Using Mobile TLA as a Logic for Dynamic I/O Automata

SUMMARY Input/Output (I/O) automata and the Temporal Logic of Actions (TLA) are two well-known techniques for the specification and verification of concurrent systems. Over the past few years, they have been extended to the so-called dynamic I/O automata and, respectively, Mobile TLA (MTLA) in order to be more appropriate for mobile agent systems. Dynamic I/O automata is just a mathematical model, whereas MTLA is a logic with a formally defined language. In this paper, therefore, we investigate how MTLA could be used as a formal language for the specification of dynamic I/O automata. We do this by writing an MTLA specification of a travel agent system which has been specified semi-formally in the literature on that model. In this specification, we deal with always existing agents as well as with an initially unknown number of dynamically created agents, with mobile and non-mobile agents, with I/O-automata-style communication, and with the changing communication capabilities of mobile agents. We have previously written a TLA specification of this system. This paper shows that an MTLA specification of such a system can be more elegant and faithful to the dynamic I/O automata definition because the agent existence and location can be expressed directly by using agent and location names instead of special variables as in TLA. It also shows how the reuse of names for dynamically created and destroyed agents within the dynamic I/O automata framework can be specified in MTLA.

key words: dynamic input/output automata, mobile agent system, formal specification, spatio-temporal logic

1. Introduction

Input/Output automata (or I/O automata) [1] constitute a state-machine framework for the modelling and verification of concurrent systems. They have been extended to dynamic I/O automata [2] in order to be more suitable for the modelling of dynamic systems consisting of interacting components, such as mobile agent systems. The difference between the ordinary I/O automata and the dynamic ones is that the latter enable direct representation of the dynamic creation and destruction of agents and allow agents to change their sets of possible actions (i.e., their signatures) during system execution. Dynamic I/O automata only prescribe a mathematical model but no formal language for system description. For this reason, in [3] and [4] we tried to use the Temporal Logic of Actions (TLA) [5] for system specification based on a model similar to dynamic I/O automata. The advantage of using this logic is that it offers a formal language and proof rules for rigorous specification and reasoning, and that the same language can be used for the specification of systems and their required properties. Like (the ordinary) I/O automata, TLA does not contain any constructs for the specification of agent existence. Hence, we used special variables to indicate whether an agent exists or not. In [4], we succeeded in specifying a mobile agent system with dynamically created agents changing their locations and signatures. Neither the ordinary or dynamic I/O automata nor TLA support the specification of location changes. We, therefore, represented locations by additional state variables.

An extension of TLA, called Mobile TLA (MTLA), is proposed in [6] and more precisely elaborated in [7]. MTLA extends TLA by adding the ability to specify and reason about the mobility of agents between locations forming a tree, and it also has the potential to specify the dynamic creation and destruction of agents. The aim of this paper is to show how MTLA could be used to write specifications based on the dynamic I/O automata model. We do this by considering the specification of a travel agent system which is specified in [2] in a semi-formal precondition-effect style in order to illustrate dynamic I/O automata. We have already written a TLA specification of this system in [4]. It will, therefore, serve us as a starting point in this paper.

The paper is organised as follows. A brief informal presentation of MTLA is given in Sect. 2. Section 3 describes dynamic I/O automata to the extent needed in this paper. In Sect. 4, the travel agent system dealt with in this paper is first informally described. Its MTLA specification is then presented and thoroughly commented on. The specification approach is further discussed in Sect. 5. Section 6 concludes the paper.

2. A Short Presentation of MTLA

MTLA is a linear-time temporal logic similar to TLA, equipped with spatial modalities. It has been devised to describe the possible runs of a mobile system [7]. A run is an infinite sequence of configurations. Informally speaking, a configuration determines the system state at a discrete time instant. However, unlike the state in TLA, which just describes the current values of system variables, it also determines the spatial structure of the system. The structure is represented as a finite tree whose nodes are labelled with unique names drawn from a denumerably infinite set of names. The tree has a root node, which is labelled with a unique implicit label ε. The nodes are meant to represent named locations of agents or the agents themselves. Every
node has its own local variables, i.e., its local state. MTLA contains formulas for describing how the structure of the tree (i.e., which locations or agents exist and how they are located with respect to each other) and the local state of the nodes (typically the values of the agents’ local variables) change over time.

Possible changes of the local states are specified by temporal formulas of the form $\Box[A]_n$, asserting that every transition between consecutive configurations that modifies the expression $v$ must satisfy formula $A$. For example, formula $n.x = 0 \land \Box[n.x' = n.x + 1]_n$ might be interpreted as asserting that the value of variable $x$ of agent $n$ in the initial configuration is 0 and that every transition either increments this variable or leaves it unchanged. Figure 1 shows a run which satisfies it. The prime sign is used to denote the value of a variable at the next instant (i.e., in the next configuration) as in TLA. In MTLA, however, it has to be evident to which node ($n$ in our example) the variable belongs. In addition, MTLA takes into account that the node at which a formula is evaluated might not exist. The value of a variable at such a node is defined to be a constant but unknown “null” value. Another difference between MTLA and TLA is that $A$ may contain the usual linear-time temporal operators including $\Box$.

Possible changes in the structure of the tree are specified by the formulas of the form $\Box[A]_{a,n}$ where $n$ is a node name and $a$ is a sequence of node names. This formula says that every transition that removes or introduces node $n$ below the subtree indicated by $a$ must satisfy $A$. A formula $\Box[A]_{a,n}$ asserts that every transition that introduces node $n$ below $a$ has to satisfy $A$. In this paper, formulas specifying possible transitions, such as formula $A$ in the presented cases, will be called action formulas analogously to TLA.

The dotted notations in all the above formulas are, in fact, kinds of abbreviations coming from the use of MTLA spatial modalities $m[-]$ and $m(\cdot)$. Informally, formula $m[F]$ means that $F$ holds at node $m$, provided that $m$ exists. Formula $m(F')$ means that $F$ holds at node $m$ and that $m$ exists [6].

3. The Dynamic I/O Automata Model

We present the dynamic I/O automata model in a somewhat simplified way. Basically, the system consists of a number of signature I/O automata [2]. Every signature I/O automaton in the system has a unique identifier. A signature I/O automaton is, essentially, a kind of state-transition system with an action labelling each transition. It has a set of states and a set of start states, which is a subset of the former. The automaton’s transition relation determines which actions can be executed (i.e., are enabled) in a state and to which state an action execution leads. As usual, an action is said to be enabled (respectively, executed) in a state if a transition labelled with it exists in the transition relation (respectively, is taken) in that state. The signature I/O automaton has a state signature, which is a function of the automaton state. The state signature is partitioned into sets of input, output, and internal actions. Their union is the set of actions which can be used as transition labels in the current state. It follows that an action may be enabled in a state only if it is in the signature of this state. For any input action, if it is in the signature of a state, it has to be enabled in that state. Only one transition and, thus, only one action may be executed at a time.

The current state of the system (i.e., the global state) is determined by the states of the component automata (i.e., the local states). In every state, any two automata in the system must have disjoint sets of internal (and, respectively, output) actions in the state signatures. Two or more automata may have the same input action, and there may exist at most one automaton which has this action in its set of output actions. We say that such an output action is complementary to the input actions, and vice versa.

The system executes as follows. In each global state, only one action may be executed at a time. Every component which has this action in its current state signature has to participate in the transition, and all the other components do nothing. It follows that in any global state, the first possibility is that one automaton executes an internal action and all the others do nothing. The second possibility is that one automaton executes an output action for which no complementary input actions exist in the state signatures of the system components and the rest of the system does nothing. Another possibility is that automata with the same input action in the state signature for which no complementary output action exists execute the input action and all the other automata are idle. If there exist some automata with the same input action and one with the complementary output action in the current state signature, then these actions may only be executed simultaneously. In this way, the sending of a message by one automaton (with an output action) to some others (i.e., to those with the complementary input action) can be represented. If some automata execute complementary actions, the rest of the automata do nothing.

Not all the system components need exist from the beginning. In fact, the global state is called configuration and is determined by pairs (automaton identifier, local state) for all existing signature I/O automata. Signature I/O automata can have “create” actions in their signatures. A “create” action is a special kind of internal or output action for which no complementary action may exist in other components. It has the identifier of the automaton to be created as its parameter. If the “create” action is executed in a configuration, an automaton with this identifier is created if it does not yet exist, i.e., it occurs in the next configuration and can be in any one of its start states there. Any action may change the
current state signature of the automaton to which it belongs. If the state signature becomes empty, the automaton is immediately destroyed.

4. The Travel Agent System in MTLA

4.1 An Informal Specification

In [2], the dynamic I/O automata model of the travel agent system is described by using a semi-formal notation and textual comments. The system’s agents are represented by signature I/O automata. Variables are introduced to store their states, including locations, and state signatures. Their transition relations are specified by stating preconditions and effects of the transitions on the variables. In this subsection, we will describe the system textually, but fairly precisely, by following this specification and by using the symbols introduced in it (cf. [4]).

The travel agent system in [2] is structured as follows. It contains a stationary, i.e. always existing and non-mobile, client agent ClientAgt, located at location c, and N stationary database agents, located at disjoint locations d1, ..., dN and meant to control flight databases at these locations. ClientAgt has a constant state signature. It receives client requests to purchase a ticket for a particular flight, given by some “flight information” f via request(f) input actions. Each f is supposed to contain all the client information, including flight data, a maximal price mp the client is willing to pay, and an identifier which is unique across all client requests, so that each request can be identified by f in the model. ClientAgt accumulates these requests in variable reqs. For each new request f, it creates request agent ReqAgt(f) by internal action create(ClientAgt, ReqAgt(f)) (notice that the first parameter, i.e. the name of the creating agent, is not part of the model).

The request agent ReqAgt(f) handles the single request f. It tries to find a flight which conforms to f and whose price is at most mp by visiting the flight databases one after another. Upon creation, ReqAgt(f) is co-located with ClientAgt and has internal actions move1(c, d) for all possible locations d of the databases in its initial state signature. By executing one of them, it moves to some d, all the move actions are removed from the signature, and input actions informd(f, flts) and confd(f, fd, ok?) as well as output actions queryd(f), buyd(f, flts), and req-agent-response(f, fd, ok?) for all possible flight data fd, flts, and Boolean indications ok? are added to the signature. It queries the database d by queryd(f) and indicates this by setting transd to true. It receives its flight data flts by executing action informd(f, flts). The effect of this action is that the agent searches in flts for flights whose prices are smaller or equal to mp. If such a flight does not exist, it ends the querying in d by setting variable transd to false and by adding actions movef(d, dn) for all the databases dn not yet queried to the signature. After the inform action for the database and location d is executed, the request agent moves to another database (dn) by executing one of the new move actions if the query at d was unsuccessful, and the complete querying procedure repeats at the new location and database. If the query at d was successful, it sends the database d a request for a ticket for a flight from flts by executing action buyd(f, flts) and receives a response by executing action confd(f, fd, ok?). The effect of this action is as follows: If the buying is successful (ok? is true), the flight ticket information fd is stored to variable tkt. Both in this case and if the answer is negative (in this case, the value of fd is undefined) and all N databases have been queried, the status of the transaction is marked as “computed”. Variable transd is set to false in any case. If the status is “computed”, ReqAgt(f) sends a positive (i.e. with ok? equal to true) or, respectively, a negative response (i.e. with ok? set to false) to ClientAgt from the current location by executing output action req-agent-response(f, fd, ok?), which empties the state signature and destroys ReqAgt(f).

We do not give a description of the database agents operation because it is also not given in [2]. Let us just assume that the database agent signatures are constant and contain complementary actions for all the actions used for database communication by any request agent.

Notice that all actions of the travel agent system have parameters. The parameters are used to model the sending of messages. Let us explain this by the example of sending the response to ClientAgt. Its state signature contains input actions req-agent-response(f, fd, ok?) for all possible values of f, fd, and ok? in the system. By definition, these input actions are always enabled. ReqAgt(f) may execute action req-agent-response(f, fd, ok?) for some f, fd, and ok? only if ClientAgt executes it simultaneously as an input action. In this way, the latter receives the response values f, fd, and ok?. It adds this triple to set resps and subsequently communicates it to the client via output action response(f, fd, ok?). At this point it also removes all record of the request from resps.

4.2 The MTLA Specification

Now, we will present the MTLA specification of the travel agent system and indicate the advantages of using MTLA instead of TLA. The reader will surely recognise the correspondence between the symbols from the informal description of the travel agent system and the MTLA specification.

Although locations are not part of the dynamic I/O automata model, we take advantage of MTLA and specify them as special nodes. We use the set of names D = {d1, ..., dN} to denote the database locations. Let Dim = {“d-1”, ..., “d-N”} be the set of their identifiers. The identifier of the client agent location named c is “c”. These identifiers are needed in the request agent action specifications because of the restricted use of MTLA names (cf. [8]). The names cannot be used as values (and vice versa). Let FInf and Bool denote the set of possible flight data and request success indications, respectively.

The MTLA formula ICAGt(clag, RAG) in Fig. 2 is an MTLA specification of the client agent. The first line first
the local variables are the same as in the TLA configuration. For the specification of the dynamic creation of agents, we follow the approach sketched at the end of [7]. There, the extension of MTLA by rigid quantification over names belonging to a set was proposed. As the number of dynamically created agents is not known in advance, we suppose that their names will be chosen from a set of names $\text{RAG}$ which are different from all the other names used in the travel agent system specification. It is specified that initially, no agent from this set exists.

The second line of the client agent specification says that $c\text{lag}$ can only execute the actions specified by formula $\text{ClAgtN}(c\text{lag}, R\text{AG})$. In fact, this formula specifies two kinds of actions: those that the agent itself can perform ($\text{ClAgtOwn}$) and those that its environment can perform ($\text{ClAgtOth}$). This is necessary in order for the MTLA system specification to be practically a conjunction of component specifications as in TLA. All the actions in the MTLA specification of the travel agent system are specified in a similar way to TLA [4]. As in the latter, a variable $Ev$ common to all the system components is used to indicate which dynamic I/O automaton action is meant to be specified by an MTLA action formula. However, like every variable in MTLA, $Ev$ has a location in MTLA. $Ev$ is short for $\epsilon.Ev$, which means that $Ev$ is located at the root node of the MTLA tree. For example, the $\text{InRequest}(c\text{lag}, f)$ MTLA action formula specifies the dynamic I/O automaton input action $\text{request}(f)$. $\text{Act}(Ev, a)$ could be defined as the action formula which is true iff action name $a$ (such as, for example, (“request”,$\langle f \rangle$) in the $\text{InRequest}(c\text{lag}, f)$ action formula) is appended to the sequence $Ev$, as in [4], or in any other way which would ensure that $Ev$ changes in different ways for different actions $a$, because this is what basically matters (cf. [9]). $\text{InRequest}(c\text{lag}, f)$ and the three action formulas following this one in Fig. 2 are parameterised action specifications. All the possible values for their parameters and, thus, all the possible actions of the client agent itself are specified by $\text{ClAgtOwn}$. By the definition of dynamic I/O automata, in any possible global state of the system containing an automaton, the automaton’s environment in the system cannot participate in an action which is not in the automaton’s state signature, without the latter participating in the action. It follows that at any time, either the client agent participates in one of its own actions (this is specified by $\text{ClAgtOwn}$) or it is idle and the environment participates in an action which is not in the client agent signature (this is specified by $\text{ClAgtOth}$). In the travel agent system, the environment of the client agent consists of the database agents and dynamically created request agents with changing signatures, and similar for the other agents. Therefore, $\text{Acts}$ denotes the set of all actions which can ever occur in the signatures of the travel agent system components (formally, all the actions from their universal signatures [2]). Observe that for simplicity, in this paper we do not specify exactly the agent signatures, but rather the union of their elements (e.g. $\text{UClAgt}$ for the client agent). It should also be noticed that similar to TLA, MTLA formulas of the form $\forall\exists[A]_v$ allow transitions which do not change $v$. It follows that our MTLA specifications of dynamic I/O automata allow, besides their regular actions, self-looping unlabelled transitions (because $Ev$ may remain unchanged). This, however, does not pose a problem because the formulas do not distinguish between systems having such transitions and those without them ([7], [8], cf. [10]).

The main advantage of using MTLA instead of TLA for dynamic I/O automata specification is that in MTLA
we do not need Boolean variables to express existence of signature I/O automata. Consequently, in action formula \(\text{IntCreate}(\text{clag}, \text{reqag}, f)\), the creation of a new request agent \(\text{reqag}\) at location \(c\) can be expressed naturally by \(\sigma_c.\text{reqag}(\text{true})\), which requires that \(\text{reqag}\) exists in the next configuration and is in accordance with the definition of dynamic I/O automata. For TLA specifications, we used an adapted dynamic I/O automata model. In TLA, we had to take that all components of a system “exist” all the time except that those that are not yet created have an activation variable which is false if the component does not really exist and true otherwise. In order to create a component, another one should set its activation variable to true, but this would be a kind of shared variable communication. In order for the model to be more in the spirit of I/O automata, we introduced input create actions in the “non-existing” components with the effect of setting their activation variables. In this way, a component could create another one by executing an output create action simultaneously with the input create action of the other.

The MTLA action formula \(\text{IntCreate}(\text{clag}, \text{reqag}, f)\) also requires that the new agent behave as specified by formula \(\text{IReqAgt}(\text{reqag}, f)\), which is given in Fig. 3. We follow the semi-formal specification in [2] in that every request agent is uniquely identified by \(f \in \text{Finf}\) upon creation, but the latter cannot be applied as a node name in MTLA. For this reason, every newly created request agent is represented by a name from \(\text{RAG}\) not currently in use for the representation of an existing agent. The names can be reused: request agents for different requests can have the same MTLA name if they do not exist at the same time. The last line of \(\text{IClAgt}\) specifies that request agents can only be created by the create actions of the client agent.

Although we knew in advance that the client agent has a constant state signature, we specified it by local variables \(\text{in}, \text{out}, \text{and int}\) to be consistent with the dynamic I/O automata model. The first conjunct in the third line of \(\text{IClAgt}\) specifies that the state signature is constant. The next conjunct specifies that the client agent is stationary.

Once created, the request agent \(\text{reqag}\) for the request \(f\) has to behave as specified by formula \(\text{IReqAgt}(\text{reqag}, f)\) in Fig. 3. It has a changing state signature, which is represented by \(\text{UReqAgt}(\text{reqag})\). The first conjunct of formula \(\text{IReqAgt}(\text{reqag}, f)\) specifies initial values of \(\text{reqag}\)'s local variables including the agent identifier and the state signature. The second conjunct specifies possible actions of \(\text{reqag}\). In a similar way to the client agent, the specification of all the possible actions \(\text{ReqAgtN}(\text{reqag}, f)\) consists of two disjuncts: \(\text{ReqAgtOwn}\) specifies those actions which can be executed by the agent and \(\text{ReqAgtOth}\) the allowed actions of the environment. The detailed specification of the agent’s actions is in Fig. 4. The specification is very similar to the specification in TLA, except for the use of location and agent names. In TLA, the local variables of every request agent were components of arrays representing the states of all possibly existent request agents for different values of \(f\). In MTLA, the local variables of the request agent \(\text{reqag}\) are simply identified by the name \(\text{reqag}\) preceding the variable name itself. Notice that some symbols denoting action formulas are indexed by variables representing location names. This is in accordance with the examples of MTLA specifications given in the literature ([6]–[8]).

In Fig. 4 it can be seen that \(\text{reqag}\) moves if an action specified by an \(\text{IntMove}f\) formula is executed and destroys itself by an action specified by \(\text{OutReqAgtResp}\). In any other action, including the environment ones, the \(\text{Stationary}\) predicate (cf. [8]) is used to specify that \(\text{reqag}\) remains where it is. It follows that although the third conjunct of \(\text{IReqAgt}\) contains formula \(\text{ReqAgtN}(\text{reqag}, f)\) for generality, it specifies that \(\text{reqag}\) can only change location or be destroyed by the actions described by \(\text{IntMove}\) and \(\text{OutReqAgtResp}\).

It should be noticed that the second and third conjuncts of \(\text{IReqAgt}\) cannot have just the form \(\Box \text{ReqAgtN}(\text{reqag}, f)\\text{V\text{IReqAgt(reqag)}}\) and, respectively, \(\Box \text{ReqAgtN}(\text{reqag}, f)\\text{I\text{IReqAgt(reqag)}}\) for all the possible locations because this would mean that \(\text{vReqAgt(reqag)}\), which includes \(\text{Ev}\), and \(\text{l Reqag}\) can only be changed by an action related to \(\text{reqag}\) identified by \(f\). This agent, however, can eventually destroy itself. For this reason, we have to use
the implication \( \text{reqag}(id = f) \Rightarrow \text{ReqAgtN}(\text{reqag}, f) \) instead of just \( \text{ReqAgtN}(\text{reqag}, f) \). In this way, it is required that if agent \( \text{reqag} \) identified by \( f \) exists, then if \( \text{Ev} \), local variables, or the location of \( \text{reqag} \) change, they change according to \( \text{ReqAgtN}(\text{reqag}, f) \). Otherwise, the changes are determined by the specifications of other agents.

Formula \( I\text{Sys}(RAG) \) in Fig. 5 is the specification of the complete travel agent system. The first and second lines specify that the system contains \( N \) database locations and location \( c \) all the time, that they are not nested in one another (i.e. each of them is a separate location), and that they have unique constant identifiers (cf. [8]). The conjunct \( DB\text{Ag}(d_i) \) is meant to be an MTLA specification of the stationary database agent at location \( d_i \) (in fact, here we identify the locations by the database agents themselves, but special nodes representing the agents could be specified), which we do not give in this paper and is also not specified in [2]. The MTLA specifications of the database agents would be similar to the client agent one without the agent creation. The next conjunct is the specification of the client agent which, in
contrast to the TLA specification, also comprises the specification of dynamically created request agents.

\( ISys(RAG) \) without the last line basically guarantees that the specified system behaves as required by the model definition in Sect. 3. At most, only one action from the union of state signatures of currently existing components can happen at a time since \( Act(\mathcal{E}_v, a) \) is supposed to change \( \mathcal{E}_v \) in a different way for every \( a \). All the agents which have the executed action in the current state participates in its execution. This is ensured by the environment action specifications of the client agent, the database agents and \( ReqAgIoth(reqag, f) \) (together with the antecedent \( reqag(id = f) \)) for all possible request agents. The formulas prohibit an action \( a \) from the state signature of an (existing) agent being executed (i.e., that \( Act(\mathcal{E}_v, a) \) is true) without this agent participating in the execution. Communication is specified properly. For example, if \( OutReqAgIothResp(reqag, f, fd, ok) \) of an existing \( reqag \) identified by \( f \) is true, then \( InReqAgIothResp(clag, f, fd, ok) \) of the client agent is true simultaneously, which has the effect that the triple \( (f, fd, ok) \) is communicated to the client agent. However, \( ISys(RAG) \) without the last line allows actions different from those in the current state signatures of existing components to happen when the latter do nothing. For example, a “move” action which represents the movement of a non-existing request agent can be executed, or a “move” action which represents the movement of an existing request agent, but is not in its current state signature, can happen. A general solution to eliminate such illegal action executions in MTLA is to require that \( Ev \) can only be changed by the transitions of existing system components, as in the last line of \( ISys(RAG) \) (cf. [8]). It should be noticed that the problem of illegal actions does not occur if all the components of the specified system exist all the time and have constant state signatures. In that case, in contrast to [8], such a requirement is not needed, but the model of such a system is, in fact, an ordinary I/O automata one, even if it includes mobility.

5. Discussion

\( ISys(RAG) \) is an MTLA counterpart of the semi-formal specification of the travel agent system given in [2]. It is more elegant and closer to the latter than the TLA specification because of the use of agent names. Although locations are not part of the dynamic I/O automata model, we presented them as MTLA nodes. This made writing the specification a little more difficult because the node names could not be used as values in the action specifications. We, therefore, introduced location identifiers as in [8]. The use of location names also forced us to precisely specify the positions of the locations with respect to each other and the duration of their existence. This specification is entirely absent from the semi-formal and TLA specifications although it might be necessary for proper verification. We could, of course, specify locations and mobility in MTLA just by special variables of agents (e.g., \( reqag.loc \)) in a similar way to [2] and [4], but such a specification would be less intuitive.

[8] proposes how to write MTLA specifications of Mobile UML state-machines, which include mobility but neither dynamic creation of objects nor changing communication capabilities. By looking at the MTLA travel agent system specification, it can be seen that dynamic I/O automata MTLA specifications can basically be written in a similar way to them, but some important differences exist. Firstly, dynamic I/O automata specifications need variables for storing state signatures. As in the specifications of UML state-machines, we use a root variable which can be accessed by all system components. However, it has a different role than in the former. There, it is only needed for modelling communication. Because it is asynchronous, this variable does not have to be included in the list of component variables denoted by \( v \) in the specification of the form \( \Box [A]_v \). The action formula \( A \) representing possible actions also does not have to include a part specifying the actions of the component’s environment.

In the specifications of dynamic I/O automata, the variable has to represent the label of every executed action and ensure the synchronous action execution. For this reason, it has to be included in \( v \), and the environment action allowed with respect to the current state signature have to be specified for every component.

As already mentioned, in the MTLA specification of the dynamic creation and destruction of agents in the dynamic I/O automata model, we can basically follow [7]. The latter also uses a travel agent system as an example, but it is different from the one in this paper and is also based on a different model. When creating an agent, for example \( n \), we cannot just require that a formula of the form \( \Box [A]_w \) where \( A \) describes possible actions of the agent (and similarly for location changes) should hold from now on as in [7], if the agent mentioned in the formula can be destroyed in the future. We first believed that we could instead require that \( n(\Box [A]_v) \) should hold. Informally, this formula asserts that \( n \) exists (now) and that \( [A]_w \) holds at \( n \) as long as \( n \) exists, i.e., even if it is eventually removed from the tree and even if \( n \) reappears later and \( [A]_w \) does not hold there [7]. For example, formula \( \Box [n.x = n.x + 1]_{n.x} \) would not hold for the run in Fig. 1 if node \( n \) was removed from its third configuration, whereas formula \( n(\Box [x' = x + 1]_{x'}) \) would, even if in addition, \( x \) at \( n \) existing from the fourth configuration on was incremented by six. It is, however, only possible to use the variables of \( n \) and the names of nodes located below it in the tree in \( [A]_w \) because only their values are defined there. It follows that at least \( Ev \) could not be used in \( A \) and \( v \) because it is above \( n \) in the tree. As shown in Fig. 3, the solution is to use \( \Box [N \Rightarrow [A]_w] \) where \( N \) expresses the condition that the specified agent exists.

6. Conclusion

We showed how systems based on the dynamic I/O automata model which include dynamically created and mobile components, as well as changing state signatures, can be specified in MTLA. The presented specification of the travel
agent system suggests that they can be specified in a rather uniform way. Of course, the main purpose of using MTLA for the specification of dynamic I/O automata is to be able to carry out formal verification in the same language. The advantage of MTLA over TLA in this respect is that it offers notation and proof rules for directly expressing and verifying properties which include nodes (e.g. signature I/O automata names, locations). A disadvantage, though, is that no tool support for verification using MTLA exists yet and that its proof rules are not yet as elaborated as for TLA. Verification of specifications such as the one presented in this paper by proving properties in MTLA will be investigated in the future.

References

[1] N.A. Lynch and M.R. Tuttle, “Hierarchical correctness proofs for distributed algorithms,” Proc. 6th ACM Symp. on Principles of Distributed Computing (PODC’87), pp.137–151, Vancouver, British Columbia, Canada, Aug. 1987.
[2] P.C. Attie and N.A. Lynch, “Dynamic input/output automata: a formal model for dynamic systems,” Proc. 12th Intl. Conf. on Concurrency Theory (CONCUR 2001), pp.137–151, Aalborg, Denmark, Aug. 2001.
[3] T. Kapus, “Modelling of agent computations using the temporal logic of actions,” Proc. Intl. Conf. on Software, Telecommunications and Computer Networks (SoftCOM 2002), pp.345–349, Split, Croatia, Oct. 2002.
[4] T. Kapus, “Mobile agent system specification using the temporal logic of actions,” Proc. IASTED Intl. Conf. on Software Engineering, pp.319–324, Innsbruck, Austria, Feb. 2005.
[5] L. Lamport, “The temporal logic of actions,” ACM Trans. Program. Lang. Syst., vol.16, no.3, pp.872–943, March 1994.
[6] S. Merz, M. Wirsing, and J. Zappe, “A spatio-temporal logic for the specification and refinement of mobile systems,” Proc. 6th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2003), pp.87–101, Warsaw, Poland, April 2003.
[7] J. Zappe, Towards a Mobile TLA, Ph.D. thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, Sept. 2005.
[8] A. Knapp, S. Merz, M. Wirsing, and J. Zappe, “Specification and refinement of mobile systems in MTLA and mobile UML,” Theor. Comput. Sci., vol.351, no.2, pp.184–202, Feb. 2006.
[9] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel, “Lazy caching in TLA,” Dist. Comp., vol.12, no.2/3, pp.151–174, June 1999.
[10] J.F. Søgaard-Andersen, N.A. Lynch, and B.W. Lampson, Correctness of Communication Protocols — A Case Study, Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, MIT, Cambridge, USA, and Technical Report ID-TR 993-129, Department of Computer Science, Technical University of Denmark, Lingby, Denmark, Nov. 1993.

Tatjana Kapus received her M.Sc. and Ph.D. degrees from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 1991 and 1994, respectively. She is now an associate professor there. Her primary research area are formal methods for specification and verification of reactive systems.