MODELLING AND VERIFYING MIGRATION: A CASE STUDY ⋆

M. CHARPENTIER, M. FILALI, P. MAURAN, G. PADIOU, P. QUÉINNEC

ABSTRACT. We propose a minimal extension to the UNITY formalism to express mobility. Basically, we assign an explicit location to variables. This approach implies to define a semantic notion of mapping correctness. In this first proposal, we only use mobile variables and replicated code. This approach aims at providing a helpful expression of algorithms to refine and prove them more easily. To illustrate it, we develop a mobility-based mutual exclusion algorithm and prove its safety.

Keywords: mobility, UNITY, mobility semantics, theorem proving, mutual exclusion.

1. INTRODUCTION

The purpose of the UNITY formalism [CM88] is to provide a framework to reason about the *correctness* of programs. It is a general model that allows the specification of parallel programs from an operational point of view (writing program statements) as well as from a declarative point of view (stating program properties). The minimalism of the UNITY formalism is one of its strong qualities: it ensures its generality, while guaranteeing its simplicity of use.

We are interested in tuning this general framework to distributed algorithm specification and proof. We have first studied the expression of communication in UNITY [CFM+96]. This problem led us to define a more abstract notion than message exchanges through channels. Communications are described in terms of a relation between so-called image variables and source variables.

Nowadays, an other relevant aspect of distributed systems is data and code mobility. Code mobility is an old feature [BFPS85] which first appeared as a tool for load balancing in early distributed operating systems [BHEL88]. It becomes up-to-date again with the new generation of distributed platforms (Java for instance). In a UNITY program, instructions are not assigned to a specific host and so, code can always be considered as mobile. Our contribution consists in introducing mapping constraints ensuring the availability of all referenced variables of an instruction on a single node. The main part of the paper

⋆International Conference On Principles Of DIstributed Systems (OPODIS'99), pages 151–166.

is devoted to data mobility. In our proposition, data mobility takes the form of an explicit migration instruction. Both our propositions are well embedded in the UNITY logic, which doesn't need to be changed to handle our extensions.

Starting from a specification of the mutual exclusion problem, we describe our extension to express mobility in UNITY and apply this approach to refine a solution to the problem using a mobile token. Then, we formalize the semantics of mobility in the UNITY model. In Sect. 6, we develop a mobility-based mutual exclusion algorithm and prove its safety. Lastly, we outline some related works and compare them with our proposition

2. THE UNITY FORMALISM

The UNITY formalism consists of two parts: a programming language, based on transition systems, and a specification language, based on a linear temporal logic [MP92], to express safety and progress properties of this transition systems. This section describes parts of UNITY syntax and semantics useful to understand the remainder of the paper. A more comprehensive presentation can be found in [CM88] or [Mis94].

2.1. **The** UNITY **Language.**

A UNITY program F describes a state transition system. It consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements.

An **initially** section specifies initial values of variables in terms of a predicate, called init.F, expressing the possible initial states for the program.

An **assign** section, denoted by assign.F, describes the transitions as a set of assignments separated by the $[\!]$ mark. A UNITY statement is a variant of Dijkstra guarded multiple assignment. Its syntax is:

$$\begin{array}{rcl} v_1, \ldots, v_n & := & e_{11}, \ldots, e_{1n} \textbf{ if } c_1 \\ & \sim & \vdots \\ & \sim & e_{m1}, \ldots, e_{mn} \textbf{ if } c_m \end{array}$$

If an expression c_i is true then the expressions $e_{i1}, \ldots, e_{in}$ are atomically assigned to the variables $v_1, \ldots, v_n$. If none of the expressions c_i are true, then such a statement does nothing. This assignment must be deterministic: if c_i and c_j are both true, $e_{i1}, \ldots, e_{in}$ and $e_{j1}, \ldots, e_{jn}$ must yield the same values. Quantification is also allowed over statements to describe a family of assignments, written as $\langle [\!] v : condition :: statement(v) \rangle$.

Semantically, executing a UNITY program consists in choosing non-deterministically one assignment and then executing it atomically. This choice is repeated infinitely often, assuming weak fairness: each statement is executed infinitely often.

2.2. UNITY **program semantics.** Since UNITY statements are deterministic and terminate, we can model each statement as a function mapping a state to a state. Then, a statement has the following type:

$$statement : state \rightarrow state$$

A UNITY program F is typed as a couple $(\text{init}.F, \text{assign}.F)$ where the first element is the initialization predicate and the second element is the set of s-tatements:

$$Unity\_prog : (state \rightarrow bool) \times \textbf{set of } statement$$

A well-known operational semantics is $\mathcal{O}_1$, presented in [UK93]. The semantics $\mathcal{O}_1.F$ of a program F is the set of sequences σ of states which satisfy:

- $\text{init}.F.\sigma_0$
- $\langle \forall i :: \langle \exists A : A \in \text{assign}.F \ :: \ \sigma_i \ A \ \sigma_{i+1} \rangle \rangle$
- $\langle \forall A : A \in \text{assign}.F :: \langle \exists_i^\infty \ :: \ \sigma_i \ A \ \sigma_{i+1} \rangle \rangle$

2.3. **The** UNITY **Logic.** Given such a semantics, all the UNITY predicates can be defined exactly in the same way. We only define the operators of the UNITY logic used in the following sections in the operational semantics $\mathcal{O}_1$. In this operational semantics, the operators are equivalent to Sanders' [San91].

- `invariant` p: in any computation, p is true in the initial state and it remains true throughout all the computation:

$$\texttt{invariant } p \texttt{ in } F$$
$$\equiv$$
$$(init.F \ \Rightarrow \ p) \ \wedge \ \langle \forall \sigma, i : \sigma \in \mathcal{O}_1.F :: p.\sigma_i \Rightarrow p.\sigma_{i+1} \rangle$$

- $p \mapsto q$: if p is true, then q will eventually become true:

$$p \mapsto q \texttt{ in } F$$
$$\equiv$$
$$\langle \forall \sigma, i : \sigma \in \mathcal{O}_1.F :: p.\sigma_i \Rightarrow \langle \exists j : j \geq i :: q.\sigma_j \rangle \rangle$$

- p `until` q: if p is true, it will stay true at least until q becomes true, and q will become true (note that p may stay false continuously):

$$p \texttt{ until } q \texttt{ in } F$$
$$\equiv$$
$$\langle \forall \sigma, i : \sigma \in \mathcal{O}_1.F :: p.\sigma_i \Rightarrow \langle \exists j : j \geq i :: q.\sigma_j \wedge \langle \forall k : i \leq k \leq j :: p.\sigma_k \rangle \rangle \rangle$$

- p `detects` q: whenever p is true, q is also true, and, q becoming true ensures that p will eventually be true also:

$$p\ \texttt{detects}\ q\ \texttt{in}\ F\ \equiv\ \texttt{invariant}\ p \Rightarrow q\ \texttt{in}\ F\ \wedge\ q \mapsto p\ \texttt{in}\ F$$

3. Expressing Mobility

To express mobility, we must be able to describe the location of data and instructions. Therefore, we assume such entities own an attribute to identify their current location. For a UNITY variable or assignment E, we denote $E@$ its location. Such "location" variables have the same predefined type $Domain$. To describe algorithms, we use an injective function node from a natural interval $1..N$ to a domain of node names:

$$\text{node} : 1..N \longrightarrow \text{Domain}$$

We assume that UNITY assignments must be performed as local actions. Therefore, we only consider programs verifying a mapping constraint: namely, an assignment including mobile objects requires a "location guard" to ensure the availability of all referenced variables on a *single* node. This constraint can be considered as a proof obligation.

Initially, entities are placed on nodes. A migration is translated into an assignment to the location attribute of one (or more) entity. Therefore, mobility is embedded in the UNITY model without a new specific statement. For instance, if a variable X located on node(s) is moved to the node node(s'), this migration is translated into:

$$X@ := \text{node}(s')$$

However, we do not consider a migration to be an instantaneous action. The migration of a mobile variable is not considered as an atomic event and therefore the destination node will detect the effective migration with a delay. Thus asynchronous transmission delays are dealt with. We attach to each variable X, a family of predicates $X@$node(s) for all s, such that the family satisfies the following properties:

Definition 1. *If the location predicate $X@$node(s) is true, the variable X must actually be on the node node(s):*

$$\texttt{invariant}\ \ X@\texttt{node}(s) \Rightarrow X@ = \texttt{node}(s)$$

Moreover, we assume a migration eventually ends at the destination: more precisely, a migration must be detected by the destination node before a new *migration step for this variable can occur:*

$$X@ = \texttt{node}(s)\ \texttt{until}\ X@\texttt{node}(s)$$

We emphasize that $\langle \exists s :: X@\text{node}(s)\rangle$ is not an invariant. Actually, from a user point of view, $\langle \forall s :: \neg X@\text{node}(s)\rangle$ means that the mobile variable is in transit.

From this definition, we can deduce the following property:

Proposition 1. *The migration towards a node s is eventually detected:*

$$X@\texttt{node}(s)\ \texttt{detects}\ X@ = \texttt{node}(s)$$

An Example. As a basic example, we describe a well-known mobile object, a circulating token. We emphasize that a variable is mobile by using the keyword **mobile**. The token migration is implemented by an assignment to the location variable $Token@$. This assignment is replicated on every node. We use the **initially** section to set the initial location of the token on node(1).

```
Program TokenMigration
   type TokenState=null;
  Token : mobile TokenState;
 initially
  Token@node(1)        -- initial mapping
 assign
 ⟨ ‖ s,s':1..N :: Token@ := node(s') if Token@node(s) ⟩
end TokenMigration
```

This description satisfies the following safety property: the token is a single variable and so exists at only one location:

$$\texttt{invariant}\ \forall s, s' : 1..N :: \text{Token@node}(s) \wedge \text{Token@node}(s') \Rightarrow s = s'$$

However, this solution does not force the token to visit any node infinitely often, namely the following fairness property is not satisfied:

$$\forall s : 1..N :: \text{true} \ \mapsto\ \text{Token@node}(s)$$

This property is easily obtained by modifying the guard of assignments so that the token migration follows a fair visit of all nodes:

$$\langle \| s : 1..N :: \text{Token@} := \text{node}(next(s))\ \textbf{if}\ \text{Token@node}(s)\rangle$$

If the $next$ function is $next(s) = s \bmod N + 1$, the token circulates along a logical ring and the fairness property is ensured. In Sect. 6, we consider an example where the *next* function is encoded by *Hint* variables; thus, the token circulates along paths which are updated dynamically and always lead to a node requesting mutual exclusion.

4. A Token-Based Solution to Mutual Exclusion

The mobile token object may be used to implement a mutual exclusion service. The user abstraction remains unchanged but the resource is now granted to a process indexed by s located on node(s) when the mobile token is also on node(s). The non-ubiquity of the token ensures the mutual exclusion property.

Moreover, when the token visits a node, it serves at most one request before moving to the following node. We assign two different states to the token to obtain this fairness property. In the movable state, the token is allowed to circulate. When the token arrives at a node, its state becomes frozen. This means either the token serves an outstanding request before leaving the node or, if no request exists, the token eventually enters the movable state again and is able to leave the node.

```
Program TokenBasedMutex
 type TokenState = {movable, frozen};
 type UserState = {outside, requesting, inside};
 Token : mobile TokenState;
 User  : array [1..N] of UserState;
initially        -- initial mapping
   Token@node(1) ∧ ⟨∀ s : 1..N :: User[s]@node(s)⟩
 ∧ Token=frozen ∧ ⟨∀ s : 1..N :: User[s]=outside⟩
assign
⟨[] s : 1..N ::      -- User code
      -- User request
     User[s] := requesting if User[s]=outside
   [] -- Visit and serve (a pending request)
     User[s] := inside
         if User[s]=requesting ∧ Token=frozen ∧ Token@node(s)
   [] -- User exit
     User[s], Token := outside, movable if User[s]=inside
   [] -- Visit and skip (no pending request)
     Token := movable
         if User[s]=outside ∧ Token=frozen ∧ Token@node(s)
   [] -- Token migration
    ⟨[] s':1..N :: Token@, Token := node(s'), frozen
              if Token=movable ∧ Token@node(s) ∧ s'=next(s)
    ⟩
⟩
end TokenBasedMutex
```

Mapping this program on an distributed architecture with N nodes consists in specifying where the various variables and instructions are located:

- the variable `Token` is mobile and may be on any node;
- the s'th element of the array `User` is on node(s);
- assignments dealing with `User[s]` (reading or writing) are located on node(s);
- the last set of assignments dealing with the token migration can be considered as mobile (migrating with the token) or distributed on the N nodes.

The correctness of the solution relies upon the following invariants:

- If a user s located on node(s) is "inside", the token is on s and frozen:

(1)
$$\texttt{invariant}\ \forall s : 1..N :: \text{User}[s] = \text{inside} \Rightarrow \text{Token} = \text{frozen}\ \wedge\ \text{Token@node}(s)$$

- If the token is present on node(s_0), then other users are not inside:

(2)
$$\forall s_0 : \texttt{invariant}\ \text{Token@node}(s_0) \Rightarrow (\forall s : 1..N \wedge s \neq s_0 :: \text{User}[s] \neq \text{inside})$$

The specification of the token migration fairness remains unchanged.

5. Program Mapping Correctness

Definition 2 (Statement mapping correctness)**.** *A statement A is correctly mapped on a node n if only variables located at n are updated or referenced.*

Formally, a statement A is considered as a function from state to state. A statement A references (or uses) only variables located at n if it does not depend on variables located at other nodes, that is, the new state after executing A is independent of the values of the variables located on the other nodes.

Definition 3 (Program mapping correctness)**.** *A* UNITY *program is correctly mapped if all its statements are correctly mapped.*

We remark that mapping correctness cannot be verified syntactically in general and requires proof obligations. For instance, in the program $TokenBasedMutex$ of Sect. 4, the mapping correctness of the exit statement cannot be established syntactically; in fact, its mapping correctness proof obligation is:

$$\texttt{invariant}\ \text{inside}[s] \Rightarrow \text{Token@node}(s)$$

which is implied by (1).

6. A Tracker-Based Mutual Exclusion Algorithm

The preceding algorithm involves a continuous circulation of the token. When no request exists, this movement is useless. Therefore, several algorithms propose solutions to avoid such a drawback. Especially, we consider the algorithm based on path reversal [NTA96]. In this algorithm, a request issued from a node s is inserted into a fifo waiting queue. The privilege migrates from node to node along the path implemented by the queue.

Therefore, each node owns a partial view of the fifo queue:

- A hint of the current last node in the queue ($Hint[s]$). This hint is exact when $Hint[s] = s$;
- A knowledge of its state with respect to its rank in the queue: a boolean $Available[s]$ is true when the node is the head of the queue and the privilege is available.

For each node s, a mobile so-called $Tracker[s]$ variable is in charge of inserting the request into the queue and of returning home (starting node) with the privilege.

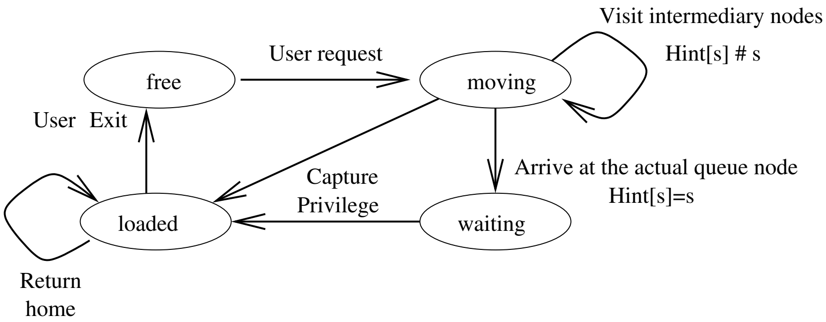

FIGURE 1. Tracker state transitions.

Figure 1 illustrates the state transitions of such mobile $Tracker[s]$ elements. When a local request occurs on a node s, the corresponding $Tracker[s]$ enters the moving state. If the privilege is free and local, the $tracker$ directly enters the loaded state. Otherwise, it migrates from node to node in the moving state until it reaches the actual queue node. Then, it stops there waiting for the availability of the privilege. Finally it captures the privilege and returns to its home node with it. The full program is given in Fig. 2.

The mobile trackers satisfy the following property:

```
tracker:
  ⟨∀ i :: User[i]=outside ⇒ Tracker[i]@node(i)⟩
∧ ⟨∀ i :: User[i]=inside ⇒ Tracker[i]@node(i)⟩
∧ ⟨∀ i :: User[i]=inside ⇒ Tracker[i]=loaded⟩
∧ ⟨∀ i :: Available[i] ⇒ Tracker[i]≠loaded⟩
∧ ⟨∀ i,j :: ((Available[i] ∨ Tracker[i]=loaded)
             ∧ (Available[j] ∨ Tracker[j]=loaded)) ⇒ i=j⟩
```

Global Invariant. The safety of the solution relies upon the following inductive *invariant*

$$\begin{array}{ll} & \langle \forall s :: User[s] = inside \Rightarrow Tracker[s] = loaded \rangle \\ \wedge & \langle \forall s :: (User[s] = outside) \equiv (Tracker[s] = free) \rangle \\ \wedge & \langle \forall s, s' :: Available[s] \wedge Available[s'] \Rightarrow s = s' \rangle \\ \wedge & \langle \forall t :: Tracker[t] \in \{loaded, free\} \Rightarrow Tracker[t]@node(t) \rangle \\ \wedge & \langle \forall t, s' :: Tracker[t] = loaded \Rightarrow \neg Available[s'] \rangle \\ \wedge & \langle \forall t, t' :: Tracker[t] = loaded \wedge Tracker[t'] = loaded \Rightarrow t = t' \rangle \\ \wedge & \langle \forall s, s' :: User[s] = inside \wedge User[s'] = inside \Rightarrow s = s' \rangle \end{array}$$

7. VALIDATION

With respect to mutual exclusion, we want to prove that:

$$\text{mutex:} \quad \forall i\ j :: \text{User}[i] = \text{inside} \wedge \text{User}[j] = \text{inside} \Rightarrow i = j$$

is *always true* (i.e. true over any execution). However, since such a property is not an inductive *invariant* (i.e. true initially and preserved by any statement), we have to strengthen it by providing a predicate str such that mutex ∧ str is actually invariant. Then, from the invariance of mutex ∧ str, it follows that mutex ∧ str is always true and consequently also mutex. As noted by Shankar [Sha93], although there is no algorithm to generate inductive invariants: “it requires invention and insight into why the system works”, there is a heuristic [SL92] based on weakest preconditions [Dij76] that often works.

In the following, we first show how we have synthesized an inductive invariant by applying heuristic steps and using our insight of the trackers’ semantics given in Fig. 1. Then, we describe the mechanization of this invariant synthesis within the PVS assistant theorem prover [COR+95].

7.1. **Invariant Synthesis.** First, let us recall that the basic idea of the heuristic proposed in [SL92] is to generate invariant requirements for a property `prop` to be always true with respect to a transition system (in our case a UNITY program). This generation is iterated until all the invariant requirements are

```
Program NT
 type    UserState = {outside, requesting, inside};
 type    TrackerState = {free, moving,waiting, loaded};
 Tracker   : array [1..N] of mobile TrackerState;
 User      : array [1..N] of UserState;
 Hint      : array [1..N] of 1..N;
 Available: array [1..N] of boolean;
initially         -- mapping
  ⟨∀s : 1..N :: ⟨User[s],Tracker[s],Available[s],Hint[s]⟩@node(s)⟩
 ∧⟨∀s : 1..N :: User[s]=outside ∧ Hint[s]=1
                ∧ Available[s]=(s=1) ∧ Tracker[s]=free⟩
assign    -- User code replication
 ⟨ ‖ s : 1..N ::
    -- User request
  User[s], Tracker[s] := requesting, moving
                                                  if User[s]=outside
 ‖  -- User enter. Serve an outstanding request
  User[s] := inside if Tracker[s]=loaded ∧ Tracker[s]@node(s)
 ‖   -- User exit
  User[s],Tracker[s],Available[s] := outside,free,true
                                                   if User[s]=inside
 ⟩
 ‖      -- Tracker migration description
 ⟨ ‖ t : 1..N ::
  ⟨ ‖ s : 1..N ∧ Tracker[t]@node(s) :: -- Tracker[t] on node(s)
    Tracker[t]@, Tracker[t],Hint[s], Available[s] :=
       -- a tracker t visits a non actual privileged node s (perhaps t=s)
     node(Hint[s]),moving,t,Available[s]
            if Hint[s]≠s ∧ Tracker[t]=moving
    ∼   -- a tracker t arrives at a tracking or privileged node
     node(s),waiting,t,Available[s]
            if Hint[s]=s ∧ ¬Available[s] ∧ Tracker[t]=moving
    ∼   -- a tracker t arrives at the node owning the privilege
     node(s),loaded,t,false
            if Hint[s]=s ∧ Available[s] ∧ Tracker[t]=moving
    ∼   -- a tracker captures the privilege
     node(s),loaded,Hint[s],false
            if Available[s] ∧ Tracker[t]=waiting
    ∼   -- a tracker leaves the node and returns at home
     node(t),loaded,Hint[s],Available[s]
            if Tracker[t]=loaded ∧ s≠t
  ⟩
 ⟩
end NT
```

FIGURE 2. Tracker-based mutual exclusion

satisfied. Initially, the set of invariant requirements is the singleton $\{\texttt{prop}\}$. Given a set of invariant requirements $\{R_0, \ldots, R_k\}$, R_i is satisfied with respect to a transition (a UNITY statement) tr if the two following predicates are true:

- init $\Rightarrow R_i$
- $R_0 \wedge \ldots \wedge R_k \Rightarrow \mathrm{wp}(tr, R_i)$

If the first predicate is not true, the heuristic has failed, if the second predicate is not true, a new invariant requirement is generated: $R_{k+1} = \mathrm{wp}(tr, R_i)$. If the iteration terminates, $R_0 \wedge \ldots \wedge R_n$ is an inductive invariant.

In the following tables, we summarize our use of the heuristic; the first one gives the name of the predicates composing the inductive invariant and the second gives the steps of the iteration process.

property	formula
mutex	$\forall i\ j :: (\text{User}[i] = \text{inside} \wedge \text{User}[j] = \text{inside}) \Rightarrow i = j$
synt_0	$\forall i\ j :: (\text{User}[i] = \text{inside} \wedge T[j] = \text{loaded}) \Rightarrow i = j$
synt_1	$\forall i\ j :: (T[i] = \text{loaded} \wedge T[j] = \text{loaded}) \Rightarrow i = j$
tracker	section 6

invariant requirements	unsatisfied transition	generated requirement	insight
{mutex}	UserEnter	synt_0	
{mutex,synt_0}	UserEnter	synt_1	
{mutex,synt_0,synt_1}	privilege capture		tracker
{mutex,synt_0,synt_1,tracker}	–	–	–

7.2. **Mechanization.** In this section we describe the mechanization of the safety proof of the tracker-based mutual exclusion described in Sect. 6. Our choice of the PVS system has been motivated by the fact that its type theory allows an easy description of parameterized systems. Here, the parameter N is the number of nodes.

We now give the main theories we have developed for the mechanization (see Fig. 3):

theory	contents
`NT_is_Mutex`	definition of synthesized and inductive invariants, validation
`NT`	algorithm description
`Mutex`	definition of mutual exclusion
`Tracker`	definition of a tracker
`Mobile`	definition of mobile data
`P_Invariant`	invariants pertaining to parameterized systems
`P_Trs`	Parameterized transition systems
`Trs`	basic transition systems, weakest preconditions calculus

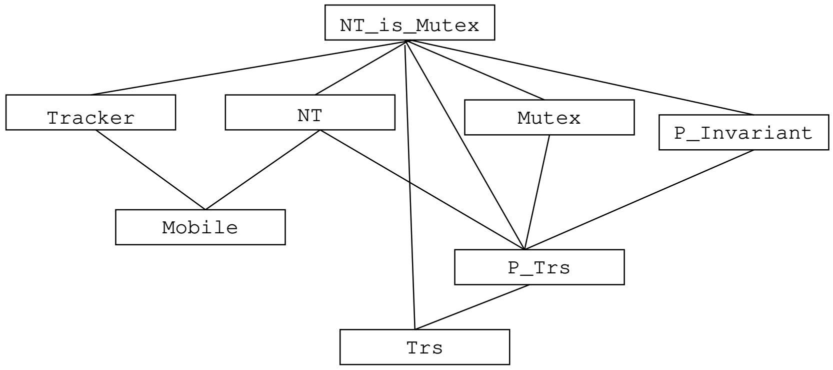

FIGURE 3. PVS theories hierarchy.

We have proven that the proposed invariant:

$$\text{mutex} \wedge \text{synt\_0} \wedge \text{synt\_1} \wedge \text{tracker}$$

is actually an inductive invariant and consequently the safety of the mutual exclusion algorithm is verified.

8. RELATED WORK

In the domain of state based formalisms, Mobile UNITY [PRM97] is a key contribution to mobility. With respect to classic UNITY, Mobile UNITY introduces two new concepts:

- *components* which are the unit of migration. A component is a usual UNITY program with a dedicated location variable λ.

- *interactions* for expressing communication and coordination between components (located at the same node or not). This aspect is very elaborated and proposes many constructs to handle specific distributed problems.

Actually, Mobile UNITY was first concerned by the dynamic reconfiguration of program components. The primary goal was to model different mobile code design paradigms (remote evaluation, mobile agent, code on demand . . .).

With respect to our approach, we are mainly concerned by proposing a formalism for modelling distributed protocols (not systems) where the explicit expression of mobility is more natural than the use of messages or shared variables. Moreover, we have adopted a simple semantics for migration which is restricted to variables, as well as for interactions which are restricted to be local. Thus, unlike Mobile UNITY, we have been able to stay within the standard UNITY operational model.

9. CONCLUSION

We have introduced an elementary extension to the UNITY formalism to describe mobility aspects of distributed computing. With this extension, we have developed and proved various mutual exclusion algorithms, two of which have been presented here.

With respect to our previous work [CFM+98] on introducing distributed aspects in UNITY, we have defined another concept: the observation [CFM+96]. It seems that there exists problems for which observations are well suited, e.g., stability problems and others for which mobility is well suited, e.g., distributed control problems. We are now dealing with a challenge: how to introduce both concepts in a coherent and homogeneous way?

REFERENCES

[BFPS85] C. Bétourné, M. Filali, G. Padiou, and A. Sayah. Distributed control through task migration via abstract networks. In *5th Int'l Conf. on Distributed Computing Systems*, pages 532–538, May 1985.

[BHEL88] A. Black, N. Hutchinson, Jul. E, and H. Levy. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 1(6):109–133, 1988.

[CFM+96] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Abstracting communication to reason about distributed algorithms. In *10th Int'l Workshop on Distributed Algorithms (WDAG'96),* volume 1151 *of Lecture Notes in Computer Science*, pages 89–104, October 1996.

[CFM+98] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Tailoring UNITY to distributed program design. In *Int'l workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA98)*, volume 1388 of *Lecture Notes in Computer Science*, pages 820–832, April 1998.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[COR+95] S. Crow, S. Owre, J. Rushby, N. Shankar, and S. Mandayam. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton*, http://www.csl.sri.com/pvs, April 1995.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs New Jersey: Prentice Hall, 1976.

[Mis94] J. Misra. A logic for concurrent programming. Technical report, The University of Texas at Austin, April 1994.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[NTA96] M. Naimi, M. Trehel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *Journal on parallel and distributed computing*, 34(1):1–13, 1996.

[PRM97] G.P. Picco, G.-C. Roman, and P.J. McCann. Expressing code mobility in mobile UNITY. Technical Report WUCS-97-02, Washington University, January 1997.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Sha93] A.U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[SL92] A.U. Shankar and S.S. Lam. A stepwise refinement heuristic for protocol construction. *ACM Transactions on Programming Languages and Systems*, 14(3):417–461, July 1992.

[UK93] R.T. Udink and J.N. Kok. On the relation between UNITY properties and sequence of states. In J.W de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications,* volume 666 of *Lecture Notes in Computer Science*, pages 594–608, 1993.

Authors addresses:

Mamoun Filali, Institut de Recherche en Informatique de Toulouse, France, filali@irit.fr

Michel Charpentier, Philippe Mauran, Gérard Padiou, Philippe Quéinnec, Institut de Recherche en Informatique de Toulouse – ENSEEIHT, France,

{charpov,mauran,padiou,queinnec}@enseeiht.fr