Information and Computation 254 (2017) 167–194

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Combining fault injection and model checking to verify fault tolerance, recoverability, and diagnosability in multi-agent systems

J. Ezekiel, A. Lomuscio *

Department of Computing, Imperial College London, London, SW7 2AZ, UK

ARTICLE INFO

Article history:
Received 3 January 2015
Available online 5 November 2016

ABSTRACT

We present an automated technique that combines fault injection with model checking to verify fault tolerance, recoverability, and diagnosability in multi-agent systems. We define a general method for mutating a multi-agent systems model representing correct behaviour by injecting faults into it, and specification patterns based on temporal-epistemic formulas to reason about the correct and faulty behaviours of the mutated model. The technique is implemented in a toolkit that can be used for injecting automatically faults into a multi-agent systems program. The usefulness of the methodology is demonstrated by injecting a number of faults into a model of the IEEE 802.5 token ring LAN protocol and analysing the protocol's fault tolerance, by verifying a number of temporal-epistemic specifications.
© 2016 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

The multi-agent systems (MAS) paradigm [1] has been employed successfully in several disciplines studying systems whose core components, or *agents*, autonomously interact with one another, engaging in communication, negotiation, co-ordination, etc. One reason for the popularity of MAS formalisms in many scenarios is the availability of rich modal logics to analyse the behaviour of agents, including the ability to reason about various notions of *knowledge* of the agents in the system [2].

In recent years several studies have been conducted to ensure that the MAS paradigm can be adopted for situations in which the system must continue to operate correctly under degraded conditions, such as the event of a failure within one or more of the agents. As a result, several *fault tolerant* architectures for MAS have been proposed (see, e.g., [3–7]). The strategies utilised by these architectures include replicating agents [3] to facilitate overall system tolerance to agent failures, and allowing agents to *diagnose* faults so that they can communicate and co-ordinate to *recover* from them [5–7]. While these strategies have been proven useful in a range of scenarios, they fall short of providing practically usable tools and techniques for *certifying* the correct behaviour of fault tolerant MAS architectures.

Formal methods and, in particular, *model checking* [8] are becoming increasingly popular for verifying the correct behaviour of systems (see, e.g., [9,10]). Model checking has previously been used to verify fault tolerant properties of a system (see, e.g., [11]), including *recoverability* [12], i.e., recovery from faults, and *diagnosability* [13], i.e., establishing whether a fault can be correctly detected from the observable events of the system [14]. Recently, a number of tools have been developed by applying *fault injection* [15] to a correctly behaving system and using model checking to verify correct operation of

* Corresponding author.
E-mail address: a.lomuscio@imperial.ac.uk (A. Lomuscio).

http://dx.doi.org/10.1016/j.ic.2016.10.007
0890-5401/© 2016 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

that system under degraded conditions [16,9,11,17]. Techniques that allow automation when injecting faults into the system model are particularly attractive to non-experts in verification due to the high level of usability implied by the *automatic* nature of both the fault injection and the verification process [9]. Unfortunately, due to their modelling formalisms and sole support of *temporal logic* [18] as a specification language, these tools are not directly applicable for verifying MAS since their specifications involve rich, AI-based primitives, such as knowledge, beliefs, desires and intentions. AI-based specifications have been shown to be useful beyond

Contribution: In this article we present an automated technique for verifying fault tolerance properties of MAS. A key feature of the technique we put forward is that it supports the reasoning about epistemic properties of the system. We show that, in addition to its widely discussed suitability for agent-based system, this further provides a natural and expressive language for expressing various notions of diagnosability. We ground this work on the MCMAS model checker [19], a model checker tailored to MAS specifications.

More specifically, we combine automatic fault injection techniques with MCMAS to introduce a general method for *mutating* a MAS model representing its correct behaviour by injecting faults into it. A mutation is an update to a MAS model in which the behaviour of one or more agents is altered. The mutated model is verified against temporal-epistemic specifications to reason about the correct and faulty behaviours of the MAS, in order to assess properties of fault tolerance, recoverability, and diagnosability. A noteworthy feature of the technique is the expressiveness in the specifications supported and its high level of automation. To evaluate the methodology we introduce a toolkit for injecting automatically faults into a MAS program for analysis with MCMAS. We report the results obtained by evaluating the IEEE 802.5 token ring LAN protocol in the context of its fault tolerance mechanisms that employ distributed diagnosis of faults to facilitate recovery from them.

The rest of this article is structured as follows. In Section 2 we provide the necessary background on model checking, faults, diagnosability, interpreted systems, and MCMAS. In Section 3 we introduce a method of automatic fault injection including failure modes and MAS model mutation via fault injection. In Section 4 we introduce a library of temporal-epistemic specification patterns to reason about fault tolerance, recoverability, and diagnosability. In Section 5 we introduce a toolkit for automatic fault injection. In Section 6 we describe the IEEE 802.5 token ring LAN protocol, and present results relating to its temporal-epistemic properties of fault tolerance, recoverability, and diagnosability, as obtained in the methodology presented. In Section 7 we discuss related work and in Section 8 we conclude and put forward future work.

2. Background

Model checking [8] is a widely adopted technique for systems verification. In model checking the system S considered for verification is represented by a logical model M_S encoding the behaviour of the system as computational traces. In this approach a specification of a property P is expressed by means of a logical formula φ_P. The model checker establishes whether or not M_S satisfies φ_P (formally, $M_S \models \varphi_P$). The satisfaction relation is implemented as a decision procedure, the *automatic* nature of which makes model checking attractive for the purpose of verification.

Model checking tools used for reactive systems such as SPIN [20], SMV [21], and NuSMV [22] express φ_P as a *temporal logic* formula [18]. Model checking tools for multi-agent systems such as MCMAS [19], Verics [23] and MCK [24] support richer specifications including epistemic logics [2]. Epistemic logic has long been advocated as an expressive and natural language to capture properties of MAS.

2.1. Model checking, faults, and diagnosability

Traditionally, model checking has been applied to provide assurances about the *correct* behaviour of the system. However, the analysis of safety-critical systems involves reasoning about the *consequences of faults*. This has been achieved by comparing correct system runs and runs in which *faulty* behaviour is injected by means of model checking tools [16,9,11,17].

Reasoning about faulty behaviour is also a recent topic in MAS. Faulty behaviour in MAS has been modelled and reasoned about for systems such as transmission protocols [25] and web services [26]. However, an automatic method for injecting faulty behaviour into MAS has not yet been developed; therefore, faulty behaviour is normally introduced manually when modelling a system.

The general problem of fault diagnosis has received considerable attention since the late 80s (see, e.g., [27]). The property of diagnosability can be analysed by establishing whether a fault can be correctly detected from the observable events of the system, and is typically defined by saying that a fault is diagnosable if there are a finite number of observations after the occurrence of the fault that correctly identify it [14]. For systems in which accurate fault diagnosis is critical, model checking has been used to verify this property [13]. However, in this approach *distributed* diagnosability is not considered, and the faulty behaviour of the system is modelled by hand, thereby hampering applications.

2.2. Interpreted systems and MCMAS

Interpreted systems [2] are a popular semantics for temporal-epistemic logic. Below we briefly summarise the framework of interpreted systems popularised in [2] to model MAS. We follow standard naming conventions and characterise each agent

$i \in \{1, \ldots, n\}$ in the system by a finite set of local states L_i and by a finite set of local actions Act_i. Actions are performed in compliance with a local protocol $P_i : L_i \to 2^{Act_i}$ specifying which actions may be performed in a given state. In this formalism the environment in which agents live is modelled by means of a special agent E. Associated with E are a set of local states L_E, a set of local actions Act_E, and a local protocol P_E. A tuple $g = (l_1, \ldots, l_n, l_E) \in L_1 \times \ldots \times L_n \times L_E$, where $l_i \in L_i$ for each i and each $l_E \in L_E$, is a *global state* describing the system at a particular instant of time.

The evolution of the agents' local states is described by a transition function $t_i : L_i \times Act_1 \times \ldots \times Act_n \times Act_E \to L_i$ which returns the next local state for agent i given the current local state of the agent, the current action by the environment as well as all the agents' actions. Similarly, the evolution of the environment's local states is described by a function $t_E : L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_E$, returning the next environment state given the current round of actions. It is assumed that in every state agents evolve simultaneously. The evolution of the global states of the whole system is described by a function $t : G \times Act \to G$, where $G \subseteq L_1 \times \ldots \times L_n \times L_E$ is the set of global states for the system reachable from a set of initial global states $I \subseteq G$, and $Act \subseteq Act_1 \times \ldots \times Act_n \times Act_E$ is the set of enabled joint actions. The function t is defined as $t(g, a) = g'$ if and only if for all $i, t_i(l_i(g), a) = l_i(g')$ and $t_E(l_E(g), a) = l_E(g')$, where $l_i(g)$ denotes the i-th component of global state g (corresponding to the local state of agent i). Finally, an interpreted system includes a set of atomic propositions AP together with a valuation function $V \subseteq AP \times G$. Formally, we define an *interpreted system* as the tuple

$$IS = \langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (L_E, Act_E, P_E, t_E), I, V \rangle$$

Interpreted systems can be used to interpret specifications in the temporal-epistemic logic CTLK, whose grammar we introduce below:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid AG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi \mid E_\Gamma\varphi \mid C_\Gamma\varphi \mid D_\Gamma\varphi$$

In the grammar above $p \in AP$ is an atomic proposition; $EX\varphi$ represents that there is a global next state of computation in which φ holds; $AG\varphi$ means in all sequences of global states (or, *runs*, see below) φ holds; $E(\varphi U \psi)$ expresses that there exists a run in which φ holds until ψ holds; $K_i\varphi$ expresses that *agent i knows* φ, $E_\Gamma\varphi$ represents that *everybody in group* Γ *knows* φ, $C_\Gamma\varphi$ means that *it is common knowledge in group* Γ *that* φ, and $D_\Gamma\varphi$ expresses that *it is distributed knowledge in group* Γ *that* φ. Distributed knowledge represents the epistemic state the group would be in if the agents in the group were able to combine their individual epistemic states. We refer the reader to [2] for details on this and related epistemic concepts widely discussed in the epistemic logic literature. We also assume the standard rewriting for other CTL operators such as AX, EF, AF, EG in terms of the triple here introduced.

Any interpreted system is associated with a model $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible states W is the set G of global states reachable from I through the temporal relation R_t. The temporal relation $R_t \subseteq W \times W$ is defined by considering the transition function t of the corresponding IS: two worlds w and w' are such that $R_t(w, w')$ if and only if there exists a joint action $a \in Act$ such that $t(w, a) = w'$. The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by considering the equality of the local components of the global states. Two worlds $w, w' \in W$ are such that $w \sim_i w'$ if and only if $l_i(w) = l_i(w')$ (i.e., two worlds w and w' are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same [2]). The labelling relation $L \subseteq AP \times W$ is defined in terms of the valuation relation V.

Formulae are interpreted in M_{IS} as standard [8,2]. Let $\pi = (w_0, w_1, \ldots)$ be a *run*, i.e., an infinite sequence of global states such that for all i, $R_t(w_i, w_{i+1})$, and let $\pi(i)$ denote the i-th world of the sequence (notice that, following our assumptions above the temporal relation is serial and thus all computation paths are infinite). We write $(M, w) \models \varphi$ to represent that a formula φ is true at a world w in a Kripke model M, associated with an interpreted system IS. Satisfaction is defined as follows.

$(M, w) \models p$ iff $(p, w) \in L$,
$(M, w) \models \neg\varphi$ iff it is not the case that $(M, w) \models \varphi$,
$(M, w) \models \varphi_1 \vee \varphi_2$ iff $(M, w) \models \varphi_1$ or $(M, w) \models \varphi_2$,
$(M, w) \models EX\varphi$ iff there exists a path π such that $\pi(0) = w$, and $(M, \pi(1)) \models \varphi$,
$(M, w) \models AG\varphi$ iff for all paths such that $\pi(0) = w$, we have $(M, \pi(i)) \models \varphi$, for all $i \geq 0$,
$(M, w) \models E(\varphi U \psi)$ iff there exists a path π such that $\pi(0) = w$, and there exists $k \geq 0$ such that $(M, \pi(k)) \models \psi$ and $(M, \pi(j)) \models \varphi$ for all $0 \leq j < k$,
$(M, w) \models K_i\varphi$ iff for all $w' \in W w \sim_i w'$ implies $(M, w') \models \varphi$,
$(M, w) \models E_\Gamma\varphi$ iff for all $w' \in W R_\Gamma^E(w, w')$ implies $(M, w') \models \varphi$,
$(M, w) \models C_\Gamma\varphi$ iff for all $w' \in W R_\Gamma^C(w, w')$ implies $(M, w') \models \varphi$,
$(M, w) \models D_\Gamma\varphi$ iff for all $w' \in W R_\Gamma^D(w, w')$ implies $(M, w') \models \varphi$.

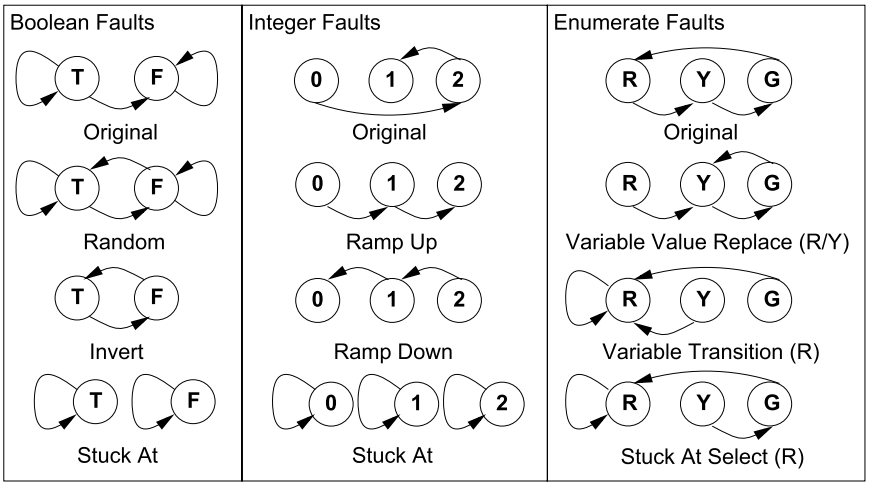


Fig. 1. Types of faults used to define failure modes.

In the definition above the relation R^E_Γ is defined as the union of the epistemic relations for the agents in $\Gamma : R^E_\Gamma = \bigcup_{i\in\Gamma} \sim_i$; the relation R^D_Γ is defined as the intersection of the epistemic relations for the agents in $\Gamma : R^D_\Gamma = \bigcap_{i\in\Gamma} \sim_i$; the relation R^C_Γ is the transitive closure of R^E_Γ.

We say that formula φ is true in an interpreted system *IS*, and write $IS \vDash \varphi$ if for all initial states $k \in I$, we have that $(M_{IS}, k) \vDash \varphi$.

MCMAS [19] is a BDD-based model checker for the automatic verification of multi-agent systems. It provides ISPL (Interpreted Systems Programming Language) as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulas as specifications of the system. The structure of an ISPL program allows the local states to be defined using *boolean*, *bounded integer*, and *enumeration* variables. ISPL programs are closely related to interpreted systems; specifically, each ISPL program describes a unique interpreted system. MCMAS supports the verification for all formulas in the language above as well as others.

3. Automatic fault injection

Automatic fault injection is the process of mechanically mutating a correct system model into one that displays a particular faulty behaviour [16,9,11,17]. In a MAS oriented context, we assume a MAS to be modelled by an interpreted system *IS*; from this we wish to derive an extended faulty system IS^{F*}, where $F*$ indicates the presence of the mutated faulty behaviour in the system. The extended faulty system contains the original behaviours of *IS* as well as some behaviours resulting from the faults. We begin by defining this extension by looking at the types of common faults that occur in systems. These have been previously defined as failure modes in [9,17]; they are here instantiated in the interpreted system formalism and extended by introducing complex timings to co-ordinate the fault with the state and actions of other agents in the system.

3.1. Failure modes

Failure modes describe behaviour relating to component failures. Common types of failures in components are those such as *random*, *stuck at* or *inverted* faults [9,17], which model faults such as a valve becoming stuck open, or accidentally closed. Failure modes can also be used to capture the persistence of faults, such as occurring in every step of the evolution of the system, a fixed number of steps, or intermittently [17].

We consider these common types of component failures as the starting point for defining failure modes. We extend them to allow for more sophisticated agent based failures by defining faults which *replace* and *skip* non-faulty agent behaviours, as well as rephrase these in the context of interpreted systems semantics and MAS. Fig. 1 illustrates the types of faults that we consider to be injectable into an agent, showing an example of an original transitions between states followed by the mutated transitions between states according to the type of fault. Different types of faults are defined for *boolean*, *integer*, and *enumerate* variables describing a component of the system.

3.1.1. Boolean faults

Boolean faults represent incorrect evolutions of boolean variables. For example, consider the case where an agent A has a boolean variable Var_b representing two possible local states F and T, i.e., $L_A = \{F, T\}$. We define three faults that can be injected into the agent on this variable.

- *Inverting* the value of Var_b (so that state F becomes state T and vice versa).
- *Sticking Var* to its current value (so that the state remains constant at its current value).
- *Randomly* setting the value of Var_b (arbitrarily choosing one of the states at every tick of the clock).

3.1.2. Integer faults

Integer faults correspond to erroneous evolutions of integer variables. For example, consider the case where an agent A has an integer variable Var_i representing three possible local states; *0*, *1*, and *2*, i.e., $L_A = \{0, 1, 2\}$. We define three faults that can be injected into the agent on this variable.

- *Ramping down* the value of Var_i (so that state *2* becomes state *1*, state *1* becomes state *0*, and state *0* remains at its current value).
- *Ramping up* the value of Var_i (so that state *0* becomes state *1*, state *1* becomes state *2*, and state *2* remains at its current value).
- *Sticking* Var_i to its current value (so that the state remains constant at its current value).

3.1.3. Enumerate faults

Enumerate faults express faulty evolutions of enumerate variables. For example, consider the case where an agent A has an enumerate variable Var_e representing three possible local states; R, Y and G, i.e., $L_A = \{R, Y, G\}$. In Fig. 1 we illustrate the original transitions between these states as those of a traffic light evolving from red to yellow to green and back to red, where R is the red state, Y is the yellow state and G is the green state. We define three faults that can be injected into the agent on this variable.

- A *variable value replace* fault denotes a situation where a value v_1 of type Var_e is updated with a value v_2 of type Var_e (in this example v_1 and v_2 can be R, Y, or G). This fault occurs when some of the correct agent behaviour is skipped. Fig. 1 illustrates a traffic light that evolves from the green state to the yellow state instead of evolving from the green state to the red state by replacing red with yellow (i.e., v_1 is R and v_2 is Y).
- A *variable transition* fault denotes a situation where Var_e is set to a value v_1 of type Var_e regardless of the state the agent is in. This fault occurs when an agent immediately evolves to a state as a result of a fault. Fig. 1 illustrates the faulty behaviour where a traffic light evolves to the red state from every state (i.e., v_1 is R).
- A *variable stuck at select* fault occurs when a value v_1 of Var_e persists whenever the current value of Var_e is v_1 of type Var_e. Other values of the variable are allowed to change when the agent evolves. Fig. 1 illustrates the faulty behaviour where a traffic light is stuck in the red state (i.e., v_1 is R). If the traffic light is not in the red state, it behaves in a non-faulty manner.

3.1.4. Fault persistence

To enhance the expressiveness of the failure types previously described, we allow for the persistence of the faults to be varied, such as occurring *constantly*, *randomly*, and at *a specific point* during a system run. This allows realistic system faults to be defined since faulty behaviour in a system may only be intermittent and happen to occur during, before, or after a specific point in time.

In the following we use these failure modes to define a notion of update from models representing correct behaviour into ones encoding faults as well.

3.2. MAS model updates via fault injection

To make our fault injection method automatic, we define a general way to extend any agent of the system A into a *faulty agent* A^{F*}. To do this we introduce a *fault injection agent* (*FI*) determining the conditions under which faulty behaviour occurs in the faulty agent. For each fault $j \in \{1, \ldots, m\}$ introduced into the mutated interpreted system, a corresponding fault injection agent FI_j is defined. By separating the fault injector from the resulting mutated agents we obtain clearer and more uniform models. As we will see later, a user will be able to select various options pertaining to the occurrence of a fault during a system run by manipulating *FI*.

We stipulate that the faulty behaviour is triggered in the faulty agent whenever the *inject* action is performed by *FI*. Conversely, the original behaviour is preserved in the faulty agent whenever any action other than *inject* is performed by *FI*. The *inject* action is performed according to the local state of the fault injection agent. This is highlighted in Fig. 2 which shows the mutation of an agent A containing correct behaviour into a faulty agent A^{F*} which contains correct and faulty behaviour.

3.2.1. The fault injection agent

The fault injection agent determines the time at which a fault is injected during a system run. To allow for varying fault persistence, we define a fault injection agent that determines that a fault is injected *constantly* as default with options for *random* (*rnd*) fault injection, fault injection after and before a *random start point* (*rstt*) and a *random stop point* (*rsto*), and fault

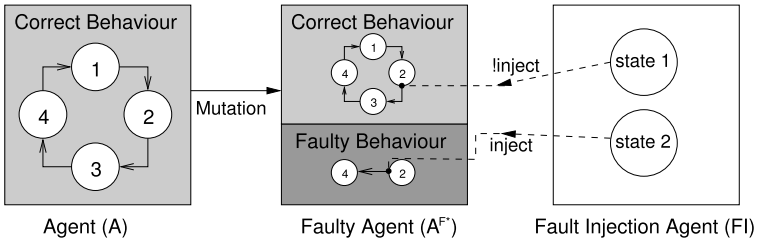


Fig. 2. Mutating a correctly behaving agent into one that may display faulty behaviour.

Table 1
Definition of local states and actions of *FI*.

Options	L_{FI}	Act_{FI}
default	{*nofault*, *fault_i*, *fault_ni*}	{*notinject*, *inject*}
rstt	∪ {*w_rstt*}	∪ {*start*}
astt	∪ {*w_astt*}	
asto	∪ {*stop_ni*}	
asto, !*rsto*	∪ {*stop_i*}	
rsto	∪ {*stop_ni*}	∪ {*stop*}
rsto, *asto*	∪ {*asto_i*}	
rsto, *asto*, *rnd*	∪ {*asto_ni*}	

Table 2
Definition of protocol of *FI*.

Options	Act_P	P_{FI}
default	{*inject*}	{(*fault_i*, Act_P), (*nofault*, {*notinject*})}
!*rnd*		∪ {(*fault_ni*, {*inject*})}
rnd	∪ {*notinject*}	∪ {(*fault_ni*, Act_P)}
rstt		∪ {(*w_rstt*, {*notinject*, *start*})}
astt		∪ {(*w_astt*, {*notinject*})}
asto		∪ {(*stop_ni*, {*notinject*})}
asto, !*rsto*		∪ {(*stop_i*, {*notinject*})}
rsto		∪ {(*stop_ni*, {*notinject*})}
rsto, !*asto*	∪ {*stop*}	
rsto, *asto*		∪ {(*asto_i*, Act_P ∪ {*stop*})}
rsto, *asto*, *rnd*		∪ {(*asto_ni*, Act_P ∪ {*stop*})}

injection after and before a *start action occurs*[1] (*astt*) and a *stop action occurs* (*asto*). Any of these options can be combined thereby defining more sophisticated faults. Given any of these faults, *FI* can be formalised by giving the corresponding local states, actions, protocols, and transitions.

We now define the fault injection agent according to these options. The **options** column in Tables 1–5 represents that all options presented in the column must be selected for the definition in the table to apply. If an option is preceded by !, the option must not be selected for the definition in the table to apply.

Table 1 defines the local states and actions of the fault injection agent according to these options. For the default fault injection agent we have $ACT_{FI} = \{notinject, inject\}$. The actions *start* and *stop* are added if random start and stop options are set.

The fault injection agent can either be in a state where it is never injecting faults (*nofault*), has injected or not injected a fault at the current tick of the clock (*fault_*[*i*, *ni*]), is waiting for a start condition (*w_*[*astt*, *rstt*]), has injected or not injected a fault at the current tick of the clock after a stop action (*asto_*[*i*, *ni*]), or has stopped injecting faults (*stop_*[*i*, *ni*]). For example, the local states of a random fault injection agent with a random start are defined as $L_{FI} = \{nofault, fault_i, fault_ni, w_rstt\}$.

In the above definitions *_i* indicates a state of the agent in which the *inject* action was performed at the current tick of the clock; *_ni* indicates a state of the agent in which the *inject* action was not performed at the current tick of the clock. Thus, a *stop_i* state can be reached when a point of a system run is reached in which the agent is no longer injecting faults, but a fault has been injected at the current tick of the clock. It follows that the fault injection agent will then evolve to a *stop_ni* state.

Table 2 defines the fault injection agent protocol P_{FI} according to the options, where the protocol function is represented for convenience as a relation, i.e., we write $P_{FI}(nofault) = \{notinject\}$ as $(nofault, \{notinject\})$, and use a variable Act_P for brevity. For the default fault injection agent we have that $P_{FI}(nofault) = \{notinject\}$, $P_{FI}(fault_i) = \{inject\}$,

[1] Start and stop actions are actions executed by any agent.

Table 3
Definition of transition relation of *FI*.

Options	Target state	Transition condition
astt	*fault_ni*	*w_astt* **and** *STA*
astt, *rstt*	*w_astt*	*w_rstt* **and** $Act_{FI} = start$
!*astt*, *rstt*	*fault_ni*	*w_rstt* **and** $Act_{FI} = start$
!*asto*	*fault_i*	(*fault_i* **or** *fault_ni*) **and** $Act_{FI} = inject$
!*asto*, *rnd*	*fault_ni*	(*fault_i* **or** *fault_ni*) **and** $Act_{FI} = notinject$
asto	*fault_i*	(*fault_i* **or** *fault_ni*) **and** $ACT_{FI} = inject$ **and** !*SPA*
asto, *rnd*	*fault_ni*	(*fault_i* **or** *fault_ni*) **and** $ACT_{FI} = notinject$ **and** !*SPA*
asto, !*rsto*	*stop_i*	((*fault_i* **or** *fault_ni*) **and** *SPA*) **and** $Act_{FI} = inject$
	stop_ni	*stop_i*
asto, !*rsto*, *rnd*	*stop_ni*	((*fault_i* **or** *fault_ni*) **and** *SPA*) **and** $Act_{FI} = notinject$
asto, *rsto*, *rnd*	*asto_i*	(((*fault_i* **or** *fault_ni*) **and** *SPA*) **or** (*asto_i* **or** *asto_ni*)) **and** $Act_{FI} = inject$
	asto_ni	(((*fault_i* **or** *fault_ni*) **and** *SPA*) **or** (*asto_i* **or** *asto_ni*)) **and** $Act_{FI} = notinject$
asto, *rsto*, !*rnd*	*asto_i*	(*fault_i* **or** *fault_ni*) **and** *SPA*
rsto	*stop_ni*	$Act_{FI} = stop$

Table 4
Definition of initial states of *FI*.

Options	I^{FI}
default	{*nofault*}
rstt	∪ {*w_rstt*}
astt, !*rstt*	∪ {*w_astt*}
!*rstt*, !*astt*	∪ {*fault_ni*}

Table 5
Definitions of states for updating the evaluation function.

Options	*IND*	Options	*NIJ*
default	{*fault_i*}	default	{*nofault*, *fault_ni*}
asto, !*rsto*	∪ {*stop_i*}	*astt*	∪ {*w_astt*}
asto, *rsto*	∪ {*asto_i*}	*rstt*	∪ {*w_rstt*}
		asto	∪ {*stop_ni*}
		rsto	∪ {*stop_ni*}

$P_{FI}(fault_ni) = \{inject\}$. As another example, for a fault injection agent with a random start and random stop we have that $Act_P = \{inject, stop\}$ therefore we have that $P_{FI}(fault_i) = \{inject, stop\}$, $P_{FI}(fault_ni) = \{notinject\}$, and $P_{FI}(w_rstt) = \{notinject, start\}$.

The transition relation for the fault injection agent is defined in Table 3. In the table $STA \subseteq Act_1 \times \ldots \times Act_n \times Act_E$ is a set of start actions which stipulate when the fault can begin occurring $SPA \subseteq Act_1 \times \ldots \times Act_n \times Act_E$ is a set of stop actions which stipulate when the fault can stop occurring. Similar to the options of the fault injection agents these actions are assigned by the user. Note that the fault injection agent remains in the same state if none of the transition conditions are met.

The initial states of the fault injection agent are defined in Table 4. The initial state is either *nofault*, which persists indefinitely, or another state determined by the options. For example, if *rstt* is set, the initial state can be either *nofault* or *w_rstt*, i.e., no faults are ever injected by the fault injection agent or a random start must occur before faults are injected.

An example of a random fault injection agent with a random start and action stop is illustrated in Fig. 3. The states of the agent, initial states, transition between states, protocol, and actions are shown to clarify how the selected options can be used to stipulate the occurrence of a fault.

3.2.2. Transitions for the faulty agent

The mutation rules for the faulty agent that denote the mutated transition relation are shown in Table 6. In the table *Trans* indicates the evolution function, *Target State* indicates the target state of the chosen variable for the injected fault, and *Transition Condition* shows the transition condition of the evolution function under original and mutated conditions. In the target state column $*_{ts}$ indicates the original target state; similarly, in the transition condition column $*_{tc}$ indicates the original transition condition. When individual variable components of the target state and transition condition are distinguished, $*_{ts}$ and $*_{tc}$ indicate the remaining component of the target state and transition condition respectively. We write $!(ACT_{FI} = inject)$ to express that the fault is not injected at the current tick of the clock. We use v_x to denote any value of a variable.

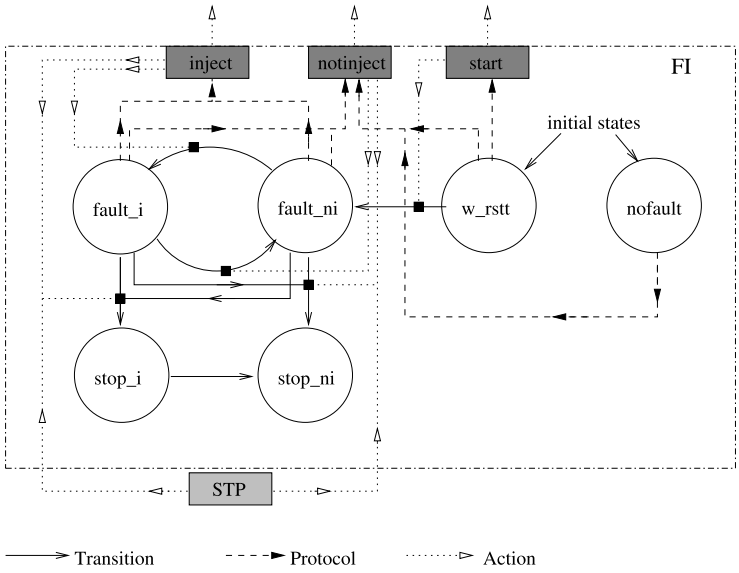


Fig. 3. Example of a random fault injection agent with a random start and action stop.

- *Boolean: Random* fault indicates that a boolean variable *var* is updated with a value either T or F in $t_{A^{F*}}$ for any value of *var* in t_A.
- *Boolean: Invert* fault denotes that a boolean variable *var* is updated with the inverse of its current value v_{var} in $t_{A^{F*}}$ for any value of *var* in t_A.
- *Boolean: Stuck At* fault represents that a boolean variable *var* remains at its current value v_{var} in $t_{A^{F*}}$ for any value of *var* in t_A.
- *Integer: Ramp Up* fault encodes that the current value v_{var} of an integer variable *var* is incremented by a value n for any value of *var* in t_A when $v_{var} + n$ is less than or equal to the maximum possible value of *var*.
- *Integer: Ramp Down* fault signifies that the current value v_{var} of an integer variable *var* is decremented by a value n for any value of *var* in t_A when $v_{var} - n$ is greater than zero.
- *Integer: Stuck At* fault expresses that an integer variable *var* remains at its current value v_{var} in $t_{A^{F*}}$ for any value of *var* in t_A.
- *Enumerate: Variable Value Replace* fault denotes an enumerate variable *var* is updated with a value v_2 in $t_{A^{F*}}$ whenever the value of *var* is updated to v_1 in t_A.
- *Enumerate: Variable Stuck At Select* fault encodes that the value v_1 of an enumerate variable *var* persists if the current value of *var* is v_1. If in t_A the variable *var* is updated to a value $v_x \neq v_1$ when $var = v_1$, the faulty behaviour in $t_{A^{F*}}$ preserves $var = v_1$.
- *Enumerate, Boolean: Variable Transition* fault represents that a variable *var* is set to the value v_1 (v_1 is T or F for a Boolean variable) in $t_{A^{F*}}$ whenever the fault is injected regardless of the state the agent is in. Thus, the original behaviour is preserved for all transitions whenever $!(ACT_{FI} = inject)$, and only *var* is updated to v_1 whenever $ACT_{FI} = inject$.

3.2.3. The mutated model IS^{F*}

We can complete the description of the mutated model that has been defined so far by introducing an appropriate set of atomic propositions to reason about faults. For each fault $j \in \{1, \ldots, m\}$ the mutated set of atomic propositions AP^{F*} is extended by using the propositions $faulty_j, injected_j, injecting_j, stopped_j$. Initially if $AP^{F*} = AP$ then in the default case we take $AP^{F*} = AP^{F*} \cup \{faulty_j\}$ where $faulty_j$ is defined to reason about whether faults are ever injected in a given run. The corresponding evaluation function V is updated so that $V^{F*}(faulty_j) = \{g \in G \mid l_{A^{F*}}(g) \neq nofault_j\}$. Similarly we have $AP^{F*} = AP^{F*} \cup \{injected_j\}$. The corresponding evaluation function is updated according to $IND_j \subset L_{FI}$, a set of local states of FI_j in which faults are injected at the current tick of the clock as defined in Table 5, so that $V^{F*}(injected_j) = \{g \in G \mid l_{A^{F*}}(g) \in IND_j\}$. For random faults if any start or stop option is selected then $AP^{F*} = AP^{F*} \cup \{injecting_j\}$ and the corresponding evaluation function is updated according to $NIJ_j \subset L_{FI_j}$ a set of local states of FI_j in which faults cannot be injected at the current tick of the clock as defined in Table 5, so that $V^{F*}(injecting_j) = \{g \in G \mid l_{A^{F*}}(g) \notin NIJ_j\}$. If any stop option is selected we have that $AP^{F*} = AP^{F*} \cup \{stopped_j\}$ and the corresponding evaluation function is updated so that $V^{F*}(stopped_j) = \{g \in G \mid l_{A^{F*}}(g) = stop_ni_j\}$. The extended faulty system is thus formally defined as follows:

$$IS^{F*} = \langle (L^{F*}{}_i, Act^{F*}{}_i, P^{F*}{}_i, t^{F*}{}_i)_{i \in \{1,\ldots,n+m\}}, L_E, Act_E, P_E, t_E), I^{F*}, V^{F*} \rangle.$$

Table 6
Mutation rules for A^{F*}.

Boolean: Random, *var* is set randomly to F or T		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = F$ **or** T **and** $*_{ts}$	$*_{tc}$ **and** $ACT_{FI} = inject$
Boolean: Invert, *var* is inverted		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = !v_{var}$ **and** $*_{ts}$	$*_{tc}$ **and** $ACT_{FI} = inject$
Boolean: Stuck At, *var* remains at its current value		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_{var}$ **and** $*_{ts}$	$*_{tc}$ **and** $ACT_{FI} = inject$
Integer: Ramp Up, the value of *var* is incremented by n		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_{var} + n$ **and** $*_{ts}$	$*_{tc}$ **and** $v_{var} + n <= max(var)$ **and** $ACT_{FI} = inject$
Integer: Ramp Down, the value of *var* is decremented by n		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_{var} - n$ **and** $*_{ts}$	$*_{tc}$ **and** $v_{var} - n > 0$ **and** $ACT_{FI} = inject$
Integer: Stuck At, *var* remains at its current value		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_{var}$ **and** $*_{ts}$	$*_{tc}$ **and** $ACT_{FI} = inject$
Enumerate: Variable Value Replace, when *var* is v_1 it is set to v_2		
Trans	Target state	Transition condition
t_A	$var = v_1$ **and** $*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$var = v_1$ **and** $*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_2$ **and** $*_{ts}$	$*_{tc}$ **and** $ACT_{FI} = inject$
Enumerate: Stuck At Select, when *var* is v_1 it remains at v_1		
Trans	Target state	Transition condition
t_A	$var = v_x$ **and** $*_{ts}$	$var = v_1$ **and** $*_{tc}$
$t_{A^{F*}}$	$var = v_x$ **and** $*_{ts}$	$var = v_1$ **and** $*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_1$ **and** $*_{ts}$	$var = v_1$ **and** $*_{tc}$ **and** $ACT_{FI} = inject$
Enumerate, Boolean: Variable Transition, *var* is set to v_1 (v_1 is either F or T for Boolean)		
Trans	Target state	Transition condition
t_A	$*_{ts}$	$*_{tc}$
$t_{A^{F*}}$	$*_{ts}$	$*_{tc}$ **and** $!(ACT_{FI} = inject)$
$t_{A^{F*}}$	$var = v_1$	$ACT_{FI} = inject$

4. Reasoning about correct and faulty behaviour

Our starting point is some of the existing approaches for the verification of MAS against temporal-epistemic specifications [19]. This analysis normally assumes the model represents all the (correct) executions of the system. We can now use the mutated model IS^{F*}, generated as described in the previous section, to reason about both the *correct* and *faulty* behaviours of the system. This allows us to examine, for instance, whether some temporal-epistemic specifications that were valid on the original model remain true should a fault occur in the system. We go a step further and establish *specification patterns* enabling us to identify classes of behaviour of interest. We are particularly concerned with fault tolerance, recoverability, and diagnosability. We analyse these below.

Given a set of m faults introduced into the mutated model and any fault $j \in \{1 \dots m\}$ in this set, we define a class of formulas Θ_j to reason about the *intermittent persistence* of a single fault j as follows:

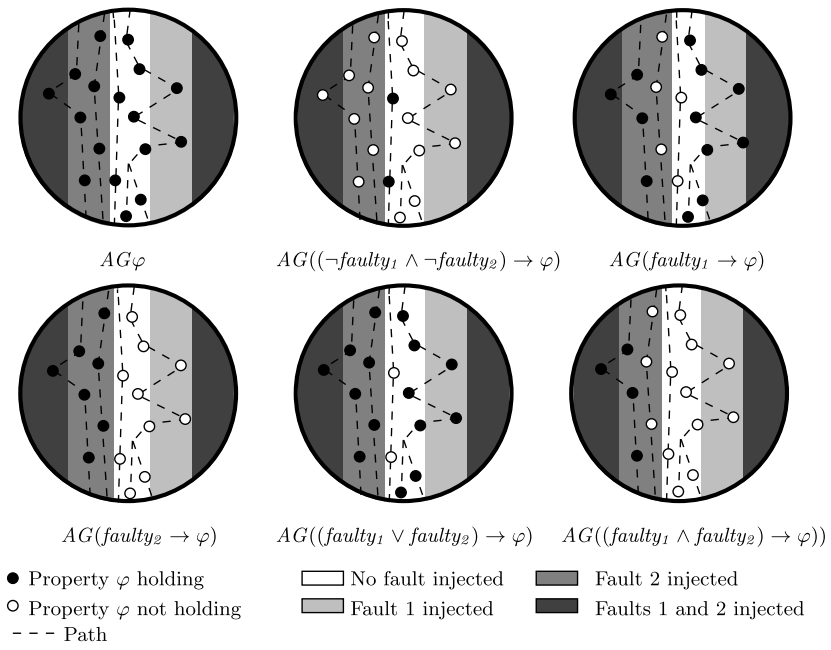


Fig. 4. Example models highlighting specifications of total tolerance.

$$\Theta_j ::= injecting_j | stopped_j | injected_j | \neg\Theta_j | \Theta_j \vee \Theta_j | \Theta_j \wedge \Theta_j$$

To reason about multiple faults we define classes of formulas for the *intermittent persistence of multiple faults* Ψ and the *overall persistence of multiple faults* Φ by means of the following syntax:

$$\Psi ::= \Theta_{j\in\{1,\ldots,m\}} | \neg\Psi | \Psi \vee \Psi | \Psi \wedge \Psi$$

$$\Phi ::= faulty_{j\in\{1,\ldots,m\}} | \neg\Phi | \Phi \vee \Phi | \Phi \wedge \Phi$$

The BNF grammars above define Boolean formulas representing properties of interest in the respective class. Observe that properties concerning multiple faults are expressed as Boolean combinations of formulas encoding single faults.

4.1. Reasoning about total tolerance to injected faults

We wish to define specification patterns that enable us to reason about the fault tolerance of a system *with respect to any specification* encoded as a logical formula φ in CTLK. We begin by considering the simple formula:

$$AG\varphi \tag{4.1}$$

In IS^{F*} this formula holds if φ is unaffected by the faulty behaviour. We can say that if the formula holds, then the system demonstrates *total* tolerance to the faults.

In general a system is likely to tolerate some faults but not others; to isolate the consequences of specific faults we therefore need to restrict our analysis. The following specification can be used to verify whether φ always holds even in the presence of specific faults:

$$AG(\Phi \rightarrow \varphi) \tag{4.2}$$

Given that formulas in Φ are path-invariant (see the definition of *faulty* in Section 3.2.1), this formula states that φ holds along a set of computational paths denoted by the overall persistence of multiple faults Φ. Effectively, the role of Φ is to restrict our analysis to the computational paths of interest, e.g., those that exhibit the *absence* of a fault or the presence *any* faulty behaviour introduced by the faults in question. This allows us to reason about the total tolerance of φ with respect to a specific fault or any combinations of faults, i.e., the fact that the specification φ holds in the system even when the fault in question is present.

4.1.1. Example specifications of total tolerance

We now introduce some models in Fig. 4 to illustrate how Formulas (4.1) and (4.2) can be used to define specifications of total tolerance. The example highlights *some* of the computational paths from the top of the circle to the bottom of the circle that we may wish to analyse in the entire reachable state space of the system (contained within the black circle), when two

types of faults can be injected. Observe that, as explained previously, the path-invariant propositions $faulty_1, faulty_2$ are true in paths where their respective fault is enabled and false otherwise. The coloured areas represent portions of the state-space where not only faults are enabled (*faulty*), but they also injected at the current tick (*injected*).

The variations include paths along which no faults are injected, paths along which faults are injected and combinations of both. Under each model we report instances of Formula (4.1) or (4.2) that hold on the particular model.

We can use Formula (4.2) to reason about only the *correct* behaviour of the system, specific faults, combinations of faults as follows:

$$AG((\neg faulty_1 \wedge \neg faulty_2) \rightarrow \varphi)$$

$$AG(faulty_1 \rightarrow \varphi)$$

$$AG((faulty_1 \vee faulty_2) \rightarrow \varphi)$$

The first specification states that φ always holds when neither fault is ever injected. The second specification stipulates that φ always holds whenever fault 1 can be injected. The third specification states that φ always holds whenever either fault 1 or fault 2 can be injected.

4.2. Reasoning about tolerance to injected intermittent faults

Reasoning about the total tolerance to system faults can be useful for systems in which a property must hold in all faulty scenarios. However, it is often useful to reason about the tolerance to faults when these occur intermittently.

The following formula can be used to verify that φ always holds at points in which an intermittent fault is injected:

$$AG(\Psi \rightarrow \varphi) \tag{4.3}$$

This formula states that φ holds in all reachable computational paths in which the intermittent persistence of multiple faults Ψ holds. This enables us to assess the tolerance of the system with respect to φ and the combination of intermittent faults in question.

We can further combine Formulas (4.2) and (4.3) to reason about intermittent faulty behaviour when specific faults are either absent or present in the system by means of the formula:

$$AG(\Phi \wedge \Psi \rightarrow \varphi) \tag{4.4}$$

4.2.1. Example specifications of tolerance to intermittent faults

Analogously to Fig. 4, Fig. 5 shows some models illustrating a few variations (amongst many) of how Formulas (4.3) and (4.4) can be used to define specifications of tolerance to intermittent faults which can be read in the same way as Fig. 4. Here we also introduce *start* and *stop* points indicating the point at which a fault begins and ends injecting in relation to the transitions from the set of states $\{w_rstt, w_astt\}$ and the transition into the set of states $\{stop_i, stop_ni\}$ of the fault injection agent.

Specifications based on Formula (4.3) can be used to reason about the behaviour of the system whenever a fault is injected, whenever a fault cannot be injected, and after a fault has stopped being injected as follows:

$$AG(injected_1 \rightarrow \varphi)$$

$$AG(\neg injecting_1 \rightarrow \varphi)$$

$$AG(stopped_1 \rightarrow \varphi)$$

The first specification determines whether φ is tolerant to fault 1 during its occurrence at the current tick. The second specification stipulates that φ holds whenever fault 1 does not occur at the current tick. The third specification states that φ always holds when fault 1 has previously been injected but cannot be injected again (at the current tick and in all future ticks).

We can utilise specifications based on Formula (4.4) to reason about the intermittent persistence of a fault when another fault is absent. For example, the formula

$$AG(\neg faulty_2 \wedge stopped_1 \rightarrow \varphi)$$

states that φ always holds whenever fault 2 cannot be injected and fault 1 has previously been injected but cannot be injected again. Formula (4.4) can therefore be used to determine whether the system is tolerant to a fault whenever another fault is not present.

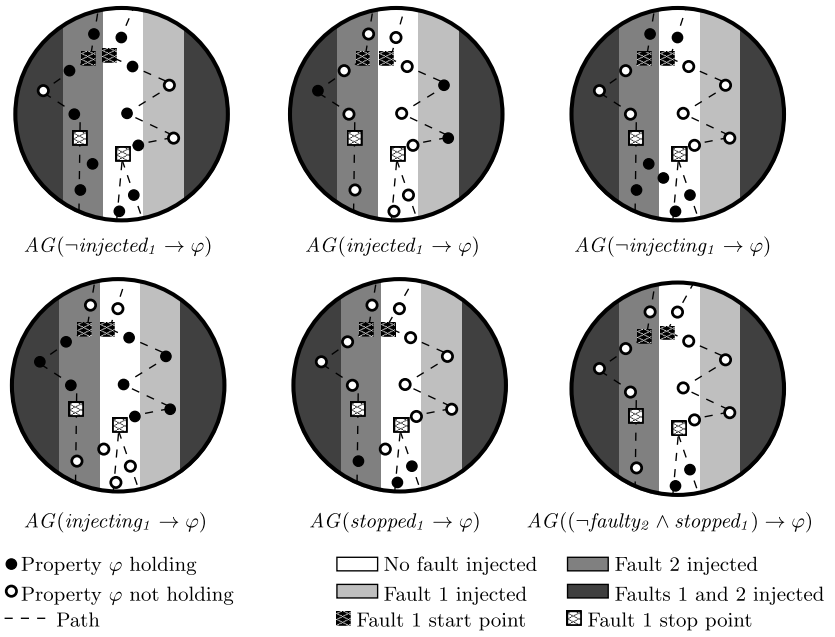


Fig. 5. Example models highlighting specifications of tolerance to intermittent faults.

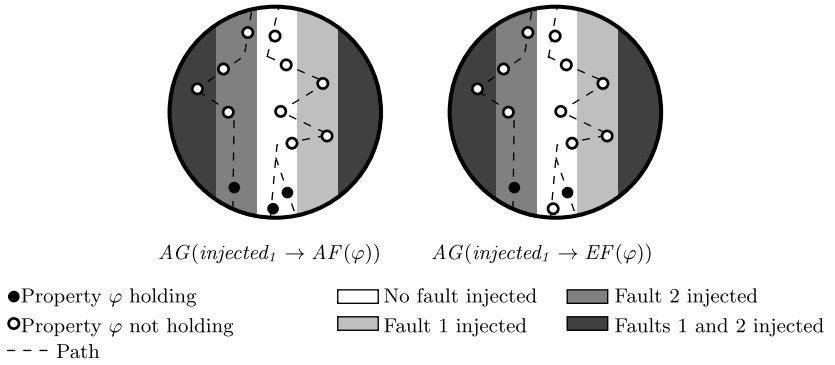


Fig. 6. Example models highlighting specifications of recoverability.

4.3. Reasoning about recoverability

The following formula can be used to reason about the recoverability of the system when a fault occurs.

$$AG(injected_j \rightarrow EF\varphi); \tag{4.5}$$

This formula captures the fact it is always true that whenever a fault is injected, along some path at some point φ holds. Thus, if this specification holds, it can be determined that the system *might* recover from the fault in terms of property φ.

We may wish to establish whether the system enjoys stronger robustness by analysing the property:

$$AG(injected_j \rightarrow AF\varphi); \tag{4.6}$$

This formula states that is always true that whenever a fault is injected, along all paths at some point φ holds. Thus, if this specification holds, the system *will* eventually recover from the fault in terms of property φ.

The models in Fig. 6 highlight recoverability specifications which are based on Formulas (4.5) and (4.6) which can be read in the same way as those in Fig. 5.

Formulas (4.2), (4.5), and (4.6) can be combined to reason about recoverability as follows:

$$AG(\Phi \wedge injected_j \rightarrow EF\varphi); \tag{4.7}$$

$$AG(\Phi \wedge injected_j \rightarrow AF\varphi); \tag{4.8}$$

In these formulas Φ is used to restrict the recovery test to states where particular faults are either present or absent in the system when reasoning about recoverability from an injected fault.

Note that by using the formulas above we cannot gain any insight in terms of *how long* it takes for the system to recover, nor *how probable* the recovery is (in the case of Formula (4.5)). These limitations are inherent in the use of CTL. They can be overcome by employing more expressive logics; we do not pursue this here.

4.4. Reasoning about diagnosability

Having so far explored the issue of resilience to faults with respect to temporal-epistemic specifications, we now turn our attention to diagnosability [14]. We express diagnosability, i.e., the ability of a system to identify a malfunctioning component, by means of epistemic specifications. We distinguish between diagnosability properties in the presence of observable and unobservable system behaviour. Observability will be used to ascertain whether the system's diagnosability mechanism functions correctly. Assumptions on unobservable behaviour will be used to refer to the traditional diagnosability setting of post-mortem analysis, i.e., when we wish to reason about faults after they have occurred.

We write Δ to indicate a diagnosis property of the system in which the diagnosis of an individual fault or group of faults from the observable behaviour of the system is ascertained. Consider the following formula:

$$AG(\Delta \rightarrow K_i(faulty_j)) \tag{4.9}$$

This formula states that whenever a diagnosis property of the system Δ occurs, agent i knows that fault j may be present (i.e., it has been injected). The distinction between paths of faulty behaviour by the diagnosis property provides an insight into the ability of agent i to determine that the faulty behaviour has occurred. This specification is useful for verifying diagnosability in agents that distinguish and act upon individual faults, e.g., those whose task is to restore functionality to the system.

In the case where a diagnosis is made for different types of faults using the same mechanism, given any fault $k \in \{1 \dots m\}$ consider the following formula:

$$AG((faulty_j \wedge faulty_k \wedge \Delta) \rightarrow (K_i(\Theta_j \vee \Theta_k) \ \wedge \neg K_i(\Theta_j) \wedge \neg K_i(\Theta_k))) \tag{4.10}$$

This formula states that whenever faults j and k can be injected and a diagnosis property of the system Δ occurs, agent i knows that either Θ_j or Θ_k (e.g., a fault j or k is being injected at the current tick of the clock, can be injected at the current tick of the clock, or has stopped being injected) but does not know specifically whether it is Θ_j or Θ_k. Thus, the formula specifies the ability of agent i to use the same mechanism for identifying a *range* of faults correctly, rather than diagnosing individual faults. Note that, in contrast to Formulas (4.10), (4.9) the formula specifies a "minimal" form of diagnosis for the disjunction of the fault. By insisting on the two final conjuncts we ensure that no incorrect diagnosis of any underlying individual fault is made by agent i. For instance, if we had $K_i(\Theta_j)$ every time $faulty_j \wedge faulty_k$ holds, then $K_i(\Theta_j \vee \Theta_k)$ would hold but this may at times correspond to an incorrect diagnosis. The specification above forces the diagnosis only to refer to the disjunction of faults, but insists on no incorrect diagnosis ever being made.

We now consider knowledge of faults in relation to the occurrence of faults. Here, we only consider specifications pertaining to the diagnosis of faults after they have first occurred. Specifications that consider the diagnosis of re-occurring faults can be further defined.

The following formula can be used to reason about diagnosability when a fault has occurred without explicitly referencing a diagnosis property of the system.

$$\neg E(\neg\Theta_j \ U \ (\Theta_j \wedge \neg AF(K_i(\Theta_j)))) \tag{4.11}$$

This formula states that there is no path in which at some point Θ_j becomes true and at which point it is not true that at some point in the future agent i knows Θ_j. The formula specifies the ability of agent i to diagnose faults correctly in relation to the occurrence of faults, such as when a fault begins occurring. The specification makes no reference to how it takes for this knowledge to become acquired; but this can be added by suitable encoding of propositional facts.

So far we have described specifications pertaining to individual agent diagnosis of faults. It is also possible to reason about group knowledge of faults. Consider the following specification:

$$\neg E(\neg\Theta_j \ U \ (\Theta_j \wedge \neg AF(D_\Gamma(\Theta_j)))) \tag{4.12}$$

This formula states that there is no path in which at some point Θ_j and at which point it is not true that at some point in the future it is distributed knowledge amongst the group Γ that Θ_j. The formula specifies the ability of a group of agents to diagnose faults correctly. It is important to assess when the group as a whole have sufficient information to know a fact, in this case a correct diagnosis. This means that there is sufficient information in the system for this fact to become known and the engineer can, in principle, devise an information sharing protocol for this fact to become explicitly known by some or all agents in the system. For more details and a discussion of further aspects of distributed knowledge we refer to [2]. When there is a need, similar specifications can obviously also be given determining whether a specific, some, or every agent in a group knows about the fault, or indeed whether a group acquires common knowledge of the fault.

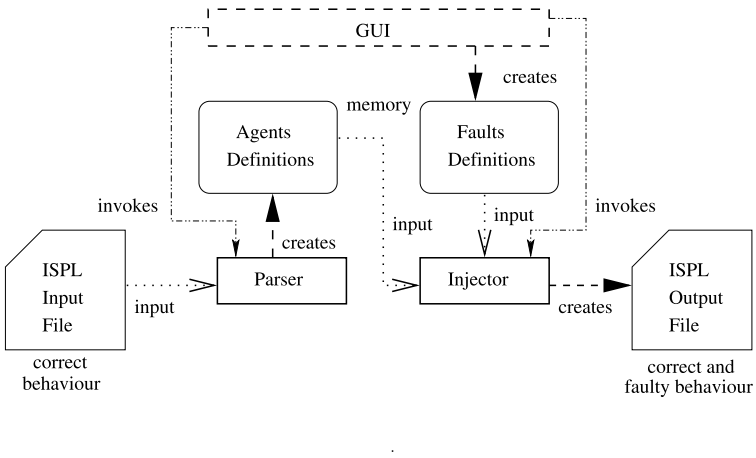


Fig. 7. Architecture of the fault injection toolkit.

Finally, we may wish to reason about the propagation of the knowledge of faults through the system, using the following specification:

$$\neg E(\neg K_i(\Theta_j)\ U\ (K_i(\Theta_j) \wedge \neg AF(E_\Gamma(\Theta_j)))) \tag{4.13}$$

This formula states that there is no path in which at some point agent i comes to know Θ_j and at which point it is not true that at some point in the future everybody in group Γ knows Θ_j. This formula specifies the propagation of the knowledge of faults to a group of agents. In other words if at some point agent i comes to know Θ_j, then eventually all agents in Γ will know this. Similar specifications can also be defined determining whether distributed knowledge of a fault amongst a group of agents results in knowledge of that fault for an individual agent, i.e., agents exchanging information so that someone knows about the fault, and whether everybody knowing about the fault leads to common knowledge of the fault.

The diagnosability formulas can be extended where required, for example we can extend Formulas (4.11), (4.12), and (4.13), to reason about the diagnosis of a range of faults in a similar manner to Formula (4.10).

We stress that the analysis above is only concerned with devising specifications *to assess* whether the system possesses some elements of resilience and diagnosability. The actual mechanisms that realise these properties are coded at system level and are not of concern here.

5. A toolkit for fault injection

To reason about a MAS diagnosability properties, we have built a prototype toolkit that operates on appropriate MAS models. The toolkit supports the injection of faults into ISPL programs in a variety of ways described below. The mutated models can then be checked by MCMAS [19], a model checker that takes ISPL files as input. The toolkit, written in C++ using GTK+ for the GUI, is targeted for Linux operating systems, and is available for public use [28].

Toolkit architecture The toolkit takes an ISPL program as input and provides a GUI which allows the user to inject faults into the MAS model and output a mutated MAS model. The architecture of the toolkit is shown in Fig. 7. The process for injecting faults is as follows:

1. The GUI invokes the *Parser* to read in the agent definitions from the ISPL input program describing the original model.
2. The agents definitions are used as input into the GUI to allow the user to define faults.
3. The faults definitions are created by the user using the GUI.
4. Once the user has finished defining faults, the GUI invokes the *Injector*.
5. The *Injector* combines the agents and faults definitions to create an ISPL output program defining the mutated model, containing both correct and faulty executions, ready to be checked by MCMAS.

Parser The *Parser* is given a filename for the ISPL input program. First, the *Parser* checks the input program to ensure it is correctly defined. Any programs that are incorrectly defined are rejected. The *Parser* then creates an object in memory containing the agent definitions. This includes the name of the agents, the variables defined in each agent, their actions, protocols, and transition relations. The object also contains information pertaining to the initial states, global variables, groups, fairness definitions, and specifications defined in the ISPL input program.

GUI The GUI facilitates the injection of any number of faults using any number of corresponding fault injection agents. Each fault injection agent is named uniquely and can be defined using several persistence options. The GUI is illustrated in

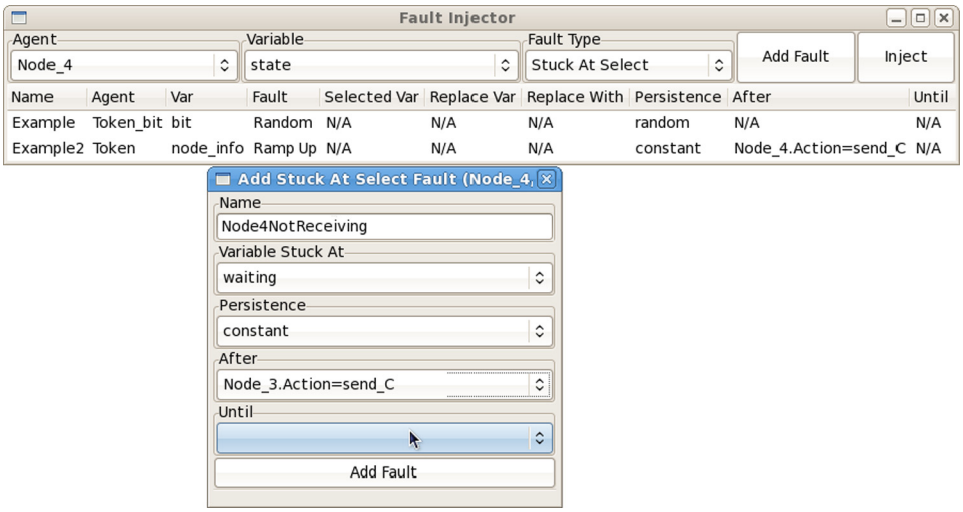


Fig. 8. The fault injection toolkit GUI.

```
For each line L in tA
  If Find(LHS(L), var + "=" + v1)
    tA* += LHS(L) + "if" + RHS(L) + "and"
         + FI + ".Action = inject"\n";
    tA* += Remove(LHS(L), var + "=" + v1)
         + "if" + RHS(L) + "and !"
         + FI + "Action = inject\n";
  else
    tA* += LHS(L) + "if" + RHS(L) + "\n";
```

Fig. 9. The *Injector* mutation function pseudo-code.

Fig. 7, which shows the way in which the agents, variables, and types of faults can be selected, and the manner in which persistence options can be defined for the fault.

After a correctly defined ISPL program is given via the GUI, the user can add faults for injection by firstly selecting the agent and variable she wishes to inject the fault on, followed by the type of fault they wish to create (Fig. 8). Each option is presented to the user using a drop down list of available agents, variables, and fault types. After these options have been selected the *Add Fault* button is used to introduce the fault. The user is then asked to name the fault and choose options relating to the faults persistence using drop down lists.

Once the user has added the faults, he or she can choose to click on the *Inject* button, which allows the user to specify an output file for the mutated ISPL program. The *Injector* is then invoked by the GUI thereby creating the ISPL output file. The ISPL file is then used as input with MCMAS.

Injector The *Injector* combines the agents and faults definitions to create an ISPL output program containing correct and faulty behaviour. The first task of the *Injector* is to output the fault injection agents. For each fault the ISPL code is created for the associated fault injection agent according to the name of the fault and the persistence options.

The next task is to output all of the non-fault injection agents evolution functions including both correct and faulty behaviour according to the faults definitions. The ISPL code is mutated by applying the mutation rules of a fault to each evolution line of the agent the fault is being injected into. The transition relation is mutated into the transition relation of the faulty agent $t_{A^{F*}}$. The mutation is performed using string find and string remove functions. The pseudo-code in Fig. 9 illustrates how the transition relation t_A of an agent A is mutated for a *stuck-at-select* fault where the variable stuck at is $v1$. In the pseudo-code *LHS* returns a string containing the target state and *RHS* returns a string containing the transition condition. The function *Find* returns a boolean value indicating whether a string has been found. The function *Remove* takes two strings as parameters, removes the second string from the first string and returns the resulting string.

To inject multiple faults on the same agent, the process is repeated for each fault on the transition relation mutated by the previous fault.

Finally, the *Injector* outputs the mutated initial states and global states according to the faults definitions, as well as the groups, fairness, and specifications from the original model.

6. Evaluation

In the previous sections we have introduced a taxonomy of specifications for reasoning about faults, recovery, and diagnosability in the context of temporal-epistemic specifications for MAS. This has formed the backbone of a methodology for

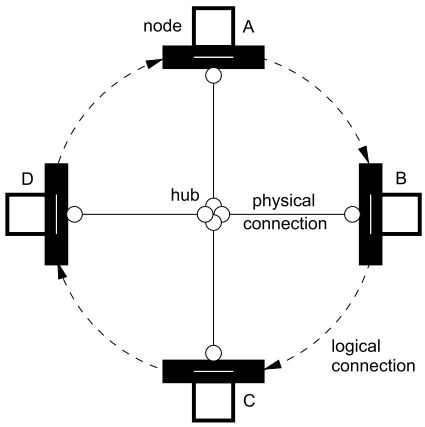


Fig. 10. The token ring protocol.

model mutation that allows to take any MAS model programmed in ISPL and inject faults automatically into it through a toolkit thereby obtaining an updated model representing the faulty behaviour under analysis. This software toolkit, released as open-source to the community, can be used in collaboration with engineers to analyse fault-tolerance, recovery, and effective diagnosability with respect to MAS-based specifications in concrete engineering scenarios.

We assume begin studying a system through its discrete model given in ISPL. Note that automatic discretisation and abstraction techniques as well as compilers from a number of languages into ISPL have been developed; see, e.g., [26]. From the ISPL model we can model check the specifications of interest and verify whether they hold. If the system is found not to meet its specifications, the usual process of refinement (facilitated by MCMAS's production of counterexamples) can be carried out. The case of interest here is when the specific system meets its specification, but there is an interest in analysing the consequences of possible faults in the system. Indeed, in many concrete engineering cases, there are very concrete indications from case studies and prototypes on what parts of the system may develop faults and how these faults could manifest themselves. A large number of them correspond to the failure modes we discussed earlier in Section 3.

In other words, the methodology and toolkit presented allow the engineer to inject automatically faults of interest in a given model, generate the resulting system, and analyse key specifications on the mutated system. Below we illustrate this process in the case of a popular networking protocol, the IEEE 802.5 Token Ring protocol [29]. Through this analysis we will observe that the protocol does satisfy some key specifications whenever no host develops a fault, but we will focus on the impact the faults have in terms of recoverability and diagnosability properties. To do so we will analyse a specific network running the protocol and mutate the behaviour of some hosts to model specific kinds of failure. By means of the methodology described so far we will then analyse the resulting properties of the instantiated example. While our analysis generally confirms the robustness of the IEEE 802.5 in the case analysed, this section is not intended to provide a validation of the IEEE 802.5. Doing so would require to analyse an arbitrary number of hosts and arbitrary faults, therefore requiring, among others, topological considerations and parametric analysis (see, e.g., [30]). Instead, the aim here is simply to illustrate the methodology on a concrete case study drawn from applications. We conducted an analogous exercise in the context of fault-tolerance and diagnosis of autonomous vehicles in [31]. We opt for the IEEE protocol here because of its broader interest.

6.1. Overview of the protocol

The IEEE 802.5 token ring protocol is a popular local area network (LAN) protocol in which the nodes of the network are logically organised in a ring topology. The data circulates in one direction in the form of a *token* passed from node to node. While the token ring is logically defined as a ring topology, it is physically defined as a star topology as illustrated in Fig. 10. This facilitates fault tolerance by allowing faulty nodes to be bypassed by physically disconnecting the faulty node and re-establishing the logical ring.

To ensure fault tolerance a node can act as a *monitor* to diagnose faults and take action to resolve them. During the normal operation of the network a token can be populated by a node with data to be sent to another node. When a fault occurs, tokens containing fault information are sent around the network thereby allowing a monitor to identify and correct faults on the network.

When the ring is initialised, a contention process takes place during which one node is designated as an *active monitor*. The active monitor has the responsibility of issuing new tokens when tokens are lost, removing orphaned tokens which circulate the ring more than once, and establishing and re-establishing a fully operational ring. The rest of the nodes act as *standby monitors* and are responsible for diagnosing faulty nodes or cable breaks. When an active monitor is unable to perform its duties correctly, a standby monitor can make a claim to become an active monitor.

The process of sending data, determining the active monitor, establishing an operational ring and diagnosing faulty nodes is made possible by using several different types of tokens. A *data token* circulates the ring when the ring is fully operational. A *claim token* is used to decide which node becomes the active monitor. When a node receives the claim token back, it becomes the active monitor. A *ringpolling* token is used by the active monitor to establish the correct operation of the ring. When the active monitor receives the ringpolling token back, it creates a new data token and circulates it. A *beaconing* token is used to signal a problem when a node is unable to receive tokens. It contains the address of the last known nearest upstream neighbour of the node that is not receiving tokens.

Determining whether there is a problem on the network is achieved by a timer on each node. After it has sent out a data token, the active monitor starts a timer that counts down until the time it takes for a token to circulate the network. If the timer reaches zero without the active monitor receiving the token back, the active monitor knows the data token has been lost and sends out a ringpolling token. If a timeout occurs for the ringpolling token, the active monitor knows it cannot establish a fully operational ring, de-activates itself as active monitor and initiates the claim token process. All other nodes timeout when they have not received a token from their nearest upstream neighbour after a specified period of time. If this timeout occurs, the node makes a claim to become the active monitor. If the claim process times out, the node goes into beaconing mode.

The goal of the beaconing process is to allow the ring to bypass any faulty nodes on the network. The beaconing node identifies the fault domain as either itself or its nearest upstream neighbour by sending out a beaconing token containing the address of its nearest upstream neighbour. If a node receives several beaconing tokens reporting it as the faulty node, it disconnects from the network. If a beaconing station sends out several beaconing tokens with no success in repairing the network it disconnects from the network.

The protocol defines that the active monitor must send regular messages around the ring to let standby monitors know it is present on the ring. If a standby monitor has not received an active monitor message for a specified period of time, it enters into the claim process. This ensures fault tolerance in situations where the active monitor becomes faulty and cannot monitor the network for faults.

6.2. ISPL implementation

We encoded the token ring protocol above described as an ISPL program. In the ISPL implementation the environment agent encodes the hub, abstracts from the timer details of the individual nodes, and manages the token being passed among nodes. A node agent represents a node of the network; we implement as many node agents as there are nodes on the ring. Each node agent is named $N[x]$ where $[x]$ is the number of the node. One node on the ring is designated as a sender and one node is designated as a receiver. The sender sends messages to the receiver until it receives an acknowledgement. A token agent contains the token data; an associated token bit agent determines whether the token has been inspected. Active monitor messages were omitted from the implementation to investigate situations in which there is no information passed between nodes indicating that the active monitor has become faulty.

We defined a network with 6 nodes: $N1, \ldots, N6$. The nodes are arranged clockwise from Node 1 to Node 6 with the token circulating clockwise; it is assumed Node 1 always wins contention for the active monitor. We declare Node 2 as a sender and Node 6 as a receiver. Node 2 repeatedly sends messages to Node 6 until Node 2 has received an acknowledgement from Node 6. The implementation is available for download [28].

6.2.1. Environment agent

The environment agent implements the physical ring hub and abstracts from the timer details of the individual nodes and manages the token passing between nodes. The variables are defined as follows.

Status	{*Wait_tok*, *Wait_ack*, *Send*, *TO_tok*, *TO_ack*}
Token	{*D*, *C*, *RP*, *B*}
SStatus	{*Processing*, *Sending*}
Timeout	0 . . . *TIMEOUT_VAL*
Curnode	1 . . . *NUM_NODES*
Amonitor	0 . . . *NUM_NODES*

where *Status* is the current status of the hub, *Token* is the token received and sent by the hub, *SStatus* indicates whether a message or token is being processed or sent, *Timeout* is a timeout counter, *Curnode* indicates the last node the token was sent to and *Amonitor* is the current active monitor.

The hub waits for tokens (*Wait_tok*) to be sent to it. In each evolution if a token has not been received by the hub, the *Timeout* timer counts down. If a token is received from a node, it sends the token (*Send*) to the nearest downstream neighbour, bypassing nodes that have been disconnected. The *Timeout* timer is then reset and the hub waits for an acknowledgement that the token has been received from the node that the token was sent to (*Wait_ack*). If an acknowledgement is received, it waits for the next token to be sent (*Wait_tok*); otherwise the timeout timer counts down.

If the timeout counter reaches zero, a timeout message is sent. If the timeout occurs waiting for a token from a node (*TO_tok*), a timeout message is sent to the active monitor; if there is no active monitor, the timeout message is sent to

the nearest downstream neighbour of the node. If the timeout occurs while waiting for an acknowledgement from a node (*TO_ack*), a timeout message is sent to the active monitor; if there is no active monitor, the timeout message is sent to the node. Thus, the timeouts occur in the correct sequence: first, the active monitor times out; then the nearest downstream neighbour of the node that is not sending tokens times out. Similarly, if a node is not receiving tokens, the active monitor times out, followed by the node.

Tokens are sent to nodes via an action *Send*_[*n*]_[*t*], where *n* is the value of the variable *Curnode* and *t* is the value of the variable *Token*. Similarly, the timeout message action is *Timeout*_[*n*], where *n* is the value of *Amonitor* or *Curnode*.

6.2.2. Node agent

The node agent is comprised of a number of enumerate, integer and boolean variables as follows:

Istatus	{*Wait*, *Process*, *TimeO*, *SetT*, *Send*, *Disconnect*}
Rstatus	{*Repeating*, *Ringpolling*, *Claiming*, *Beaconing*}
Token	{*D*, *RP*, *C*, *B*}
Amonitor	boolean
Bfailed	1 . . . *NUM_FAILED*
Breceived	1 . . . *NUM_RECEIVED*
Rec	boolean
Sent	boolean

where *Istatus* is the internal status of the agent; *Rstatus* is the status on the ring; *Token* is the current token being processed or sent by the agent; *Amonitor* defines whether the node is an active monitor; *Bfailed* indicates the number of beaconing tokens that have been sent unsuccessfully; *Breceived* indicates the number of beaconing tokens that have been received; *Sent* is only applicable to the sender nodes and indicates that it has sent a message to a receiver; similarly, *Rec* indicates whether a receiver has received a message or a sender has received an acknowledgement.

In terms of behaviours, a node agent starts off by waiting for a token (*Wait*). When a token is received, it processes the token (*Process*). After this, it can set the token (*SetT*) before sending the token (*Send*) at which point it returns to waiting. When a token has not been received after a specified period of time, the node receives a message from the environment stating that the timer has timed out (*TimeO*). If the node receives or sends several beaconing tokens, it disconnects (*Disconnect*), and remains in this state. Tokens are sent using an action *Send*_[*t*], where [*t*] is the type of token.

A node can populate the token using the *Set_token_D* action. If the node is an active monitor it can free the token using the *Free_token* action or set a flag to indicate the token has been seen using the *Set_Token_bit* action. A receiver that receives a token destined for it sets *Rec* to true and populates the token with an acknowledgement. If the sender receives an acknowledgement, it sets *Rec* to true and stops sending messages to the receiver.

6.2.3. Token agents

The *Token_data* agent contains a single integer variable *Node_number* which represents the data of the token (i.e., a destination for the token) and is set according the actions of node agents. If the sender node agent performs a *Set_token_D* action, the value is set to the number of the receiver node. If the receiver node agent performs a *Set_token_D* action, the value is set to the number of the sender node. If a node agent performs a *Set_token_C* action, the token is set to the number of the node performing the action. If a node agent performs a *Set_token_B* action, the token is set to the number of the nodes nearest upstream neighbour. If a node agent performs a *Free_token* action, *Node_number* is set to zero. The action associated with *Node_number* is the value of *Node_number* (i.e., action One for value 1, action Two for value 2 etc.), which allows a node agent to determine whether the token is destined for it.

The *Token_bit* agent contains a single boolean variable *Bit* which is set to *true* whenever a node agent performs a *Set_Token_bit* action and *false* if a *Free_token* action is performed. The associated actions with *Bit* are *Set* and *Not_set* according to whether it is set to *true* or *false* respectively.

6.3. Choice of injected faults

Having implemented the protocol in ISPL we checked with MCMAS whether the network we modelled satisfied some key specifications. For example, among others, we checked that the formulas below are true on the model:

$$AF(msgsent)$$

$$AG(msgsent \rightarrow (AF(recack)))$$

$$AG(\neg am \rightarrow (AF(am)))$$

$$AG(\neg beaconing)$$

$$AG(\neg timeout)$$

This provided reassurance that the model is sound.

Table 7
Injected faults on the token ring protocol.

Fault	Agent	Fault type	Persistence
TKwd	*Token_data*	*Variable transition* (*Node_number*) *v1* = *7*	*rstt*, *astt*, *asto* *STA* = {*N2.Send_D*} *SPA* = {*TKwd.injected*}
N1sm	*N1*	*Boolean Set* (*Amonitor*) *v1* = *false*	*rstt*, *astt*, *asto* *STA* = {*N1.Send_D*} *SPA* = {*N1sm.injected*}
sN2ns	*N2*	*State replace* (*Istatus*) *v1* = *Sending* *v2* = *Waiting*	*rstt*, *astt*, *asto* *STA* = {*N1.Send_D*} *SPA* = {*N1.Send_C*}
hN3ns	*N3*	*State replace* (*Istatus*) *v1* = *Sending* *v2* = *Waiting*	*rstt*, *astt*, *asto* *STA* = {*N2.Send_D*, !*N3.Disconnected*} *SPA* = {*N3.Disconnected*}
hN4nr	*N4*	*Stuck at select* (*Istatus*) *v1* = *Waiting*	*rstt*, *astt*, *asto* *STA* = {*N3.Send_D*} *SPA* = {*N4.Disconnected*}
hN6ns	*N6*	*State replace* (*Istatus*) *v1* = *Sending* *v2* = *Waiting*	*rstt*, *astt*, *asto* *STA* = {*N5.Send_D*} *SPA* = {*N6.Disconnected*}

To evaluate the methodology we injected a number of soft and hard faults into the model to evaluate the protocol's response. These are shown in Table 7 where *Fault* indicates the name chosen for the fault, *Agent* is the node that the fault is injected on, *Fault Type* is the type of fault injected and the corresponding parameters, and *Persistence* indicates the persistence options for the fault. The meaning of the faults is as follows:

TKwd: A populated data token becomes destined for the wrong workstation.
N1sm: Node 1 becomes a standby monitor if it is an active monitor.
sN2ns: Node 2 stops sending tokens (soft fault).
hN3ns: Node 3 stops sending tokens (hard fault).
hN4nr: Node 4 stops receiving tokens (hard fault).
hN6ns: Node 6 stops sending tokens (hard fault).

Our use of soft and hard faults is derived from a standard classification of fault types for token ring networks [32]. A *soft* fault is an intermittent error cause by (amongst other things) degradation of the electrical signal. The ring can recover from a soft fault such as a packet loss without entering the beaconing process. A *hard* fault is a persistent error caused by disruption of the electrical signal path at some point on the ring. This prevents tokens from circulating until the faulty node is removed. While the primary aim of this exercise is to provide an illustration of the methodology, the combination chosen is rich and relatively realistic in terms of faults that may actually happen on a network. Further faults could be added, but it would be unreasonable to expect that properties of fault-tolerance from the protocol could be reasoned about when, for example, a hard fault occurs on all of the nodes, since this would result in a total network failure.

In order to keep a sufficient number of nodes on the ring so that message delivery and acknowledgement can take place following the occurrence of hard faults, we stipulated that the ring does not enter a state where non-faulty nodes become disconnected. To achieve this, the start actions for the hard and soft faults are set so that the faults occur at different times, and only when the ring is sending data tokens. Furthermore, since Node 3 can become disconnected by a hard fault on Node 4, *hN3ns* can only begin occurring when Node 3 is not disconnected. To distinguish between soft and hard faults, the stop action for the soft fault *sN2ns* is set so that it stops occurring when there is no active monitor on the ring. The stop actions for the hard faults imply that a hard fault stops occurring once the node causing a problem has become disconnected.

Both the *TKwd* and *N1sm* faults stop occurring immediately after they first occur, which is sufficient to create the desired failures. The token destination can become incorrect when the sender (Node 2) sends a data token. Node 1 can become a standby monitor whenever it has sent a data token, at which point it is assumed to be an active monitor. These faults allow us to investigate faulty scenarios in which there is an orphaned token on the ring and faulty scenarios in which there is no active monitor on the ring.

Fairness is imposed on the faults so that in any path where *faulty* is true for a fault, eventually a random start is invoked for the fault.

6.4. Code mutation

We now illustrate automatic ISPL code mutation as performed by the fault injection toolkit on the *hN4nr* fault. The corresponding fault injection agent is defined in ISPL as follows:

```
Agent hN4nr
 Vars:
   status: {not_fault,w_astt,fault_i,stop_i, w_rstt};
 end Vars

 Actions = {dont_inject,inject,start};

 Protocol:
  status = stop_i : {dont_inject};
  status = fault_i : {inject};
  status = w_astt :  {dont_inject};
  status = w_rstt : {dont_inject,start};
  status = not_fault : {dont_inject};
 end Protocol

 Evolution:
  status = stop_i if status = fault_i and Node_4.Action = disconnected;
  status = fault_i if status = w_astt and Node_2.Action = send_N;
  status = w_astt if hN4nr.Action = start;
 end Evolution
end Agent
```

The corresponding evolution function in the Node 4 agent is updated to contain a *stuck-at-select* fault where *v1* is *waiting*. The ISPL code is defined as follows with the boxed code containing the update to the original evolution function:

```
Evolution:
 Istatus = Process and Token = D if Istatus = Wait and
   Environment.Action = send_node_4_D
     and !hN4nr.Action = inject ;
 Istatus = Process and Token = C if Istatus = Wait and
   Environment.Action = send_node_4_C
     and !hN4nr.Action = inject ;
 Istatus = Process and Token = RP if Istatus = Wait and
   Environment.Action = send_node_4_RP
     and !hN4nr.Action = inject ;
 Istatus = Process and Token = B if Istatus = Wait and
   Environment.Action = send_node_4_B
     and !hN4nr.Action = inject ;
 Token = D if Istatus = Wait
   and Environment.Action = send_node_4_D
   and hN4nr.Action = inject;
 Token = C if Istatus = Wait
   and Environment.Action = send_node_4_C
   and hN4nr.Action = inject;
 Token = RP if Istatus = Wait
   and Environment.Action = send_node_4_RP
   and hN4nr.Action = inject;
 Token = B if Istatus = Wait
   and Environment.Action = send_node_4_B
   and hN4nr.Action = inject;
   .
   .
end Evolution
```

Table 8
Boolean formulas used in the evaluation of the token ring protocol.

Proposition	Condition
msgsent	$N2.Sent = true$
recack	$N2.Rec = true$
timeout	$N1.Istatus = TimeO \vee N2.Istatus = TimeO \vee, \ldots, \vee\ N6.Istatus = TimeO$
timeoutnoam	$N2.Istatus = TimeO \vee N3.Istatus = TimeO \vee, \ldots, \vee\ N6.Istatus = TimeO$
beaconing	$N1.RStatus = Beaconing \vee N2.RStatus = Beaconing \vee, \ldots, \vee\ N6.RStatus = Beaconing$
token_free	$Token_data.Node_Number = 0$
am	$N1.Amonitor = true \vee N2.Amonitor = true \vee \ldots \vee\ N5.Amonitor = true \vee\ N6.Amonitor = false$
$N3_d$	$N3.Istatus = Disconnect$
$N4_{sb}$	$N3.RStatus = Beaconing \wedge N3.Bfailed = 0$
$N1^{to}_{am}$	$N1.Istatus = TimeO \wedge N1.Amonitor = true$
$N1^{to}_{\neg am}$	$N1.Istatus = TimeO \wedge N1.Amonitor = false$
$hN3ns_{is}$	$hN3ns_i \vee hN3ns_s$
$hN4nr_{is}$	$hN4nr_i \vee hN4nr_s$
$hard_f$	$hN3ns_f \vee hN4nr_f \vee hN6ns_f$
$hard_i$	$hN3ns_i \vee hN4nr_i \vee hN6ns_i$
$hard_{is}$	$hN3ns_{is} \vee hN4nr_{is} \vee hN6ns_i \vee hN6ns_s$
$soft_i$	$sN2ns_i$
$soft_f$	$sN2ns_f$
$allhs_f$	$hN3ns_f \wedge hN4nr_f \wedge hN6ns_f \wedge sN2ns_f$
$anyhs_f$	$hard_f \vee soft_f$
$anyhs_i$	$hard_i \vee soft_i$
any_f	$N1sm_f \vee TKwd_f \vee anyhs_f$

The ISPL code containing the global and initial states is updated to include the atomic propositions and initial states for *hn4nr* fault injection agent as follows:

```
Evaluation
 .
 .
 hN4nr_injecting if hN4nr.status = fault_i;
 hN4nr_faulty if !hN4nr.status = not_fault;
 hN4nr_stopped if hN4nr.status = stop_i;
 .
 .
end Evaluation
InitStates
.
.
 and (hN4nr.status =  w_rstt
    or hN4nr.status = not_fault)
.
.
end InitStates
```

The full mutated code can be found at [33].

6.5. Verifying fault tolerance, recoverability, and diagnosability in the token ring protocol

We used the mutated ISPL program of the protocol and our specification patterns to reason about fault tolerance, recoverability, and diagnosability in the token ring protocol. We performed preliminary verification on the protocol to ensure that: 1) all the faults can enter a start and stop state; 2) Node 1 is the only active monitor; 3) Nodes 1 and 4 can reach a timeout state; 4) Nodes 3, 4, and 6 are the only nodes that can disconnect. These represent additional properties that we expected the system to satisfy. MCMAS verified all the specifications above and those discussed in the rest of this section in approximately 32 hours on an Intel Core 2 Quad 8400 2.66 GHz processor running a 2.6.32-33 Linux Kernel with 4 GB of memory. The number of reachable states for the mutated model was approximately 4.96×10^6.

To reason about the injected faults, we defined a number of atomic propositions. These are reported in Table 8 where *Proposition* is the name of the atomic proposition and *Condition* is the condition making the atomic proposition true. As naming conventions, *hard* indicates any hard fault; *soft* indicates a soft fault; *allhs* indicates all hard and soft faults; *anyhs* indicates any hard or soft fault; $_f$ is the *faulty* persistence; $_i$ is the *injecting* persistence; $_s$ is the *stopped* persistence; $_{is}$ is either the injecting or stopped persistence; $_d$ indicates that a node is disconnected; $_{sb}$ indicates that a node has started sending beacons; $_{to}$ indicates a node has timed out; $_{am}$ indicates that the node is the active monitor. The *injected* persistence is the same as the *injecting* persistence for all faults as the fault is injected constantly.

To begin with, we established some properties of the protocol under faulty and non-faulty behaviour that are fundamental to our study. The following specifications stipulate that at some point a message is sent; at some point under faulty behaviour an acknowledgement is received; at some point under non-faulty behaviour an acknowledgement is received.

$$AF(msgsent)$$

$$AF(any_f \rightarrow recack)$$

$$AF(\neg any_f \rightarrow recack)$$

MCMAS reported the first and third specifications as true, and the second as false. This establishes that messages are always sent at some point, and acknowledgements are always received under non-faulty behaviour, but are not always received under faulty behaviour.

We now turn our attention to investigating properties of the token ring protocol under faulty behaviour. The following specification (an instantiation of Formula (4.1)) states that when there is no active monitor on the ring, then eventually an active monitor is re-established.

$$AG(\neg am \rightarrow AF(am))$$

MCMAS reported this specification as false and reported a counterexample showing a system run of 330 global states (approximately 550 kb) in which a token circulates without an active monitor on the ring. This result allows us to determine that the faulty behaviour prevents the ring from correctly designating an active monitor when one is absent. To further investigate this behaviour, consider the following specification (corresponding to Formula (4.2)) stating that if Node 1 cannot exhibit the faulty behaviour of becoming a standby monitor when it is an active monitor, and there is no active monitor on the ring, then always at some point in the future there is an active monitor on the ring.

$$AG((\neg N1sm_f \wedge \neg am) \rightarrow AF(am))$$

MCMAS verified the specification as true. This and the previous specification establish that, in the absence of active monitor messages, the ring has no tolerance to the faulty active monitor behaviour since there is no mechanism to let standby monitors know when there is no active monitor on the ring. However, this specification also confirms that under this faulty behaviour the ring operates correctly in terms of establishing an operational active monitor.

The following specification states that whenever a token has a destination, then always at some point in the future the token becomes free.

$$AG(\neg token_free \rightarrow AF(token_free))$$

MCMAS reported this specification to be false and showed a counterexample that indicates a system run of 476 global states (approximately 800 kb) in which an orphaned token repeatedly circulates the ring. Thus, the faulty behaviour prevents the ring from correctly freeing a token whenever it has a destination.

We can look more closely at this behaviour by using the following specification based on Formula (4.2) which states that if Node 1 cannot exhibit the faulty behaviour of becoming a standby monitor when it is an active monitor or the token destination is not corrupted, and a token has a destination, always at some point in the future the token becomes free.

$$AG((\neg N1sm_f \wedge \neg TKwd_f \wedge \neg token_free) \rightarrow AF(token_free))$$

MCMAS verified this specification as true. This signifies that without a mechanism to let standby monitors know when there is no active monitor on the ring, the combination of a missing active monitor and an orphaned token prevents a token from being correctly freed.

The next specification of interest we analysed (matching Formula (4.1)) stipulates that the ring never enters the beaconing process.

$$AG(\neg beaconing)$$

MCMAS found this specification to be false, and a counterexample of 128 global states (approximately 216 kb) was generated in which the status of Node 4 is beaconing. Thus, the faulty behaviour causes the ring to enter the beaconing process.

To investigate the beaconing process further, we can use a specification (corresponding to Formula (4.2)) stating that if there is not a hard fault on the ring, the ring does not enter the beaconing process.

$$AG(\neg hard_f \rightarrow \neg beaconing)$$

The specification was verified as true, which means that hard faults are the cause of beaconing. To investigate the faulty behaviour further, we can check whether any of the node timers ever reaches zero.

$$AG(\neg timeout)$$

MCMAS found a counterexample to this formula showing a system run of 66 global states (approximately 115 kb) in which the active monitor enters a timeout state.

Given the formula was verified as true in the model without faults, we can therefore conclude that, as a result of the faulty behaviour, one of the nodes enters a timeout state.

To investigate hard and soft faults further consider the following specifications (corresponding to Formula (4.2)). They state that: if there is not a hard fault on the ring, then none of the node timers ever reaches zero; if there is not a soft fault on the ring, then none of the node timers ever reaches zero; if there are no hard and soft faults on the ring, then none of the node timers ever reaches zero.

$$AG(\neg hard_f \rightarrow \neg timeout)$$

$$AG(\neg soft_f \rightarrow \neg timeout)$$

$$AG((\neg hard_f \ \wedge \ \neg soft_f) \rightarrow \neg timeout)$$

MCMAS reported the first two specifications to be false and provided counterexamples of system runs with a length of 61 global states (approximately 100 kb), and 66 global states (approximately 115 kb) respectively, demonstrating that the active monitor enters a timeout state. However, the last specification was verified as true signifying that both hard and soft faults cause a node on the ring to timeout.

To illustrate how soft and hard faults affect the nature of the fault detection, consider the following specifications. They stipulate that: if there is not an active monitor fault and soft fault on the ring, then no standby monitor ever times out; if there is not an active monitor fault nor hard fault on the ring, then no standby monitors ever time out.

$$AG((\neg N1sm_f \ \wedge \ \neg soft_f) \rightarrow \neg timeoutnoam)$$

$$AG((\neg N1sm_f \ \wedge \ \neg hard_f) \rightarrow \neg timeoutnoam)$$

The first specification was reported to be false and a counterexample was given showing with a length of 98 global states (approximately 165 kb) in which Node 4 enters a timeout state. The second specification was verified as true. We conclude that hard faults can cause standby monitors to time out whereas soft faults only cause the active monitor to timeout.

In summary, by using Formulas (4.1) and (4.2) we have shown that: 1) without a mechanism to let the standby monitors know when there is no active monitor on the ring, the ring cannot correctly designate an active monitor and free orphaned tokens; 2) hard faults cause the ring to enter the beaconing process; 3) soft faults cause the active monitor to timeout; 4) hard faults cause the standby monitors to timeout. We now proceed to consider intermittent faults.

Verifying aspects of tolerance to intermittent faults The following specification, based on Formula (4.3), states that if the active monitor has exhibited the faulty behaviour of becoming a standby monitor, then there is no active monitor on the ring.

$$AG(N1sm_s \rightarrow \neg am)$$

MCMAS reported this specification to be false and provided a counterexample showing a system run with a length of 225 global states (approximately 378 kb). The result appears counter-intuitive since one may expect that if an active monitor has become a standby monitor, then there will never be an active monitor due to the omission of messages informing standby monitors that there is no active monitor on the ring. However, the counterexample reported that Node 1 became a standby monitor when *N1sm* occurred, Node 1 then later became an active monitor after *hN3ns* occurred. The counterexample also showed other nodes in the ring claiming to become active monitors.

We further examine this behaviour by using a specification (corresponding to Formula (4.4)) which states that if the active monitor has exhibited the faulty behaviour of becoming a standby monitor and there no hard or soft faults can occur, then there is no active monitor on the ring.

$$AG((N1sm_s \wedge \neg anyhs_f) \rightarrow \neg am)$$

MCMAS verified this specification as true. This indicates that hard and soft faults may cause an active monitor to become designated in the absence of active monitor messages.

We can further demonstrate this is the case by checking a specification based on Formula (4.3), which states that if the active monitor has exhibited the faulty behaviour of becoming a standby monitor and any of the hard or soft faults are currently occurring, then always at some point in the future there will be an active monitor on the ring.

$$AG((N1sm_s \ \wedge \ anyhs_i) \rightarrow AF(am))$$

The result of checking this specification is true. The conclusion from the above investigation is that if a hard or soft fault occurs, then an active monitor becomes designated even in the absence of active monitor messages.

6.6. Verifying aspects of recoverability

To reason about recovery, consider the specification below, based on Formula (4.6), stating that whenever any hard or soft faults are injected and a message has been sent, then always at some point in the future the sender will receive an acknowledgement.

Table 9
Specifications of diagnosability for the token ring protocol.

A	$AG((\neg N1sm_f \wedge N1^{to}_{am}) \rightarrow K_{N1}(hN6ns_f)))$	**T**	(4.9)
B	$AG((\neg N1sm_f \wedge all_f \wedge N1^{to}_{am}) \rightarrow (K_{N1}(anyhs_i) \wedge \neg K_{N1}(hard_i) \wedge \neg K_{N1}(soft_i)))$	**T**	(4.10)
C	$\neg E((\neg N1sm_f \wedge \neg anyhs_i)\ U\ (anyhs_i \wedge \neg AF(K_{N1}(anyhs_i))))$	**T**	(4.11)
D	$\neg E((\neg N1sm_f \wedge \neg anyhs_i)\ U\ (anyhs_i \wedge \neg AF(E_{ALL}(anyhs_i))))$	**F**	(4.12)
E	$\neg E((\neg N1sm_f \wedge \neg D_{ALL}(anyhs_i))\ U\ (D_{ALL}(anyhs_i) \wedge \neg AF(E_{ALL}(anyhs_i))))$	**F**	(4.13)
F	$AG((\neg N1sm_f \wedge allhs_f \wedge N4_{sb}) \rightarrow (K_{N4}(hN3ns_{is} \wedge hN4nr_{is}) \wedge \neg K_{N4}(hN3ns_{is}) \wedge \neg K_{N4}(hN4nr_{is})))$	**T**	(4.10)
G	$AG((\neg N1sm_f \wedge allhs_f \wedge N3_d) \rightarrow (K_{N3}(K_{N4}(hN3ns_{is} \vee hN4nr_{is}) \wedge \neg K_{N4}(hN3ns_{is}) \wedge \neg K_{N4}(hN4nr_{is}))))$	**T**	(4.10)
H	$\neg E((\neg N1sm_f \wedge \neg hN3ns_i)\ U\ (hN3ns_i \wedge \neg AF(K_{N4}(hN3ns_{is} \vee hN4nr_i))))$	**T**	(4.11)
I	$\neg E((\neg N1sm_f \wedge \neg hN4nr_i)\ U\ (hN4nr_i \wedge \neg AF(K_{N4}(hN3ns_{is} \vee hN4nr_{is}))))$	**T**	(4.11)
J	$\neg E((\neg N1sm_f \wedge \neg D_{ALL}(hN3ns_i \vee hN4nr_i))\ U\ (D_{ALL}(hN3ns_{is} \vee hN4nr_{is}) \wedge \neg AF(E_{ALL}(hN3ns_{is} \vee hN4nr_{is}))))$	**F**	(4.13)
K	$\neg E((\neg N1sm_f \wedge \neg D_{ALL}(hN3ns_i \vee hN4nr_i))\ U\ (D_{ALL}((hN3ns_{is} \vee hN4nr_{is})) \wedge \neg AF(E_{ALL}(hard_{is}))))$	**T**	(4.13)
L	$\neg E((\neg N1sm_f \wedge \neg E_{ALL}(hard_{is}))\ U\ (E_{ALL}(hard_{is}) \wedge \neg AF(C_{ALL}(hard_{is}))))$	**F**	(4.13)

$$AG((anyhs_i \wedge msgsent) \rightarrow AF(recack))$$

MCMAS reported this specification as false and provided a counterexample showing a system run of 300 global states (approximately 504 kb) in which a token circulates the ring without an acknowledgement reaching the sender. This means that, in the presence of all the faults here analysed, the ring will not necessarily deliver all acknowledgements to messages.

To try and isolate the issue in more detail, we tested the following specification (corresponding to Formula (4.8)) expressing that if an active monitor does not exhibit the faulty behaviour of becoming a standby monitor, and if a token is never orphaned, then whenever any hard or soft fault is injected and a message has been sent, then always at some point in the future the sender will receive an acknowledgement.

$$AG((\neg N1sm_f \ \wedge\ \neg TKwd_f \ \wedge\ anyhs_i \wedge msgsent) \rightarrow AF(recack))$$

The result of checking this specification is true.

These results demonstrate that the combination of an orphaned token and active monitor incorrectly becoming a standby monitor prevents an acknowledgement from always being delivered. Thus, the considerations above highlight the requirement for informing the standby monitors whenever there is no active monitor on the ring.

6.7. Verifying aspects of diagnosability

We now report the results we obtained while verifying diagnosability in the token ring protocol using specifications that rely on the epistemic states of the nodes in the system. The specifications we used are given in Table 9, along with the result that MCMAS returns when checking the formula, and the corresponding diagnosability specification pattern used to define the specification. In our analysis, we omit the scenario in which the active monitor becomes a standby monitor, as active monitor messages are required in the diagnosis process.

Specification A states that whenever Node 1 is not an active monitor and enters a timeout state, then it knows that there is a hard fault on Node 6. Specification B states that whenever all hard and soft faults occur in a run of the system, if Node 1 is an active monitor and enters a timeout state, then it knows that there is a fault occurring on the ring, but does not know whether it is a soft or hard fault.

To reason about the diagnosability without referring to a diagnosis property, Specification C states that it is not possible that a fault occurs without eventually Node 1 (the active monitor) knowing that a fault has occurred. Similar specifications can be obtained for the other nodes. Specification D expresses the fact that a fault cannot occur without this eventually becoming distributed knowledge amongst all nodes. Specification E concerns the transfer of knowledge of faults: it states that a fault cannot be distributedly known without eventually all of the nodes knowing this. Again, similar specifications referring to individual nodes, faults or subsets of either can be checked if required.

To express the diagnosability of the beaconing process, Specification F expresses that whenever all hard and soft faults occur in a run of the system, if Node 4 enters a state where it has begun sending beacons, then it knows that either it is not receiving tokens or Node 3 is not sending tokens, but does not know specifically which of these faults has occurred. Specification G states that whenever all hard and soft faults occur in a run of the system, if Node 3 becomes disconnected, then it knows that Node 4 knows that either Node 3 is either not sending tokens, or Node 4 is not receiving tokens, but does not know specifically which of these nodes is not sending or receiving tokens.

To verify the diagnosability of the beaconing process without referencing a diagnosis property, Specifications H and I state that it is not possible that Node 3 is unable to send tokens or Node 4 cannot receive tokens without eventually Node 4 knowing that one of these faults has occurred. Specification J states that it is not possible that there is distributed knowledge of one of these faults without eventually all of the nodes knowing that one of these faults has occurred. Similarly,

Specification K states that it is not possible that there is distributed knowledge of one of these faults without eventually all of the nodes knowing that a hard fault has occurred. Specification L states that it is not possible that all of the nodes know a hard fault has occurred without eventually becoming common knowledge amongst all of the nodes that a hard fault has occurred.

The verification results for these formulas are reported in the table. From these we can infer the following properties: 1) A standby monitor can diagnose a fault on its nearest upstream neighbour and an active monitor can diagnose any hard and soft fault on the ring; 2) The protocol is not capable of translating distributed knowledge of a fault into everyone knowing the fault; 3) Not every node on the ring comes to know about a fault since the same mechanism that is used to diagnose soft faults is used to establish the ring when it initialises, and the standby monitors are not able to differentiate between initialisation and diagnosis of soft faults; 4) The beaconing process allows a node to diagnose and resolve a hard fault that has occurred between itself or its nearest upstream neighbour; 5) In contrast to soft faults, the knowledge of the occurrence of a hard fault is propagated during the beaconing process so that the nodes of the ring can co-ordinate during the beaconing process; 6) Hard faults do not become common knowledge amongst the nodes as this would require a mechanism for broadcasting knowledge of the fault simultaneously to all nodes. These properties are not discussed in the original token ring specification [29] and contribute to our understanding of the protocol, its resilience to faults, and diagnosability properties.

7. Related work

In previous work we have put forward a preliminary methodology for reasoning about fault-injection and diagnosability in the context of MAS [34,35]. The present article consolidates and expands in a systematic way the material published there. The methodology was also further applied to a usecase from the autonomous system domain not discussed here [36].

From a theoretical point of view, various notions of faults have received considerable attention in the distributed systems community and in knowledge-based approaches. Specifically, “crash failure”, “omission failure” and “Byzantine failure” have been defined and analysed, particularly in the context of protocol analysis and fault analysis [2]. The notions of model mutation discussed in this paper are inspired from these and extend them by considering more sophisticated variants. The fundamental difference is not the generality of the approach but its different overall emphasis and objective. For example, in the knowledge-based analysis of Byzantine agreement [37–39], as well as in other protocols, the objective is to establish whether epistemic notions, notably common knowledge, can be established in entire class of protocols, e.g., agreement schema such as the attacking generals. Instead, here we are not concerned with what epistemic states some protocols can achieve but on a methodology that can be used to decide whether particular temporal-epistemic specifications hold in the presence of faults, and whether epistemic properties representing diagnosability are realised *in the concrete MAS under analysis*.

Much closer to the techniques here presented and a direct inspiration for our work is previous research, described below, on safety-analysis methods combining fault-injection with model checking for the verification of safety-critical systems [16,9,11,17]. In this literature a wealth of mature technologies have been developed to assess the robustness of systems. However, these are limited to temporal specifications, whereas here we are concerned with MAS, which are typically specified by means of a variety of modal logics, notably epistemic logic as we do here. Furthermore, these approaches do not consider the automated analysis of diagnosability, an aspect here explored.

The techniques used so far involve the popular model checker NuSMV [9], process algebras such as CCS/Meije [16,11], and the commercial SCADE tool by Esterel Technologies coupled with the SCADE Design Verifier model checker [17]. Our classification of failure modes is based upon the faults defined in [9,17].

In [9] an integrated tool for injecting faults into a system model defined in NuSMV is applied to verify safety-critical avionics systems. The tool automatically mutates the NuSMV code according to a library of failure modes. The tool provides a library of temporal logic formulas for safety requirements whose definition is pattern based. Due to the high level of automation of the tool in specifying safety requirements, injecting faults, and producing fault trees, the tool is successful in improving the usability for non-experts in formal verification. However, the tool is limited to reactive systems specified in temporal logic and specifications for verifying diagnosability are not considered.

Related to this, is the extensive work within the EU project Compass [40] aimed at providing a model-based approach to systems co-engineering and tools to identify critical faults in systems for the aerospace domain. Within this project not only tools for assessing functional correctness via model checking have been developed [41], but also fault-injection and dependability analysis techniques have been put forward [42,43]. The emphasis of this work is different from ours. Firstly, it mostly targets hardware and reactive systems for space exploration; therefore it is tailored to temporal specifications in a probabilistic setting. In contrast we here focus on MAS characterised by epistemic specifications. Secondly, the Compass project is concerned with actual deployment in the aerospace domain, whereas here we intended to use temporal-epistemic specifications as a unifying formalism for both the initial MAS specifications and to express diagnosability.

In [17] faults are injected into SCADE for a model of a wheel brake system. Verification is performed on the faulty model by reasoning about the faults using temporal logic. Specifications are included to check whether safety properties hold under faulty conditions. These are specific to the faulty model and not generically extended from verification of the correct model. The fault injection is not automatic which limits the level of automation of the tool.

In [11] fault tolerance is verified via model checking using mechanisms for handling faults modelled for process algebras such as CCS by applying special-purpose process operators. Similarly, in [16] a modelling approach for formalising fault tolerant systems is proposed for the CCS/Meije process algebra and model checking applied to verify fault tolerance and recoverability. The proposed formalisms are not suitable for verifying MAS, and the work is not extended to provide a practically usable tool.

The previous work on analysing diagnosability considers discrete event systems [14,44], and model based diagnosis systems [13,45]. The main focus of the work in [45,14,44] is the formalisation of the diagnosability problem and less attention is given to the practicality of the proposed algorithms for analysis. This makes the techniques difficult to apply for non-experts in formal verification. Additionally, some of the assumptions made in some of the above literature entail a heavy computational cost. For example, [14] assumes a perfect recall semantics, whereas our setting is only observational.

A practical approach to verifying diagnosability using temporal logic model checking is given in [13]. In this line, a coupled twin model of the diagnosis system must be constructed so that diagnosability can be expressed as a temporal specification. This implies that the modelling component of the technique is significantly more difficult for non-experts to perform in comparison to injecting automatically faulty behaviour. The practicality of this approach is not examined for systems in which distributed diagnosis is present. In contrast with all the lines of research cited above, our contribution employs concepts based on epistemic logic to define notions of diagnosability. In the setting of epistemic logic we here adopt, the knowledge of an agent (e.g., the knowledge of a fault or of a recoverability property) does not refer to what the agent knows *explicitly*, but, instead, expresses whether the agent in question has sufficient information to deduce with certainty, if given unlimited resources, whether a given property holds. While this notion does not provide an algorithmic procedure for computing the knowledge of the agents, it has nonetheless been shown useful as it provides a bound for the knowledge of the agents in an idealised setting. The diagnosability properties we here explore are to be interpreted in the same way.

After the research reported here was conducted, [46] put forward a notion of diagnosability and maximality based on epistemic logic. The setup includes temporal operators to reason about the past and adopts a synchronous, perfect-recall semantics for the epistemic operator representing the diagnoser. Checking perfect-recall knowledge operators is undecidable in some settings and has exponential complexity in others. We are not aware of any existing model checker supporting epistemic modalities with perfect recall semantics for more than 2 agents. While the approaches share the theme of a knowledge-based representation of the diagnoser, they differ in the technical details; moreover while [46] considers a single diagnoser we are here concerned with distributed diagnosis. Also recently, [47] defined various notions of diagnosability in terms of probabilistic and epistemic properties of systems. Given we do not employ probabilities in our setup the approaches cannot readily be compared.

8. Conclusions

In this article we presented an automated approach to verifying fault tolerance, recoverability, and diagnosability in MAS. We put forward a methodology for mutating MAS models in order to produce systems exhibiting faulty behaviour, and defined a library of temporal and epistemic specifications to reason about the correct and faulty behaviours of the mutated system. A tool was developed and paired with a MAS model checker to help engineers study faults and diagnosability properties automatically through model checking the resulting models. The methodology was evaluated on the token ring protocol, a networking protocol that incorporates mechanisms utilising distributed diagnosis to facilitate recovery from faults.

Contribution. While verification of MAS through model checking temporal-epistemic specifications has received considerable attention in the AI and MAS communities recently [24,23,19], much remains to be done in terms of developing robust methodologies for analysing properties of MAS operating in degraded circumstances. Being able to validate aspects of fault-tolerance, recovery and correct diagnosability is essential in key MAS application areas such as autonomous vehicles as, in the future, formal techniques may provide a basis for the certification of these systems. In this context, we see the contribution of this paper as threefold. Firstly, the fault-injection technique developed here is model-based and independent of the specifications used, we employed it on models on which generic temporal-epistemic specifications (those typically used to analyse agent systems) can be checked. This enables us to assess MAS in the presence of faults against MAS-based specifications. Secondly, the notion of diagnosability has been formalised in temporal-epistemic languages so that diagnosability analysis can be conducted through model checking mutated models and diagnosability mechanisms of MAS can be verified. Thirdly, an open-source toolkit has been released to assist with analysing the fault-tolerance and diagnosability of autonomous systems in the presence of a large class of possible faults. These three aspects contribute to our long term ambition of helping engineers to certify the correctness of autonomous agents.

Limitations. The analysis conducted on the token ring protocol only applies to the specific configuration considered and the faults injected. It is not intended to certify the protocol in question. Doing so would require parametric verification and abstraction on the topology of the network. This is in principle possible, but it is not the objective of the scenario presented. The application discussed is meant to be an illustration of the methodology put forward that involves experimenting with degraded conditions of interest to the engineer.

As with any approach based on model checking, the state-space explosion remains the bottleneck of any model checking call. Model checkers such as MCMAS mitigate this problem by means of symbolic representations of the state-space, typically

by using binary decision diagrams (BDDs). Even if this enables the verification of systems several orders of magnitude larger than what is feasible with explicit approaches, state-spaces much larger than those discussed in the scenario analysed here become challenging for plain BDD-based approaches. To limit these difficulties a number of methodologies are available including predicate abstraction and symmetry reduction.

Future work. We intend to use the current approach to verify aspects of fault tolerance and diagnosability in autonomous systems. Additional timing options for re-occurring faults will be added to the fault injection agent in order to allow for the broadening of our diagnosability specification patterns to reason about them. We envision the extension of the toolkit to allow for user defined mutation rules and automatic generation of typical fault tolerance, recoverability, and diagnosability specifications.

We conducted the analysis in the context of the BDD-based model checker MCMAS. The methodology was instantiated on concrete ISPL files. On large MAS this forces us to use ISPL generators to produce the model of the system. A promising alternative is to use a parameterised version of ISPL [48–50] so that the analysis cold be based on templates instead.

References

[1] M.J. Wooldridge, Reasoning About Rational Agents, MIT Press, 2000.
[2] R. Fagin, J.Y. Halpern, M.Y. Vardi, Y. Moses, Reasoning About Knowledge, MIT Press, 1995.
[3] A. Fedoruk, R. Deters, Improving fault-tolerance by replicating agents, in: Proceedings of the 1st International Conference on Autonomous Agents and Multiagent Systems, AAMAS'02, ACM Press, 2002, pp. 737–744.
[4] Z. Guessoum, J.P. Briot, S. Charpentier, O. Marin, P. Sens, A fault-tolerant multi-agent framework, in: Proceedings of the 1st International Conference on Autonomous Agents and Multiagent Systems, AAMAS'02, ACM Press, 2002, pp. 672–673.
[5] M. Kalech, G.A. Kaminka, On the design of social diagnosis algorithms for multi-agent teams, in: Proceedings of the 18th International Joint Conference on Artificial intelligence, IJCAI'03, Morgan Kaufmann, 2003, pp. 370–375.
[6] R. Micalizio, P. Torasso, G. Torta, On-line monitoring and diagnosis of a team of service robots: a model-based approach, AI Commun. 19 (4) (2006) 313–340.
[7] N. Roos, A. Teije, C. Witteveen, A protocol for multi-agent diagnosis with spatially distributed knowledge, in: Proceedings of the 2nd International Conference on Autonomous Agents and Multiagent Systems, AAMAS'03, ACM Press, 2003, pp. 655–661.
[8] E. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[9] M. Bozzano, A. Villafiorita, The FSAP/NuSMV-SA safety analysis platform, Int. J. Softw. Tools Technol. Transf. 9 (1) (2007) 5–24.
[10] A. Cimatti, Industrial applications of model checking, in: Proceedings of the 4th International Summer School on Modeling and Verifying Parallel Processes, MOVEP'00, in: LNCS, vol. 2067, Springer, 2001, pp. 153–168.
[11] G. Bruns, I. Sutherland, Model checking and fault tolerance, in: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology, AMAST'97, in: LNCS, vol. 1349, Springer, 1997, pp. 45–59.
[12] F. Raimondi, A. Lomuscio, C. Pecheur, Applications of model checking for multi-agent systems: verification of diagnosability and recoverability, in: Proceedings of the 14th International Workshop on Concurrency, Specification, and Programming, CS&P'05, Warsaw University, 2005, pp. 433–444.
[13] A. Cimatti, C. Pecheur, R. Cavada, Formal verification of diagnosability via symbolic model checking, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI'03, Morgan Kaufmann, 2003, pp. 363–369.
[14] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, D. Teneketzis, Diagnosability of discrete-event systems, IEEE Trans. Autom. Control 40 (9) (1995) 1555–1575.
[15] R. Iyer, Experimental evaluation, in: Proceedings of the 25th International Symposium on Fault-Tolerant Computing, FTCS'95, IEEE, 1995, pp. 115–132.
[16] C. Bernardeschi, A. Fantechi, S. Gnesi, Model checking fault tolerant systems, Softw. Test. Verif. Reliab. 12 (4) (2002) 251–275.
[17] A. Joshi, M.P.E. Heimdahl, Model-based safety analysis of Simulink models using SCADE design verifier, in: Proceedings of the 24th International Conference on Computer Safety, Reliability, and Security, SAFECOMP'05, in: LNCS, vol. 3688, Springer, 2005, pp. 122–135.
[18] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, FOCS'77, IEEE, 1977, pp. 46–57.
[19] A. Lomuscio, H. Qu, F. Raimondi, MCMAS: a model checker for the verification of multi-agent systems, in: Proceedings of the 21st International Conference on Computer-Aided Verification, CAV'09, in: LNCS, vol. 5643, Springer, 2009, pp. 682–688.
[20] G.J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295.
[21] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inf. Comput. 98 (2) (1992) 142–170.
[22] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model verifier, in: Proceedings of the 11th International Conference on Computer Aided Verification, CAV'99, in: LNCS, vol. 1633, Springer, 1999, pp. 495–499.
[23] M. Kacprzak, W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, B. Wozna, A. Zbrzezny, Verics 2007 – a model checker for knowledge and real-time, Fundam. Inform. 85 (1–4) (2008) 313–328.
[24] P. Gammie, R. van der Meyden, MCK: model checking the logic of knowledge, in: Proceedings of the 16th International Conference on Computer-Aided Verification, CAV'04, in: LNCS, vol. 3114, Springer, 2004, pp. 479–483.
[25] A. Lomuscio, M.J. Sergot, A formalisation of violation, error recovery, and enforcement in the bit transmission problem, J. Appl. Log. 2 (1) (2004) 93–116.
[26] A. Lomuscio, H. Qu, M. Solanki, Towards verifying contract regulated service composition, Auton. Agents Multi-Agent Syst. 24 (3) (2012) 345–373.
[27] J. de Kleer, B.C. Williams, Diagnosing multiple faults, Artif. Intell. 32 (1) (1987) 97–130.
[28] J. Ezekiel, A. Lomuscio, MCMAS fault injection compiler, http://vas.doc.ic.ac.uk/software/tools/ispl-fault-injector/.
[29] R. Donnan, IEEE Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications, IEEE Standard 802.5-1989, IEEE, 1989.
[30] Y. Hanna, D. Samuelson, S. Basu, H. Rajan, Automating cut-off for multi-parameterized systems, in: Proceedings of the 12th International Conference on Formal Engineering Methods, ICFEM'10, in: LNCS, vol. 6447, Springer, 2010, pp. 338–354.
[31] J. Ezekiel, A. Lomuscio, L. Molnar, S.M. Veres, Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle, in: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI'11, AAAI Press, 2011, pp. 1659–1664.
[32] N.C. Strole, A local communications network based on interconnected token-access rings: a tutorial, IBM J. Res. Dev. 27 (5) (1983) 481–496.
[33] J. Ezekiel, A. Lomuscio, ISPL model for token ring with faults, http://vas.doc.ic.ac.uk/software/tools/ispl-fault-injector/tokenring/.
[34] J. Ezekiel, A. Lomuscio, Combining fault injection and model checking to verify fault tolerance in multi-agent systems, in: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS'09, IFAAMAS, 2009, pp. 113–120.
[35] J. Ezekiel, A. Lomuscio, An automated approach to verifying diagnosability in multi-agent systems, in: Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods, SEFM'09, IEEE, 2009, pp. 51–60.

[36] J. Ezekiel, A. Lomuscio, L. Molnar, S. Veres, Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle, in: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI'11, AAAI Press, 2011, pp. 1659–1664.
[37] C. Dwork, Y. Moses, Knowledge and common knowledge in a byzantine environment: crash failures, Inf. Comput. 88 (2) (1990) 156–186.
[38] M.J. Fischer, N.A. Lynch, A lower bound for the time to assure interactive consistency, Inf. Process. Lett. 14 (4) (1982) 183–186.
[39] G. Neiger, M.R. Tuttle, Common knowledge and consistent simultaneous coordination, Distrib. Comput. 6 (3) (1993) 181–192.
[40] Correctness, modelling and performance of aerospace systems, http://compass.informatik.rwth-aachen.de/.
[41] M. Bozzano, A. Cimatti, J.P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri, R. Wimmer, A model checker for AADL, in: Proceedings of the 22nd International Conference on Computer-Aided Verification, CAV'10, Springer, 2010, pp. 562–565.
[42] M. Bozzano, A. Cimatti, J.P. Katoen, V.Y. Nguyen, T. Noll, M. Roveri, The COMPASS approach: correctness, modelling and performability of aerospace systems, in: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security, SAFECOMP'09, in: LNCS, vol. 5775, Springer, 2009, pp. 173–186.
[43] B. Ern, V.Y. Nguyen, T. Noll, Characterization of failure effects on AADL models, in: Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security, SAFECOMP'13, in: LNCS, vol. 8153, Springer, 2013, pp. 241–252.
[44] Y. Wang, T.-S. Yoo, S. Lafortune, Diagnosis of discrete event systems using decentralized architectures, Discrete Event Dyn. Syst. 17 (2) (2007) 233–263.
[45] L. Console, C. Picardi, M. Ribaudo, Diagnosis and diagnosability analysis using PEPA, in: Proceedings of the 14th European Conference on Artificial Intelligence, ECAI'00, IOS Press, 2000, pp. 131–135.
[46] M. Bozzano, A. Cimatti, M. Gario, S. Tonetta, Formal design of fault detection and identification components using temporal epistemic logic, in: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'14, in: LNCS, vol. 8413, Springer, 2014, pp. 326–340.
[47] X. Huang, Diagnosability in concurrent probabilistic systems, in: Proceedings of the 12th International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS'13, IFAAMAS, 2013, pp. 853–860.
[48] P. Kouvaros, A. Lomuscio, A cutoff technique for the verification of parameterised interpreted systems with parameterised environments, in: Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI'13, AAAI Press, 2013, pp. 2013–2019.
[49] P. Kouvaros, A. Lomuscio, A counter abstraction technique for the verification of robot swarms, in: Proceedings of the 29th AAAI Conference on Artificial Intelligence, AAAI'15, AAAI Press, 2015, pp. 2081–2088.
[50] P. Kouvaros, A. Lomuscio, Parameterised verification for multi-agent systems, Artif. Intell. 234 (2016) 152–189.