. As an example, consider the following `call` rule used to model the definition of the *stick* process:

```
rl [call] : (AL[stick | P] | N) => (AL[S(out(s,inf,(out(a,inf,
              ((out(d,inf,stop,stop)) | (out(d,inf,stop,stop))),
              stick)),stick)) | P] | N) .
```

We can now consider analysing $Md(SP)$ using Maude's built–in model checking command `search`. However, as the model stands its use would be very restricted due to the state space explosion problem that arises given the number of locations available to be selected during each derivation step. To address this problem we can go back to the definition of our rewriting strategy captured by the `step` rule and update this strategy so that only locations containing robot processes are considered. This helps to make analysis tractable and illustrates the flexibility of Maude and its metaprogramming capabilities.

Consider the following `search` test that confirms robots are able to collaborate to pull up a stick:

```
search [1] (grid | (L(1,1)[robot]) | (L(2,2)[stick]) |
           (L(3,3)[robot])) =>+ N1 such that not(areSticks(N1)) .
```

The test makes use of a function `areSticks : Nets -> Bool` which captures the property of there being no sticks in a grid. This function is straightforward to define equationally and again this illustrates the flexibility afforded by Maude.

As an example of a further test, consider the more complex test given below which again confirms the robots abilities to collaborate:

```
search [1] (grid | (L(1,1)[robot]) | (L(1,3)[robot]) |
    (L(1,2)[stick]) | (L(2,2)[stick]) | (L(3,1)[robot]))
    =>+ N1 such that not(areSticks(N1)) .
```

A range of interesting analysis can be applied to the model using MAUDE and its various analysis tools, such as the LTL model checker [24]. A detailed analysis of the model (and its potential shortcomings) is beyond the scope of this paper and will be considered in future work.

6 Concluding Remarks

Aiming to bridge the gap between the existing theoretical approach of process calculi and forthcoming realistic programming languages for distributed systems, TIMO was introduced as a rather simple calculus. We can see TIMO as a prototyping language for multi-agent systems, featuring mobility and local interaction. Multi-agent systems typically consist of a large number of agents which exhibit autonomic behaviour depending on their timeouts and actions. The mobility of agents and interaction between the agents through communication may introduce new and sometimes unexpected behaviours. Components can be highly heterogeneous, each operating at different temporal scales and having different objectives. Verifying these systems is becoming increasingly necessary because they are extremely complex and often used in various critical application domains such as e-commerce and distributed collaborative systems. Thus, it is important to have modelling techniques and tools which are able to describe such systems, and to reason about their behaviour in both qualitative and quantitative terms. We see the work on TIMO as an important step towards this goal and this highlights one of the many important contributions to this field made by Maciej Koutny.

After the initial version of TIMO introduced in [9, 10], several variants of TIMO were developed during the last years: a version with access permissions given by a type system [18], a real-time version RTIMO [1], a probabilistic extension pTIMO [14], and a version with costs cTIMO .

Inspired by TIMO , a flexible software platform was presented in [12] to support the specification of agents allowing timed migration in a distributed environment. A verification tool called TIMO@PAT was developed by using an extensible platform for model checkers [15]. A probabilistic temporal logic called PLTM was introduced in [14] to verify properties of pTIMO processes making explicit reference to specific locations, and using temporal constraints over local clocks and multisets of actions. A formal relationship between RTIMO and timed automata allows us to use the model checking capabilities provided by the software tool UPPAAL [2]. TIMO was used to describe a railway control system, and then a new behavioural congruence over real-time systems (named strong open time-bounded bisimulation) was used to check which behaviours are closer to an optimal and safe behaviour [3]. In [17] it is defined a general framework for reasoning about systems specified in TIMO by using the Event-B modelling method and the Rodin platform.

In this paper we have extended existing work [13, 16] to develop a new semantic translation from TiMo to Maude. The aim here was to provide support for analysing TiMo specifications by allowing the range of interesting model checking tools provided by Maude to be applied. We illustrated our approach with a simple CAS robot swarm example based on the stick pulling problem. This case study involved developing a new TiMo specification of the stick pulling problem and illustrated the analysis possible using Maude. In particular, it highlighted the considerable flexibility provided by Maude and its metaprogramming capabilities.

Acknowledgements

We enjoy every moment shared with Maciej (as scientist and as friend), and express our thanks for these moments. *Gabriel Ciobanu*

I would like to sincerely thank Maciej for his support, guidance and friendship over the last 25 years. He is an inspirational researcher and I count myself very luck to have been able to work with him. *Jason Steggles*

References

1. B. Aman, G. Ciobanu. Real-Time Migration Properties of rTiMo Verified in Uppaal. *Lecture Notes in Computer Science* **8137**, 31–45 (2013).
2. B. Aman, G. Ciobanu. Timed Mobility and Timed Communication for Critical Systems. *Lecture Notes in Computer Science* **9128**, 146–161 (2015).
3. B. Aman, G. Ciobanu. Verification of Critical Systems Described in Real-Time TiMo. *Int'l Journal on Software Tools for Technology Transfer*, 1–14 (2016).
4. B. Aman, G. Ciobanu, M. Koutny. Behavioural Equivalences over Migrating Processes with Timers. *Lecture Notes in Computer Science* **7273**, 52–66 (2012).
5. V. Andrikopoulos , A. Bucchiarone, S. Gómez Sáez, D. Karastoyanova, C.A. Mezzina. Towards Modeling and Execution of Collective Adaptive Systems. In: Lomuscio A.R., Nepal S., Patrizi F., Benatallah B., Brandić I. (eds), *Service-Oriented Computing - ICSOC 2013.* LNCS 8377, pages 69–81, Springer, 2014.
6. E.Balland, P.Brauner, R.Kopetz, P.-E.Moreau, and A.Reilles. Tom: Piggybacking rewriting on java. In: RTA'07, LNCS 4533, pages 36–47, Springer Verlag, 2007.
7. 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).
8. G. Ciobanu and C. Prisacariu: Timers for Distributed Systems. *Electronic Notes in Theoretical Computer Science* 164 (2006) 81–99.
9. G. Ciobanu and M. Koutny: Modelling and Verification of Timed Interaction and Migration. Proc. of *FASE'08*, Springer, LNCS 4961 (2008) 215–229.
10. G. Ciobanu and M. Koutny: Timed Mobility in Process Algebra and Petri Nets. *Journal of Algebraic and Logic Programming* 80(7) (2011) 377–391.
11. G. Ciobanu and M. Koutny: Timed Migration and Interaction with Access Permissions. Proc. of *FM'11*, Springer, LNCS 6664 (2011) 293–307.

12. G. Ciobanu, C. Juravle. Flexible Software Architecture and Language for Mobile Agents. *Concurrency and Comput. Practice and Experience* **24**, 559–571 (2012).
13. 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.
14. G. Ciobanu, A. Rotaru. A Probabilistic Logic for pTiMo. *Lecture Notes in Computer Science* **8049**, 141–158 (2013).
15. G. Ciobanu, M. Zheng. Automatic Analysis of TiMo Systems in PAT. *Engineering of Complex Comp. Systems (ICECCS)*, IEEE Computer Society, 121–124 (2013).
16. G. Ciobanu, M. Koutny, and L. J. Steggles. Strategy based semantics for mobility with time and access permissions. *Formal Aspects of Computing*, 27(3):525–549, 2014.
17. G. Ciobanu, T.S. Hoang, A. Stefanescu. From TiMo to Event-B: Event-Driven Timed Mobility. *ICECCS 2014* (best paper), IEEE Computer Society, 1–10 (2014).
18. G. Ciobanu and M. Koutny: PerTiMo: A Model of Spatial Migration with Safe Access Permissions. *The Computer Journal* 58(5) (2015) 1041–1060.
19. M. Clavel, *et al.*: Maude as a Metalanguage. In: C. Kirchner and H. Kirchner (eds), Proc. of *WRLA '98*, *Electronic Notes in Theoretical Computer Science* 15 (1998).
20. M. Clavel, *et al*: Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science* 285(2) (2002) 187–243.
21. M. Clavel, *et al. Maude Manual (Version 2.7)* http://maude.cs.illinois.edu/ Accessed April 2016
22. 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.
23. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, K. Sonmez. Pathway Logic: Executable models of biological networks. In: F. Gadducci, U. Montanari (eds.), *Proc. of WRLA 2002*, *Electronic Notes in Theoretical Computer Science*, 71:144–161, 2004.
24. S. Eker, J. Meseguer, A. Sridharanarayananb. The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.
25. S. Eker, N. Martí-Oliet, J. Meseguer, A. Verdejo. Deduction, Strategies, and Rewriting. *Proc. of STRATEGIES 2006*, *Electronic Notes in Theoretical Computer Science*, 174(11):3–25, 2007.
26. J. Hillston. Challenges for Quantitative Analysis of Collective Adaptive Systems. In: Abadi M., Lluch Lafuente A. (eds), *Trustworthy Global Computing. TGC 2013*, LNCS 8358, pages 14–21, Springer, 2014
27. A. J. Ijspeert, A. Martinoli, A. Billard, and L. M. Gambardella. Collaboration Through the Exploitation of Local Interactions in Autonomous Collective Robotics: The Stick Pulling Experiment. *Autonomous Robots*,11(2):149–171, 2001.
28. A. Martinoli, K. Easton, and W. Agassounon. Modeling Swarm Robotic Systems: a Case Study in Collaborative Distributed Manipulation. *The International Journal of Robotics Research*, 23(4-5):415-436, 2004.
29. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In: D.M.Gabbay and F.Guenthner (eds), *Handbook of Philosophical Logic (Second Edition)*, Vol. 9, pages 1–87, Kluwer Academic Publishers, 2002.
30. J. Meseguer: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(2) (1992) 73–155.
31. V. Nigam, R. Donaldson, M. Knapp, T. McCarthy and C. Talcott. Inferring Executable Models from Formalized Experimental Evidence. Computational Methods in Systems Biology 9308, pages 90–103, Springer Verlag, 2015.

32. L. J. Steggles. Rewriting Logic and Elan: Prototyping Tools for Petri Nets with Time. *Applications and Theory of Petri Nets 2001*, LNCS 2075, pages 363-381, Springer Verlag, 2001.
33. L. J. Steggles, R. Banks, O. Shaw, A. Wipat. Qualitatively modelling and analysing genetic regulatory networks: a Petri net approach. *Bioinformatics*, 23(3):336-343, 2006.
34. M-O. Stehr, J. Meseguer, and P. C. Ölveczky Rewriting Logic As a Unifying Framework for Petri Nets. In: H. Ehrig, *et al.* (eds), Unifying Petri Nets: Advances in Petri Nets, LNCS 2128, pages 250–303, Springer Verlag, 2001.
35. 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.