Formal Specification of Fault-Tolerant Multi-agent Systems

Elena Troubitsyna(✉)

KTH – Royal Institute of Technology, Stockholm, Sweden
elenatro@kth.se

Abstract. Multi-agent systems (MAS) are increasingly used in critical applications. To ensure dependability of MAS, we need to formally specify and verify their fault tolerance, i.e., to ensure that collaborative agent activities are performed correctly despite agent failure. In this paper, we present a formalisation of fault tolerant MAS and use it to define specification and refinement patterns for modelling MAS in Event-B.

Keywords: Formal modelling · MAS · Fault tolerance · Event-B

1 Introduction

Mobile multi-agent systems (MAS) are complex decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programs acting autonomously on behalf of a person or organisation, while coordinating their activities by communication [8,14]. MAS are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas etc. [1,6,7,9]. However, widespread use of MAS is currently hindered by the lack of methods for ensuring their dependability, and in particular, fault tolerance.

In this paper we focus on studying fault tolerance of agent cooperative activities. However, ensuring correctness of complex cooperative activities is a challenging issue due to faults caused by agent disconnections, dynamic role allocation and autonomy of the agent behaviour [4,5,10,11]. To address these challenges, we need the system-level modelling approaches that would support formal verification of correctness and facilitate discovery of restrictions that should be imposed on the system to guarantee its safety.

In this paper we propose a formalisation of properties of fault tolerant MAS and then demonstrate how to specify and verify them in Event-B [3]. The main development technique of Event-B is refinement. It is a top-down approach to formal development of systems that are correct by construction. The system development starts from an abstract specification which defines the main behaviour and properties of the system. The abstract specification is gradually transformed (refined) into a more concrete specification directly translatable into a system implementation. Correctness of each refinement step is verified by proofs.

F. Dignum et al. (Eds.): PAAMS 2021, LNAI 12946, pp. 291–302, 2021.
https://doi.org/10.1007/978-3-030-85739-4_24

These proofs establish system safety (via preservation of safety invariant properties expressed at different levels of abstraction) and liveness (via the provable absence of undesirable system deadlocks). Transitivity of the refinement relation allows us to guarantee that the system implementation adheres to the abstract and intermediate models. The Rodin platform [15] provides the developers with automated tool support for constructing and verifying system models in Event-B.

Our reliance of abstraction and stepwise refinement allows us to rigorously define and verify correctness of agent cooperative activities in presence of agent failure. We consider a hierarchical agent system, i.e., distinguish between the supervisor and subordinate agents. This introduces intricate details into handling the failures of different kinds and performing cooperative error recovery. Event-B allowed us to consider fault tolerance as a system-level property that can be verified by proofs. Hence,we argue that Event-B offers a useful formalisation framework for specification and verification of complex fault tolerant MAS.

2 Fault Tolerant MAS

2.1 Fault Tolerance

The main aim of fault tolerance is to ensure that the system continues to provide its services even in presence of faults [13]. Typically, fault occurrence leads to a certain service degradation. However, it is important to ensure that the system behaves in a predictable deterministic way even in presence of faults.

The main techniques to achieve fault tolerance are error processing and fault treatment [13]. Fault treatment is usually performed while the system is not operational, i.e., during the scheduled maintenance. In this paper, we focus on error processing part of fault tolerance.

Error processing comprises the fault tolerance measures applied while the system is operational. The purpose of error processing is to eliminate an error from the computational state and preclude failure occurrence. Error processing is usually implemented in three steps: error detection, error diagnosis, and error recovery. Error detection determines the presence of error. Error diagnosis evaluates the amount of damage caused by the detected error. Error recovery aims at replacing an erroneous system state with the error-free state.

There are three types of error recovery methods: backward recovery, forward recovery and compensation. Backward recovery tries to return the system to some previous error-free state. Typically, backward recovery is implemented by checkpointing, i.e., periodically, during the normal system operation, the state of the system is stored in the memory. In case of a failure, the system retrieves the information about the error-free state from the memory and resumes its functioning from this states. When implementing forward recovery, upon detection of an error, the system makes a transition to a new error-free state from which it continues to operate. Exception handling is a typical example of forward error recovery. Compensation, typical for complex transactions, is used when the erroneous state contains enough redundancy to enable its transformation to error-free state.

To implement fault tolerance, it is important to understand the types of faults that might occur in the system. A fault can be characterized by their nature, duration or extent [13]. When considering the nature of a fault, we distinguish between random, e.g., hardware failures and systematic faults, e.g., design errors.

Faults can also be classified in terms of their duration into permanent and transient faults. Once permanent fault has occurred, it remain in the system during its entire operational life, if no corrective actions are performed. Transient faults can appear and then disappear after a short time. Moreover, faults can be categorised according to their effect on the system as localized and global ones. Localized faults affect only a single agent. Global faults permeate throughout the system and typically affect some set of agents.

2.2 Fault Tolerant MAS

To achieve fault tolerance while developing MAS, we formally define MAS and the properties that its design should ensure.

Definition 1. A multi-agent system $\mathcal{MAS}$ is a tuple $(\mathcal{A}, \mu, \mathcal{E}, \mathcal{R})$, where $\mathcal{A}$ is a collection of different classes of agents, μ is the system middleware, $\mathcal{E}$ is a collection of system events and $\mathcal{R}$ is a set of dynamic relationships between agents in a MAS.

Each agent belongs to a particular class or type of agents $A_i,\ i \in 1..n$ such that $A_i \in \mathcal{A}$. An agent $a_{ij} \in A_i$ is characterised by its local state that consists of variables determining its behaviour and static attributes. Since agent might fail and be replaced by other agents, the set of agents in each class is dynamic. An agent might experience a transient failure and hence spontaneously disappear from the class and reappear again. Moreover, an agent might fail permanently, i.e., permanently disappear from its class. In a system with redundancy, a failed agent can be replaced by another agent, i.e., a new agent can appear in a class instead of the failed one. An agent might also leave a class in a normal predefined way when its function in the system is completed.

The system middleware μ can be considered as an agent of a special kind that is always present in the system and belongs to its own class. Middleware is fault free, i.e., it always provides its services. The responsibility of the middleware is to maintain the communication between the agents and provide some basic fault tolerance. For instance, middleware is responsible for detecting agent failures. Initially, a failure is considered to be transient. However, if an agent does not recover within a certain deadline then the middleware considers this agent to be failed permanently. When a new agent appears in the system, e.g., to replace the failed agent, middleware provides it with the connectivity with the rest of the agents.

Often some agents in MAS experience a transient loss of connectivity. In this case, middleware maintains their status and state to resume normal operation when the connection is re-established, i.e., provides a backward recovery service.

The system events $\mathcal{E}$ include all internal and external system reactions. An execution of an event may change the state of the middleware or agents.

Usually, a failure of an agent affects it capability to perform its functions, i.e., it might prevent a progress in some collaborative activities with the other agents. Each collaborative activity between different agents (or an agent and the middleware) is composed of a set of events. Hence, an agent failure might disable some events. Therefore, while modelling the behaviour of a MAS, we should also define the functions of the middleware as a set of events and reactions specifying the behaviour in case of transient and permanent faults, as well as explicitly specify the events representing error detection and recovery. Moreover, we should represent the impact of failures on collaborative activities via constraining the set of enabled events. Now we are ready to introduce the first property that a fault tolerant MAS should preserve.

The collaborative actions of fault tolerant MAS should preserve the following enabledness property:

Property 1. *Let $\mathcal{A}_{act}$ and $\mathcal{A}_{ina}$ be sets of active and inactive agents correspondingly, where $\mathcal{A} = \mathcal{A}_{act} \cup \mathcal{A}_{ina}$ and $\mathcal{A}_{act} \cap \mathcal{A}_{ina} = \varnothing$. Let $\mathcal{EAA}$ and $\mathcal{EA}\mu$ be all the collaborative activities (sets of events) between agents and agents and between agents and middleware respectively. Moreover, for each $A \in \mathcal{A}$, let $\mathcal{E}A$ be a set of events in which the agent A is involved. Then*

$$\forall A \cdot \ A \in \mathcal{A}_{act} \Rightarrow \mathcal{E}A \in \mathcal{EAA}$$

and

$$\forall A \cdot \ A \in \mathcal{A}_{ina} \Rightarrow \mathcal{E}A \in \mathcal{EA}\mu$$

This property defines the restrictions on agent behaviour in presence of failures. Essentially, it postulates that if an agent failed, i.e., has become inactive then it cannot participate in any collaborative activities until it recovers, i.e., becomes active. This property can be ensured by checking the status attributes of each agent that should be involved into a collaborative activity.

A collection of system events $\mathcal{R}$ consists of dynamic relationships or connections between active agents of the same or different classes. An agent relationship is modelled as a mathematical relation

$$R(a_1, a_2, ..., a_m) \subseteq C_1^* \times C_2^* ... \times C_m^*,$$

where $C_j^* = C_j \cup \{?\}$. A relationship can be *pending*, i.e., incomplete. This is indicated by question marks in the corresponding places of R, e.g., $R(a_1, a_2, ?, a_4, ?)$. Pending relationships are typically occur during the error recovery. If an agent fails then the middleware detects it, saves the status of an agent and actives the timer bounding the time of error recovery. If an agent recovers before the timeout then the relationships become complete, i.e., all the corresponding events become enabled. However, if an agent fails to recover, its failure is considered to be permanent. Then the middleware tries to replace the failed agent by a healthy one. If it succeeds in doing this then the relationships become complete.

Property 2. *Let $\mathcal{A}_{act}$ be a set of active agents. Let $\mathcal{EAA}$ be all the collaborative activities in which these active agents are involved. Moreover, for each agent $A \in$*

$\mathcal{A}_{act}$, let $\mathcal{R}_A$ be all the relationships it is involved. Finally, for each collaborative activity $CA \in \mathcal{EAA}$, let $\mathcal{A}_{CA}$ be a set of the involved agents in this activity. Then, for each $CA \in \mathcal{EAA}$ and $A_1, A_2 \in \mathcal{A}_{CA}$,

$$\mathcal{R}_{A_1} \cap \mathcal{R}_{A_2} \neq \varnothing$$

This property restricts the interactions between the agents – only the agents that are linked by relationships (some of which may be pending) can be involved into cooperative activities.

The system middleware μ keeps a track of pending relationships and tries to resolve them by enquiring suitable agents to confirm their willingness to enter into a particular relationships. Additional data structure $Pref_R$ associated with a relationship $R \in \mathcal{R}$ can be used to express a specific preference of one agents over other ones. The middleware then enforces this preference by enquiring the preferred agents first. Formally, $Pref_R$ is an ordering relation over the involved agent classes. Thus, for $R \subseteq C_1^* \times ... \times C_m^*$,

$$Pref_R \ \in \ C_1 \times ... \times C_m \ \leftrightarrow \ C_1 \times ... \times C_m.$$

A responsibility of the middleware is detect situations when some of the established or to be established relationships become pending and guarantee "fairness", i.e., no pending request will be ignored forever, as well as try to enforce the given preferences, if possible.

While developing a critical MAS, we should ensure that certain cooperative activities once initiated are successfully completed. These are the activities that implement safety requirements. The ensure safety we have to verify the following property:

Property 3. *Let $\mathcal{EAA}_{crit}$, where $\mathcal{EAA}_{crit} \subseteq \mathcal{EAA}$, be a subset containing critical collaborative activities. Moreover, let $\mathcal{R}_{pen}$ and $\mathcal{R}_{res}$, where $\mathcal{R}_{pen} \subseteq \mathcal{R}$ and $\mathcal{R}_{res} \subseteq \mathcal{R}$, be subsets of pending and resolved relationships defined for these activities. Finally, let $\mathcal{R}_{CA}$, where $CA \in \mathcal{EAA}$ and $\mathcal{R}_{CA} \subseteq \mathcal{R}$, be all the relationships the activity CA can affect. Then, for each activity $CA \in \mathcal{EAA}_{crit}$ and relationship $R \in \mathcal{R}_{CA}$,*

$$\Box((R \in \ \mathcal{R}_{pen}) \rightsquigarrow \ (R \in \ \mathcal{R}_{res}))$$

where $\Box$ designates "always" and $\rightsquigarrow$ denotes "leads to".

This property postulates that eventually all pending relationships should be resolved for each critical cooperative activity. It guarantees that error recovery terminates (either successfully or not).

"The system state p always leads to the state q" or, using the temporal logic notation, "$\Box(p \ \rightsquigarrow \ q)$".

3 Formal Specification in Event B

We start by briefly describing our formal development framework. The Event-B formalism is a variation of the B Method [2], a state-based formal approach

that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems.

Modelling in Event-B. In Event-B, a system specification (model) is defined using the notion of an *abstract state machine* [3]. An abstract state machine encapsulates the model state represented as a collection of model variables, and defines operations on this state, i.e., it describes the dynamic part (behaviour) of the modelled system. A machine may also have the accompanying component, called *context*, which contains the static part of the system. In particular, a context can include user-defined carrier sets, constants and their properties, which are given as a list of model axioms.

The machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialised in the $Init$ event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **Events** clause. Generally, an event can be defined as follows:

$$\textbf{evt} \mathrel{\widehat{=}} \textbf{any } vl \textbf{ where } g \textbf{ then } S \textbf{ end}$$

where vl is a list of new local variables (parameters), the guard g is a state predicate, and the action S is a statement (assignment). In case when vl is empty, the event syntax becomes **when** g **then** S **end**. If g is always true, the syntax can be further simplified to **begin** S **end**.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of assignments. The assignments can be either deterministic or non-deterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A non-deterministic assignment is denoted either as $x :\in Set$, where Set is a set of values, or $x :| P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to Set or according to P.

Event-B Semantics. The semantics of an Event-B model is formulated as a collection of *proof obligations* – logical sequents. Below we describe only the most important proof obligations that should be verified (proved) for the initial and refined models. The full list of proof obligations can be found in [3].

The semantics of Event-B actions is defined using so called before-after (BA) predicates [3]. A before-after predicate describes a relationship between the system states before and after execution of an event, as shown in Fig. 1. Here x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after-state.

Action (S)	$BA(S)$
$x := E(x, y)$	$x' = E(x, y) \ \wedge \ y' = y$
$x :\in Set$	$\exists z \cdot (z \in Set \wedge x' = z) \ \wedge \ y' = y$
$x :\mid P(x, y, x')$	$\exists z \cdot (P(x, z, y) \wedge x' = z) \ \wedge \ y' = y$

Fig. 1. Before-after predicates

The initial Event-B model should satisfy the event feasibility and invariant preservation properties. For each event of the model, evt_i, its feasibility means that, whenever the event is enabled, its before-after predicate (BA) is well-defined, i.e., exists some reachable after-state:

$$A(d, c),\ I(d, c, v),\ g_i(d, c, v) \ \vdash \ \exists v' \cdot BA_i(d, c, v, v') \tag{FIS}$$

where A is model axioms, I is the model invariant, g_i is the event guard, d are model sets, c are model constants, and v, v' are the variable values before and after the event execution.

Each event evt_i of the initial Event-B model should also preserve the given model invariant:

$$A(d, c),\ I(d, c, v),\ g_i(d, c, v),\ BA_i(d, c, v, v') \ \vdash \ I(d, c, v') \tag{INV}$$

Since the initialisation event has no initial state and guard, its proof obligation is simpler:

$$A(d, c),\ BA_{Init}(d, c, v') \ \vdash \ I(d, c, v') \tag{INIT}$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. These new events correspond to stuttering steps that are not visible at the abstract level. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables.

To verify correctness of a refinement step, we need to prove a number of proof obligations for a refined model. The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the Rodin platform [15]. Proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of such complex distributed systems as multi-agent systems.

In the next section, we outline main principles of formal reasoning about MAS and their properties.

4 Specification of Fault Tolerant MAS in Event-B

In the Event-B specification of fault tolerant MAS, we are interested in verifying the properties related to convergence and correctness of fault tolerance:

- all pending relationships are eventually resolved;
- the given relationship preferences are enforced.

Properties of the first kind, i.e., eventuality properties, are especially important for multi-agent systems. Such properties are often of the form "The system state p always leads to the state q" or, using the temporal logic notation, "$\Box(p \rightsquigarrow q)$".

Often we are interested in formulating the properties similar to the examples below:

- $\Box$(new subordinate agent $\rightsquigarrow$ assigned supervisor agent);
- $\Box$(an agent leaves the system $\rightsquigarrow$ all its relationships are removed);
- $\Box$(a supervisor agent leaves the system $\rightsquigarrow$ all its subordinates are re-assigned);
- $\Box$(a supervisor agent fails $\rightsquigarrow$ all its subordinate agents are re-assigned).

The responsibility of the middleware is detect situations when some of the established/ or to be established relationships become pending due to failres and guarantee "fairness", i.e., no pending request for collaboration will be ignored forever.

Next we present modelling patterns that allow us to express properties described above in the Event-B framework. The abstract machine MAS (omitted for brevity) defines two general types of agents defined by sets $ATYPE1$ and $ATYPE2$, which are subsets of the generic type $AGENT$. The status of agents (i.e., whether they active or not, i.e., failed) is stored in a function variables $status1$ and $status2$, which for agents of different types returns a value of the enumerated set $STATUS = \{active, inactive\}$. We encapsulate the other variables of the machine by the abstract variable $state$. The machine models recovery of the agents in the location, i.e. the operating system, and non-deterministic changes of their statuses due to failure or recovery. In Fig. 2 we define a machine $MAS1$. Essentially, the specification $MAS1$ introduces a new event $CooperativeActivity$ in the machine MAS. The we can define the following proposition:

Proposition 1. The machine $MAS1$ refines MAS and preserves Property 1, where
$\mathcal{A}_{act} = \{a \mid a \in a\_t1 \wedge status1(a) = active) \vee (a \in a\_t2 \wedge status2(a) = active\}$
and $\mathcal{A}_{ina} = \{a \mid (a1 \in a\_t1 \wedge status1(a1) = inactive) \vee (a \in a\_t2 \wedge status2(a) = inactive)\}$ and $\mathcal{EAA} = \{CooperativeActivity\}$
and $\mathcal{EA}\mu = \{Status1, Status2\}$

Proof: The proof of the proposition follows from two facts:

1. The rules REF_INV, REF_GRD and REF_SIM defined in Sect. 2 are satisfied
2. The event *CooperativeActivity* is enabled only for active, i.e., healthy agents, i.e., the agents whose status evaluates to $TRUE$

In a MAS, the agents often fail only for a short period of time. After the recovery, the agent should be able to continue its operations. Therefore, after detecting an agent failure, the middleware should not immediately disengage the disconnected agent but rather set a deadline before which the agent should recover. If the failed agent recovers before the deadline then it can continue its normal activities. However, if the agent fails to do so, the location should permanently disengage the agent.

In the refined specification we define the variable $failed$ representing the subset of active agents that are detected as transiently failed $failed \subseteq coop\_agents$.

Moreover, to model a timeout mechanism, we define the variable *timer* of the enumerated type $\{inactive, active, timeout\}$. Initially, for every active agent, the *timer* value is set to *inactive*. As soon as active agent fails, its id is added to the set $failed$ and its timer value becomes *active*. This behaviour is specified in the new event *FailedAgent*.

An agent experiencing a transient failure can succeed or fail to recover, as modelled by the events *RecoverySuccessful* and *RecovertFailed* respectively. If the agent recovers before the value of timer becomes *timeout*, the timer value is changed to *inactive* and the agent continues its activities virtually uninterrupted. Otherwise, the agent is removed from the set of active agents. The following invariant ensures that any disconnected agent is considered to be inactive:

$$\forall a \cdot (a \in coop\_agents \wedge timer(a) \neq inactive \ \Leftrightarrow a \in disconnected)$$

The introduction of an agent failure allows us to make a distinction between two reasons behind leaving the system by a supervisory agent – because its duties are completed or due to the disconnection timeout. To model these two cases, we split the event *AgentLeaving* into two events *NormalAgentLeaving* and *DetectFailedFreeAgent* respectively.

While modelling failure of a supervisory agent, we should again have to deal with the property stating that if a supervisory agent fails then all its subordinate agents are re-assigned. In a similar way as above, the event *ReassignSupervisor* is decomposed into two events *NormalReassignSupervisor* and *DetectFailedAgent*. The second refinement step resulted in a specification ensuring that no subordinate agent is left unattached to the supervisor neither because of its normal termination or failure.

The second refinement step resulted in a specification ensuring that no subordinate agent is left unattached to the supervisor neither because of its normal termination or failure.

5 Related Work

Formal modelling of MAS has been undertaken by [6,8,10–12]. Our approach builds on these work. It is different from numerous process-algebraic approaches used for modelling MAS. Firstly, we have relied on proof-based verification that does not impose restrictions on the size of the model, number of agents etc. Secondly, we have adopted a system's approach, i.e., we modelled the entire system and extracted specifications of its individual components by decomposition. Such an approach allows us to express and formally verify correctness of the overall

Machine $MAS1$
Variables $a\_t1,\ a\_t2,\ status1,\ status2,\ state$
Invariants
$inv_1 : a\_t1 \subseteq ATYPE1$
$inv_2 : a\_t2 \subseteq ATYPE1$
$inv_3 : status1 \in a\_t1 \rightarrow STATUS$
$inv_4 : status2 \in a\_t2 \rightarrow STATUS$
$inv_5 : state : STATE$
Events
$Initialisation \mathrel{\widehat{=}}$
begin
$a\_t1 := \varnothing$
$a\_t2 := \varnothing$
$status1 := \varnothing$
$status2 := \varnothing$
$state :: STATE$
end

$Populate1 \mathrel{\widehat{=}}$
any $a1$
when
$a1 \in ATYPE1$
$a1 \notin a\_t1$
then
$a\_t1 := a\_t1 \cup \{a1\}$
$status1(a1) := active$
end

$Status1 \mathrel{\widehat{=}}$
any $a1$
when
$a1 \in a\_t1$
then
$status1(a1) :\in STATUS$
$state :\in STATE$
end

$CooperativeActivity \mathrel{\widehat{=}}$
any $a1, a2$
when
$a1 \in a\_t1$
$a2 \in a\_t2$
$status(a1) = active$
$status(a2) = active$
then
$state :\in STATE$
end
END

Fig. 2. Specification of a MAS with cooperative activity

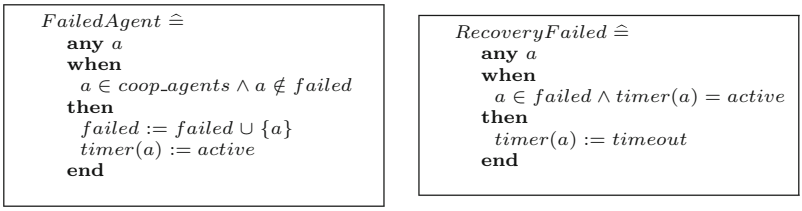

Fig. 3. Failed agent and recovery failed events

```
DetectFailedAgent ≙
  Refines ReassignSupervisor
  any a, a_new
  when
    a ∈ ran(assigned_supervisor) ∧ a ∉ last_cooperated[cooperated]∧
    a_new ∈ coop_agents ∧ a_new ≠ a∧
    a ∈ disconnected ∧ timer(a) = timeout∧
    a_new ∉ disconnected ∨ (a_new ∈ disconnected ∧ timer(a_new) = active)
  then
    coop_agents := coop_agents \ {a}
    assigned_supervisor := assigned_supervisor⩤
                (dom(assigned_supervisor ▷ {a}) × {a_new})
    disconnected := disconnected{a}
    timer := {a} ⩤ timer
  end
DetectFailedFreeAgent ≙
  Refines AgentLeaving
  any a
  when
    a ∈ coop_agents ∧ a ∉ ran(assigned_supervisor)∧
    a ∉ last_cooperated[cooperated] ∧ a ∈ disconnected ∧ timer(a) = timeout
  then
    coop_agents := coop_agents \ {a}
    disconnected := disconnected{a}
    timer := {a} ⩤ timer
  end
```

system, i.e., we indeed achieve verification of fault tolerance as a system level property. Finally, the adopted top-down development paradigm has allowed us to efficiently cope not only with complexity of requirements but also with complexity of verification. We have build a large formal model of a complex system by a number of rather small increments. As a result, verification efforts have been manageable because we merely needed to prove refinement between each two adjacent levels of abstraction.

6 Conclusion

In this paper, we have presented an approach to formal specification of fault tolerant MAS. We formalised the main properties of fault tolerant MAS that perform cooperative activities and supported by the middleware to achieve fault tolerance. We defined the specification and refinment patterns for the formal development of fault tolerant MAS in Event-N.

In our development we have explicitly modelled the fault tolerance mechanism that ensures correct system functioning in the presence of agent failures. We have verified by proofs the correctness and termination of error recovery. Formal verification process has not only allowed us to systematically capture the complex error detection and recovery but also facilitated derivation of the constraints that should be imposed on the behaviour of the agents of different types to guarantee a correct implementation of fault tolerant. As a future work, we are planning to apply the proposed approach to modelling interaction of autonomous agents that are subject of malicious rather than random faults.

References

1. Majd, A., Ashraf, A., Troubitsyna, E.: Online path generation and navigation for swarms of UAVs. In: Scientific Computing, pp. 1–12 (2020)
2. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Abrial, J.-R.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
4. Majd, A., Troubitsyna, E.: Data-driven approach to ensuring fault tolerance and efficiency of swarm systems. In: Proceedings of Big Data, vol. 2017, pp. 4792–4794 (2017)
5. Majd, A., Troubitsyna, E.: Towards a realtime, collision-free motion coordination and navigation system for a UAV fleet. In: Proceedings of ECBS, vol. 2017, pp. 111–119 (2017)
6. Troubitsyna, E., Pereverzeva, I., Laibinis, L.: Formal development of critical multi-agent systems: a refinement approach. In: Proceedings of European Dependable Computing Conference, pp. 156–161 (2015)
7. Vistbakka, I., Troubitsyna, E.: Modelling autonomous resilient multi-robotic systems. In: Calinescu, R., Di Giandomenico, F. (eds.) SERENE 2019. LNCS, vol. 11732, pp. 29–45. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30856-8_3
8. Vistbakka, I., Troubitsyna, E.: Modelling resilient collaborative multi-agent systems. J. Comput. **103**(4), 1–23 (2020)
9. Vistbakka, I., Troubitsyna, E.: Pattern-based goal-oriented development of fault-tolerant MAS in Event-B. In: Proceedings of International Conference on Practical Applications of Agents and Multi-Agent Systems, pp. 327–339 (2020)
10. Vistbakka, I., Majd, A., Troubitsyna, E.: Deriving mode logic for autonomous resilient systems. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 320–336. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02450-5_19
11. Majd, A., Vistbakka, I., Troubitsyna, E.: Formal reasoning about resilient goal-oriented multi-agent systems. Sci. Comput. Program. **148**, 66–87 (2019)
12. Majd, A., Vistbakka, I., Troubitsyna, E.: Multi-layered safety architecture of autonomous systems: formalising coordination perspective. In: Proceedings of 9th International Symposium on High Assurance Systems Engineering (HASE), pp. 58–65 (2019)
13. Laprie, J.C.: From dependability to resilience. In: 38th IEEE/IFIP International Conference on Dependable Systems and Networks, pp. G8–G9 (2008)
14. OMG Mobile Agents Facility (MASIF). www.omg.org
15. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project. http://rodin.cs.ncl.ac.uk/