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

Jonathan Ezekiel
Department of Computing
Imperial College London
jezekiel@doc.ic.ac.uk

Alessio Lomuscio
Department of Computing
Imperial College London
alessio@doc.ic.ac.uk

ABSTRACT

The ability to guarantee that a system will continue to operate correctly under degraded conditions is key to the success of adopting multi-agent systems (MAS) as a paradigm for designing complex agent based fault tolerant systems. In order to provide such a guarantee, practically usable tools and techniques for verifying fault tolerant MAS architectures are urgently required. In this paper we address this requirement by combining automatic fault injection with model checking to verify fault tolerance in MAS. We present a generic method to mutate a model of a correctly behaving system into a faulty one, and show how the mutated model can be used to reason about fault tolerance, which includes recovery from faults. The usefulness of the proposed method is demonstrated by injecting automatically a fault into a sender-receiver protocol, and verifying temporal and epistemic specifications of the protocol's fault tolerance using the MCMAS model checker.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking

General Terms

Verification

Keywords

Model checking, fault tolerance, fault injection, epistemic logic

1. INTRODUCTION

The multi-agent systems (MAS) paradigm [22] has been employed successfully in several disciplines based on systems in which the core components, or agents, autonomously interact with one another, engaging in communication, negotiation, coordination, etc. One of the reasons MAS formalisms have been adopted in many scenarios is the availability of rich modal logics to analyse the behaviour of agents, including the ability to reason about the *knowledge* of agents [8].

Cite as: Combining Fault Injection and Model Checking to Verify Fault Tolerance in Multi-Agent Systems, Jonathan Ezekiel, Alessio Lomuscio, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009),* Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 113–120

To date, several *fault tolerant* architectures for MAS have been proposed (see, e.g., [9, 11]), which allow the system to continue to operate correctly under degraded conditions, such as the event of a failure within one or more of the agents. Strategies such as replicating agents [9] are used to ensure overall system tolerance to agent failures. However, the development of practically usable tools and techniques for certifying the correct behaviour of fault tolerant MAS architectures is urgently required if the MAS paradigm is to be widely employed in real-world applications.

Formal methods and, in particular, *model checking* [7] are becoming increasingly popular for verifying the correct behaviour of systems (see, e.g., [2, 5]). Recently, a number of tools have been developed by applying *fault injection* [13] to a correctly behaving system and using model checking to verify correct operation of that system under degraded conditions [1, 2, 3, 14]. Tools 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 verification process [2]. Unfortunately, due to their modelling formalisms and use of only *temporal logic* [21] for writing specifications, these tools are not directly applicable for verifying MAS.

In this paper we begin to develop usable tools for certifying the correct behaviour of fault tolerant MAS architectures by proposing and demonstrating a method for verifying fault tolerance in MAS. The method discussed combines automatic fault injection with the ability to use both temporal and *epistemic logic* [22] to write specifications for verifying fault tolerance. We ground this work on the MCMAS model checker [17], a model checker tailored to MAS specifications.

To combine automatic fault injection with MCMAS we introduce a general method for mutating a MAS model of correct behaviour by injecting commonly occurring faults into it. We then suggest a way in which temporal logic specifications of correctness for the MAS model can be extended to reason about the correct and faulty behaviours of the system, for the purposes of verifying fault tolerance, including *recoverability* i.e., recovery from faults. To move towards a usable method for verifying fault tolerance in MAS we inject automatically faults into a MAS program, and show how fault tolerant properties of the bit-transmission protocol can be analysed.

The rest of the paper is structured as follows. In Section 2 we provide the background on model checking, interpreted systems and MCMAS, and model checking faulty behaviour. In Section 3 we introduce our proposed method of automatic

fault injection including failure modes, generic MAS model extension via fault injection, and reasoning about faults. In Section 4 we apply the method to a bit-transmission protocol example, illustrate the functionalities of a compiler that we developed for automatic fault injection, and verify fault tolerance. In Section 5 we discuss the related work and in Section 6 we conclude and put forward future work.

2. BACKGROUND

Model checking [7] is a widely adopted technique for systems verification. In model checking the system considered for verification S is represented by a logical model M_S which encodes 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, $M \vDash \varphi_P$). The satisfaction relation is implemented as a decision procedure, the *automatic* nature of which makes model checking attractive for the purpose of verification [7].

Model checking tools that are used for reactive systems such as SPIN [12], SMV [4], and NuSMV [6] express φ_P as a *temporal logic* formula [21]. Model checking tools for multi-agent systems such as MCMAS [17], Verics [20] and MCK [10] facilitate a richer way of expressing φ_P by using a number of different modal logics including temporal, ATL, and epistemic logics [22]. In particular, epistemic logic can be used to reason about the *knowledge* of the agents over time.

2.1 Interpreted systems and MCMAS

The modelling of MAS is typically conducted by using interpreted systems [8]. We summarise the framework of interpreted systems as presented in [8] to model MAS. Each agent $i \in \{1, \cdots, n\}$ in the system is characterised by a finite set of local states L_i and by a finite set of actions Act_i. Actions are performed in compliance with a protocol $P_i : L_i \rightarrow 2^{Act_i}$, specifying which actions may be performed in a given state. In this formalism, the environment in which agents "live" may be modelled by means of a special agent E. Associated with E are a set of local states L_E, a set of actions Act_E, and a protocol P_E. A tuple $g = (l_1, \cdots, l_n, l_E) \in L_1 \times \cdots \times L_n \times L_E$ where $l_i \in L_i$ for each i and each $l_E \in L_E$, is a *global state* and describes the system at a particular instant of time.

The evolution of the agents' local states is described by a function $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \rightarrow L_i$, which returns a local state (the "next" local state) for agent i given the "current" local state of the agent, the "current" local state of the environment and all the agents' actions. Similarly the evolution of the environment's local states is described by a function $t_E : L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \rightarrow L_E$. It is assumed that in every state, agents evolve simultaneously. The evolution of the global states of the system is described by a function $t : S \times Act \rightarrow S$, where $S \subseteq L_1 \times \cdots \times L_n \times L_E$, and $Act \subseteq Act_1 \times \cdots \times Act_n \times Act_E$. The function t is defined as $t(g,a) = g'$ iff 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 states g (corresponding to the local state of agent i). Given a set $I \subseteq S$ of possible initial global states a set $G \subseteq S$ of reachable global states is generated by all possible runs of the system. Finally, the definition includes a set of atomic propositions AP together with a valuation function $V \subseteq AP \times S$. We define an *interpreted system* as the tuple:

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

The syntactical constructs and the semantic model that are presented in [17] are adopted for the interpretation of temporal-epistemic formulae in interpreted systems. Specifically, we consider the following syntax defining our specification language:

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

In the grammar above $p \in AP$ is an atomic proposition; EX is a temporal operator expressing that there exists a next state in which φ holds; AG is a temporal operator expressing that in all runs φ holds globally; $E(\varphi U \psi)$ is a temporal operator expressing that there exists a run in which φ holds until ψ holds; $K_i\varphi$ expresses that *agent i knows* φ [8].

IS is associated to a model $M_{IS} = (W, R_t, \sim_1, \cdots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible worlds W is the set G of reachable global states. The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by considering the temporal transition t. Two worlds w and w' are such that $R_t(w, w')$ iff there exists a joint action $a \in Act$ such that $t(w, a) = w'$, where t is the transition relation of IS. 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 $w \sim_i w'$ iff $l_i(w) = l_i(w')$ (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 [8]). The labelling relation $L \subseteq AP \times W$ is equivalent to the valuation relation V.

Formulae can be interpreted in M_{IS} in a standard way [7, 18, 8] as follows. Let $\pi = (w_0, w_1, \cdots)$ be 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 standard conventions we assume that the temporal relation is serial and thus all computation paths are infinite). We write $(M, w) \vDash \varphi$ 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.

$(M, w) \vDash p$	iff	$(p, w) \in L$,
$(M, w) \vDash \neg\varphi$	iff	$M \nvDash \varphi$,
$(M, w) \vDash \varphi_1 \vee \varphi_2$	iff	either $M \vDash \varphi_1$ or $M \vDash \varphi_2$,
$(M, w) \vDash EX\varphi$	iff	there exists a path π such that $\pi(0) = w$, and $(M, \pi(1)) \vDash \varphi$,
$(M, w) \vDash AG\varphi$	iff	for all paths we have that $\pi(0) = w$, and $(M, \pi(i)) \vDash \varphi$, for all $i \geq 0$,
$(M, w) \vDash E(\varphi U \psi)$	iff	there exists a path π such that $\pi(0) = w$, and there exists $k \geq 0$ and $(M, \pi(j)) \vDash \varphi$ such that $(M, \pi(k)) \vDash \psi$ and $(M, \pi(j)) \vDash \varphi$ for all $0 \leq j < k$,
$(M, w) \vDash K_i\varphi$	iff	for all $w' \in W$, $w \sim_i w'$ implies $(M, w') \vDash \varphi$.

We say that a formula φ is true in the model and we write $M \vDash \varphi$ if $(M, w) \vDash \varphi$ for all $w \in W$. Similarly to [8], we say that a formula φ is true in an interpreted system IS, and we write $IS \vDash \varphi$, if $M \vDash \varphi$. *A formula is true in an interpreted system if it is true in the associated Kripke model.*

MCMAS [17] provides ISPL as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulas as specifications of the system. ISPL programs are closely related to interpreted systems; specifically

each ISPL program describes an interpreted system. The model checking algorithm in MCMAS allows these specifications of the system to be automatically verified.

2.2 Model checking and faults

Traditionally, temporal logic model checking has been applied to provide assurances about the *correct* behaviour of the system. However, in safety-critical systems analysis there is also an interest in injecting faulty behaviour and analysing both the *correct and faulty* behaviours of systems by means of model checking [1, 2, 3, 14].

Reasoning about faulty behaviour is also a recent topic in MAS (see e.g., [16, 15, 19]). Faulty behaviour of MAS has been modelled and reasoned about for systems such as transmission protocols [19] and web services [15, 16]. However, an automatic method for injecting faulty behaviour into MAS has not yet been developed, meaning that faulty behaviour must be introduced by hand when modelling a system.

Despite the previous research in the area of model checking faulty behaviour, specifications for reasoning about fault tolerance are still not yet well defined. For this reason, as well as providing a manner in which faults can be automatically injected into MAS, in this paper we also extend the topic of using temporal logic to reason about correct and faulty behaviours of the system.

3. AUTOMATIC FAULT INJECTION

To provide a usable tool for verifying fault tolerant behaviour, automatic fault injection consists in taking a description of a correct system model and allowing faults to be automatically injected into the system in order to create a mutated faulty model. In a MAS oriented context, given a description of an interpreted system IS we wish to derive an extended faulty system IS^{F*} that contains the original behaviours of IS as well as some mutated behaviours. We begin defining this extension by looking at the types of common faults that occur in systems, which have been previously defined as failure modes in [2, 14].

3.1 Failure modes

Failure modes describe behaviour relating to component failures. Common types of failures in components include *random*, *stuck at* or *inverted* faults [2, 14]. 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 [14].

For the purposes of developing an automatic method for injecting faults into MAS, we consider the common types of component failures as the starting point for defining failure modes. Consider the case where agent A has a variable Var representing only two possible local states 0 and 1, i.e., $L_A = \{0, 1\}$. We define three faults that can be injected into the agent on this variable to alter the local states. *Inverting* the value of Var (so that state 0 becomes state 1 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 (arbitrarily choosing one of the states at every tick of the clock). In the following we use these failure mode definitions to define a general extension of a model representing correct behaviour into one including faults via automatic fault injection.

Table 1: Transition relation for FI.

target state	transition condition
$faulty, not\_injected$	$FI.Action = dont\_inject$
$faulty, injected$	$FI.Action = inject$

3.2 Generic MAS model extension 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*}. In order to reason about the correct and faulty behaviours of A^{F*}, we set the fault to be intermittent.

To describe an intermittent fault we define a fault injector agent FI which determines under what conditions a fault is to be injected into A^{F*}. FI contains four local states:

$$L_{FI} = \{(not\_faulty, not\_injected), (not\_faulty, injected), (faulty, not\_injected), (faulty, injected)\}$$

which indicate whether or not agent A^{F*} has in the past or will at present inject a fault. We define this agent so that $(not\_faulty/faulty)$ indicates whether faults are *ever* injected. This is decided non-deterministically when setting either $faulty$ or not_faulty as its initial state, this state then persists in the future. The pair $(not\_injected/injected)$ indicates whether a fault is being injected *at the current tick* of the clock according to $(not\_faulty/faulty)$ and the non-deterministic actions of FI.

For the fault injection agent we define in this paper we do not allow an evolution into the state $(not\_faulty, injected)$, i.e., it is not possible that faults are not injected and a fault is currently being injected. These states have been introduced so that $not\_faulty/faulty$ can be used to reason about overall correct and faulty behaviours of the mutated system, and $not\_injected/injected$ can be used to reason about intermittent faults.

In the fault injection agent we define a set of two actions $Act_{FI} = \{dont\_inject, inject\}$ indicating whether the fault is to be injected. The protocol is defined as:

$P_F(not\_faulty) = \{dont\_inject\}$,
$P_F(faulty) = \{dont\_inject, inject\}$.

Thus, the action of injecting a fault can be selected non-deterministically for intermittent faults. FI can also be defined to inject faults in a more complex way, for the purposes of varying the persistence of faults. For example, we can define the transition relation to set $injected$ to true for n evolutions of the system. We currently define the transition relation for FI to select $(dont\_inject/inject)$ non-deterministically as shown in Table 1.

In order to inject the fault into agent A^{F*} we need to mutate the evolution function t_A to contain the desired faulty behaviour in $t_{A^{F*}}$. We define the extension of the transition relation for our faults in Table 2 where *trans* indicates the evolution function, *fault* indicates the type of fault, *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, where '[...]' indicates the original transition condition.

We introduce the following atomic propositions to reason about correct and faulty behaviours of IS^{F*}. $AP^{F*} = AP \cup \{fault, injected\}$. The corresponding evaluation function

Table 2: Extended transition relation for fault injection.

trans	fault	target state	transition condition
t_A	N/A	$state = 0$	[...]
t_{AF*}	all	$state = 0$	[...] and $Act_{FI} = dont\_inject$
t_{AF*}	random	$state = 0$ or $state = 1$	[...] and $Act_{FI} = inject$
t_{AF*}	invert	$state = 1$	[...] and $Act_{FI} = inject$
t_{AF*}	stuck at	$state = state$	[...] and $Act_{FI} = inject$
t_A	N/A	$state = 1$	[...]
t_{AF*}	all	$state = 1$	[...] and $Act_{FI} = dont\_inject$
t_{AF*}	random	$state = 0$ or $state = 1$	[...] and $Act_{FI} = inject$
t_{AF*}	invert	$state = 0$	[...] and $Act_{FI} = inject$
t_{AF*}	stuck at	$state = state$	[...] and $Act_{FI} = inject$

V is updated so that $V^{F*}(fault) = \{g \in G \mid l_{A^{F*}(g)} = faulty\}$ and $V^{F*}(injected) = \{Act_{FI} = inject\}$. All that remains now is to update I to initialise the local state of FI either to not_faulty or to $faulty$ to allow for faulty or correct behaviour respectively. The extended faulty system is defined as follows:

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

3.3 Reasoning about faults

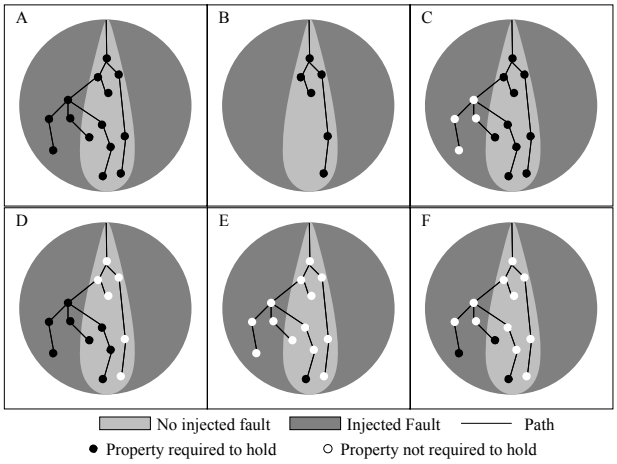

Figure 1: Computational paths under correct and faulty behaviours of the system.

Once we have obtained a mutated model IS^{F*}, both the *correct* and *faulty* behaviours of the system can be analysed. Our goal is to develop specifications that are useful for guaranteeing the correct operation of the system under degraded conditions. We achieve this by defining a number of specifications to reason about the correct and faulty behaviours of the system. This enables us to assess the extent to which faulty behaviour *affects* properties of the system and the system's ability to *recover* from faulty states.

To put this into context, consider a MAS in which a property of the system is the receipt of a message. Depending on the system architecture, a system may be deemed fault tolerant if the faulty behaviour of an agent never affects the receipt of the message. Alternatively, a system may be deemed fault tolerant if the faulty behaviour of an agent initially prevents the receipt of the message, but at a later time the system corrects itself and the message is received. Thus, the *fault tolerance specification patterns* we define here represent the direct and indirect effects of injected faults.

For the purposes of defining these specifications, Figure 1 highlights some of the computational paths we may wish to analyse in the reachable state space of the system (contained within the grey circle). These include paths along which no faults are injected, paths along which faults are injected and combinations of both. The figure shows properties which are required or not required to hold along computational paths according to the fault tolerance specification pattern we now define. These specifications can be used to extend any logical formula φ, to reason about fault tolerance.

Consider the formula:

$$AG\varphi \tag{3.1}$$

This states that it is always true that φ. In IS^{F*}, this formula holds if the property of the system is *unaffected* by the faulty behaviour. In this case the property would hold along the paths in Figure 1.A. If the formula holds the system would be completely tolerant to the fault.

The following formula can be used to verify that φ always holds when the system behaves correctly:

$$AG(\neg fault \rightarrow \varphi) \tag{3.2}$$

This states that φ always holds when a fault is never injected into the model, such as along the path in Figure 1.B. This formula can be used to reason about only the *correct* behaviour of the mutated system.

To reason about how the faulty behaviour affects the system consider the following formula:

$$AG(\neg injected \rightarrow \varphi) \tag{3.3}$$

This stipulates that φ always holds when the fault is (at the current tick) not injected into the system, such as along the paths in Figure 1.C. This is useful for deciding whether the system is *unaffected* by the injected fault. In this case, the specification can be used to determine whether the property is tolerant to the faulty behaviour when a fault has been injected before its occurrence.

To reason about property φ when no fault has been injected prior to its occurrence consider the following formula:

$$\neg E(\neg injected\ U\ (injected \wedge \neg AG\varphi)) \tag{3.4}$$

This formula states that there is no path in which at some point a fault is injected and at which point it is not true that φ always holds forever in the future, which is demonstrated along the paths in Figure 1.D. This formula is useful when deciding whether a property of the system is *unaffected* by the injected fault and tolerant to it *once that property of the system holds*.

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

$$AG(injected \rightarrow EF\varphi); \tag{3.5}$$

This formula captures the fact it is always true that whenever there is a fault 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, such as along the paths in Figure 1.E.

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

$$AG(injected \rightarrow AF\varphi); \tag{3.6}$$

This formula captures the fact it is always true that whenever there is a fault injected, along all paths at some point φ holds. Thus, if this specification holds, the system *will* recover from the fault in terms of property φ, such as along the paths in Figure 1.F.

These formulas provide useful insights to explore the extent to which the system is affected by faulty behaviour and its ability to recover from it. These are specifications of interest, but other variations exist. We now turn our attention to applying these techniques to illustrate their usefulness.

4. EXAMPLE

We illustrate the automatic fault injection method that is presented here by applying it to the example of the bit-transmission problem [19]. Specifically, we present a compiler which automatically injects a fault into the model. We then extend the specification of the protocol's correct behaviour to verify fault tolerance using some of the specifications above.

The bit-transmission scenario involves a sender S wishing to transmit the value of a bit to a receiver R via an unreliable communication channel that can drop messages. The channel may fail to deliver a message in either direction between the sender and receiver, or may only be able to deliver messages one way. A protocol is therefore required to facilitate reliable communication between the sender and receiver by re-sending the value of the bit when the message is dropped. A suitable protocol can be defined by enforcing S to repeatedly send the bit to R until an acknowledgement has been received from R. R remains silent and continuously transmits an acknowledgement following receipt of the bit.

4.1 Modelling correct behaviour

In this section we model the standard (i.e., correct) behaviour of the sender-receiver protocol [19]. The communication channel can be represented using the Agent E that models the environment. There are four possible local states for the environment $L_E = \{(.,.), (sendbit,.), (.,sendack), (sendbit, sendack)\}$, where $(.,.)$ represents the state in which the communication channel drops the message in both directions, $(sendbit,.)$ represents the state in which the bit can be sent, $(.,sendack)$ represents the state in which the acknowledgment can be sent and $(sendbit, sendack)$ represents a fully functioning communication channel.

The actions for the environment are defined in a similar manner $Act_E = \{-, S \rightarrow, \leftarrow R, S \rightleftarrows R\}$, where $-$ represents the action in which the communication channel drops the message in both directions, $S \rightarrow$ represents the action in which the environment enables the transmission from S to R, $\leftarrow R$, represents the action in which the environment enables the transmission from R to S and $S \rightleftarrows R$ represents the action in which in which the environment enables the transmission in both directions. The protocol defines a non-deterministic choice of action from any local state of the environment, $P_E(l_E) = Act_E = \{-, S \rightarrow, \leftarrow R, S \rightleftarrows R\}$, for all $l_E \in L_E$. This completes the modelling of the behaviour of the communication channel.

Table 3: Local transition relation for S.

target state	transition condition
$(0, ack)$	$(l_S = 0)$ and $Act_R = sendack$ and $Act_E = (\leftarrow R$ or $S \rightleftarrows R)$
$(1, ack)$	$(l_S = 1)$ and $Act_R = sendack$ and $Act_E = (\leftarrow R$ or $S \rightleftarrows R)$

Table 4: Local transition relation for R.

target state	transition condition
0	$(l_S = 0)$ and $Act_S = sendbit$ and $Act_E = (S \rightarrow$ or $S \rightleftarrows R)$
1	$(l_S = 1)$ and $Act_S = sendbit$ and $Act_E = (S \rightarrow$ or $S \rightleftarrows R)$

The behaviour of the sender S can be modelled using four local states $L_S = \{0, 1, (0, ack), (1, ack)\}$, where local states 0 and 1 indicate that the sender is sending the respective bits and states $(0, ack)$ and $(1, ack)$ are states in which the corresponding bit was sent and the acknowledgement has been received. The corresponding actions are $Act_S = \{sendbit(0), sendbit(1), \lambda\}$, where $sendbit(n)$ indicates that bit n is being sent by the receiver and λ represents the null action. The protocol for the sender is therefore formally defined as $P_S(0) = sendbit(0)$, $P_S(1) = sendbit(1)$, $P_S((0, ack)) = P_S((1, ack)) = \lambda$. The initial state for the sender is 0 or 1 and the local transition relation is given in Table 3.

The behaviour of the receiver R can be modelled using four local states $L_R = \{0, 1, (0, \epsilon), (1, \epsilon)\}$, where states 0 and 1 indicate that the receiver has received the respective bits and states $(0, \epsilon)$,$(1, \epsilon)$ indicate neither bit has been received. The corresponding actions are $Act_R = \{sendack, \lambda\}$, where $sendack$ indicates the acknowledgement is being sent and λ represents the null action. The protocol for the receiver is therefore defined as $P_R(0) = P_S(1) = sendack$, $P_S(0, \epsilon) = P_S(1, \epsilon) = \lambda$. The initial state for the receiver is $(0, \epsilon)$ or $(1, \epsilon)$ and the local transition relation is given in Table 4.

The following atomic propositions are introduced to allow for reasoning about the behaviour of the protocol $AP = \{bit = 0, bit = 1, recack\}$, and the corresponding evaluation function is defined as follows:

$V(bit = 0) = \{g \in G \mid \text{either } l_s(g) = 0 \text{ or } l_s(g) = (0, ack)\}$
$V(bit = 1) = \{g \in G \mid \text{either } l_s(g) = 1 \text{ or } l_s(g) = (1, ack)\}$
$V(recack) = \{g \in G \mid \text{either } l_s(g) = (0, ack) \text{ or } l_s(g) = (1, ack)\}$

We have thus described an interpreted system which models the correct behaviour of the sender-receiver protocol. We denote this system as IS_{BTP}.

4.2 A compiler for automatic fault injection

It is straightforward to implement the bit-transmission protocol in ISPL for it to be checked by MCMAS. To facilitate automatic fault injection, we developed a generic compiler to inject automatically a fault into any ISPL program. The compiler is based on the generic MAS model extension via fault injection that we defined in section 3.2. The injection program is given the name of the ISPL file, the name of the agent we wish to inject the fault on ($FaultyAgent$), the variable we wish to inject the fault on ($FaultyVariable$) and the type of fault that needs to be injected ($FaultType$). The

type of fault can presently be of type *inversion*, *random* or *stuck at*. The compiler automatically mutates the code of the ISPL program to create a mutated model for which extended specifications can be used to verify fault tolerance.

We illustrate how the compiler works by injecting a single fault on IS_{BTP}. The compiler mutates the ISPL code of IS_{BTP} to create ISPL code defining ${IS^{F*}}_{BTP}$. We instruct the compiler to inject an inversion fault on the *rec* variable of the receiver agent in IS_{BTP}, so that the receiver erroneously acknowledges that it has received a bit without one having been sent. The input for the compiler includes the ISPL filename for IS_{BTP}, $FaultyAgent$ is $Receiver$, $FaultyVariable$ is rec and $FaultType$ is $inversion$. The correct behaviour of the Receiver is defined in ISPL as follows:

```
Agent Receiver
 Vars:
  rbit : {r0, r1};  rec : boolean;
 end Vars

 Actions = {nothing,sendack};

 Protocol:
  rec=false : {nothing};
  rbit=r0 and rec=true: {sendack};
  rbit=r1 and rec=true: {sendack};
 end Protocol

 Evolution:
  (rec = true and rbit=r0) if ((rec=false)  and
   (Sender.Action=sb0) and ((Env.Action=SR)
   or (Env.Action=S)));
  (rec = true and rbit=r1) if ((rec=false) and
   (Sender.Action=sb1) and ((Env.Action=SR)
   or (Env.Action=S)));
 end Evolution
end Agent
```

Thus, $rbit = r0$ and $rec = false$ correspond to state $(0, \epsilon)$ in R and $rbit = r0$ and $rec = true$ corresponds to state 0 in R. The same correspondence between variable and states applies when $rbit = r1$.

The first task the compiler performs is inserting the fault injection agent FI into the code as follows:

```
Agent Receiver_FI_rec
 Vars:
  inject: boolean;  injected: boolean;
 end Vars

 Actions = {dont_inject,inject_fault};

 Protocol:
  inject=true : {dont_inject,inject_fault};
  inject=false: {dont_inject};
 end Protocol

 Evolution:
  injected=true if
    (Receiver_FI_rec.Action=inject_fault);
  injected=false if
    (Receiver_FI_rec.Action=dont_inject);
 end Evolution
end Agent
```

where $inject = false$ and $inject = true$ corresponds to $not\_faulty/faulty$ in the state definitions of FI. Similarly $injected = false$ and $injected = true$ corresponds to $not\_injected/injected$ in the state definitions of FI. The corresponding naming convention of the fault injection agent is $FaultyAgent$_FI_$FaultyVariable$.

The next task of the compiler is mutating the code of the receiver agent, by altering t_R to contain the desired faulty behaviour in $t_{R^{F*}}$, the transition conditions of which are shown in in Table 5.

Table 5: Mutated transition relation for R.

target state	transition condition
0	$(l_S = 0)$ and $Act_S = sendbit$ and $Act_E =$ $(S \rightarrow$ or $S \rightleftarrows R)$ and $Act_{FI} = dont\_inject$
1	$(l_S = 1)$ and $Act_S = sendbit$ and $Act_E =$ $(S \rightarrow$ or $S \rightleftarrows R)$ and $Act_{FI} = dont\_inject$
0	$(Receiver.state = (0, \epsilon))$ and $Act_{FI} = inject$
1	$(Receiver.state = (1, \epsilon))$ and $Act_{FI} = inject$
$(0, \epsilon)$	$(Receiver.state = (0)$ and $Act_{FI} = inject$
$(1, \epsilon)$	$(Receiver.state = (1)$ and $Act_{FI} = inject$

The corresponding code of the mutated receiver agent is defined as follows, where the mutated code is highlighted with a box:

```
Evolution:
 (rec=true and rbit=r0) if ((rec = false)
  and (Sender.Action=sb0) and ((Env.Action=SR)
  or (Env.Action=S))
  and (Receiver_FI_rec.Action=dont_inject));
 (rec=true and rbit=r1) if ((rec=false) and
  (Sender.Action=sb1)  and ((Env.Action=SR)
  or (Env.Action=S))
  and (Receiver_FI_rec.Action=dont_inject));
 (rec=true) if (rec=false)
  and (Receiver_FI_rec.Action=inject_fault);
 (rec= false) if (rec= true)
  and (Receiver_FI_rec.Action=inject_fault);
end Evolution
```

The final role of the compiler is to mutating the definitions of V and I to V^{F*} and I^{F*} in the code as follows, where the mutated code is highlighted with a box:

```
Evaluation
 recack if (Sender.ack=true );
 bit0 if (Sender.bit=b0);
 bit1 if (Sender.bit=b1);
 fault if (Receiver_FI_rec.inject=true);
 injected if (Receiver_FI_rec.injected=true);
end Evaluation

InitStates
 ((Sender.bit=b0) or (Sender.bit=b1)) and
  (Receiver.rec=false) and ( Sender.ack=false) and
 (Env.state=none)
 and ((Receiver_FI_rec.inject=true)
 or (Receiver_FI_rec.inject=false));
end InitStates
```

The compiler therefore performs the function of parsing the ISPL code of IS, inserting the fault injection agent into it, adding the mutated transition conditions to $FaultyAgent$ for $FaultyVariable$, and updating the evaluation and initial states to create ISPL code defining IS^{F*}.

4.3 Verifying fault tolerance

Now that we have a mutated model ${IS^{F*}}_{BTP}$, we wish to reason about its correct and faulty behaviours to verify fault tolerance by modifying formulas that verified correct behaviour in IS_{BTP}. For IS_{BTP} we consider the following temporal-epistemic specification to verify the correct behaviour of the protocol.

$AG(recack \rightarrow K_S(K_R(bit = 0) \vee K_R(bit = 1)))$;

This formula states that when the sender receives an acknowledgement, the sender knows that the receiver knows the value of the bit. In IS_{BTP}, this specification is verified by MCMAS as true. In the mutated ${IS^{F*}}_{BTP}$ model,

this specification corresponds to the specification pattern of Formula 3.1. For ${IS^{F*}}_{BTP}$, MCMAS returns the specification as false, since the injected inversion fault can cause the receiver to send the acknowledgement before the value of the bit has been received. We conclude that the protocol is intolerant to this specific injected fault.

What is significant is that we are now in the position of checking on the new program a variety of other specifications. For instance we may want to check that when in ${IS^{F*}}_{BTP}$ no faults are actually injected then the original property is still true. To do so we consider the following extended specification corresponding to the specification pattern of Formula 3.2.

$$AG(\neg fault \rightarrow (recack \rightarrow K_S(K_R(bit = 0) \vee K_R(bit = 1))));$$

This formula states that when no faults are ever injected into the model and the sender receives an acknowledgement, the sender knows that the receiver knows the value of the bit. MCMAS verifies the specification as true since the protocol is acting under correct behavioural conditions.

To reason about the influence of injected faults on the specification we now consider the following formula corresponding to the specification pattern of Formula 3.3.

$$AG(\neg injected \rightarrow (recack \rightarrow K_S(K_R(bit = 0) \vee K_R(bit = 1))));$$

This formula states that when there is no fault injected into the model and the sender receives an acknowledgement, the sender knows that the receiver knows the value of the bit. MCMAS returns the formula as false, since it is possible that a fault has been previously injected into the model that caused the receiver to send the acknowledgement before the value of the bit has been received. The protocol is therefore indirectly affected by the the injected fault with regards to the original property we are reasoning about, and intolerant to the fault when the fault has been previously injected.

We now consider a specification corresponding to the specification pattern of Formula 3.4 where there is no fault injected until a fault is injected and at which point the original specification holds.

$$\neg E(\neg injected\ U\ (injected \wedge \neg AG(recack \rightarrow K_S(K_R(bit = 0) \vee K_R(bit = 1)))));$$

This formula states that there is no path in which at some point a fault is injected and at which point it is not true that when the sender receives an acknowledgement, then the sender knows that the receiver knows the value of the bit. MCMAS verifies the specification as false, which means that the original property is intolerant to the fault if it is injected when the original property holds. This is because *recack* may be false for the original property to hold, thus the injected fault can still affect the acknowledgment when the original property holds.

We now consider a specification corresponding to the specification pattern of Formula 3.5 to reason about the recoverability of the protocol in relation to the original property.

$$AG(injected \rightarrow EF(recack \rightarrow K_S(K_R(bit = 0) \vee K_R(bit = 1))));$$

This formula states that when there is fault injected into the model it is possible that when the sender receives an acknowledgement the sender knows that the receiver knows the value of the bit. MCMAS returns this specification as false which means that in relation to the original property there is no possibility that it is able to recover from the injected fault. This is what we would expect because once an injected fault has caused an acknowledgement to be sent, another one will not follow. The stronger specification corresponding to the specification pattern of Formula 3.6 also returns false since the protocol is not always able to recover.

By reasoning about the correct and faulty behaviours of the mutated sender-receiver protocol we have shown that the original property is intolerant to the injected fault generally, indirectly, and when the original property holds. We have also shown that in relation to the original property the protocol is unable to recover from the fault and thus intolerant to the fault in terms of recovery.

5. RELATED WORK

Previous work on combining fault injection [1, 2, 3, 14] and model checking is limited to model checkers that use temporal logic to reason about properties of the system. Formalisms used are the language of the popular model checker NuSMV [2], process algebras such as CCS/Meije [1, 3], and the commercial SCADE tool by Esterel Technologies coupled with the SCADE Design Verifier model checker [14]. Our classification of failure modes is based upon the commonly defined faults in [2, 14].

In [2] a 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.

In [14] faults are injected into SCADE for a model of a wheel brake system. The fault injection is not automatic. 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.

In [3] 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 [1], 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 although feasible, the work is not extended to provide a practically usable tool.

6. CONCLUSION

In this paper we developed a methodology for automatically injecting common faults into a MAS program to generate a mutated system exhibiting faulty behaviour. We used temporal and epistemic logic to reason about the correct and faulty behaviours of the mutated system in order to verify fault tolerance. To test the practical application of our methods we developed a compiler to inject faults automatically, and used it to analyse the degraded performance in the bit-transmission problem. In this way we were able to verify automatically issues pertaining to fault tolerance

and recoverability.

We regard the presented results as interesting for addressing the challenging problem of how to verify complex fault tolerant agent based systems. In particular, the methods discussed here are either automatic or can be extended to be automatic. This is crucial to usability for those who are not experts in verification.

In future work we will proceed with the goal in mind of creating a powerful automatic tool to verify fault tolerance of a MAS. Inspired by the formulas we discussed in this article, we wish to investigate a taxonomy of specifications modelling various aspects of faults and recovery. The compiler should also allow for automatic injection of multiple faults into the system and the fault injection agent needs to be extended to vary the persistence of faults. Finally, we further aim to determine how to inject faults on the protocols of the MAS as well as on the states.

Acknowledgements.

The research described in this paper is partly supported by EPSRC funded project EP/E02727X/1.

7. REFERENCES

[1] C. Bernardeschi, A. Fantechi, and S. Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability*, 12(4):251–275, 2002.

[2] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA safety analysis platform. *Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[3] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *Proceedings of AMAST'97*, volume 1349 of *LNCS*, pages 45–59. Springer, 1997.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] A. Cimatti. Industrial applications of model checking. In *Proceedings of MOVEP'00*, volume 2067 of *LNCS*, pages 153–168. Springer, 2001.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 495–499. Springer, 1999.

[7] E. Clarke, O.Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

[8] R. Fagin, J. Y. Halpern, M. Y. Vardi, and Y. Moses. *Reasoning about knowledge*. MIT Press, Cambridge, 1995.

[9] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *Proceedings of AAMAS'02*, pages 737–744. ACM press, 2002.

[10] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 479–483. Springer, 2004.

[11] Z. Guessoum, J. P. Briot, S. Charpentier, O. Marin, and P. Sens. A fault-tolerant multi-agent framework. In *Proceedings of AAMAS'02*, pages 672–673. ACM Press, 2002.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Enginnering*, 23(5):279–295, 1997.

[13] R. Iyer. Experimental evaluation. In *Proceedings of FTCS-25*, pages 115–132. IEEE, 1995.

[14] A. Joshi and M. P. E. Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *Proceedings of SAFECOMP'05*, volume 3688 of *LNCS*, pages 122–135. Springer, 2005.

[15] A. Lomuscio, H. Qu, M. Sergot, and M. Solanki. Verifying temporal epistemic properties of web service compositions. In *Proceedings of ICSOC'07.*, volume 4749 of *LNCS*, pages 456–461. Springer, 2007.

[16] A. Lomuscio, H. Qu, and M. Solanki. Towards verifying compliance in agent-based web service composition. In *Proceedings of AAMAS'08*, pages 265–272. IFAAMAS, 2008.

[17] A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In *Proceedings of TACAS'06*, volume 3920 of *LNCS*, pages 450–454. Springer, 2006.

[18] A. Lomuscio and M. J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[19] A. Lomuscio and M. J. Sergot. A formalisation of violation, error recovery, and enforcement in the bit transmission problem. *Journal of Applied Logic*, 2(1):93–116, 2004.

[20] A. Niewiadomski, W. Penczek, and M. Szreter. Verics 2004: A model checker for real time and multi-agent systems. In *Proceedings of CS&P'04*, Informatik-Berichte, pages 88–99, 2004.

[21] A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS'77*, pages 46–57. IEEE, 1977.

[22] M. J. Wooldridge. *Reasoning about Rational Agents*. MIT Press, Cambridge, 2000.