From TiMo to Event-B: Event-Driven Timed Mobility

Gabriel Ciobanu
Romanian Academy,
Inst. of Computer Science,
Iaşi, Romania

Thai Son Hoang
ETH Zürich,
Inst. of Information Security,
Switzerland

Alin Stefanescu
University of Bucharest
and University of Pitesti,
Romania

***Abstract*—Mobile distributed systems involve specific aspects such as migration, communication and concurrency, usually under temporal constraints. In this paper, we deal with formal modelling of timed migrating and communicating processes, as provided by the TIMO calculus. In this framework, mobile processes can move between different locations and communicate when co-located, all this happening in the presence of local timers. Our contribution is a general framework for reasoning about systems specified using TIMO. We use the Event-B modelling method as the target for translating TIMO specifications. Subsequently, we utilise the supporting Rodin platform of Event-B to verify system properties using the embedded theorem-provers and model checkers. The main feature of our encoding include a generic model capturing the syntax and semantics of TIMO, together with a concrete model corresponding to each specific TIMO specification. We illustrate our approach by a non-trivial example featuring different concepts of TIMO.**

I. INTRODUCTION

We consider a process calculus named TIMO (Timed Mobility), with explicit migration allowing the use of timers for controlling process mobility and interaction [9]. Migration involves several explicit locations. Each location has a local clock; these local clocks attempt to specify distributed systems in a more accurate way. Timing constraints for migration allow to specify a temporal timeout after which a mobile process must move to another location. Two processes may communicate only if they are present at the same location. TIMO allows a maximal parallelism of actions. Using all these features, we can specify and analyse complex timing systems in a new way, different from traditional ones. In terms of verification, interesting properties described by TIMO regarding process migration, time constraints, bounded liveness and optimal reachability could be analysed and checked using model-checking [13].

In this paper, we provide a formal encoding of TIMO in a framework based on theorem proving, called Event-B [2]. We follow a stepwise modelling approach using several refinements levels, which starts with an abstract model and gradually introduces the different concepts and rules of TIMO calculus. The correctness of the stepwise construction of formal Event-B models is ensured by (automatically or interactively) discharging a set of proof obligations generated by the Rodin tool set [3]. Moreover, we can encode not only the general TIMO rules but also a concrete TIMO system and analyse on it different types of properties, by applying the theorem proving and model checking technology available in Rodin. We exemplify this with an extended version of the TIMO model from the literature [10]. In particular, the parameterised version of the system is sophisticated and challenging to verify.

The contributions of our work are listed below:

- The most important feature of our Event-B encoding is its genericity. We do not encode only concrete systems in Event-B as it is usually done in the literature, but also the TIMO syntax and operational rules. Thus we can reason about general properties of TIMO systems, such as the preservation of well-formedness, showing that they hold using theorem-proving. This is not possible using classical model-checking techniques, which need concrete instantiations in order to generate the state space for analysing.
- There is a strong mathematical relation between the concrete encoding and the generic encoding, namely an Event-B refinement, which we formally proved using Rodin. Properties proved on the generic model are utilised to reason about properties of the concrete systems. This is different from the literature, where either only a concrete encoding of a (distributed) system is given, see e.g., [17], or there is a transformation algorithm based on the rules of the input formalism which converts an input system into a concrete Event-B model, see e.g., [6], [4].
- We give an Event-B implementation of local clocks and relative time suited for TIMO. We are not aware of any other Event-B specifications dealing with similar timing constraints. The majority of the Event-B approaches adapt a modelling time pattern based on a global clock and absolute time [8], [20].
- Through Event-B/Rodin, we make theorem proving technologies available to TIMO. Until now, only reasoning through model-checking was possible [13]. In particular, we are able to reason about parameterised systems, which is not feasible in the TIMO verification based on model checking.

The paper is structured as follows. We give a brief overview of TIMO and of Event-B in Section II and Section III. In Section IV, we present our approach to encode TIMO specifications in Event-B. We provide the results of our analysis on the Event-B encoding and the running example in Section V. We conclude and discuss related work in Section VI.

II. TIMO: TIMED MOBILITY IN DISTRIBUTED SYSTEMS

Several process calculi are used to model complex distributed systems in a compositional way. Various features were introduced in these process calculi, including explicit locations in distributed π-calculus [16], explicit migration and timers in timed distributed π-calculus [12]. Most of the papers assume the existence of a global clock; however, there are

TABLE I. TIMO SYNTAX.
(LENGTH OF $\vec{u}$ IS THAT OF $\vec{X}$, AND LENGTH OF $\vec{v}$ IN $id(\vec{v})$ IS m_{id})

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

several applications and systems for which a global clock is inappropriate. The process calculus TIMO was introduced in [9] as a formalism for mobile systems in which it is possible to add local timers to control process mobility and interaction. Processes are equipped with communication capabilities which are active up to a predefined time deadline (timer). A local clock is assigned to each location, and each local clock determines the timing of actions executed at the corresponding location [10].

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 $(x_1,\ldots,x_k)$ whenever it does not lead to confusion. The syntax of TIMO is given in Table I, 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. It is assumed that: (i) $a \in Chan$ is a channel, and $t \in \mathbb{N}$ 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) Ⓢ is a special symbol used to state that a process is temporarily 'stalled'.

The only variable binding construct is $a^{\Delta t}\,?\,(\vec{u}{:}\vec{X})$ `then` P `else` P'; it binds the variables $\vec{u}$ within P, but *not* within P'. We denote by $fv(P)$ the free variables of a process P. 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 alpha-conversion, and $\{v/u\}P$ is obtained from P by replacing all free occurrences of a variable u by v, 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 $\{v_1/u_1, v_2/u_2, \ldots, v_k/u_k\}P$.

A process $a^{\Delta t}\,!\,\langle\vec{v}\rangle$ `then` P `else` P' attempts to send a tuple of values $\vec{v}$ over channel a for t time units. If successful, it continues as process P; otherwise, as the alternative process P'. A process $a^{\Delta t}\,?\,(\vec{u}{:}\vec{X})$ `then` P `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 $\texttt{go}^{\Delta t}\ l$ `then` P which moves from the current location to the location l within t time units. Since l can be a variable, migration actions support a flexible scheme for moving processes around a network. Processes are further constructed from the terminating process `stop` and parallel composition $P|P'$. Finally, process expressions of the form Ⓢ P represent a technical device used in the structural operational semantics of TIMO; intuitively, Ⓢ specifies that a process P is temporarily *stalled* (until a clock tick), 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 $N \mid N'$.

TABLE II. TIMO OPERATIONAL SEMANTICS.
(IN (PAR) AND (EQUIV) ψ IS AN ACTION, AND IN (TIME) l IS A LOCATION.)

(EQ1) $\quad N \mid N' \equiv N' \mid N$

(EQ2) $\quad (N \mid N') \mid N'' \equiv N \mid (N' \mid N'')$

(EQ3) $\quad l\,[\![\,P \mid P'\,]\!] \equiv l\,[\![\,P\,]\!] \mid l\,[\![\,P'\,]\!]$

(CALL) $\quad l\,[\![\,id(\vec{v})\,]\!] \xrightarrow{id@l} l\,[\![\,\text{Ⓢ}\,\{\vec{v}/\vec{u}\}P_{id}\,]\!]$

(MOVE) $\quad l\,[\![\,\texttt{go}^{\Delta t}\ l'\ \texttt{then}\ P\,]\!] \xrightarrow{l'@l} l'\,[\![\,\text{Ⓢ}\,P\,]\!]$

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

(PAR)
$$\frac{N \xrightarrow{\psi} N'}{N \mid N'' \xrightarrow{\psi} N' \mid N''}$$

(EQUIV)
$$\frac{N \equiv N' \quad N' \xrightarrow{\psi} N'' \quad N'' \equiv N'''}{N \xrightarrow{\psi} N'''}$$

(TIME)
$$\frac{N \not\xrightarrow{}_l}{N \xrightarrow{\surd_l} \phi_l(N)}$$

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 II 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 II 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), $N \not\xrightarrow{}_l$ 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 $!\langle\vec{v}\rangle$ or $?\,(\vec{u}{:}\vec{X})$) 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.

A complete computational step is captured by a *derivation* of the form $N \stackrel{\Psi}{\Longrightarrow} N'$, where $\Psi = \{\psi_1, \ldots, \psi_m\}$ ($m \geq 0$) is a finite multiset[1] of l-actions for some location l (i.e., actions of the form $id@l$ or $l'@l$ or $a\langle\vec{v}\rangle@l$) 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. Note that whenever there is only a time progression at a location, we have $N \stackrel{\varnothing}{\Longrightarrow} N'$. The 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 (concurrent) individual executions. Moreover, derivations preserve well-formedness of networks [10].

Example: We use an example presented and illustrated in [10] describing simple e-shops. In this example we consider two customer processes initially residing in their respective home locations $homeA$ and $homeB$, and looking for (the address of) an e-shop where the same desirable e-item can be purchased. To find this out, each customer moves to the location $info$ in order to acquire the relevant address (this move takes up to 5 time units). After waiting for 2 time units at location $info$ without getting the desired address, the e-item loses its importance and the customer is no longer interested in acquiring it. The location $info$ contains a broker who knows all about the e-shops stocking the desired e-item. For up to 5 time units the right e-shop is that at the location $shopA$, and after that for up to 7 time units at location $shopB$ (these changes of availability are cyclical and happen also if a location is communicated to a customer). We consider the evolution in which the active customer is initially residing in location $homeB$, and then moving to location $info$ to acquire the address of an e-shop. After receiving such an address from the broker, the customer moves to the corresponding location $shopA$. The essential features of this example are captured by the following TIMO network whose evolution together with several results are described in [10]:

$$homeA[\![\,customer(homeA)\,]\!] \mid homeB[\![\,customer(homeB)\,]\!] \mid \\ \mid info[\![\,broker\,]\!]\ ,$$

where the process identifiers are defined as:

$$\begin{aligned} customer(home{:}Loc) &\stackrel{\text{df}}{=} \mathtt{go}^{\Delta 5}\ info\ \mathtt{then} \\ &\qquad a^{\Delta 2}\,?\,(shop{:}Loc) \\ &\qquad \mathtt{then}\ \mathtt{go}^{\Delta 2}\ shop\ \mathtt{then\ stop} \\ &\qquad \mathtt{else}\ \mathtt{go}^{\Delta 5}\ home\ \mathtt{then\ stop} \\ broker &\stackrel{\text{df}}{=} a^{\Delta 5}\,!\,\langle shopA\rangle\ \mathtt{then}\ broker'\mathtt{else}\ broker' \\ broker' &\stackrel{\text{df}}{=} a^{\Delta 7}\,!\,\langle shopB\rangle\ \mathtt{then}\ broker\mathtt{else}\ broker \end{aligned}$$

[1]According to Proposition 2 in [10], the result of executing a (multi-)set of rules is independent of the order of executing them.

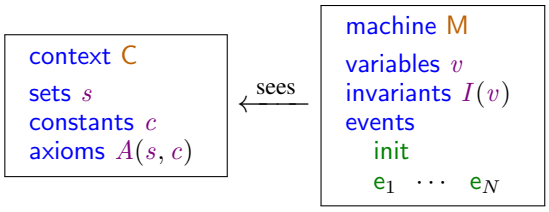

Fig. 1. The general form of an Event-B context and a machine

III. Introduction to Event-B

The classical B method (B), introduced in the 90's by Abrial [1], proved to be successful in industry. Several large implementations, e.g., in transportation, were based on B. However, B mainly focused on software development. To enlarge the scope of formal development to system level, Event-B [2] was proposed in the 00's and developed over the course of several European projects, involving partners from both academia and industry. The formalism is supported by a mature platform, Rodin [3], and proved to be very useful in different practical scenarios as presented in [19]. The tool is open-source and Eclipse-based, and integrates various plugins for modelling, theorem proving, simulation, model-checking and many other features.

An Event-B model contains *contexts* (capturing its static behaviour) and *machines* (capturing its dynamic behaviour). Fig. 1 gives the general syntax of a context C and a machine M. Context C contains carrier sets (static types) s and constants c, constrained by axioms $A(s, c)$. Machine M corresponds to a transition system, where the states are captured by global variables v, constrained by invariants $I(v)$, and the transitions are modelled by *events* e_i. By *seeing* context C, machine M can have access to s and c. To simplify our presentation, in the subsequent, we omit possible references to s and c in machine M. The general form of an event is:

$$\mathsf{e} \mathrel{\widehat{=}} \mathsf{any}\ u\ \mathsf{where}\ G(u, v)\ \mathsf{then}\ v := E(u, v)\ \mathsf{end}\ ,$$

where u is a (possibly empty) list of local parameters. The guard $G(u, v)$ is a (possibly empty) list of predicates describing the condition under which the event can be executed. The event e is said to be enabled at a state when the guard G holds in that particular state. The action $v := E(u, v)$ is a (non-empty) set of assignments specifying changes to v when the event is carried out.[2]The init event is a special event without parameters and guards.

The execution semantics of an Event-B machine is as follows. First, the init event is executed. Afterwards, at each step an enabled event is non-deterministically selected and its actions are executed simultaneously. If no events are enabled, then the system deadlocks. The invariants $I(v)$ ensure that the variables are properly typed and also that the system has desirable properties. Proof obligations are generated to ensure that I is established by init and maintained by all events e_i.

The Event-B modelling approach is top-down, starting with an abstract model, which is gradually concretised in a series of refinement steps. Two mechanisms exist to support this development process: *context extension* and *machine refinement*. A (concrete) context D extends an (abstract) context C by having

[2]We do not consider non-deterministic Event-B actions in this paper.

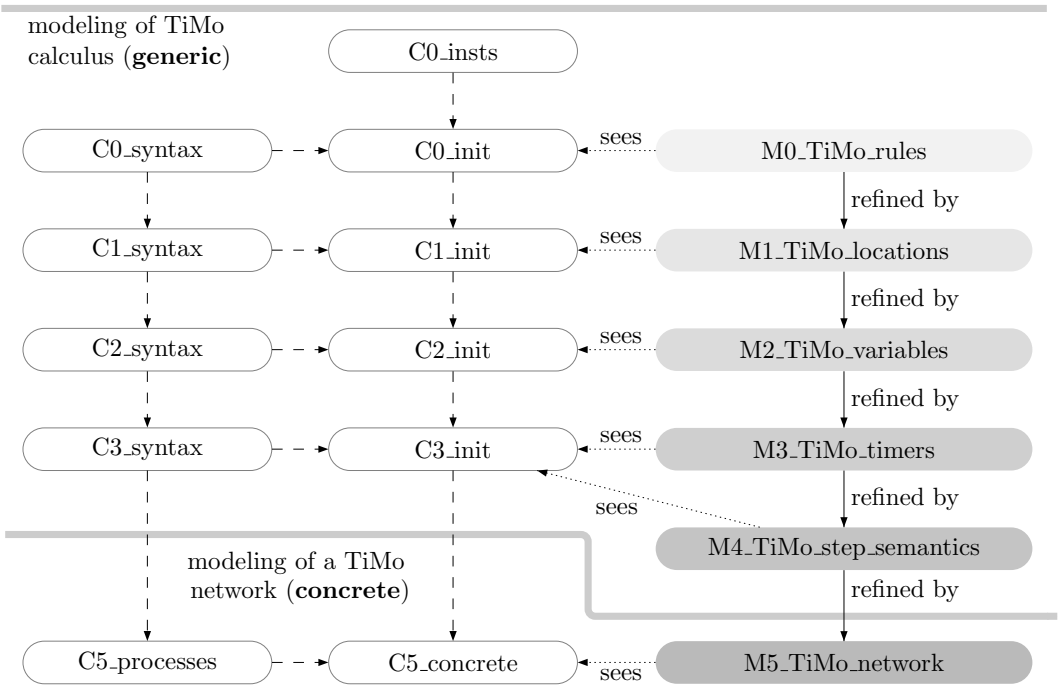

Fig. 2. The hierarchy of refinements of our Event-B encoding of TiMo (where a dashed arrow from C to C' means "C is extended by C")

additional carrier sets t and constants d with further axioms relating the two contexts. A (concrete) machine N refines an (abstract) machine M by replacing abstract variables v by concrete variables w. The states of M and N are linked by gluing invariants $J(v, w)$. Intuitively, M is refined by N if M can simulate any behaviour of N, taking into account the gluing invariants J. Reasoning about machine refinement can be established event-wise. Each abstract event e is refined by a concrete event f (this one-to-one relationship can be relaxed). Assume that e and f are as follows:

$$\mathsf{e} \mathrel{\widehat{=}} \textsf{any } u \textsf{ where } G(u, v) \textsf{ then } v := E(u, v) \textsf{ end}$$
$$\mathsf{f} \textsf{ refines } \mathsf{e} \mathrel{\widehat{=}} \textsf{any } y \textsf{ where } H(y, w) \textsf{ with } W(u, y, w) \textsf{ then } w := F(y, w) \textsf{ end}$$

Somewhat simplifying, we say that f refines e if (1) the concrete guard H is stronger than the abstract guard G (*guard strengthening*), and (2) the abstract action simulates the concrete action (*simulation*) via gluing invariant J. Notice that a witness is required in the form of a predicate $W(u, y, w)$ linking the abstract parameters u and the concrete parameters y.

A special form of refinement is when the abstract variables v are retained in the concrete machine. The proof obligations for refinement are adapted accordingly. Further details on the Event-B modelling method can be found in [2].

Event-B is supported by the Rodin platform. Theorem-proving is the main technology used for verifying the consistency (i.e., invariants preservation) and correctness of refinements. Rodin will automatically generate proof obligations for proving the consistency of machines and their refinements, which can be discharged either automatically or interactively. Further properties (e.g., temporal properties) of Event-B models can be verified using the ProB model checker [18], which is integrated as a plugin in Rodin.

IV. Modelling of TiMo in Event-B

In this section, we show how we can capture the TiMo concepts and semantics in Event-B. This will be done gradually, using five Event-B refinements, as depicted in Fig. 2. Note that there are two main parts: a generic part, modelling the TiMo calculus, and a concrete part, modelling a concrete TiMo network (see the last row of the figure). We use refinement to introduce the different features of TiMo such as locations, variables, timers, into the formal model gradually. The full model is available at: http://is.gd/eventb_timo

For the generic part, at each refinement level, the encoding uses two contexts and a machine, each specifying a different part of TiMo formalism. More precisely, we address separately the TiMo syntax, the process instances, and the operational semantic rules. The syntax of TiMo is coded progressively in the contexts C0_syntax–C3_syntax. The set of process instances is first defined in C0_insts and further extended in C0_init–C3_init, which define different functions that associate to each process instance its initial location, free variables and so on. Finally, the TiMo operational semantic rules such as (MOVE) or (COM) are abstractly defined as events in the M0_TiMo_rules and successively refined by M1_TiMo_locations, M2_TiMo_variables, M3_TiMo_timers, and M4_TiMo_step_semantics. These machines and their associated ('seen') contexts introduce the notions of locations, variables, local timers, and the step semantics based on locations, respectively.

For the concrete part, i.e., to model a concrete TiMo network (such as the one in the e-shop running example), we add an extra refinement layer. More precisely, C3_syntax and C3_init are extended by C5_processes and C5_concrete, respectively, in which we instantiate the constants with the concrete processes and the number of instances in the given TiMo network. Also, we refine

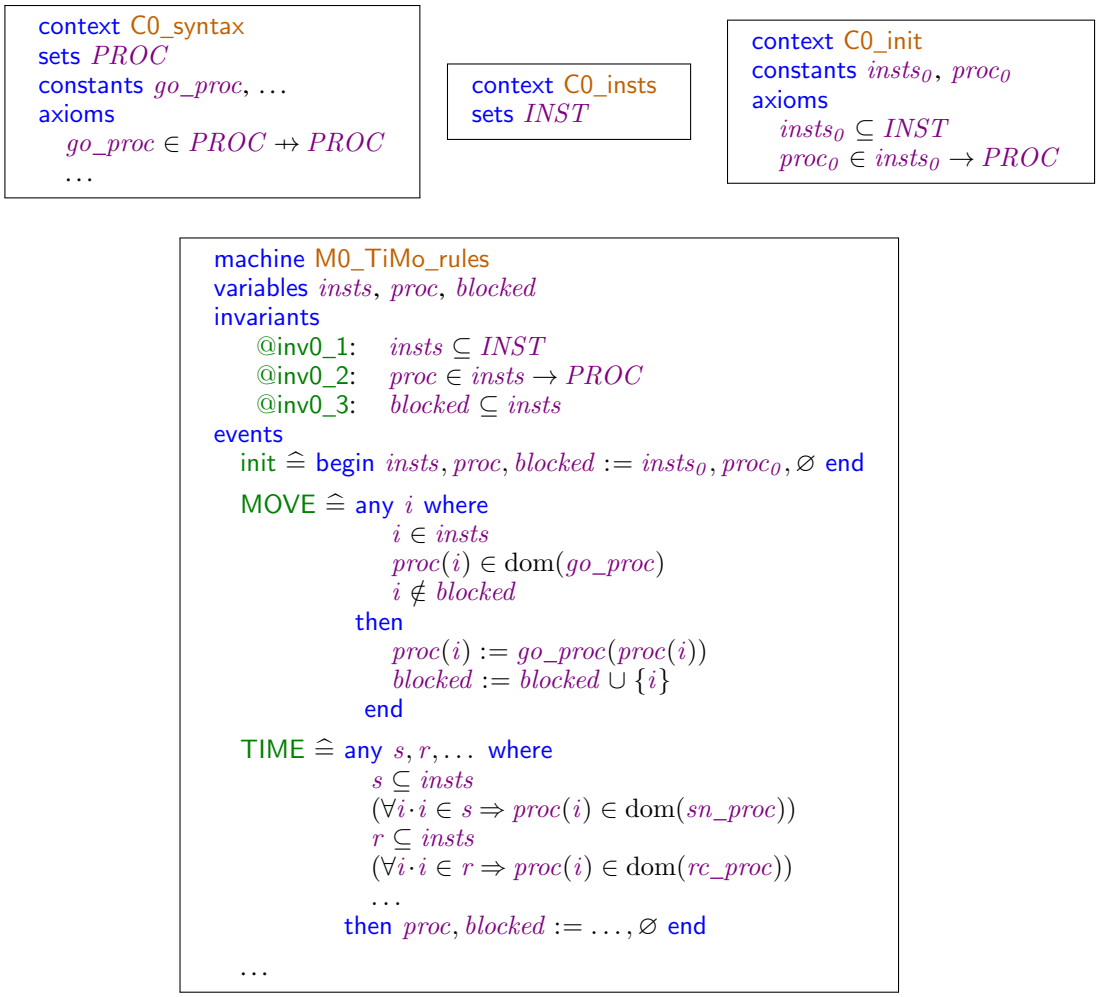

Fig. 3. The initial model

M4_TiMo_step_semantics by M5_TiMo_network in which each step of a TiMo process is mapped to a separate event. Notice that, given the concrete context C5_processes and C5_concrete, the behaviour of M4_TiMo_step_semantics is the same as that of M5_TiMo_network. To be more precise, we can perform analysis of the concrete network by having M4_TiMo_step_semantics seeing C5_processes and C5_concrete. We chose to tailor M5_TiMo_network towards the concrete TiMo network in order to illustrate the direct mapping between them with a view for automatic translation.

In the rest of the section, we overview the main data structures and variables of our model together with the design decisions made during the refinement process. We focus on the modelling of 'go-processes' (i.e., of the form "MOVEΔtlP"), and the (MOVE) and (TIME) rules. Other rules (e.g., (CALL), (COM), (PAR)) are modelled similarly.

Initial model (Fig. 3): In this initial model, we focus on the TiMo syntax of processes, abstracting away from details such as locations and timers. Carrier set $PROC$ models the set of processes. Constant go_proc (defined as a partial function) formalises the link between a go-process and its sub-process. The other relationships between processes are defined as partial function constants similarly. Machine M0_TiMo_rules contains three variables: $insts$ represents the set of active process instances of the network, $proc$ maps each instance to the corresponding process, and $blocked$ represents the set of blocked instances. Variables $insts$ and $proc$ are initialised to some constants defined in C0_init, while initially, there are no blocked instances.

Event MOVE has a parameter i representing the process to move. The guard of MOVE ensures that the instance is not blocked. The action of MOVE updates the pointer $proc(i)$ to the sub-process, i.e., $go\_proc(proc(i))$, and blocks i. The set of blocked instances is cleared when the TIME event occurs as specified by the TiMo semantics. Event TIME has parameters representing the set of process instances that require to update the process position, namely, the set of send-processes s and receive-processes r whose timers reached value zero (timed out). We omit the detail how $proc$ is updated according to these parameters.

First refinement (Fig. 4): This refinement introduces the notion of locations for process instances. The set of locations is defined as a constant LOC, a subset of some carrier set $DATATYPE$.

A variable loc is added to M1_TiMo_locations to keep track of the location for each process instance. Event MOVE now has an additional parameter l specifying the destination of

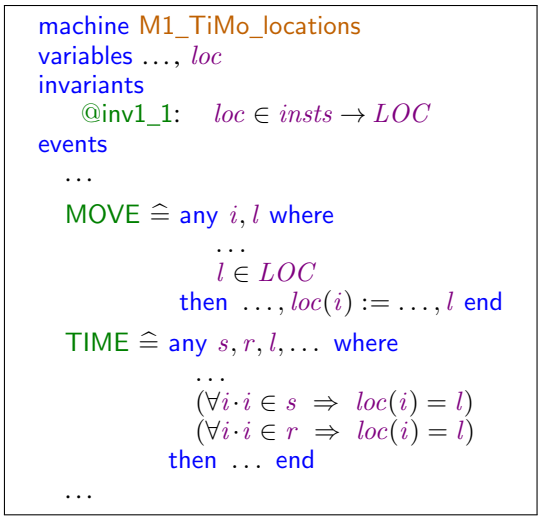

Fig. 4. The first refinement

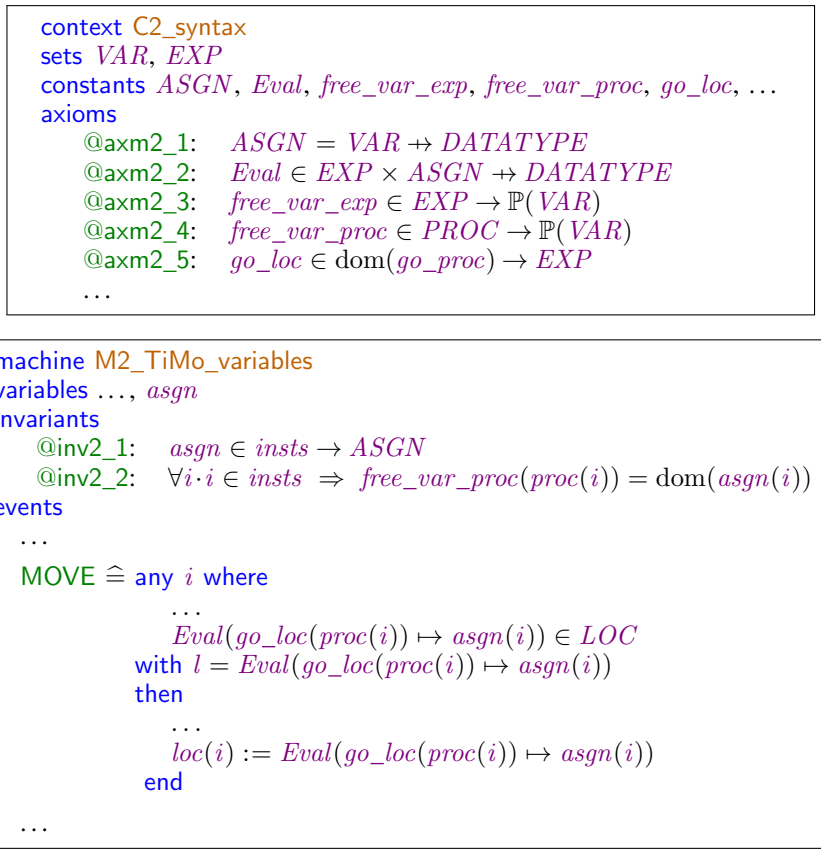

Fig. 5. The second refinement

the process instance i. At the moment, l is randomly chosen. Event TIME has an additional parameter l specifying the location where the time progresses. The new guard specifies that the process instances in s (those requires update to their processes) are at location l.

Second refinement (Fig. 5): Now, we introduce the notion of variables, variables assignments, expressions, and their evaluations. Carrier sets VAR and EXP abstractly define the set of (TiMo-) variables and expressions. Assignments ($ASGN$) are partial functions from variables to their values (@axm2_1). Evaluation of an expression e, given a (valid) variable assignment a, is a $DATATYPE$ value $Eval(e \mapsto a)$[3] (see @axm2_2). Expressions and processes are associated with a set of their free variables (@axm2_3 and @axm2_4). Finally, for every go-process, its target location is specified by an expression (@axm2_5).

Machine M2_TiMo_variables introduces a new variable $asgn$ capturing the variable assignments associated with each

[3]Event-B expression $a \mapsto b$ denotes the ordered pair (a, b).

context C3_syntax
constants $Time$, $proc\_timer$
axioms
@axm3_1: $Time = \mathbf{N}$
@axm3_2: $proc\_timer \in PROC \to Time$

machine M3_TiMo_timers
variables ..., $timer$
invariants
@inv3_1: $timer \in insts \to Time$
@inv3_2: $\forall i \cdot i \in insts \Rightarrow timer(i) \leq proc\_timer(proc(i))$
events
...
MOVE $\widehat{=}$ any i where ... then ..., $timer(i) := \ldots, proc\_timer(go\_proc(proc(i)))$ end
TIME $\widehat{=}$ any $s, r, l, o, \ldots$ where
$\forall i \cdot i \in insts \setminus blocked \wedge proc(i) \in \mathrm{dom}(go\_proc) \wedge loc(i) = loc \Rightarrow timer(i) \neq 0$
$s = \{i \mid i \in insts \setminus blocked \wedge proc(i) \in \mathrm{dom}(sn\_proc) \wedge loc(i) = l \wedge timer(i) = 0\}$
$r = \{i \mid i \in insts \setminus blocked \wedge proc(i) \in \mathrm{dom}(rc\_proc) \wedge loc(i) = l \wedge timer(i) = 0\}$
$o = \{i \mid i \in insts \setminus blocked \wedge \wedge loc(i) = l \wedge timer(i) \neq 0\}$
...
then $proc, blocked, timer := \ldots, \varnothing, \ldots$ end
...

Fig. 6. The third refinement

machine M4_TiMo_step_semantics
variables ..., $picked$, $curr$
invariants
@inv4_1: $curr \in LOC$
@inv4_2: $picked = \mathrm{FALSE} \Rightarrow blocked = \varnothing$
events
...
pick $\widehat{=}$ any l where $picked = \mathrm{FALSE} \wedge (\exists i \cdot i \in insts \wedge loc(i) = l)$ then $picked, curr := \mathrm{TRUE}, l$ end
MOVE $\widehat{=}$ any i where
$i \in insts$
$proc(i) \in \mathrm{dom}(go\_proc)$
$i \notin blocked$
$Eval(go\_loc(proc(i)) \mapsto asgn(i)) \in LOC$
$picked = \mathrm{TRUE}$
$loc(i) = curr$
then
$proc(i) := go\_proc(proc(i))$
$blocked := blocked \cup \{i\}$
$loc(i) := Eval(go\_loc(proc(i)) \mapsto asgn(i))$
$timer(i) := proc\_timer(go\_proc(proc(i)))$
end
...

Fig. 7. The fourth refinement

process instance (@inv2_1). Invariant @inv2_2 states that for every process instance, there are no unassigned free variables. We refine event MOVE accordingly. The abstract target location l is specified by evaluating the target location expression of the go-process ($go\_loc(proc(i))$ using the current variable assignment of the instance i, i.e., $asgn(i)$. The guard of MOVE ensures that the evaluation results in a location value. Event TIME is unchanged in this refinement.

Third refinement (Fig. 6): In this refinement, we introduce the timers and their associated semantics. We represent time by a natural number (@axm3_1). For each process, a default timer value is defined (@axm3_2). Machine M3_TiMo_timers introduces a new variable $timer$ representing the current timer value for each instance (@inv3_1). When an instance is switched to a new process, the timer is set to the default value. For instance, in MOVE, the timer for

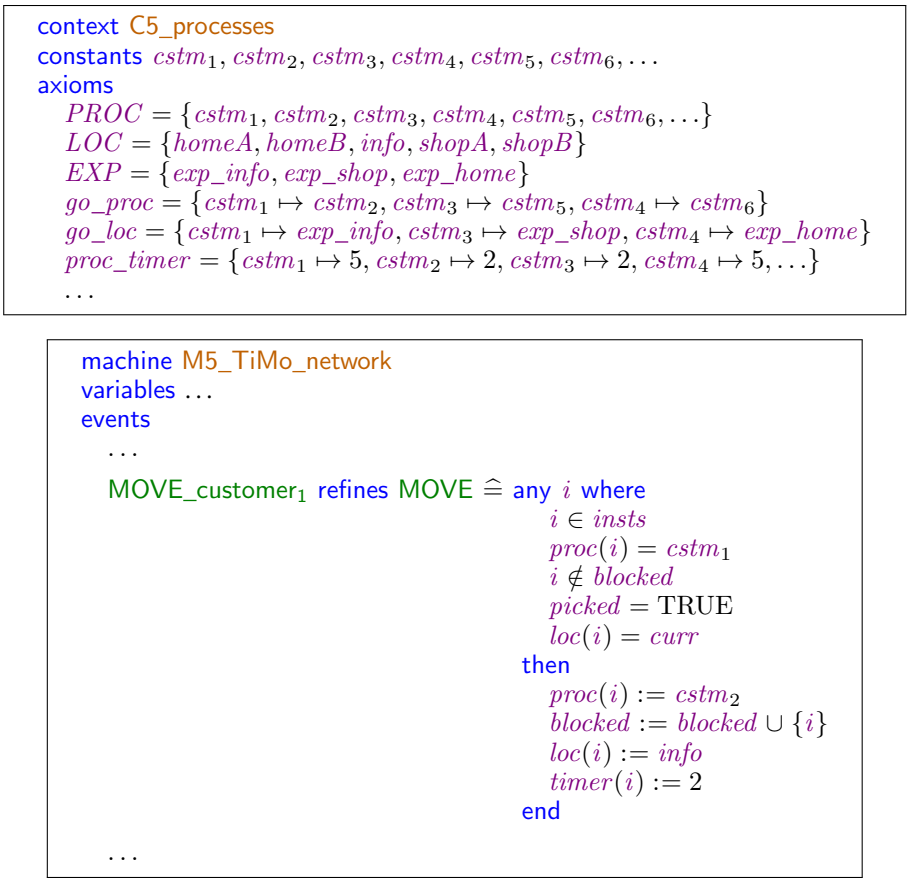

Fig. 8. The fifth refinement

the instance i is updated to $proc\_timer(go\_proc(proc(i)))$. For event TIME, we can now specify the condition under which time can progress at location given by the parameter l. For example, the first guard of TIME in Fig. 6 specifies that there cannot be a pending go-process which is timed out. Furthermore, we can now model precisely the set of instances that need to be updated by the TIME event: s is the set of timed out send-process instances, r is the set of timed out receive-process, and o is the set of instances that has not been timed out.

Fourth refinement (Fig. 7): Here we introduce the step semantics of TIMO. We add a Boolean variable $picked$ to indicate a location has been picked and variable $curr$ representing the current picked location. A new event pick is introduced to pick a random location that has some instances. New guards are added to MOVE to specify that the a location has been picked the instance i is at the current location $curr$. Flag $picked$ is unset in event TIME. Furthermore, parameter l is refined by variable $curr$. An important invariant of this refinement is @inv4_2 stating that there are no blocked instances when a location is not picked.

Our Event-B model at this stage corresponds to a generic TIMO network. In particular, the semantics of TIMO rules has been encoded into the corresponding events. For instance, event MOVE in Fig. 7 clearly captures the TIMO rule (MOVE) specified in Table. II. The formal proof of the equivalence between TIMO's operation semantics and semantics captured by our Event-B model is not within the scope of this paper.

Below we discuss how to instantiate the generic model to any concrete TIMO network.

Fifth refinement - encoding of a concrete TIMO *network (Fig. 8):* To obtain a model for a concrete TIMO network, we instantiate the generic constants introduced in the contexts earlier with concrete values corresponding to the actual network. For instance, the instantiation corresponding to the e-shop running example can be seen in context C5_processes in Fig. 8. For instance, constants $cstm_i$ represent different sub-processes of the TIMO $Customer$ process. For example, $cstm_1$ corresponds to the "$\text{go}^{\Delta 5}$ $info$" process, $cstm_2$ corresponds to the "$a^{\Delta 2}$? $(shop{:}Loc)$" process, $cstm_3$ corresponds to the "$\text{go}^{\Delta 2}$ $shop$" process, $cstm_4$ corresponds to the "$\text{go}^{\Delta 5}$ $home$" process. Constants exp_info, exp_shop, exp_home model TIMO constants *info* and variables $shop$ and $home$, accordingly.

Concrete machine M5_TiMo_network (Fig. 8) is a refinement of M4_TiMo_step_semantics, where the generic events are split according to the actual network. For instance, the MOVE_customer$_1$ event corresponds to the (MOVE) rule for process $cstm_1$. The fact that MOVE_customer$_1$ is a refinement of MOVE is guaranteed by the given instantiation of the carrier sets and constants.

V. Formal Analysis and Verification

We used the Rodin platform to verify the correctness and analyse different properties of the system. First of all,

TABLE III. PROOFS STATISTICS FOR THE EVENT-B MODEL IN FIG. 2

Model	Number of proof obligations	Automatically discharged	Interactively discharged
C2_syntax	16	16	0
C2_Init	1	1	0
C3_init	1	1	0
M0_TiMo_rules	35	34	1
M1_TiMo_locations	9	9	0
M2_TiMo_variables	19	10	9
M3_TiMo_timers	48	45	3
M4_TiMo_step_semantics	16	16	0
M5_TiMo_network	92	48	44
Total	237	180	57

we proved the correctness of different refinement levels by discharging the proof obligations generated by Rodin. The proof obligation statistics are presented in Table III. There is a high percentage of manual proofs for M2_TiMo_variables (due to the complexity in defining variables, assignments expressions, and evaluation) and M5_TiMo_network (due to the concreteness of instantiated values for constants). Note that if we consider only the generic model (i.e. without M5_TiMo_network), there are very few manual proof needed (13 out of 145, i.e., less than one tenth).

Analysis of the generic model: Having a generic encoding of the TIMO calculus enables us to verify general properties of the encoding using theorem proving. This would not be possible using model checking, which usually needs concrete instantiations or state enumeration. For instance, one of the properties that are manually proved in [10] (Proposition 3 in there) is the preservation of network well-formedness by the TIMO derivations (see Section II for definitions). Basically, network well-formedness states that there are no network free variables and that there are no blocked instances when the location is not picked (i.e., between two derivations). We capture network well-formedness property as invariants (@inv2_2, @inv4_2) in our Event-B model (see Fig. 5 and Fig. 7).

Analysis of the concrete model: Furthermore, we can analyse the running example, which is modelled in C5_processes, C5_concrete and M5_TiMo_network. We use the ProB model-checker [18] to verify some properties. Firstly, since the broker alternates his answer between 'shopA' and 'shopB', we check that

> **Property 1** *Customers cannot be at the same shop at the same time.*

Moreover, since the broker is always at location 'info', we want to verify that

> **Property 2** *Once a customer left home, s/he will not go home.*

ProB found counterexamples for both properties. In particular, **Property 2** does not hold because in process "$a^{\Delta 2}$? $(shop{:}Loc)$", the waiting time for the customer to receive an answer from the broker (i.e., 2 ticks) is insufficient. We emphasise here that fact that formal verification helps us to discover errors in our specification. To remedy this situation, we increase the customer waiting time to 3 ticks (instead of 2 ticks). This time ProB confirms that **Property 2** holds.

Analysis of the concrete parameterised system: Consider again **Property 2**. In fact, the customer waiting time depends on the number of customers in the network. We consider the system with an arbitrary number of customers, given as a parameter N to the system.[4] To verify **Property 2** for the extended parameterised version of the system, we relied on theorem proving functionality of Rodin instead of ProB. This is because model checking can only check the property for a specific value of N, and will not be able to explore the state space for a large value of N due to the classical state space explosion. For this verification, several invariants are invented (this is in constrast with verification using ProB for the non-parameterised version of the system). There are 306 proof obligations for the concrete machine, among which 117 POs are proved automatically (38%), the other are discharged interactively. The low percentage of automatic proofs indicates that **Property 2** is indeed a challenging property to prove for the parameterised system. In particular, the proofs involve reasoning about arithmetic and cardinality which are known to be difficult for automated theorem provers.

VI. CONCLUSION AND RELATED WORK

In this paper we provided an Event-B encoding of a timed mobility process calculus which uses local clocks, process migration, and local maximal concurrency of actions. The clocks take care of the relative time of migration and interaction of the processes residing at the same location. The modelling followed a formal refinement approach and enabled us to verify properties of such systems at both generic (framework properties) and concrete levels (system-specific properties). In particular, the encoding in Event-B of TIMO allows us to analyse the systems using both theorem proving and model checking techniques provided by the supporting Rodin platform. It is also important to note that our approach of having both generic model (capturing the syntax and semantics of the process calculus) and concrete model (corresponding to specific systems) is general and can be apply to encoding process calculi other than TIMO.

We formally specified in Event-B the notions of relative time, mobility and communication using the TIMO semantics. First, since Event-B does not incorporate a notion of time, we had to implement ourselves the concept of a local timer that can be reset. Our time modelling pattern is different from existing works dealing with time in Event-B. They usually adapt the time pattern introduced in [8], which simulates a global clock that can only go forward. This pattern based on a global clock is further refined to notions of deadlock, delay, or expiry in [20]. A variation of same time pattern in the presence of asynchronous communication, not synchronous as in our case, is given in [7]. Event-B specifications for local communication are proposed in [4] for web service compositions, and in [17] for mobile agents communicating via message routing algorithms. However, no explicit locations, migration or local timers are present in these works. Finally, [6] defines a method for encoding a process algebra in Event-B. However, they follow a very different approach from ours. Firstly, they do not give a generic encoding of the process algebra rules as we do, but a methodology based on refinement of constructing

[4] Strictly speaking, TIMO syntax does not yet support this notion of parameters.

the concrete Event-B model for a given process instance of the algebra. Essentially, they capture the algebra's syntax and semantics using machines. In our work, the syntax is modelled in contexts and the semantics is formalised separately using machines. Secondly, the process algebra exemplified in [6] includes none of the TIMO operators.

Regarding TIMO, there are several process calculi in the literature. However, we are not aware of any approach combining all these aspects regarding mobility with timing constraints, local clocks, and local maximal concurrency of actions. Several TIMO variants were developed during the last years: a probabilistic extension PTIMO [15], a real-time version RTIMO [5], and access permissions given by a type system in PERTIMO [14]. A flexible software platform was introduced in [11] to support agents allowing timed migration in a scalable distributed environment. Model checking capabilities are introduced for TIMO and its extensions in [13], [5], [15]. However, no theorem proving technologies for TIMO were available until now. In particular, our analysis of the parameterised e-shop system is out-of-scope for the model checking technique.

As future work, we plan to cover in our Event-B model some of the TIMO extensions, e.g., the access permissions as defined in [14]. This can easily integrated into the current development as an extra refinement. The model can be then checked against safe access properties. Moreover, we want to investigate dynamic aspects of such systems, in which process instances are created and destroyed on-the-fly, during the execution of the system. Dealing with dynamic aspects is usually a strength of theorem proving over model checking techniques. Finally, we will also plan to generalise our approach by providing a set of Event-B guidelines for encoding a process algebra in Event-B (in a similar vein with [6], but based on our pattern of generic and concrete dimensions).

Acknowledgments:: This work was supported by Romanian National Authority for Scientific Research through grants PN-II-ID-PCE-2011-3 no. 0688 and 0919.

REFERENCES

[1] Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996).

[2] Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010).

[3] Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6) (2010) 447–466. Tool webpage: http://rodin-b-sharp.sourceforge.net

[4] Aït-Sadoune, I., Aït-Ameur, Y.: Stepwise development of formal models for web services compositions: Modelling and property verification. T. Large-Scale Data- and Knowledge-Centered Systems **10** (2013) 1–33.

[5] Aman, B., Ciobanu, G.: Real-time migration properties of rTiMo verified in Uppaal. In: Proc. of SEFM'13. Volume 8137 of LNCS, Springer (2013) 31–45.

[6] Aït-Ameur, Y., Baron, M., Kamel, N., Mota, J.M.: Encoding a process algebra using the Event B method. STTT **11**(3) (2009) 239–253.

[7] Bryans, J.W., Fitzgerald, J.S., Romanovsky, A., Roth, A.: Patterns for modelling time and consistency in business information systems. In: Proc. of ICECCS'10, IEEE Computer Society (2010) 105–114.

[8] Cansell, D., Méry, D., Rehm, J.: Time constraint patterns for Event B development. In: Proc. of B'07. Volume 4355 of LNCS, Springer (2007) 140–154.

[9] Ciobanu, G., Koutny, M.: Modelling and Verification of Timed Interaction and Migration. In: Proc. of FASE'08. Volume 4961 of LNCS, Springer (2008) 215–229.

[10] Ciobanu, G., Koutny, M.: Timed mobility in process algebra and Petri nets. J. Log. Algebr. Program. **80**(7) (2011) 377–391.

[11] Ciobanu, G., Juravle, C.: Flexible software architecture and language for mobile agents. Concurrency and Computation: Prac. and Exper. **24**(6) (2012) 559–571.

[12] Ciobanu, G., Prisacariu, C.: Timers for distributed systems. *ENTCS* **164**(3) (2006) 81–99.

[13] Ciobanu, G., Zheng, M.: Automatic analysis of TiMo systems in PAT. In: Proc. of ICECCS'13, IEEE (2013) 121–124.

[14] Ciobanu, G., Koutny, M.: PerTiMo: A model of spatial migration with safe access permissions. Computer Journal (2014) to appear.

[15] Ciobanu, G., Rotaru, A.: A probabilistic logic for pTiMo. In: Proc. of ICTAC'13. Volume 8049 of LNCS, Springer (2013) 141–158.

[16] Hennessy, M.: A Distributed π-calculus. Cambridge University Press (2007).

[17] Kamali, M., Laibinis, L., Petre, L., Sere, K.: Formal development of wireless sensor-actor networks. Sci. Comput. Program. **80** (2014) 25–49.

[18] Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2) (2008) 185–203.

[19] Romanovsky, A., Thomas, M., eds.: Industrial Deployment of System Engineering Methods. Springer (2013).

[20] Sarshogh, M.R., Butler, M.J.: Specification and refinement of discrete timing properties in Event-B. ECEASST **46** (2011) 1–15.