Monitoring Hierarchical Agent-based Simulation Traces

Benjamin Herd, Simon Miles, Peter McBurney, Michael Luck
Department of Informatics
King's College London
London, United Kingdom
{benjamin.c.herd, simon.miles, peter.mcburney, michael.luck}@kcl.ac.uk

ABSTRACT

Due to their internal complexity, agent-based simulations are rarely amenable to conventional formal verification. With its focus on individual traces, *runtime verification* represents an interesting alternative for correctness assessment. Here, execution traces produced by the running system are observed by a monitor and checked for correctness *on-the-fly*. If the truth or falsity of a given property cannot be determined at time t, then the monitor creates an *obligation* that the current trace needs to satisfy at time $t + 1$ in order for the whole property to become true.

With different observational levels, traces produced by agent-based simulations have an inherently *hierarchical* nature which complicates the structure and manipulation of obligations significantly. However, it turns out that this problem is general enough to be dealt with in an abstract, language-independent way.

In this paper, we provide a general framework for the monitoring of hierarchical traces. It introduces different types of obligations and appropriate manipulation procedures along with minimal requirements that a property specification language needs to satisfy in order to be monitorable. We provide a full formalisation of the framework and an example implementation of a subset in Haskell.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Verification, Theory, Algorithms

Keywords

Runtime verification; agent-based simulation; monitoring; temporal logic

1. INTRODUCTION

Agent-based simulation (ABS) is rapidly emerging as a popular paradigm for the simulation of complex systems that exhibit non-linear and emergent behaviour [14]. Similar to other software systems, *correctness* also plays a central role in the development of ABSs and the demand for tailored verification techniques increases.

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, 2015, Istanbul, Turkey.*
Copyright © 2015, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

In the more general area of multiagent systems (MAS), temporal logic model checking has been used successfully to verify a wide range of properties [13]. Despite impressive advances made in recent years, exponential growth of the underlying finite-state model (the so-called 'state space explosion') remains a central problem which renders the verification of large-scale real-world systems difficult or even impossible. Due to their high level of complexity, the exhaustive verification of ABSs is, in general, far beyond the capabilities of contemporary model checking techniques and tools.

Runtime verification represents a promising alternative for the evaluation of large-scale systems [8]. It refers to the idea of observing the execution of a program at runtime and perform correctness checks on-the-fly. A central component of any runtime verification approach is a *monitoring process* which observes an execution trace against the background of a given correctness criterion (e.g. a linear temporal logic formula) and reports for each state in the trace whether the given property has already been satisfied or violated or whether a clear verdict cannot be made yet. In the latter case, an *obligation* is produced; it describes those aspects of the original property that the system needs to satisfy in the next step in order to satisfy the overall property.

Runtime verification of software systems has been investigated extensively in the past twenty years, yet in the area of MAS in general and ABS in particular, approaches are still largely missing. One reason for that is complexity. Conventional runtime verification assumes traces to be *monolithic*, i.e. composed of atomic, individual states. In a multiagent setting, however, traces have a more fine-grained internal structure: Traces still represent sequences of states (*population states*) yet, at the same time, they also represent sets of sequences of *group states* as well as sets of sequences of *individual agent states*. This has significant implications on the nature of obligations and their manipulation at runtime. Whilst some existing approaches to runtime verification and monitoring in the area of general MAS have to deal with certain aspects of obligation manipulation, a general and specification language-independent treatment of the monitoring problem is still missing. It turns out that the nature of obligations in a multiagent context as well as their manipulation is *independent* of the underlying specification language and can (and should) thus be treated *entirely separately* from the latter's design and implementation.

This paper makes the following contributions:

- we present a *language-independent framework for the monitoring of hierarchical traces* with a particular focus on the *manipulation of obligations*. The framework serves as an *abstract description of a monitoring procedure* which, due to its general nature, can be used as a blueprint for developers to implement concrete monitors for different specification languages;

- we give a *full typechecked formalisation* of the framework using the Z notation; and
- we show the correspondence of the abstract framework and an implementation by translating a subset of the framework into executable Haskell code.

We start with an overview of related work in Section 2 and some background in Section 3. We then provide an informal description of the idea of hierarchical traces in Section 4 which is succeeded by a formal treatment of the obligation manipulation problem in Section 5. Instead of tying the monitoring problem to a particular specification language, we aim to be as *general* as possible and treat the problem in a *language-independent* way. The purpose of this endeavour is to isolate the problem of obligation manipulation in order to allow a language designer to focus solely on the core language. We do this by developing a *monitoring framework for ABS traces* for the purpose of runtime verification together with *requirements* that a specification language needs to satisfy in order to be usable within the framework. To this end, we introduce the concepts of *joinability*, *agent-monitorability*, and *multiagent-monitorability*. In order to show the closeness of the framework to an executable implementation, we translate a subset of the framework into Haskell code. The integration of a simple example specification language into the framework is illustrated in Section 6.

2. RELATED WORK

In the wider area of MAS, runtime verification has not yet been dealt with in a general way. Whereas a range of problem-specific and language-specific testing and debugging approaches have been proposed for general MAS [9], their simulation counterparts have been largely neglected to date. Simulation-specific monitoring and testing approaches that have been presented tackle the monitoring problem in an ad hoc fashion [16, 21, 4]; formal approaches to monitor construction are still missing.

The approach closest to ours is the one presented by Dastani and Meyer [5]. They propose MDL, a specification language for the formulation of properties about BDI-based multiagent programs together with a debugging, i.e. runtime verification procedure. MDL allows for the specification of temporal properties about BDI-based systems and, in addition to temporal operators, provides a means to limit the *scope* of a property to an individual agent, to a group of agents, or to the overall population[1]. As opposed to the approach described above, Dastani and Meyer give MDL a two-valued semantics, according to which properties are either true (if clearly satisfied) or false (if either violated or not yet satisfied). The algorithm is thus not optimal w.r.t. the early detection of violations. Furthermore, as opposed to producing obligations, they seem to re-verify the entire trace prefix produced so far once a new state has been processed. Giving t time steps, $(t+1)/2$ states thus need to be verified. This offline nature of the verification procedure circumvents the obligation problem discussed in this paper, yet it also renders the approach non-optimal w.r.t. the number of checks necessary to answer a given property.

Sharpanskykh and Treur presented *TTL*, the *temporal trace language* [17]. It subsumes languages based on temporal logics and differential equations and thus allows for the expression of properties which go beyond purely temporal statements. The specification language takes into consideration the hierarchical nature of traces obtained from MAS executions and offers the possibility to observe behaviours on different levels of observation. The authors also describe a verification procedure on sets of traces. However, the procedure is *exhaustive* in nature, i.e. it expects full traces to be present prior to verification. Similar to Dastani and Meyer's work described above, this circumvents the need to deal with obligations.

[1] In that respect, it bears a strong similarity with *simLTL*, a property specification language for ABS traces [10, 11].

3. BACKGROUND

Linear temporal logic (LTL).

The treatment of time in temporal logic can be roughly subdivided into *branching time* (CTL, CTL*) and *linear temporal logic* (LTL) [2]. Branching time logics assume that there is a choice between different successor states at each time step and thus views time as an exponentially growing tree of possible worlds. Linear time logic views time as a linear sequence of states. The problem addressed in this paper is based upon the analysis of individual finite execution traces. Since each path comprises a sequence of states, it is natural to assume linear temporal flow. We thus focus on LTL, the syntax of which is given below:

$$\phi ::= \mathbf{true} \mid p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\, \phi \mid \mathbf{X}\phi \mid \phi\, \mathbf{U}\, \phi$$

The basic building blocks are atomic propositions p, the Boolean connectives $\wedge$ ('and'), $\vee$ ('or') and $\neg$ ('not') and the temporal connectives $\mathbf{X}$ ('next') and $\mathbf{U}$ ('until'). LTL formulae are evaluated over paths. For formula ϕ and state s, $\mathbf{true}$ always holds, p holds iff it is true in s, $\phi_1 \wedge \phi_2$ holds iff ϕ_1 holds and ϕ_2 holds, $\phi_1 \vee \phi_2$ holds iff either ϕ_1 or ϕ_2 holds, $\neg\, \phi$ holds iff ϕ does not hold and $\mathbf{X}\phi$ holds iff ϕ holds in the direct successor state of s. For formulae ϕ_1 and ϕ_2, $\phi_1\, \mathbf{U}\, \phi_2$ holds in state s iff ϕ_1 holds until ϕ_2 holds at some point in the future. Other logical connectives such as '$\Rightarrow$' or '$\Leftrightarrow$' can be derived in the usual manner: $\phi_1 \Rightarrow \phi_2 \equiv \neg\, \phi_1 \vee \phi_2$ and $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$. Additional temporal operators such as $\mathbf{F}$ ('eventually'), $\mathbf{G}$ ('always') and $\mathbf{W}$ ('weak until') can be derived as follows: $\mathbf{F}\phi \equiv \mathbf{true}\, \mathbf{U}\, \phi$ (*ϕ holds eventually*), $\mathbf{G}\phi \equiv \neg\, \mathbf{F}(\neg\, \phi)$ (*ϕ holds always*) and $\phi_1 \mathbf{W} \phi_2 \equiv (\phi_1\, \mathbf{U}\, \phi_2) \vee \mathbf{G}\phi_1$ (ϕ_1 may be succeeded by ϕ_2).

Runtime verification.

As indicated above, the application of conventional model checking is severely constrained by the size of the underlying system. Runtime verification attempts to circumvent this problem by focussing on the *current execution* of a system instead of its universal behaviour [12]. In that respect, runtime verification bears a strong similarity with testing. However, in contrast to conventional testing, runtime verification typically allows for the formulation of the system's desired behaviour in a less ad hoc and more rigorous, e.g. logic-based, way and can thus be considered more formal.

Runtime verification represents a large research area in its own right and we can only give a very superficial description here. A typical runtime verification approach has the following three characteristics: (i) the behaviour of the system under consideration is observed while the system is running; (ii) a property describing the system's desired behaviour is checked against the current execution trace; and (iii) a result is reported as soon as the property has been satisfied (or violated). Due to its focus on individual execution traces, runtime verification views time as a linear flow and properties are thus often formulated in a variant of LTL. Those properties are then translated into a *monitor* which is used to observe the execution of the system and report any satisfaction or violation that may occur. In order for a monitoring approach to be efficient, it necessarily needs to be *forward-oriented*; having to 'rewind' the execution of a system in order to determine the truth of a property is generally not an option. In terms of monitor construction, two

$$\mathbf{X}\phi \equiv \mathbf{true} \wedge \mathbf{X}\phi \quad (1)$$
$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2) \quad (2)$$

Table 1: Expansion laws for the elementary LTL operators

different approaches — *automaton-based* and *symbolic* — can be distinguished. They are briefly described below.

The automaton-based approach is related to automata-theoretic model checking of LTL properties and involves the construction of an automaton, for example a finite state automaton, a Büchi automaton or an alternating finite automaton [19]. The automaton represents the language $\mathcal{L}(\phi)$ defined by a given LTL property ϕ which describes all allowed traces of the system. One typical problem in runtime verification is that the truth of a property ϕ at time t may need additional information about the future behaviour of the trace which is not yet available. From the monitor's point of view at a particular point in time, the future is not foreseeable. Depending on the type of automaton being used, transitions may thus need to be taken nondeterministically which raises questions about the correct *traversal mode*. In the case of depth-first traversal, the automaton can only ever be in a single state, yet it may require backtracking if the wrong path has been taken. In the case of breadth-first traversal, the automaton can be in a number of states at the same time, yet the number of currently active states may grow exponentially over time.

The second, symbolic approach is based on the idea of *formula manipulation*. Instead of constructing a full automaton as mentioned above, the monitor is represented as a formula which describes (i) the requirements that the currently observed state needs to satisfy in order to either satisfy (or at least not violate) a given property ϕ, and (ii) those requirements which the rest of the trace needs to fulfil in order for ϕ to be eventually satisfied. To that end, the symbolic approach makes use of the recursive nature of temporal logic by exploiting *expansion laws* (the laws for the basic LTL operators are shown in Table 1). Informally, expansion laws allow for the decomposition of an LTL formulae into two parts: the fragment of the formula that needs to hold in the *current* state and the fragment that needs to hold in the *next* state in order for the whole formula to be true. It is useful to view both fragments as *obligations*, i.e. aspects of the formula that the trace under consideration needs to satisfy *immediately* and aspects that it promises to satisfy *in the next step*. Consider, for example, the expansion law for the 'until' operator shown in Table 1. The equivalence states that, in order for a formula $\phi_1 \,\mathbf{U}\, \phi_2$ to be satisfied at time t, either (i) ϕ_2 needs to be satisfied at time t, or (ii) ϕ_1 needs to be satisfied at time t and $\phi_1 \,\mathbf{U}\, \phi_2$ needs to be satisfied at time $t + 1$.

Expansion laws play an important role in the idea of runtime verification, since they form the basis of a decision procedure that a monitor can use to decide in a certain state whether a given property has already been satisfied or violated. By decomposing a formula into an immediate and a future obligation, optimality can be achieved: as soon as the immediate obligation is satisfied and no future obligation is created, the formula is satisfied and evaluation ends. Expansion laws work well if the trace under consideration is infinite. In the presence of finite traces, however, certain problems arise. A thorough discussion of the finite trace problem is beyond the scope of this paper, for further information, see e.g. [3].

4. HIERARCHICAL TRACES

Although the monitoring of hierarchical traces is a general problem in the area of MAS, we put a particular focus on simulations here. We can assume that simulation traces are, by definition, finite and therefore *maximal*, i.e. there is no 'unknown future' that a monitor needs to take into account. As a consequence, we can restrict our focus to a *three-valued* truth domain, i.e. $true$, $false$, and $undecided$ (the latter of which only occurs in the middle of a trace, not at its end) [3]; furthermore, in a simulation context, the focus is typically on the purely *state-based behaviour* of the system rather than on epistemic, deontic or doxastic issues, as it is often the case in the multiagent world [13].

As opposed to other types of software system, traces produced by an ABS are not monolithic but hierarchical in nature. In addition to describing the evolution of the overall *agent population*, each trace also describes at the same time the evolution of all possible *groups of agents* as well as the evolution of all *individual agents*. An example of a hierarchical trace is shown in Figure 1. It describes the temporal evolution of four agents in a simple epidemiological context. The agents are grouped into two categories (*male* and *female*) and can be in one of two states (*healthy* or *infected*). Each vertical sequence of boxes below an agent name can be seen as the evolution of the respective agent's state over time. For example, Agent 2 is initially healthy but becomes infected at time 3.

Due to the logical grouping of agents into a male and a female fragment, the simulation trace exhibits a hierarchical structure; it can be seen as comprising four agent traces, two group traces and one population trace. Each group can be understood as a *meta agent* which is characterised by its own state and its own behaviour. For example, the state of the group of male agents at time t is defined as the union of the states of male agents 1 and 2 at time t. The same holds for the entire population whose state at time t is defined as the union of the states of all agents at time t. It is important to note that groups are not just containers that host their children. They can also possess attributes which are functions of their children's states but nevertheless attributes of the group itself. For example, we might be interested in the *number of infected female agents*. This is clearly a group attribute that cannot be expressed as a combination of individual agent attributes. The same hierarchy that exists within a full trace also exists within a single time step. For example, the state of the entire population at time t can be subdivided into the state of all male agents and the state of all female agents, each of which in turn can be itself subdivided into the states of its constituents. The hierarchical structure thus permeates the trace on all levels — temporal and atemporal. The diagram on the right hand side of Figure 1 illustrates the tree structure of a hierarchical trace. The leaf nodes represent individual agents, the composite nodes represent groups of agents. Groups can be seen as meta agents which, besides serving as containers for other groups or individual agents, have their own state and behaviour.

In a verification context, the presence of hierarchical traces has important implications. It requires the ability to formulate hierarchical correctness properties, i.e. properties about *individual agents*, about *groups of agents*, as well as about the whole population. Consequently, when the correctness of a simulation model is to be assessed, this can (and, in fact, has to) be done on multiple levels. Each component of a hierarchical trace, i.e. each node in the tree structure, is a potential candidate for correctness assessment.

Formally, we view an agent's evolution over time as a simple sequence of states. We refer to such a sequence as an *agent trace*:

$$Tr_a == \mathrm{seq}\, AState$$

where $AState == Name \rightarrow Value$ is a mapping from the set of attribute names to the set of attribute values. Each group of agents is assumed to be finite and composed of n individual agents. Consequently, a *group state* is defined as a function from the set of

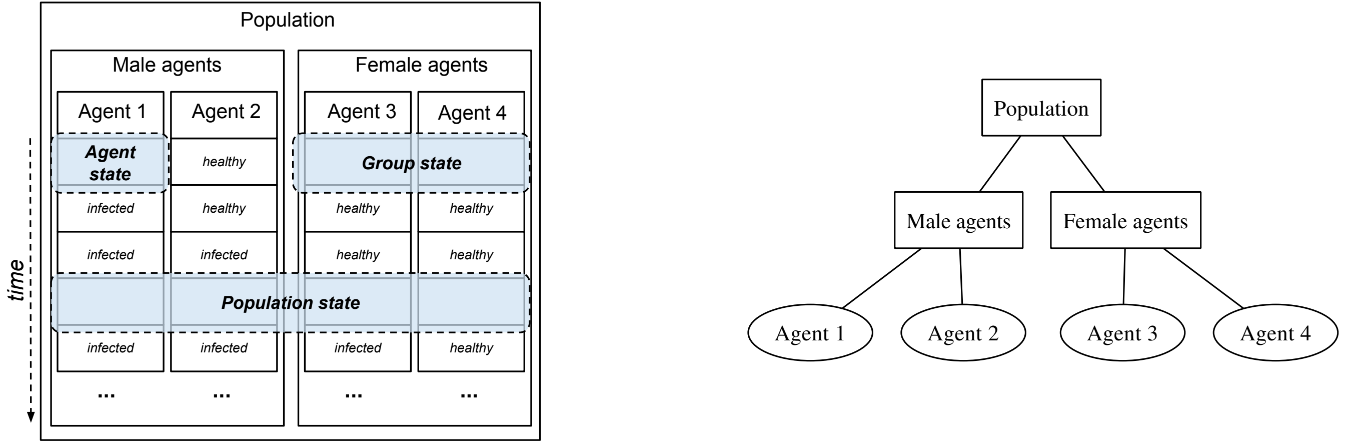

Figure 1: Example of a hierarchical trace (left) and its internal structure represented as a tree (right)

agent identifiers Ag to the set of agent states $AState$:

$$GState == Ag \to AState$$

Since agents evolve over time, groups also evolve. A *group trace* is thus defined as a sequence of group states:

$$Tr_g == \text{seq}\, GState$$

The next section discusses the verification of temporal multi-level properties on hierarchical traces.

5. MONITORING HIERARCHICAL TRACES

Due to the presence of individual agents as well as groups of agents which may themselves be arbitrarily nested, the analysis of hierarchical traces poses particular challenges. For example, in the presence of individual agents (each with its own distinct life history), obligations are no longer atomic but also hierarchical in nature. Furthermore, obligations referring to groups of agents may be understood in different ways — *collectively* or *individually*.

The crucial point is that those complications are essentially independent of the specification language being used. They are, in fact, common to all languages which allow for the formulation of statements about individuals as well as about groups. A developer of such a specification language is thus guaranteed to eventually face the problem of obligation manipulation which, given its generality, should be solved on a higher level of abstraction.

The purpose of this section is to address this problem and provide a formal treatment of the monitoring problem for hierarchical traces *independent of the concrete shape of the underlying specification language*. We tackle the problem in a layered manner, starting with the monitoring of individual agents in Section 5.1, followed by the monitoring of groups in Section 5.2. Our formalisation is based on the Z notation [18] which, together with its different extensions, has been shown to be useful as a specification language for MAS [6, 7, 15]. In order to keep the specifications as succinct as possible, we follow a functional style and make use of two commonly used functions, *map* and *zip*[2].

An essential ingredient of the subsequent formalisation is a distinction between *agent* and *group properties*. Before going into more detail we introduce the notion of a $Context == Agent \times Group$ which describes context information for the monitor about the current point in time. The context consists of the *agent in scope*, i.e. the currently focussed agent within a group (in the case of agent properties) as well as the *group* (i.e. a finite set of agents) that the monitor is currently operating on (in the case of group properties).

5.1 Monitoring agent properties

Agent properties — as the name suggests — are properties about individual agents. If one assumes atomic propositions to describe Boolean facts about agents, then LTL as described in Section 3 can, for example, be used as an agent property language. Our goal here is to abstract away from any concrete language and reduce it to a set of mere requirements that it needs to satisfy in order to be usable in a runtime verification context.

To that end, we first define what it means for a language to be *joinable*. Informally, for any language $\mathcal{L}$, we want to be able to both *conjoin* and *disjoin* instances of $\mathcal{L}$. This is important because — as illustrated further below — the manipulation of obligations for any language $\mathcal{L}$ may require the construction of logical conjunctions or disjunctions of any two formulae of type $\mathcal{L}$. In order for that to be possible, $\mathcal{L}$ needs to provide implementations of the logical 'and' and 'or' operators. This leads to the following definition.

Definition 1. A language $\mathcal{L}$ is **joinable** if it supports logical conjunction and disjunction, i.e. if implementations of both operators $\wedge\colon \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ and $\vee\colon \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ are given.

Essentially, the monitoring of individual agent properties is no different from the monitoring of conventional linear time properties as described in Section 3. In a three-valued setting and at any point in time, a linear time property $\phi : \mathcal{L}$ may be *satisfied*, *violated* or *undecided*. As a consequence, the result of a property check is either a definite truth value ($\mathbf{true}$, $\mathbf{false}$) or an obligation $\psi : \mathcal{L}$, the latter of which is again a property of language $\mathcal{L}$. In order to describe this formally, we define the following algebraic data type $AResult$ which describes the result of an individual agent property check of language $\mathcal{L}$:

$$AResult ::= ASuccess \mid AFailure \mid AO\langle\langle \mathcal{L} \rangle\rangle$$

The first two constructors, $ASuccess$ and $AFailure$ are obvious: they denote the immediate success or failure of the given agent property. The third constructor denotes an *agent obligation*, i.e. a

[2] *map* accepts a sequence and a function, applies the function to each element in the sequence and returns the resulting sequence; *zip* takes two sequences and combines them into a single sequence of tuples.

property of type $\mathcal{L}$ that the respective agent promises to satisfy in the next step in order for the whole property to become true.

During verification, we may also need to conjoin and disjoin monitoring results; to that end, we need to define appropriate operators. We start with conjunction:

$$
\begin{array}{l}
_ \wedge _ : AResult \times AResult \rightarrow AResult \\
\hline
\forall a, b : AResult \bullet \\
\quad (ASuccess \wedge b = b \wedge \\
\quad a \wedge ASuccess = a \wedge \\
\quad a \wedge AFailure = AFailure \wedge \\
\quad AFailure \wedge b = AFailure) \vee \\
\quad a \wedge b = AO(\mathbf{a} \wedge \mathbf{b})
\end{array}
$$

The definition states that a conjunction of $ASuccess$ with b always results in b, a conjunction of $AFailure$ with something else always results in $AFailure$, and all other cases result in a new obligation which represents the conjunction of the respected formulae. Disjunction can be defined accordingly:

$$
\begin{array}{l}
_ \vee _ : AResult \times AResult \rightarrow AResult \\
\hline
\forall a, b : AResult \bullet \\
\quad (ASuccess \vee b = ASuccess \wedge \\
\quad a \vee ASuccess = ASuccess \wedge \\
\quad a \vee AFailure = a \wedge \\
\quad AFailure \vee b = b) \vee \\
\quad a \vee b = AO(\mathbf{a} \vee \mathbf{b})
\end{array}
$$

The definition states that the disjunction of $ASuccess$ with something else always results in $ASuccess$, a disjunction of $AFailure$ with b always results in b, and all other cases result in a new obligation which represents the disjunction of the respected formulae. We see here that the conjunction and disjunction of instances of $AResult$ requires $\mathcal{L}$ to be joinable, i.e. it has to provide appropriate conjunction and disjunction operators (in bold).

Formally, $AResult$ is now now closed under conjunction and disjunction. Theoretically, $AResult$ should also be closed under negation since it may be the case that an element of $AResult$ is to be negated as part of the obligation manipulation process. In the case of $AResult$, negation is trivial and can be defined easily; in the case of group properties described further below, however, negation becomes more intricate. In order to circumvent the problem of obligation negation altogether, we thus assume any specification language to be in *Positive Normal Form (PNF)* where negations are only allowed to appear in front of atomic predicates [2]. If the underlying language is in PNF, then the obligation manipulation does not need to take into account the negation of obligations.

According to the description above, once a property $\phi : \mathcal{L}$ is evaluated on a state of a trace at time t, a result of type $AResult$ is produced. In order to advance the monitoring process, this result now needs to be 'fed into' the evaluation of the next state at time $t + 1$. Again, this evaluation procedure is a general problem which is independent of the underlying specification language and can thus be treated in a general way. Intuitively, monitoring of a given formula $\phi : \mathcal{L}$ should stop once a definite result (success, failure) to ϕ has been found; if an obligation has been produced, it is to be fed into the evaluation of the next time step. We can thus devise the following evaluation function for instances of $AResult$:

$$
\begin{array}{l}
evalAR : Context \times AResult \rightarrow AResult \\
\hline
\forall c : Context;\ a : \mathcal{L} \bullet \\
\quad evalAR(c, ASuccess) = ASuccess \wedge \\
\quad evalAR(c, AFailure) = AFailure \wedge \\
\quad evalAR(c, AO(a)) = \mathbf{evalA}(c, a)
\end{array}
$$

As emphasised in bold in the last line, the definition of function $evalAR$ implies that a language $\mathcal{L}$ needs to provide an implementation of an evaluation function $evalA$ that accepts an instance of $\mathcal{L}$ as well as additional context information and returns either success, failure, or an individual obligation of type $\mathcal{L}$. This leads to the following general definition.

Definition 2. A language $\mathcal{L}$ is **agent-monitorable** if it provides the following components:

1. a representation in PNF;
2. logical conjunction and disjunction; and
3. an evaluation function that returns an instance of $AResult$, i.e. success, failure, or an obligation of type $\mathcal{L}$.

5.2 Monitoring group properties

The obligation manipulation problem is slightly more complex in the case of group properties. Similar to the agent layer, we first need to think about what a verification result looks like when a property is checked for a group of agents. Consider, for example, the following property: *"all agents will eventually satisfy ϕ"*. In this case, each agent within a given group is expected to satisfy *"eventually ϕ"* individually and separately. Obligations are thus also produced separately for each agent. As a consequence, we may end up with a set of n obligations (where n denotes the number of agents), each of which needs to be satisfied separately in order for the overall formula to be satisfied. We call this type of obligation a *group obligation*. Since group properties concern groups of individual agents, several possible results may arise:

1. the current group *as a whole* needs to satisfy an obligation in the next step;
2. several agents within the current group need to satisfy an obligation *as a group* in the next step;
3. several agents within the current group need to satisfy obligations *individually* in the next step;
4. a logical conjunction or disjunction of cases 1-3 needs to be satisfied in the next step; or
5. no future obligation exists (property is immediately satisfied or violated).

In order to formalise the idea of a group result, we define again an algebraic data type $GResult$ which captures these different cases:

$$
\begin{array}{rl}
GResult ::= & GSuccess \mid GFailure \\
& \mid CGO\langle\langle Group \times \mathcal{L}\rangle\rangle \\
& \mid IGO\langle\langle \mathbb{N} \times \mathrm{seq}(GResult \times Group)\rangle\rangle \\
& \mid GAnd\langle\langle GResult \times GResult\rangle\rangle \\
& \mid GOr\langle\langle GResult \times GResult\rangle\rangle
\end{array}
$$

The first two constructors are straightforward, they denote *group success* and *group failure*, respectively. The third constructor, CGO, describes a *collective group obligation*, i.e. an obligation which needs to be satisfied *collectively* by a certain group of agents; it accommodates the first two cases in the list above. As described above, an obligation of an individual agent in a language $\mathcal{L}$ is always another $\mathcal{L}$ formula. The first argument of constructor CGO is a group, i.e. a finite set of agents, for which the formula must be satisfied, the second argument is a formula $\phi \in \mathcal{L}$. This type of obligation may, for example, result from the evaluation of an *aggregate group property*, i.e. a property which makes a statement about the *group as a whole*. The fourth constructor, IGO represents an *individual group obligation* and accommodates the third

case in the list above. Here, different obligations are to be satisfied by different groups of agents. The obligation accepts a sequence of tuples, each of which contains an obligation as well as an agent; it also contains a *satisfaction constraint* in the form of a natural number which describes the number of agents within the group that need to satisfy the given criterion. The evaluation of this type of obligation can be seen as an iteration over the given group of agents, for each of which the given sub-obligation is evaluated. This type of obligation may, for example, result from the evaluation of a universally quantified group formula. Similar to $AResult$ discussed above, it is necessary for instances $GResult$ to be closed under conjunction and disjunction. However, as opposed to $AResult$ where conjunction and disjunction always resulted in either $ASuccess$, $AFailure$, or AO, it is not always possible to simplify combinations of CGO and IGO such that the result can be described as a single $GSuccess$, $GFailure$, CGO, or IGO. In order to solve this problem, constructors 5 and 6, $GAnd$ and GOr, allow for the recursive description of conjunctions or disjunctions of obligations and thus accommodate the fourth case above.

We can now look at the conjunction and disjunction of instances of $GResult$. Similar to $AResult$, we define operators $\wedge$ and $\vee$ as shown below.

$$
\begin{aligned}
& _ \wedge _ : GResult \times GResult \rightarrow GResult \\
\hline
& \forall\, a, b : GResult \bullet \\
& \quad (GSuccess \wedge b = b \wedge \\
& \quad a \wedge GFailure = GFailure \wedge \\
& \quad GFailure \wedge b = GFailure \wedge \\
& \quad a \wedge GSuccess = b \wedge \\
& \quad (\forall\, g_1, g_2 : Group;\ a_1, a_2 : Lang \bullet \\
& \qquad (CGO(g_1, a_1)) \wedge (CGO(g_2, a_2)) = \\
& \qquad\quad \mathbf{if}\ g_1 = g_2\ \mathbf{then}\ CGO(g_1, a_1 \wedge g_2) \\
& \qquad\quad \mathbf{else}\ (\mathbf{let}\ o_1 == CGO(g_1, a_1); \\
& \qquad\qquad\qquad o_2 == CGO(g_2, a_2) \bullet \\
& \qquad\qquad\qquad\quad GAnd(o_1, o_2)))) \vee \\
& \quad a \wedge b = GAnd(a, b)
\end{aligned}
$$

The definition states that a conjunction of $GSuccess$ and any other instance b always results in b, and a conjunction of $GFailure$ and any other instance always results in $GFailure$. Things are slightly more complex for the conjunction of collective group obligations. Here, we have to distinguish between different cases: if both obligations refer to the same group of agents, then we can create a single new obligation containing the conjunction of the respective formulae; otherwise, we cannot perform any further simplifications and thus construct an obligation of type $GAnd$ which represents a conjunction of obligations. This is also done for all other combinations of obligations, as described in the last line[3].

Disjunction can be given accordingly:

$$
\begin{aligned}
& _ \vee _ : GResult \times GResult \rightarrow GResult \\
\hline
& \forall\, a, b : GResult \bullet \\
& \quad (GSuccess \vee b = GSuccess \wedge \\
& \quad a \vee GSuccess = GSuccess \wedge \\
& \quad GFailure \vee b = b \wedge \\
& \quad a \vee GFailure = a) \vee \\
& \quad a \vee b = GOr(a, b)
\end{aligned}
$$

A disjunction of $GSuccess$ with any other instance always results in $GSuccess$, a disjunction of $GFailure$ with any other instance b always results in b, and a disjunction of all other combinations always results in GOr[3].

[3]Further simplifications are possible but, for clarity and space limitation, we keep the description simple.

We can now describe how instances of $GResult$ are to be manipulated over time. Similar to $AResult$, monitoring should stop once success or failure of a property has been determined. In all other cases, the obligations have to be fed into the evaluation of the next step. This is described by function $evalGR$ which accepts as an input the current context as well as an instance of $GResult$ and produces another instance of $GResult$:

$$
\begin{aligned}
& evalGR : Context \times GResult \rightarrow GResult \\
\hline
& \forall\, c : Context \bullet \\
& \quad evalGR(c, GSuccess) = GSuccess \wedge \\
& \quad evalGR(c, GFailure) = GFailure \wedge \\
& \quad (\forall\, a : Ag;\ g, g' : Group;\ o : Lang \bullet \\
& \qquad evalGR((a, g), CGO(g', o)) = \mathbf{evalG}((a, g'), o)) \wedge \\
& \quad (\forall\, a : Ag;\ g : Group;\ k : \mathbb{N};\ os : \mathrm{seq}(GResult \times Group) \bullet \\
& \qquad evalGR((a, g), IGO(k, os)) = \\
& \qquad\quad (\mathbf{let}\ fct == (\lambda\, t : (GResult \times Group) \bullet \\
& \qquad\qquad evalGR((a, second\, t), first\, t)) \bullet \\
& \qquad\qquad\quad (\mathbf{let}\ chks == map(fct, os) \bullet \\
& \qquad\qquad\qquad kSat(k, zip(chks, snds(os)))))) \wedge \\
& \quad (\forall\, x1, x2 : GResult \bullet \\
& \qquad evalGR(c, GAnd(x1, x2)) = \\
& \qquad\quad (\mathbf{let}\ r1 == evalGR(c, x1);\ r2 == evalGR(c, x2) \bullet \\
& \qquad\qquad r1 \wedge r2)) \wedge \\
& \quad (\forall\, x1, x2 : GResult \bullet \\
& \qquad evalGR(c, GOr(x1, x2)) = \\
& \qquad\quad (\mathbf{let}\ r1 == evalGR(c, x1);\ r2 == evalGR(c, x2) \bullet \\
& \qquad\qquad r1 \vee r2))
\end{aligned}
$$

If the result to be checked is either $GSuccess$ or $GFailure$, then there is nothing else to be done. In the case of a collective group obligation (CGO), the two parameters of CGO (the group in scope together with the obligation to be fulfilled) are passed to a language-specific evaluation function $evalG$ (in bold) which itself returns an instance of $GResult$. In the case of an individual group obligation, the situation is slightly more complicated. As opposed to a collective group obligation which wraps a group *formula*, the individual group obligation wraps a *list of result-group tuples* together with a numeric value that describes a *satisfaction constraint*, i.e. the required number of successful checks. To this end, an individual, recursive call to $evalGR$ is performed for each entry in the list (using map). The resulting list of check results is then 'zipped' with the list of groups. This results in another list which is checked for satisfaction w.r.t. the numeric constraint using a function $kSat$ which checks whether at least k elements in the given list are equal to $GSuccess$. The manipulation of obligation conjunctions and disjunctions is straightforward and not described in further detail here. This leads to the following final conclusion.

Definition 3. A language $\mathcal{L}$ is **multiagent-monitorable** if it provides the following components:

1. a representation in positive normal form;
2. logical conjunction and disjunction;
3. a group evaluation function that returns an instance of $GResult$, i.e. *success*, *failure*, a *collective group obligation*, an *individual group obligation*, or a conjunction or disjunction of the previous two types of obligation; and
4. an agent evaluation function that returns an instance of $AResult$, i.e. *success*, *failure*, or an *individual agent obligation*.

This concludes the description and the formalisation of the monitoring framework. The next section illustrates the implementation of the agent layer of the framework in Haskell.

5.3 Implementation

The description of the monitoring framework has been purely formal in nature so far. As mentioned in the introduction, its purpose is to serve as a blueprint for developers to construct their own monitoring procedures. To illustrate this process and show that the formal description is sufficiently close to an executable implementation, we provide here a formulation of the framework in Haskell. The code shown below is part of $\mathtt{MC^2MABS}$, a practical statistical runtime verification framework for large-scale agent-based simulation models [10, 1]. For space limitations, we restrict our description to the agent layer here; the group layer can be translated accordingly. A more comprehensive description of $\mathtt{MC^2MABS}$ including implementation details, complexity considerations, and case studies is given elsewhere [10]; the source code of $\mathtt{MC^2MABS}$ together with additional documentation is available online [1].

Haskell is a strongly typed, purely functional language, so we first need to think about the necessary types. In analogy to the Z description given above, $AResult$ can be defined as an algebraic data type with a generic parameter a that denotes the underlying language type:

```
AResult a ::= ASuccess | AFailure | AO a
```

Requirements that a type needs to satisfy can be described by means of *type classes* in Haskell; consequently, all types that belong to a given type class need to satisfy the given constraints. The notion of *joinability* can thus be described conveniently as follows.

```
class Joinable a where
  and  :: a → a → a
  or   :: a → a → a
```

The type class requires that any type a has to provide both an 'and' and an 'or' function, which is precisely the definition of joinability given above. Using this information, we can now describe the conjunction of elements of type `AResult`. As described in Section 5.1, in order for elements of type `AResult` to be conjoinable, formulae of the underlying language also need to provide an appropriate 'and' function. This can be formulated as a *type constraint* utilising the `Joinable` type class in the signature of the function:

```
andAR :: Joinable a ⇒ AResult a → AResult a → AResult a
andAR ASuccess b  = b
andAR _ AFailure  = AFailure
andAR AFailure _  = AFailure
andAR b ASuccess  = b
andAR b1 b2       = do r1 ← b1
                       r2 ← b2
                       return $ r1 `and` r2
```

Disjunction can be described accordingly:

```
orAR :: Joinable a ⇒ AResult a → AResult a → AResult a
orAR ASuccess _ = ASuccess
orAR _ ASuccess = ASuccess
orAR AFailure b = b
orAR b AFailure = b
orAR b1 b2      = do r1 ← b1
                     r2 ← b2
                     return $ r1 `or` r2
```

In order for elements of type $AResult$ to be evaluable by function $evalAR$, the underlying specification language also needs to provide an appropriate evaluation function $evalA$. This in combination with joinability (and the PNF which, for simplicity, is omitted here), provides the basis for the notion of *agent monitorability* which is described by the following type class[4]:

[4]Context information is dealth with using the *State Monad* [20].

```
class AgentMonitorable a
  evalA :: a → State Context (AResult a)
```

We now have everything that we need for evaluation of agent formulae which is described by function $evalAR$ (see Section 5.1). Similar to the functions above, it can be translated into a corresponding Haskell function in a straightforward way, utilising the constraint imposed upon the underlying language by type class `AgentMonitorable`.

```
evalAR :: AgentMonitorable a ⇒
          AResult a → State Context (AResult a)
evalAR ASuccess = return ASuccess
evalAR AFailure = return AFailure
evalAR (AO a)   = do res ← evalA a
                     return res
```

This concludes the implementation of the agent layer. Group-related types and functions can be translated accordingly. It is interesting to note that, despite exhibiting the same functionality and the same level of rigour, the Haskell notation is significantly more succinct than the Z description given above.

The next section illustrates the application of the ideas by describing how a simple example specification language can be made monitorable by satisfying the necessary requirements.

6. EXAMPLE: QUANTIFIED LTL

The nature of property specification languages depends critically on the type of system to be analysed. The concepts of joinability, agent-monitorability, and multiagent-monitorability introduced in the previous section serve as a basic interface that a specification language needs to implement in order to be monitorable in a three-valued runtime verification context.

In order to illustrate this idea, we introduce here *qLTL*, a variant of LTL which supports the formulation of properties about *individual agents* as well as about *groups of agents*. In order to achieve that, we make a syntactic distinction between *agent formulae* and *group formulae*, as described further below. It is important to note that the focus here is on illustration rather than on the development of a full specification language; the description is thus kept deliberately simple and superficial. A comprehensive description of $simLTL$, a conceptually similar, yet more complex and realistic specification language that is used within $\mathtt{MC^2MABS}$ is given elsewhere [10, 11]. The syntax of qLTL formulae is described by the following abstract data type.

$$
\begin{aligned}
QLTL ::= {} & Atom\langle\langle Prop\rangle\rangle \mid \\
& And\langle\langle QLTL \times QLTL\rangle\rangle \mid Or\langle\langle QLTL \times QLTL\rangle\rangle \mid \\
& Next\langle\langle QLTL\rangle\rangle \mid Until\langle\langle QLTL \times QLTL\rangle\rangle \mid \\
& ForAll\langle\langle QLTL\rangle\rangle \mid ForAgent\langle\langle QLTL\rangle\rangle
\end{aligned}
$$

The basic building blocks are atomic propositions $p \in Prop$, Boolean conjunction and disjunction, temporal operators 'next' and 'until', and quantifiers 'forAll' and 'forAgent'. In the case of 'forAll', the enclosed qLTL formula is checked for each agent in the current group; in the case of 'forAgent', the enclosed formula is checked on the agent that is currently in scope[5].

qLTL provides an implicit distinction between *agent* and *group formulae*. Agent formulae are those that are enclosed by a 'forAgent' statement; all other formulae are group formulae. The semantics of agent formulae are formulated over *agent traces* formally introduced in Section 4. Let $tr_a : Tr_a$ be an agent trace.

[5]Note that the syntax allows for the construction of semantically invalid formulae. For example, $ForAgent(ForAll(Prop(p)))$ is syntactically correct but, due to the enclosing of a 'forAll' statement in a 'forAgent' statement, semantically questionable. In a real-world scenario, the syntax would have to be crafted more carefully in order to avoid such problems.

Then, for all $p, p_1, p_2 : qLTL$, tr_a satisfies p (denoted $tr_a \models p$) iff p is true in state $tr_a[0]$, tr_a satisfies $Atom(p)$ (denoted $tr_a \models Atom(p)$) iff p is true in state $tr_a[0]$, tr_a satisfies $And(p_1, p_2)$ iff $tr_a \models p_1$ and $tr_a \models p_2$, tr_a satisfies $Or(p_1, p_2)$ iff $tr_a \models p_1$ or $tr_a \models p_2$, tr_a satisfies $Next(p)$ iff $tr_a[1..] \models p$ and, finally, $tr_a \models Until(p_1, p_2)$ iff $tr_a \models p_2$ or $tr_a \models p_1$ until eventually $tr_a \models p_2$. The semantics of group formulae are defined over *group traces*. The basic operators are similar to the agent case. Let $tr_g : Tr_g$ be a group trace. Then, tr_g satisfies $ForAll(p)$ iff p holds for all agents in the current group. and tr_g satisfies $ForAgent(p)$ iff p holds for the agent currently in scope.

As described in Section 5, any specification language needs to be *joinable*, *agent monitorable*, and *multiagent monitorable* in order to be usable in a runtime verification context for ABSs; furthermore, its formulae need to be in PNF. In the qLTL grammar given above, negation is not supported, so the PNF requirement is already satisfied. Furthermore, qLTL is closed under logical conjunction and disjunction, i.e. the requirement of joinability is also already satisfied. It remains just to discuss the requirements of agent and multiagent monitorability. In order to satisfy both requirements, we need to provide appropriate evaluation functions.

We start with the evaluation of qLTL agent formulae. Informally, in any state, an agent formula is either true, false, or not yet decidable. In the latter case, an obligation (which is again a qLTL agent formulae) is produced. This is described by the algebraic datatype $AResult$ that was introduced in Section 5.1 above. We can thus describe the evaluation of agent formulae by a function $evalA$ shown below; it accepts context information and an agent formula as input and returns an instance of $AResult$. Obligations are only produced in the case of formulae that contain temporal operators ('next' or 'until'); in all other cases, a verdict can be produced immediately.

$$
\begin{array}{l}
evalA : Context \times QLTL \rightarrow AResult \\
\hline
\forall c : Context \bullet \\
\quad (\forall p : Prop \bullet evalA(c, Atom(p)) = holds(c, p)) \land \\
\quad (\forall a, b : QLTL \bullet evalA(c, And(a, b)) = \\
\quad\quad evalA(c, a) \land evalA(c, b)) \land \\
\quad (\forall a, b : QLTL \bullet evalA(c, Or(a, b)) = \\
\quad\quad evalA(c, a) \lor evalA(c, b)) \land \\
\quad (\forall a : QLTL \bullet evalA(c, Next(a)) = AO(a)) \land \\
\quad (\forall a, b : QLTL \bullet evalA(c, Until(a, b)) = \\
\quad\quad \mathbf{if}\ evalA(c, b) = ASuccess\ \mathbf{then}\ ASuccess \\
\quad\quad \mathbf{else\ if}\ evalA(c, a) = ASuccess \\
\quad\quad \mathbf{then}\ AO(Until(a, b))\ \mathbf{else}\ AFailure)
\end{array}
$$

The evaluation of group formulae differs from that for agent formulae only w.r.t. the temporal operators and the quantifiers. The evaluation of a 'next' formula results in the creation of a collective group obligation CGO which can be explained as follows: if the group as a whole needs to satisfy formula p in the next step (as denoted by the 'next' operator), then this can only become true if it promises to satisfy p in the next step — hence the collective obligation. In the evaluation of an 'until' formula, if the second subformula b is satisfied, then the overall formula is satisfied. Otherwise, if the first subformula, a, is already satisfied, then the group as a whole promises to satisfy $Until(a, b)$ in the future. In this case, again, a collective group obligation is produced. If neither a nor b is satisfied, then there is no chance for the group to satisfy the overall property. The evaluation of a universally quantified formula is slightly different. Here, the nested agent formula p is evaluated *separately for each agent in the current group*. If we assume heterogeneity in the group then, clearly, there may also be different evaluation results for the agents. As a consequence, an individual group obligation IGO is returned. It represents a list of individual agent obligations that need to be satisfied in the next step. A formal description of function $evalG$ is given below.

$$
\begin{array}{l}
evalG : Context \times qLTL \rightarrow GResult \\
\hline
\ldots \\
(\forall a : Agent;\ g : Group \bullet \\
\quad (\forall p : QLTL \bullet evalG((a, g), Next(p)) = CGO(g, p)) \land \\
\quad (\forall a, b : QLTL \bullet evalG((a, g), Until(a, b)) = \\
\quad\quad \mathbf{if}\ evalG((a, g), b) = GSuccess\ \mathbf{then}\ GSuccess \\
\quad\quad\ \mathbf{else\ if}\ evalG((a, g), a) = GSuccess \\
\quad\quad\quad \mathbf{then}\ CGO(g, Until(a, b))\ \mathbf{else}\ GFailure) \land \\
\quad (\forall p : QLTL \bullet evalG((a, g), ForAll(p)) = \\
\quad\quad (\mathbf{let}\ os == \{n : \mathbb{N};\ a' : Agent \mid n \in 1 \ldots \# g \land a' \in g \bullet \\
\quad\quad\quad n \mapsto (evalG((a, g), p), g)\} \bullet IGO(\# g, os))) \land \\
\quad (\forall p : QLTL \bullet evalG((a, g), ForAgent(a, p)) = \\
\quad\quad (\mathbf{let}\ res == evalA((a, g), p) \bullet \\
\quad\quad\quad \mathbf{if}\ res = ASuccess\ \mathbf{then}\ GSuccess \\
\quad\quad\quad \mathbf{else\ if}\ res = AFailure\ \mathbf{then}\ GFailure \\
\quad\quad\quad \mathbf{else}\ CGO(\{a\}, p))))
\end{array}
$$

This concludes the description of qLTL. With a PNF representation, joinability, agent monitorability and multiagent monitorability, qLTL now satisfies all requirements in order for it to be used in a three-valued runtime verification context.

7. CONCLUSIONS

Monitoring properties about hierarchical traces is a complex, yet essentially language-independent problem and can (and should) thus be tackled in a general way. This paper addresses the monitoring problem on a high level of abstraction and proposes a language-independent solution. We first introduced formally the notion of hierarchical which allows for the evaluation of properties on two different observational levels — *groups of agents* and *individual agents*. This serves as the semantic basis for a *general, language-independent monitoring framework* for hierarchical traces in a three-valued setting. The existence of observational levels has an important impact on the nature of *obligations* produced during monitoring. We introduced different *types of obligations* — both on the agent and the group level — together with *algorithms for their manipulation* at runtime. Language independence is achieved by the definition of *minimal requirements* or *contracts* which a specification language needs to satisfy in order to be used in a runtime verification setting. To this end, we introduced the concepts of *joinability*, *agent monitorability* and *multiagent monitorability*. Along with the conceptual description of the framework, we also provided a full typechecked formalisation in Z.

The overall purpose of the framework is to serve as an abstract description of a general monitoring procedure which can, for example, be implemented as a generic library in a particular programming language. In order to illustrate this and show the close correspondence between the formal description and the executable implementation, we gave an example implementation of the agent-related functions and data types in Haskell. The implementation is part of $\texttt{MC}^2\texttt{MABS}$, a practical statistical runtime verification framework for large-scale agent-based simulation models [10, 1].

The integration of an existing specification language into the framework by implementing appropriate evaluation functions as required by the notions of agent monitorability and multiagent monitorability was illustrated using a simple example language, conceptually similar to the one used by $\texttt{MC}^2\texttt{MABS}$.

REFERENCES

[1] MC^{2}MABS website. https://github.com/bherd/mc2mabs. Last access: 02/15.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, June 2010.

[4] I. Cakirlar, Ö. Gürcan, O. Dikenelli, and S. Bora. RatKit: A repeatable automated testing toolkit for agent-based modeling and simulation. In *Proc. 15th Int. Workshop on Multi-Agent-Based Simulation*, 2014.

[5] M. Dastani and J.-J. C. Meyer. Correctness of multi-agent programs: A hybrid approach. In M. Dastani, K. V. Hindriks, and J.-J. C. Meyer, editors, *Specification and Verification of Multi-agent Systems*, pages 161–194. Springer US, 2010.

[6] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2004.

[7] M. d'Inverno, M. Luck, M. Georgeff, D. Kinny, and M. Wooldridge. The dMARS architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9:5–53, July 2004.

[8] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, 2012.

[9] M. A. C. Gatti and A. von Staa. Testing & debugging multi-agent systems: a state of the art report. Technical report, Departamento de Informatica, Pontifical Catholic University of Rio de Janeiro, 2006.

[10] B. Herd. *Statistical runtime verification of agent-based simulations*. PhD thesis, King's College London, 2015.

[11] B. Herd, S. Miles, P. McBurney, and M. Luck. An LTL-based property specification language for agent-based simulation traces. Technical Report 14-02, King's College London, Oct 2014.

[12] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[13] A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proc. 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 161–168, 2006.

[14] C. M. Macal and M. J. North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[15] T. Miller and P. McBurney. Multi-agent system specification using TCOZ. In T. Eymann, F. KlÃijgl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Multiagent System Technologies*, volume 3550 of *Lecture Notes in Computer Science*, pages 216–221. Springer, 2005.

[16] M. Niazi, A. Hussain, and M. Kolberg. Verification and validation of agent based simulations using the VOMAS approach. In *Proc. of the 3rd Workshop on Multi-Agent Systems and Simulation*, 2009.

[17] A. Sharpanskykh and J. Treur. A temporal trace language for formal modelling and analysis of agent systems. In M. Dastani, K. V. Hindriks, and J.-J. C. Meyer, editors, *Specification and Verification of Multi-agent Systems*, pages 317–352. Springer US, 2010.

[18] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[19] M. Vardi. Automata-theoretic model checking revisited. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.

[20] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer Berlin Heidelberg, 1995.

[21] C. J. Wright, P. McMinn, and J. Gallardo. Towards the automatic identification of faulty multi-agent based simulation runs using MASTER. In F. Giardini and F. Amblard, editors, *Multi-Agent-Based Simulation XIII*, volume 7838 of *LNCS*, pages 143–156. Springer, 2013.