Protocols with Exceptions, Timeouts, and Handlers: A Uniform Framework for Monitoring Fail-Uncontrolled and Ambient Intelligence Systems

Davide Ancona, Daniela Briola and Viviana Mascardi
Department of Informatics, Bioengineering, Robotics and System Engineering
University of Genova, Via Dodecaneso 35, 16146, Genova, ITALY
Email: {davide.ancona,daniela.briola,viviana.mascardi}@unige.it

***Abstract*—This paper describes an approach for designing, formalizing and implementing *sentinels* that detect errors in *fail-uncontrolled multiagent systems*, and *controllers* that identify particular situations in *ambient intelligence (AmI) systems*. The formalism we use for representing the expected patterns of actions along with exceptions, timeouts, and their handlers, is that of *constrained global types* extended with features for dealing with these new constructs. We provide the syntax and semantics of the extended constrained global types and examples of their use, in the different contexts of fail-uncontrolled and AmI systems.**

I. Introduction

Multiagent protocols are usually intended as a means to regulate communicative interactions among agents. The literature on agent interaction protocols is huge and despite its long tradition dating back to the dawn of the research on agents and multiagent systems [1], [2], it is still a hot research topic [3], [4], [5], [6], [7].

In this paper we adhere to a general definition of protocols as a means to define *legal sequences or concurrent combinations of actions without regard to the meanings of those actions* [8]: our protocols define patterns of actions in the environment, rather then just conversations.

The purpose of our research is to provide the multiagent system developer with a framework for designing, formalizing and implementing the following monitoring agents on top of existing agent environments:

1) *sentinels* that detect errors in *fail-uncontrolled multiagent systems* and
2) *controllers* that identify particular situations in *ambient intelligence (AmI) systems*.

Despite the different contexts where sentinels and controllers are expected to operate, both must be able to monitor the system's behavior and to properly deal with failures and exceptional situations that may arise during the multiagent system (MAS for short) functioning.

The language we propose for formalizing the expected patterns of actions in the environment is that of *constrained global types* [9], [10], [11] extended with features for dealing with exceptions, timeouts, and their handlers. These extensions are used to describe specific situations that must be managed before they lead to an error or to a dangerous or unwanted situation.

The basic mechanism for runtime verification that an event which took place in the environment is compliant with the protocol has already been described in our previous works [9], [6], [12], where we presented a working framework for Jason [13], chosen as a paradigmatic example of logic-based MAS language and framework, and for JADE [14], chosen because of its wide adoption both in academic and industrial environments.

In this paper we describe the extensions to *constrained global types* to cope with the newly introduced concepts such as events, timeouts etc, as described in Section III. Although the updating of the existing and working prototypes for Jason [9] and JADE [6] to include these new features is still under way, we are confident on the positive outcomes of our efforts. In fact, we already implemented an SWI Prolog extended interpreter that properly deals with these new features and which is presented in Section III, and we tested it as a standalone application (Section IV). The missing final step is to substitute the interpreter for monitoring MASs that is already integrated on top of Jason (resp. JADE) with this new, enhanced interpreter. This requires to take care of minor syntactic issues so it does not come completely for free, but is should not even raise relevant technical problems.

We consider two different types of systems, fail-uncontrolled MASs and AmI systems, to prove the usability of constrained global types for representing patterns of events (that we also name situations when referred to AmI systems, or protocols, coherently with the terminology used in the literature about security) that should drive the runtime monitoring activity of sentinels and controllers.

In *fail-uncontrolled multiagent systems*, the protocol modeling the normal expected behavior of the MAS is *explicitly represented* and involves events of any type (not limited to communicative ones); sentinels observe the system under monitoring and execute user-defined, domain-dependent code when a deviation from the expected behavior takes place. Deviations are modeled as *exceptions* to the normal flow of actions that the protocol models, and the user code must implement the exception handlers.

In *AmI systems*, where usually there is no explicit protocol driving the agents' behavior, but rather agents are free to act as they wish in any ambient, we only model particular situations that must be identified as soon as they arise: in this way we can respect the autonomy of agents and deal with unforeseen

events, but we are still able to check that specific situations are identified as "wrong" ones and their consequences are avoided.

By exploiting the abstraction of "event type" supported by constrained global types, we can model events which are observed in the monitored system and that are relevant for the monitored application, but also events which are not interesting and hence should be discarded (*uninteresting events*). *Unexpected exceptions* can be captured and managed as well.

Since sentinels and controllers are agents like any other one, they can be "part of the protocol" as well, hence being both promoters and targets of the monitoring activities[1]. For example, if the protocol states that – when an exception has been detected – the controller must always send a message to the agent that caused it, and this does not happen, the controller can realize that it is violating the protocol and should repair its own code.

In order to avoid bottlenecks, *the monitoring activities may be distributed among many different and independent sentinels (resp. controllers)*, each observing a subset of the events based on their type, the location where they take place, or other criteria. An approach for distributing the monitoring activity by projecting the protocol onto subsets of agents involved in it has been explored in [15].

The proposed approach is general enough to be implemented on top of any MAS or AmI framework where events can be observed and translated into a suitable representation, amenable for formal reasoning.

The paper is structured in the following way: Section II provides the background to this work. Section III describes the extension of constrained global types with exceptions, timeouts, and their handlers. Section IV provides examples in different contexts. Section V concludes.

II. Background and related work

The classification of system failures and the definition of fail-uncontrolled system we refer to, is that provided in [16], which in turn refers to [17]:

> *The most generally accepted failure classification can be found in [17]:*
>
> 1) *A crash failure means a component stops producing output; it is the simplest failure to contend with.*
> 2) *An omission failure is a transient crash failure: the faulty component will eventually resume its output production.*
> 3) *A timing failure occurs when output is produced outside its specified time frame.*
> 4) *An arbitrary (or byzantine) failure equates to the production of arbitrary output values at arbitrary times.*
>
> *Given this classification, two types of failure models are usually considered in distributed environments:*
>
> - *fail-silent, where the considered system allows only crash failures, and*
> - *fail-uncontrolled, where any type of failure may occur.*

In such a context,

> *a sentinel is an agent, and its mission is to guard specific functions or to guard against specific states in the society of agents. The sentinel does not partake in domain problem solving, but it can intervene if necessary, choosing alternative problem solving methods for agents, excluding faulty agents, altering parameters for agents, and reporting to human operators. Being an agent, the sentinel interacts with other agents using semantic addressing. Thereby it can, by monitoring agent communication and by interaction (asking), build models of other agents. It can also use timers to detect crashed agents (or a faulty communication link)* [18].

Exceptions can be informally defined as:

> *[...] abnormal conditions that arise during the execution of a process. The importance of exceptions stems from the simple fact that they are an essential feature of real-life processes. Businesses, for example, entertain exceptional requests from customers in the interest of better customer service. Conversely, exceptions that occur in a process may lead to poor user satisfaction. Therefore, businesses must accommodate exceptions in their underlying systems and their interactions with other businesses. For concreteness, let us review a classification of exceptions proposed by Eder and Liebhart [19]:*
>
> 1) *Basic failures, which are system-level failures such as network failures.*
> 2) *Application failures, such as database transaction failures.*
> 3) *Expected exceptions, which are deviations from the normal flow that occur infrequently but often enough to be incorporated into the process model.*
> 4) *Unexpected exceptions, which are not modeled and hence require a change in the design of the process when they are discovered.*
>
> *An alternate classification of exceptions distinguishes among system level exceptions, programming language exceptions, and pragmatic exceptions. Among these, pragmatic exceptions are the most acute and the most difficult to handle* [20].

According to our analysis of the state of the art in fault tolerant multi-agent systems, we can distinguish two broad classes of approaches:

1) approaches where the multiagent protocol is represented in an explicit way, and fault tolerance amounts to handling exceptions due to a non-compliant behavior w.r.t. the protocol [8], [20]; in such approaches, multiagent protocols are usually limited to express constraints on communicative actions;
2) approaches where no protocol exists, and fault tolerance amounts to detecting crashed agents and replicate them in a transparent way [21], [16].

[1]This "introspective" model can work only if sentinels and controllers perform the very same monitoring activity whatever the agent being monitored, including themselves.

Our approach falls in the first category.

With respect to *AmI systems*, the idea of implementing them using an agent-based approach dates back to the late nineties. Among the very first projects in this area we may mention the Intelligent Room project at MIT [22] that concentrated on making the room responsive to the occupant by adding intelligent sensors to the user interface, the ACHE system [23] which also aimed at energy saving and increased personal comfort by learning the user preferences automatically by observing the behavior of the persons in the building, and the work by Paul Davidsson and Magnus Boman [24] who used multiagent principles to control building services, with agents that decompose systems by function rather than behavior. The synergy between ambient intelligence and intelligent software agents has become stronger and stronger, and nowadays there are tens of papers and projects in this area. Among the most relevant ones, we may mention [25], [26], [27]. The impressive growth in the last years [28], [29], [30], [31], [32] is also witnessed by the Multi-Agent Systems and Ambient Intelligence (MASAI) special track at the Practical Applications of Agents and Multi-Agent Systems (PAAMS) 2014 and 2015 conferences.

Among the most recent works in the MAS & AmI area, the closer to our proposal is [33]. That paper presents a concrete software architecture dedicated to AmI features and requirements. The proposed behavioral model, called Higher-order Agent (HoA) captures the evolution of the mental representation of the agent and the one of its plan simultaneously. Plan expressions are written and composed using a formal algebraic language named AgLOTOS which is very similar to our formalism based on constrained global types although less expressive because of the lack of concatenation and recursion. In [33], the planning process is improved with two new services which provide a guidance for the mental process and a model checking approach to analyze some temporal properties over the agent plan. Both services are based on a transition system called Contextual Planning Systems that, similarly to the traces of our constrained global types, can represent the different execution traces the agent could perform from a given HoA configuration, assuming the information context of the HoA configuration holds. With AgLOTOS it is possible to handle actions and plans failures, as we do. The main differences between the approaches, besides the lower expressive power of AgLOTOS w.r.t. the formalism that we adopt, are that our approach is more general than the one discussed in [33] and it can be implemented on top of any existing MAS framework, as the only requirement is that the sentinels and controllers can observe events taking place in the environment, but the approach based on AgLOTOS is in a more mature development stage and has been experimented on the field in a Smart-Campus Project.

Finally, we may observe that our approach of supervising the ongoing activities of the MAS, being it an *AmI system* or a more generic distributed open system, is similar to the "Monitoring by overhearing" approach presented in [34], where an overhearing agent monitors the exchanged communications between the system's agents and uses the observed communications to independently assemble and infer the needed monitoring information.

III. Multiagent Protocols with Timeouts, Exceptions, and Handlers

Our proposal is general enough to be applied to many languages and environments for MASs. The only needed requirement is that some agents (the sentinels or controllers) can be given the power and the capability to observe (some of) the actions taking place in the environment.

As far as sentinels are concerned, the four types of failure identified in [17] are all resorted to a behavior that does not comply with the multiagent protocol:

- a timing failure can be detected thanks to implicit and explicit timeouts on the actions: if an expected action does not take place within the given timeout, then a time failure takes place. More successive timing failures may mean either an omission or a crash failure;
- a byzantine failure can be detected by verifying that the occurred action was (or was not) expected by the protocol in the current state and time.

Sentinels implementation verifies at run-time that perceived events are compliant with the protocol. This verification activity is necessary when the multiagent protocol is designed and the MAS is tested, but cannot guarantee any fault tolerance, as the output of the verification in any given time instant is just "yes" or "no". In order to capture deviations from the expected flow and manage them, so to possibly recover to some acceptable state when the verification fails, the sentinel must pro-actively intervene by executing some recovery plan. We have designed the mechanism that allows a sentinel to capture the expected failures of the protocol by means of *exceptions*, and the recovery intervention by means of *exception handlers*.

In this section we present the formalism of constrained global types introduced by Ancona, Mascardi et al. [9], [10], [11], extended to cope with any kind of observable event instead of only communicative actions and to model exceptions and timeouts. Although never exploited, dealing with uninteresting events was already possible with the original versions of the formalism and hence does not represent a real extension to it.

In the sequel, we will take the perspective of constrained global types as a powerful means to specify multiagent protocols, but they can be used to specify AmI situations as well, and for the same reasons. For readability, we will avoid repeating "or AmI situations", but we emphasize that our assertions hold for them as well.

Intuitively, a constrained global type represents the state of an multiagent protocol from which several transition steps to other states (that is, to other constrained global types) are possible, with a resulting event.

Exceptions are the mechanism we have introduced in order to cope with byzantine failures, whereas timeouts are used to detect and cope with omission, timing and crash failures.

A. Syntax

Event. An event e is any observable event taking place in the MAS environment, including communicative actions,

actions performed by agents, agent location and move, and actions performed by artifacts. We do not face the transduction problem and assume that events are translated into symbols that agents can manipulate by some mediator between the agents and the environment. For the purposes of our work, events are symbolic expressions.

Example 1:

```
transport(policeman(marcus), prisoner(alice),
from(jail), to(room1), by(car)).
```

Example 2:

```
collect_parcel(agent2, parcel3, parcelweight(4, kg)).
```

The event "transport" is characterized by the involved entities (the policeman and the prisoner, identified by their names), the departure and arrival places and the means of transport.

Event type. A "normal" event type η is a predicate on events. Its interpretation is the set of events that verify η; we write $e \in \eta$ to mean that η is true on e, and we also say that e has type η.

Example 1:

```
transport(policeman(marcus), prisoner(alice), from(jail),
to(room1), by(car)) ∈ move(alice, jail, room1).
```

Example 2:

```
collect_parcel(agent2, parcel3, parcelweight(4, kg))
∈ collect_parcel.
```

With respect to the actual event that took place in the environment and that was transduced into a symbolic form, the event type may abstract some details which are not relevant for the sentinel monitoring activities (like in Example 2), and can be identified by a different predicate with different arguments (like in Example 1, where "move(...)" is the event type of "transport(...)" event).

Exception type. An exception type $\mathcal{E}$ is – from a logical point of view – a quadruple <type of the event that will fire the exception, type of the exception, pointer to the exception handler, arguments of the exception handler>. We represent exceptions in a more compact way, as a couple whose second element has the type of the exception as functor, and the pointer to the exception handler along with its arguments as argument.

An exception type is a way for stating that perceiving a specific event in the current state of the protocol represents an exceptional situation and must be handled as such, calling an ad-hoc piece of code.

Example:

```
(exception(move(A, key_room, treasure_room),
     illegal_move_exc(
     entering_treasure_room_without_permission(A)))).
```

The events whose type is `move(A, key_room, treasure_room)` may raise an exception (if they take place in the state of the protocol where the exception type may be reached) whose type is `illegal_move_exc`, and that must be handled by the code pointed to by `entering_treasure_room_without_permission`, with argument `A`.

Set timeout type. From a logical point of view, a set timeout type $\mathcal{ST}$ is a couple <type of the event that will cause the alarm to be set, list of triples (timeout label l , delay timeout d and its omission handler, crash timeout c and its crash handler)>.

A timeout is set on occurrence of one event whose type is specified as first element of the couple and causes two alarms to be set and, later on, raised in an automatic way by the system. The first alarm will be automatically raised by the system after the delay timeout, unless an event tagged with label l took place before, and will be handled using the associated "omission handler". The second alarm will be automatically raised by the system after the crash timeout, unless an event tagged with label l took place before. The crash timeout must be greater than the delay timeout and will be handled using the associated "crash handler".

The link between the two alarms set and the condition under which they should have no consequences on the system behavior is given by the label l. In fact, if – after having set the alarms – an event is perceived and it matches a check timeout (explained in the sequel) identified by the same label l, then it means that the condition associated with the alarms has been satisfied and they can be switched off.

From an implementation viewpoint, a `set_timeout` event should save the information in the associated list for successive retrieval using the label as a key. In our SWI-Prolog prototype described in Section III-B, this is done by calling the `set_alarm` predicate which, for each element in the list, asserts it, computes the time when the two alarms should be raised based on the current time and the delay and crash timeouts, and generates the two awake events which will actually take place in the system at due time.

Example:

```
set_timeout( (first_move(A, _SomeRoom, key_room),0),
[timeout_setting(t1(A),
d(1000, handlerOmission(A)),
c(2000, handlerCrash(A)) ) ] ).
```

When one event whose type is `first_move(A, _SomeRoom, key_room)` takes place[2], two timeouts labeled with `t1(A)` together with their handlers are set:

- one whose expiration may be due to omission failures `d(1000, handlerOmission(A))`, meaning that if some check timeout event labeled with `t1(A)` will not take place within `1000` time units from the current time, then the agent or component that should make the condition associated with label `t1(A)` become true will be considered delayed;

[2]The fact that the event type is represented as a couple `(first_move(A, _SomeRoom, key_room),0)` with a `0` as second element means that the type is a "producer event type". This notion is introduced in the sequel.

- and one whose expiration may be due to crashes `c(2000, handlerCrash(A))`, meaning that if the check timeout events labeled with `t1(A)` will not take place within `2000` time units from the current time, then the agent or component that should make the condition associated with label `t1(A)` become true will be considered crashed.

When the timeout will expire, the corresponding event of type "awake" (explained later) will be generated by the sentinel, and the handler will be used to manage the failure.

In this specific example, since the agent identity is part of the label itself, the agent whose delay or crash is under monitoring is `A`. As soon as `A` makes its `first_move` into the `key_room`, two countdowns start. If `A` does not ask the `key` to the `key_keeper` (see example below) within 1000 time units, he will be considered delayed. If he does not ask the `key` in 2000 time units, he will be considered crashed.

Check timeout type. A check timeout type $\mathcal{CT}$ is, from an abstract point of view, a quadruple <type of the event that should take place within the timeout, label of the timeout, pointer to the timeout exception handler, arguments of the timeout exception handler>. As for the exception type, we use a more compact representation as a term with its functor and arguments.

Example:

```
check_timeout((ask(A, key_keeper, key),0),timeout_exc(t1(A),
handlerEventWithDelay(ask(A, key_keeper, key))).
```

When one event of type `ask(A, key_keeper, key)` takes place, a check if it is on time or not is performed, by retrieving the information associated with the label `t1(A)`. In our implementation, this means looking for a fact previously asserted by `set_alarm` whose label unifies with `t1(A)`. If the event is on time, the protocol is respected and alarms associated with label `t1(A)` are switched off (namely, it is tagged as "done" and the awake events that have been already generated by set_alarm will have no effect). If the event is not on time, then one or both alarms associated with `t1(A)` have expired and the corresponding handlers have been executed. Note that the crash handler is supposed to be executed when there is a high degree of confidence that the expected event will not take place anymore. Since handlers could be generic ones, we leave the developer the possibility to specify some further code, `handlerEventWithDelay(ask(A, key_keeper, key))` in this case, to be executed when the delayed event takes place.

Awake type. An awake event type $\mathcal{AW}$ is a couple <awake type (delay or crash), label of the timeout>.

It is automatically generated when the countdown associated with a delay or crash alarm reaches 0.

Example:

```
awake_delay(t1(A)) (resp. awake_crash(t1(A))).
```

When one event typed with these awake types takes place, then the delay (resp. crash) handler defined when the timeout for `t1(A)` was set will be called in order to cope with the omission (resp. crash) failure.

Uninteresting events. Events which are not interesting for the monitoring purposes can be tagged as uninteresting ones just by exploiting the event type. For example, the following definition of the `has_type` predicate states that all the actual events observed in the system which do not unify either with `truck_at_dock(_, _, _)`, or with `drop_parcel(_, _)`, etc, have type `uninteresting_event`.

```
has_type(Event, uninteresting_event) :-
   /* must list explicitly all the interesting events, as
        different (\=) from the current one */
   Event \= truck_at_dock(_, _, _),
   Event \= drop_parcel(_, _),
   Event \= move_to_free_shelf(_, _, _, _),
   Event \= collect_parcel(_, _),
   Event \= move_to_truck(_, _, _, _).
```

Any multiagent protocol can be defined as the interesting part of the protocol, interleaved with the uninteresting one. Interleaving is modeled by the fork operator "`|`" (see more details below) and the branch of the protocol modeling the uninteresting events can be defined in a standard way as

```
DISCARD = (uninteresting_event, 0): DISCARD
```

meaning that a `DISCARD` constrained global type is defined as an infinite sequence (sequence operator "`:`") of uninteresting events.

Producers and consumers. In order to model constraints across different branches of a constrained fork (explained later in this section), we introduce two different kinds of event types, called *producers* and *consumers*, respectively. Each occurrence of a producer event type must correspond to the occurrence of a new event; in contrast, consumer event types correspond to the same event specified by a certain producer event type. The purpose of consumer event types is to impose constraints on event sequences, *without introducing new events*. A consumer is an event type η, whereas a producer is an event type η equipped with a natural superscript n. In the Prolog concrete syntax that we use, η^n is represented by the couple (η, n).

Constrained global types. A constrained global type τ represents a set of possibly infinite sequences of events, and is defined on top of the following type constructors:

- λ (empty sequence): the singleton set $\{\epsilon\}$ containing the empty sequence ϵ.
- $\eta^n{:}\tau$ (*seq-prod*): the set of all sequences whose first element is an event e matching type η ($e \in \eta$), and the remaining part is a sequence in the set represented by τ. The superscript n specifies the number n of corresponding consumers that coincide with the same event type η; hence, n is the required number of times $e \in \eta$ has to be "consumed" to allow a transition labeled by e.
- $\eta{:}\tau$ (*seq-cons*): a consumer of event e matching type η ($e \in \eta$), and followed by any sequence in the set represented by τ.

- $\tau_1 + \tau_2$ (*choice*): the union of the sequences of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*fork*): the set obtained by shuffling the sequences in τ_1 with those in τ_2.
- $\tau_1 \cdot \tau_2$ (*concat*): the concatenation of the sequences of τ_1 and τ_2.
- The "meta-construct" *fc* (for *finite composition*) takes τ, the constructor cn, and the positive natural number n as inputs and generates the "normal" constrained global type (τ cn τ cn ... cn τ) (n times, hence $fc(\tau, cn, 1)$=τ).

Constrained global types are regular terms, that is, can be cyclic (recursive), and hence they can be represented by a finite set of syntactic equations. To make the treatment simpler, we limit our investigation to *contractive* (a.k.a. *guarded*) and *deterministic* constrained global types. A constrained global type τ is *contractive* if all infinite paths[3] in τ contain an occurrence of the ":" constructor. Determinism ensures that dynamic checking can be performed efficiently without backtracking. Intuitively, a constrained global type is deterministic if, in case more transition rules can be applied when event e takes place, they lead to equivalent constrained global types. The formal definition is given in the next section.

B. Semantics

The state of a constrained global type τ can be represented by τ itself. In this section, when talking about constrained global types we will refer to their current state. Also, we will use "constrained global type" and "protocol" interchangeably.

The interpretation of a constrained global type is based on the notion of transition, a total function δ:$\mathbb{N} \times \mathcal{CT} \times \mathcal{A} \rightarrow \mathcal{P}_{fin}(\mathcal{CT} \times \mathbb{N})$, where $\mathcal{CT}$ and $\mathcal{A}$ denote the set of contractive and constrained global type and of events, respectively. The `next/5` Prolog predicate discussed below implements the δ transition function that we do not show in this paper for space constraints. If τ_1 represents the current state of the protocol and the event e takes place, then the protocol can move to τ_2 iff $\delta(0, \tau_1, e) = (\tau_2, 0)$, that we write as $\tau_1 \overset{e}{\rightarrowtail} \tau_2$.

Moving from the definition of δ to its Prolog implementation, the following relationship holds:

$\delta(N, T1, Ev) = (T2, M)$ iff `next(N,T1,Ev,T2,M).`

The auxiliary function ϵ(_), implemented by the `empty/1` Prolog predicate, specifies the constrained global types whose interpretation contains the empty sequence ϵ. Intuitively, a constrained global type τ s.t. $\epsilon(\tau)$ holds specifies a protocol that is allowed to successfully terminate.

Let τ_0 be a contractive and constrained global type. A *run* ρ for τ_0 is a sequence $\tau_0 \overset{e_0}{\rightarrowtail} \tau_1 \overset{e_1}{\rightarrowtail} \ldots \overset{e_{n-1}}{\rightarrowtail} \tau_n \overset{e_n}{\rightarrowtail} \tau_{n+1} \overset{e_{n+1}}{\rightarrowtail} \ldots$ such that (1) either the sequence is infinite, or it has finite length $k \geq 0$ and the last constrained global type τ_k verifies $\epsilon(\tau_k)$; and (2) for all τ_i, e_i, and τ_{i+1} in the sequence, $\tau_i \overset{e_i}{\rightarrowtail} \tau_{i+1}$ holds. We denote by $A(\rho)$ the possibly empty or infinite sequence of events $e_0 e_1 \ldots e_n \ldots$ contained in ρ. The interpretation $[\![\tau_0]\!]$ of τ_0 is the set $\{A(\rho) \mid \ \rho$ is a run for $\tau_0\ \}$. A contractive constrained global type τ is deterministic if for any possible run ρ of τ and any possible τ' in ρ, if $\tau' \overset{e}{\rightarrow} \tau''$, and $\tau' \overset{e}{\rightarrow} \tau'''$, then $[\![\tau'']\!] = [\![\tau''']\!]$.

[3]By "path of a constrained global type" we mean "path in the possibly infinite tree corresponding to the term that represents the constrained global type".

The next and empty predicates.

The `next` predicate implements the δ function defining the semantics of constrained global types extended with constructs to deal with exceptions, timeouts and their handlers. Besides the definition of transition rules for "normal" constrained global types, in the sequel we describe the additional transition rules for the new constructs introduced in this paper.

Basic cases for "normal" constrained global types.

If the constrained global type is a sequence (":" operator) consisting of a producer event type EvType with associated number N, followed by the constrained global type T (second argument), and the actual event perceived in the environment is Ev (third argument), and Ev has type EvType, then (EvType,N):T can move to T (fourth argument). If there were no events of type EvType to be consumed (first argument of next is zero), now N events of EvType are generated and must be consumed (fifth argument of next is N, which is the number associated with EvType in the second argument)

```
next(0,(EvType,N):T,Ev,T,N) :- has_type(Ev, EvType).
```

This rule is similar to the previous one, but the event type EvType is a consumer (in fact, it has no number associated) and, if the actual event Ev perceived from the environment has type EvType, then EvType:T1 moves to T1 and the events of type EvType still to be consumed are decreased by one.

```
next(M,EvType:T1,Ev,T1,N) :-
EvType \= (_,_), has_type(Ev, EvType),
M > 0, N is M - 1.
```

If T1 can move to T2, then the choice between T1 and any other constrained global type, T1+_, can move to T2 (and the converse for the second rule)

```
next(M,T1+_,Ev,T2,N) :- next(M,T1,Ev,T2,N).
next(M,_+T1,Ev,T2,N) :- !, next(M,T1,Ev,T2,N).
```

If T1 can move to T3, then the interleaving between T1 and T2, T1|T2, can move to T3|T2 (and the converse for the second rule)

```
next(N,T1|T2,Ev,T3|T2,M) :- next(N,T1,Ev,T3,M).
next(N,T1|T2,Ev,T1|T3,M) :- next(N,T2,Ev,T3,M).
```

If T1 can move to T3 and T2 can move to T4, and T1 produces event types which can be consumed by T2, then T1 and T2 can synchronize and T1|T2 can move to T3|T4 (and the converse for the second rule)

```
next(N,T1|T2,Ev,T3|T4,P) :-
   next(N,T1,Ev,T3,M), M > 0, next(M,T2,Ev,T4,P).
next(N,T1|T2,Ev,T4|T3,P) :-
   !,next(N,T2,Ev,T3,M), M > 0, next(M,T1,Ev,T4,P).
```

If T1 can move to T3, then the concatenation T1*T2 can move to the concatenation T3*T2; if T1 contains ϵ, that is, empty(T1) holds (second rule), then T1*T2 can move to T3 if T2 can move to T3

```
next(M,T1*T2,Ev,T3*T2,N) :-
   next(M,T1,Ev,T3,N).
next(M,T1*T2,Ev,T3,N) :-
   !,empty(T1), next(M,T2,Ev,T3,N).
```

Rule for finite composition of constrained global types.

```
fc(T, T1, C, N) :- N>1, copy_term(T1, Fresh1),
N1 is N-1, fc(T2, T1, C, N1),  T =.. [C, Fresh1, T2].
fc(Fresh1, T1, _C, 1) :- copy_term(T1, Fresh1).
```

Additional rule for dealing with exceptions.

If the constrained global type is a sequence consisting of an event of type EvType, which raises an exception to be handled with the Handler code, followed by T, and the actual event perceived in the environment is Ev, and Ev has type EvType, then the Handler is executed and exception(EvType,Code):T moves to T.

```
next(0, exception(EvType,Handler):T, Ev,T,0) :-
   has_type(Ev, EvType), call(Handler).
```

Additional rules for dealing with timeouts.

If the constrained global type is a sequence consisting of a set_timeout event with its parameters, followed by T, and the perceived event Ev has type EvType, then the alarm is set by calling the set_alarm code. We do not show the rule dealing with the case where EvType is a consumer event type rather than a producer.

```
next(0,set_timeout((EvType,N), List):T, Ev,T,N) :-
   has_type(Ev,EvType), set_alarm(List).
```

If the constrained global type is a sequence consisting of a check_timeout event with its parameters, including the label of the timeout handler to check and the Code to execute if the timeout is expired, followed by T, and the perceived event Ev has type EvType, then the actual time is retrieved together with the timeout settings associated with label L. If the timeout is expired, then Handler is executed, otherwise the two alarms are switched off by setting their handlers to "done". The constrained global time moves to T. We do not show the rule dealing with the case where EvType is a consumer event type rather than a producer.

```
next(0,check_timeout((EvType,N), timeout_exc(L, Handler)):T,
Ev,T,N) :-
   has_type(Ev, EvType), get_time(CurrentTime),
   timeout_setting(L, d(Delay, _HandlDTimeExp), _C),
   (CurrentTime > Delay ->
     call(Handler);
     (retract(timeout_setting(L, d(Delay, _), c(Crash, _))),
      assert(timeout_setting(L, d(Delay, done), c(Crash,
           done))))
    ).
```

If the constrained global type is a sequence consisting of an awake_delay event with a label L as argument, followed by T, and the perceived event is awake_delay(L) which is automatically generated when a timeout expires, then the code associated with that awake event is executed and this fact is recorded by setting the handler to "done" for avoiding successive re-executions. The constrained global time moves to T.

```
next(N,awake_delay(L):T,awake_delay(L),T,N) :-
   timeout_setting(L, d(Delay, HandlDTimeExp), C),
   HandlDTimeExp \== done,
   call(HandlDTimeExp),
   retract(timeout_setting(L, d(Delay, HandlDTimeExp), C)),
   assert(timeout_setting(L, d(Delay, done), C)), !.

next(N,awake_crash(L):T,awake_crash(L),T,N) :-
   timeout_setting(L, D, c(Crash, HandCTimeExp)),
   HandCTimeExp \== done,
   call(HandCTimeExp),
   retract(timeout_setting(L, D, c(Crash, HandCTimeExp))),
   assert(timeout_setting(L, D, c(Crash, done))), !.
```

Code for setting the alarms.

```
set_alarm([]).

set_alarm(
   [timeout_setting(L, d(D, HandlDTimeExp),
                       c(C, HandCTimeExp))|T]) :-
   asserta(timeout_setting(L, d(D, HandlDTimeExp),
                              c(C, HandCTimeExp))),
   get_time(CurrentTime), !,
   DT is CurrentTime + D,
   CT is CurrentTime + C,
   generate_event(DT, awake_delay(L)),
   generate_event(CT, awake_crash(L)),
   set_alarm(T).
```

The `empty` predicate is true on the empty constrained global type `lambda`. If $T1$ or $T2$ contains ϵ, then $T1 + T2$ contains ϵ. If both $T1$ and $T2$ contain ϵ, then $T1|T2$ and $T1 * T2$ contains ϵ.

```
empty(lambda) :- !.
empty(T1+T2) :- (empty(T1),!;empty(T2)).
empty(T1|T2) :- !,empty(T1),empty(T2).
empty(T1*T2) :- !,empty(T1),empty(T2).
```

IV. EXAMPLES

In all the examples of this paper, event types have the same syntax of the events they refer to, apart from the "uninteresting_event" type which must be explicitly defined as all the events which do not unify with an interesting event. This means that, apart from uninteresting ones, event e has event type e.

A. Example 1: illegal moves

This protocol states that, in order to enter the treasure room, an agent first moves to the key room, asks for the key of the treasure room to the key keeper within 1000 secs, waits for an acceptance, enters the treasure room and then exits it. If the key keeper refuses to give the key, the agent must exit the key room. One exception to this normal flow is that from an external room (we do not care which one) the agent enters the treasure room with some trick without asking the permission to the key keeper and without waiting for the key. In this case, an exception is raised and the exception handler associated with this "illegal move" exception must be executed. Among the actions implemented by the handler, some will be executed (and consequently monitored too) by the sentinel. In particular, the sentinel will sanction the agent, which will be transported out of the treasure room, and in jail. After some time, the agent will be set free again. Note that here, the sentinel monitors one of its actions, `sanction(sentinel, A)`, implementing a form of introspection.

The last line of the protocol models the case when the event that has been sensed is unknown. If the `unknown_event(A)` event type holds on all the events that the sentinel is not able

to manage, then this exception will be thrown whenever the sentinel is facing an unexpected exception. The handler can be some general code trying to ensure the vital functions of the system even if the normal flow has some fail.

```
T = ((set_timeout((move(A, _SomeRoom, key_room),0),
     [timeout_setting(
        t1(A),
        d(1000, handlerDTimeoutExpiration(A)),
        c(2000, handlerCTimeoutExpiration(A)))]):
  ((check_timeout((ask(A, key_keeper, key),0),
     timeout_exc(
     t1(A),
     handlerEventWithDelay(ask(A, key_keeper, key)))):
     (
       ((accept(key_keeper, A, key),0):(give(key_keeper, A,
           key),0):
       (move(A, key_room, treasure_room),0):(move(A,
           treasure_room, key_room),0):
       (give(A, key_keeper:key),0):lambda)
     +
       ((refuse(key_keeper, A, key),0):
       (move(A, key_room, _AnotherRoom),0):lambda)
    )*T)
+
   ((exception(move(A, _SomeRoom, treasure_room),
     illegal_move_exc(
          entering_treasure_room_without_permission(A)))):
    (sanction(sentinel, A),0):(transport(A, treasure_room,
         key_room),0):
    (transport(A, key_room, jail),0):(transport(A, jail,
         _AnyRoom),0):lambda)
  )
 )
+
 ((exception(unknown_event(A), unknown_event_exc(A))):lambda
      )).
```

Now we show some events traces compliant with the protocol, generated using SWI Prolog implementing the `next` predicate defined in Section III and the `accept` predicate that accepts (or generates, if the free variables can be grounded when `has_type` is called) sequences of events with a given length `N`, which respect the constrained global type `T`. By using the `findall` builtin predicate on `accept`, we were able to generate all the accepted traces of a given length.

```
accept(N,T,[*]) :- empty(T), N==0,!.
accept(0,_,[]) :- !.
accept(N,T1,[Ev|L]) :-
   next(0,T1,Ev,T2,0), M is N-1, accept(M,T2,L).
```

Correct traces with length 6 include:

```
move(alice, room1, key_room),
ask(alice, key_keeper, key),
accept(key_keeper, alice, key),
give(key_keeper, alice, key),
move(alice, key_room, treasure_room),
move(alice, treasure_room, key_room)

move(alice, room1, key_room),
ask(alice, key_keeper, key),
refuse(key_keeper, alice, key),
move(alice, key_room, room1),
move(alice, room1, key_room),
ask(alice, key_keeper, key)

move(alice, room1, key_room),
ask(alice, key_keeper, key),
refuse(key_keeper, alice, key),
move(alice, key_room, room2),
move(alice, room2, key_room),
ask(alice, key_keeper, key)
```

Some traces with length 6 where an exception is found and the sentinel sanctions alice are:

```
move(alice, room1, key_room),
move(alice, key_room, treasure_room),
sanction(sentinel, alice),
transport(alice, treasure_room, key_room),
transport(alice, key_room, jail),
transport(alice, jail, room1)

move(alice, room1, treasure_room),
sanction(sentinel, alice),
transport(alice, treasure_room, key_room),
transport(alice, key_room, jail),
transport(alice, jail, room1),
move(alice, room1, key_room)
```

B. Example 2: dock loading and unloading

This example is a simplified version of the scenario described in [35]:

> *In the loader dock several workers wander around or carry parcels between incoming trucks and the shelves. Their job is to unload parcels from a full truck, or to deliver parcels to an empty one, whenever trucks arrive at the store. The dock itself contains shelves where parcels can be placed temporally. The shelves are separated by corridor ways, which are used by workers to transport the parcels. ... When a worker makes an intention to move somewhere, it communicates this intention to other ones for the purpose of coordination. A trajectory of the movement, described by points and exact time-values, is sent to each worker, so it can update its beliefs about future changes in the environment. This information is stored in a domain time model representing the beliefs concerning prospective workers' positions. This information guides the planning process of each worker, so none of them intersects a path of another.*

In this example we assume that workers and trucks are represented as agents, which are free to execute many actions including those related to the work in the dock loader. The next code manages a simplification of the described example, without monitoring for example that agents will not crash while moving: its aim is to give a flavor of how we can use the uninteresting events to specify only the subset of the actions that are relevant for the controller.

The protocol starts when a `Truck` arrives at the dock (`UNLOAD_TRUCK`), carrying `NumberOfParcels` parcels, which must be unloaded (so, after this event, the protocol moves on with `UNLOAD_ALL_PARCELS`, that is created repeating the trace `UNLOAD_PARCEL` for each parcel in a parallel way using the | constructor). The protocol part identified by `UNLOAD_PARCEL` states that the correct sequence is that firstly an agent comes to the `Truck` position, then it collects a parcel, moves to a free shelf and leaves there the parcel.

We model the positions inside the dock as couples of points, and the time slots as couples of minutes.

To model the fact that we are only interested in monitoring the events specified in the previous traces, but not the others (that is, if one event that is not related to this protocol takes place during the protocol, it is not a violation of it!), we add in the trace `UNLOAD_TRUCK` a fork (| operator) with `DISCARD`, so that any event of type `uninteresting_event`, at any time, is caught by the trace `DISCARD` (that simply

skips that event and waits for the next one). The type `uninteresting_event` is defined differently from all the other events: an `uninteresting_event` is whatever event that is not `move_to_truck`, `collect_up_parcel`, and so on.

```
UNLOAD_PARCEL =
(move_to_truck(Agent, _FromPosition, TruckPosition,
     _TimeSlot),0):
(collect_parcel(Agent, P),0):
(move_to_free_shelf(Agent, TruckPosition, _FreeShelfPosition
     , _SuccTimeSlot),0):
(drop_parcel(Agent, P),0):
lambda,
fc(UNLOAD_ALL_PARCELS, UNLOAD_PARCEL, |, NumberOfParcels),
UNLOAD_TRUCK =
  (((truck_at_dock(_TruckId,TruckPosition,NumberOfParcels)
       ,0):
     UNLOAD_ALL_PARCELS)|
  DISCARD),
DISCARD = (uninteresting_event, 0): DISCARD.
```

An example of a trace (including events not interesting for the protocol) which is recognized as compliant with the protocol is:

```
snoring(a1),
truck_at_dock(truck1, (0, 0), NumberOfParcels),
move_to_truck(a1, (6, 21), (0, 0), (10, 15)),
collect_parcel(a1, p1),
move_to_truck(a2, (12, 3), (0, 0), (15, 18)),
snoring(a1),
smiling(a1),
move_to_free_shelf(a1, (0, 0), (54, 3), (15, 22)),
drop_parcel(a1, p1),
move_to_truck(a1, (54, 3), (0, 0), (10, 15)),
collect_parcel(a2, p2),
snoring(a2),
dancing_tiptap(a1),
move_to_free_shelf(a2, (0, 0), (43, 78), (18, 21)),
drop_parcel(a2, p2),
move_to_truck(a1, (4, 9), (0, 0), (30, 35)),
collect_parcel(a1, p3),
move_to_truck(a3, (44, 9), (0, 0), (31, 37)),
collect_parcel(a3, p4),
move_to_free_shelf(a3, (0, 0), (6, 24), (37, 49)),
drop_parcel(a3, p4),
move_to_free_shelf(a1, (0, 0), (1, 24), (35, 43)),
drop_parcel(a1, p3),
snoring(truck),
smiling(truck),
drinking_a_beer(truck)
```

whereas the following trace is not compliant because parcel `p1`, collected up by agent `a1` (third event), is dropped by agent `a4` (fourth event) instead of being dropped by `a1`.

```
truck_at_dock(truck1, (0, 0), NumberOfParcels),
move_to_truck(a1, (6, 21), (0, 0), (10, 15)),
collect_parcel(a1, p1),
drop_parcel(a4, p1).
```

C. Example 3: air conditioning

The last example relates to the AmI context, where we assume to manage a building for social events (concerts, workshops and so on): in this building there are many rooms, and we would add some intelligence to the system itself, specifying some rules that should be respected in the building. In particular, if there are more then 1000 visitors and the temperature inside the building is >30, then if the air conditioning is closed, it must be switched on, otherwise, no more visitors can enter the building.

This situation is (partially) described by the constrained global type AC:

```
SAFETY_CHECK =
(
  (
     ((counted_more_than_1000_visitors,0):
      lambda)
     |
     ((temperature_higher_than_30_celsius,0):
      lambda)
  )
  *
  (
     ((air_conditioning_off,0):
      (switch_air_conditioning_on,0):
      lambda
     )
     +
     ((air_conditioning_on,0):
      (do_not_admit_other_visitors,0):
      lambda
     )
   )
)
*
SAFETY_CHECK,
AC = (SAFETY_CHECK | DISCARD),
DISCARD = (uninteresting_event, 0): DISCARD.
```

Some examples of traces which are recognized as compliant with the protocol are:

```
counted_more_than_1000_visitors,
temperature_higher_than_30_celsius,
air_conditioning_on,
do_not_admit_other_visitors.

counted_more_than_1000_visitors,
temperature_higher_than_30_celsius,
air_conditioning_off,
switch_air_conditioning_on

temperature_higher_than_30_celsius,
concert_begins,
counted_more_than_1000_visitors,
air_conditioning_off,
switch_air_conditioning_on
```

In this case the constrained global type is used to control that if a specific situation happens (visitors $>$ 1000 and temperature $>$ 30, with events perceived in any order), some specific action (`do_not_admit_other_visitors` or `switch_air_conditioning_on`) is taken.

V. Conclusions

In this paper we described an approach for designing, formalizing and implementing *sentinels* that detect errors in *fail-uncontrolled multiagent systems*, and *controllers* that identify particular situations in *AmI systems*. Our approach – like all the approaches aimed at runtime verification of system properties – requires that a formal representation of the system property (a MAS protocol in our case) is available. We used *constrained global types extended with constructs for modeling exceptions, timeouts, and handlers* as representation formalism. We implemented the SWI Prolog interpreter for this formalism and we run different tests in different application domains.

Representing a protocol using our formalism may not be trivial, but the more expressive is the formalism, the more difficult is using it. When used for runtime verification, our formalism is more expressive than linear time temporal logic (LTL [36]) because it can represent a protocol like $a^n b^n$ which

cannot be expressed using LTL[4]. For this reason it should not be surprising that our formalism is more complex than, for example, LTL.

There are however many benefits in using such a formalism:

- having a protocol formalization can help testing the MAS itself, since our Prolog translation can be exploited to generate and test correct, an not correct, simulations of the system, and constrained global types have already been proven a powerful language, able to model complex protocols (as shown in [12]);
- in *AmI systems*, *controllers* can be added on top of the already existing agents, as an independent layer, improving the modularity of the overall system and simplifying the design, development and maintenance phases while, in *fail-uncontrolled MASs*, *sentinels* may help to identify and recover from error situations by separating the errors management from the normal protocol flow;
- the basic version of our approach, without the extensions discussed in this paper, works for JADE and Jason, two among the most used platforms for MAS development; the transition rules providing the semantics of the improved formalism presented in this paper have been already translated into Prolog, the language we used for implementing our approach on top of JADE and Jason, so we claim that extending the JADE and Jason prototypes with exceptions, timeouts, and handlers should raise no technical problems;
- the extensions presented in this paper enhance the "MAS control layer" with the possibility to manage both MASs where a predefined global protocol does not exist (*AmI systems*) and MASs where errors may arise and must be formalized and managed in a controlled way.

The formalism that we have discussed in this paper seems suitable for being integrated into a sensor network in order to capture what happens in the environment, as we did in [12]. Thanks to its modularity it could be used to monitor and supervise environments where physical agents (robots) can fail leading to exceptional situations and need to be either reconfigured or replaced at runtime.

The feasibility of the approach is limited to situations where a formal representation of entities, messages and in general events, exists: so, when considering dynamic and open large-scale MASs as those in AmI and wireless sensor networks, we assume that all the agents that can join the MAS (maybe developed by third parties), are foreseen in the formalization. Our work is meant to provide a set of tools and languages for monitoring distributed systems, without claiming to be able to solve all the related problems.

For example, although we can automatically project the protocol onto subsets of agents (it means that the protocol itself can be split into separated parts) and then allocate to different controllers the supervision of those parts of the MAS, we have not solved the problem of *how to* identify the best protocol distribution, which is not even always possible. In a similar way, the decision of what to do in case of a failure is demanded to the protocol formalization or to the actual agent code, and is out of the scope of our work.

As part of our future work, we are investigating how expressing more complex behaviors when specifying timeout, and how managing timeouts related to the same agent. From a more theoretical point of view, the relationships between our formalism and timed linear time temporal logic [37] should be deeply analyzed.

[4]The protocol where one agent sends n messages of type a to another agent (with n that can be any positive number) and the receiver answers with n messages of type b.

ACKNOWLEDGMENT

This work has been partially supported by the MIUR PRIN Project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society, prot. 2010LHT4KM.

REFERENCES

[1] Y. Demazeau, "From interactions to collective behaviour in agent-based systems," in *In Proceedings of the 1st European Conference on Cognitive Science. Saint-Malo*, 1995, pp. 117–132.

[2] M. Barbuceanu and M. S. Fox, "COOL: A language for describing coordination in multi agent systems," in *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, V. R. Lesser and L. Gasser, Eds. The MIT Press, 1995, pp. 17–24.

[3] Y. Abushark, J. Thangarajah, T. Miller, and J. Harland, "Checking consistency of agent designs against interaction protocols for early-phase defect location," in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2014, pp. 933–940.

[4] M. Baldoni, C. Baroglio, and F. Capuzzimati, "Typing multi-agent systems via commitments," in *Engineering Multi-Agent Systems*. Springer, 2014, pp. 388–405.

[5] F. Al-Saqqar, J. Bentahar, K. Sultan, and M. El Menshawy, "On the interaction between knowledge and social commitments in multi-agent systems," *Applied intelligence*, vol. 41, no. 1, pp. 235–259, 2014.

[6] D. Briola, V. Mascardi, and D. Ancona, "Distributed runtime verification of JADE multiagent systems," in *Intelligent Distributed Computing VIII - Proceedings of the 8th International Symposium on Intelligent Distributed Computing, IDC 2014, Madrid, Spain, September 3-5, 2014*, ser. Studies in Computational Intelligence, D. Camacho, L. Braubach, S. Venticinque, and C. Badica, Eds., vol. 570. Springer, 2014, pp. 81–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10422-5_10

[7] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi, "Global protocols as first class entities for self-adaptive agents," in *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), May 4–8, 2015, Istanbul, Turkey*, R. Bordini, E. Elkind, G. Weiss, and P. Yolum, Eds. IFAAMAS/ACM, 2015.

[8] P. Yolum and M. P. Singh, "Reasoning about commitments in the event calculus: An approach for specifying and executing protocols," *Ann. Math. Artif. Intell.*, vol. 42, no. 1-3, pp. 227–253, 2004.

[9] D. Ancona, S. Drossopoulou, and V. Mascardi, "Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason," in *DALT X. Revised, Selected and Invited Papers*, ser. LNAI, vol. 7784. Springer, 2012.

[10] D. Ancona, M. Barbieri, and V. Mascardi, "Constrained global types for dynamic checking of protocol conformance in multi-agent systems," 2013, sAC 2013. ACM.

[11] V. Mascardi and D. Ancona, "Attribute global types for dynamic checking of protocols in logic-based multiagent systems (technical communication)," 2013, to appear in Theory and Practice of Logic Programming, On-line Supplement, as technical communication of the ICLP 2013 conference.

[12] V. Mascardi, D. Briola, and D. Ancona, "On the expressiveness of attribute global types: The formalization of a real multiagent system protocol," in *AI*IA 2013: Advances in Artificial Intelligence - XIIIth International Conference of the Italian Association for Artificial Intelligence, Turin, Italy, December 4-6, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Baldoni, C. Baroglio, G. Boella, and R. Micalizio, Eds., vol. 8249. Springer, 2013, pp. 300–311. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03524-6_26

[13] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[14] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[15] D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert, "Efficient verification of MASs with projections," in *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Revised Selected Papers*, ser. Lecture Notes in Computer Science, F. Dalpiaz and J. D. M. B. van Riemsdijk, Eds., vol. 8758, 2014, pp. 246–270.

[16] N. Faci, Z. Guessoum, and O. Marin, "DimaX: A fault-tolerant multi-agent platform," in *EUMAS*, ser. CEUR Workshop Proceedings, B. Dunin-Keplicz, A. Omicini, and J. A. Padget, Eds., vol. 223. CEUR-WS.org, 2006.

[17] D. Powell, Ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing (Research Reports Espirit Project 818/2252, Delta 4, Vol. 1)*, 1992.

[18] S. Hägg, "A sentinel approach to fault handling in multi-agent systems," *Second Australian Workshop on Distributed Artificial Intelligence, Cairns, QLD, Australia, August 27, 1996*, vol. s. 181-95, 1997.

[19] J. Eder and W. Liebhart, "The workflow activity model WAMO," in *Proc. 3rd Int. Conf. Cooperative Inf. Syst.*, 1995, pp. 87–98.

[20] A. U. Mallya and M. P. Singh, "Modeling exceptions via commitment protocols," in *AAMAS*, F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, Eds. ACM, 2005, pp. 122–129.

[21] Z. Guessoum, M. Ziane, and N. Faci, "Monitoring and organizational-level adaptation of multi-agent systems," in *AAMAS*. IEEE Computer Society, 2004, pp. 514–521.

[22] M. H. Coen, "Design principles for intelligent environments," in *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, J. Mostow and C. Rich, Eds. AAAI Press / The MIT Press, 1998, pp. 547–554. [Online]. Available: http://www.aaai.org/Library/AAAI/1998/aaai98-077.php

[23] M. C. Mozer, "The neural network house: An environment that adapts to its inhabitants," in *Proc. Am. Assoc. Artificial Intelligence Spring Symp. Intelligent Environments*. AAAI Press, 1998, pp. 110–114.

[24] P. Davidsson and M. Boman, "Saving energy and providing value added services in intelligent buildings: A MAS approach," in *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zürich, Switzerland, September 13-15, 2000, Proceedings*, ser. Lecture Notes in Computer Science, D. Kotz and F. Mattern, Eds., vol. 1882. Springer, 2000, pp. 166–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45347-5_14

[25] H. Hagras, V. Callaghan, M. Colley, G. Clarke, A. Pounds-Cornish, and H. Duman, "Creating an ambient-intelligence environment using embedded agents," *IEEE Intelligent Systems*, vol. 19, no. 6, pp. 12–20, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MIS.2004.61

[26] F. Doctor, H. Hagras, and V. Callaghan, "A fuzzy embedded agent-based approach for realizing ambient intelligence in intelligent inhabited environments," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 35, no. 1, pp. 55–65, 2005. [Online]. Available: http://dx.doi.org/10.1109/TSMCA.2004.838488

[27] C. Ramos, J. C. Augusto, and D. Shapiro, "Ambient intelligence - the next step for artificial intelligence," *IEEE Intelligent Systems*, vol. 23, no. 2, pp. 15–18, 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MIS.2008.19

[28] J. M. Gascueña, E. Navarro, P. Fernández-Sotos, A. Fernández-Caballero, and J. Pavón, "Idk and icaro to develop multi-agent systems in support of ambient intelligence," *Journal of Intelligent and Fuzzy Systems*, 2014.

[29] M. Villaverde, D. Perez, and F. Moreno, "Cooperative learning model based on multi-agent architecture for embedded intelligent systems," in *Industrial Electronics Society, IECON 2014-40th Annual Conference of the IEEE*. IEEE, 2014, pp. 2724–2730.

[30] A. Olaru, "Context-awareness in multi-agent systems for ambient intelligence," in *Context in Computing*. Springer, 2014, pp. 541–556.

[31] M. Ruta, F. Scioscia, G. Loseto, and E. Di Sciascio, "Semantic-based resource discovery and orchestration in home and building automation: a multi-agent approach," *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 1, pp. 730–741, 2014.

[32] A. Freitas, D. Schmidt, A. Panisson, R. H. Bordini, F. Meneguzzi, and R. Vieira, "Applying ontologies and agent technologies to generate ambient intelligence applications," in *Agent Technology for Intelligent Mobile Services and Smart Societies*. Springer, 2015, pp. 22–33.

[33] A. Chaouche, A. E. Fallah-Seghrouchni, J. Ilié, and D. Saïdouni, "A higher-order agent model with contextual planning management for ambient systems," *T. Computational Collective Intelligence*, vol. 8780, pp. 146–169, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44871-7_6

[34] G. Gutnik, "Monitoring large-scale multi-agent systems using overhearing," in *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '05. New York, NY, USA: ACM, 2005, pp. 1377–1377. [Online]. Available: http://doi.acm.org/10.1145/1082473.1082785

[35] A. Walczak, L. Braubach, A. Pokahr, and W. Lamersdorf, "Augmenting bdi agents with deliberative planning techniques," in *Proceedings of the 4th International Conference on Programming Multi-agent Systems*, ser. ProMAS'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 113–127. [Online]. Available: http://dl.acm.org/citation.cfm?id=1759324.1759334

[36] A. Pnueli, "The temporal logic of programs," in *FOCS*, IEEE, Ed., 1977, pp. 46–57.

[37] J.-F. Raskin, "Logics, automata and classical theories for deciding real-time," Ph.D. dissertation, Namur, Belgium, 1999.