

An automated approach to verifying diagnosability in multi-agent systems

Jonathan Ezekiel and Alessio Lomuscio

Department of Computing

Imperial College London

London, UK

Email: jezekiel@doc.ic.ac.uk, alessio@doc.ic.ac.uk

Abstract—This paper addresses the issue of guaranteeing the correctness of fault diagnosis mechanisms in multi-agent systems. We propose an automated approach to verifying the property of diagnosability by combining fault injection with model checking. In particular we show how to reason about individual agent’s and system wide knowledge of faults, which is essential for the agents to cooperate and coordinate to recover from them. The multi-agent system model checker MCMAS is used for verification and epistemic specifications are defined to specify that a system accurately diagnoses faults.

Keywords—model checking; fault tolerance; fault injection; epistemic logic;

I. INTRODUCTION

In recent years considerable interest has been shown towards the use of multi-agent systems (MAS) [18] as a paradigm for software engineering (see e.g., [10]). Most of this interest is due to the high level of complexity required to engineer software architectures in which the core components, or agents, autonomously interact with one another, engaging in communication, negotiation, coordination, etc. Although it has been argued that MAS are a natural way to engineer complex software architectures, confidence in the robustness of MAS is a practical concern when considering whether to adopt it as a software engineering paradigm.

Central to the issue of robustness is ensuring fault tolerance in MAS. One way in which fault tolerance can be improved is by employing MAS architectures in which agents are able to *diagnose* faults so that they can communicate and co-ordinate to recover from them [12], [15], [16]. The general problem of fault diagnosis has received considerable attention since the late 80s (see e.g., [6]). Further to this, studies have been conducted on the property of *diagnosability*, i.e., establishing whether a fault can be correctly detected from the observable events of the system [17]. For systems in which accurate fault diagnosis is critical, automated verification techniques such as *model checking* [5] have been used to verify diagnosability [4]. However, in that approach *distributed* diagnosability is not considered, and the faulty behaviour of the system is modelled by hand.

In this paper we propose an automated approach to verifying diagnosability in MAS. The analysis of faulty behaviour is automated using *fault injection*, which has recently been combined with model checking to verify fault

tolerance in MAS [8]. In contrast to ad-hoc analysis, in this approach faults can be automatically injected into a model of a correctly behaving MAS in order to mutate it into one that exhibits possible faulty behaviour. Once a mutated model is available, diagnosability can be verified in it by using the MAS based model checker MCMAS [13]. This enables us to verify the correct and faulty behaviour of agents, as well as their *knowledge* about the behaviour in a temporal-epistemic logic setting [9]. Specifically, we define temporal-epistemic formulas to reason about the *knowledge of faults* that have been injected into the system. These specifications allow us to verify distributed diagnosability, and appear more intuitive than purely temporal diagnosability specifications [4].

To model realistic faulty behaviour, we define complex faults with varying persistence and develop a graphical tool to inject them automatically. We highlight the usefulness of our approach by using it to verify both individual agent and system wide diagnosability in a model we constructed of the IEEE 802.5 token ring LAN protocol, which utilises distributed diagnosis to achieve fault tolerance.

The rest of the paper is structured as follows. In Section II we provide the background on model checking, interpreted systems, MCMAS, and fault injection into MAS. In Section III we extend the fault injection approach presented in Section II by defining new faults with varying persistence, and introducing a fully functional compiler for injecting faults into a MAS program. In Section IV we define temporal-epistemic specifications that can be used for verifying diagnosability. In Section V we describe a version of the token ring protocol and its implementation in the MCMAS input language. In Section VI we show how our approach is applied to verify diagnosability in the token ring protocol. In Section VII we discuss the related work and in Section VIII we conclude and put forward future work.

II. BACKGROUND

Model checking [5] 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, $M \models \varphi_P$). The satisfaction relation is implemented as a decision procedure, whose *automatic* nature makes model checking attractive for the purpose of verification [5].

In the case of MAS φ_P is often expressed by using a number of rich modal logics including temporal, ATL, and epistemic logics [18]. Particularly relevant to diagnosability is temporal-epistemic logic, which can be used to reason about the *knowledge* of the agents over time.

A. Interpreted systems and MCMAS

Interpreted systems [9] are a popular semantics for temporal-epistemic logic. We summarise the framework of interpreted systems in [9] to model MAS. Each agent $i \in \{1, \dots, n\}$ 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 $g = (l_1, \dots, l_n, l_E) \in L_1 \times \dots \times L_n \times L_E$ 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 function $t_i : L_i \times L_E \times Act_1 \times \dots \times Act_n \times \dots \times Act_E \rightarrow L_i$, 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 $t_E : L_E \times Act_1 \times \dots \times Act_n \times \dots \times Act_E \rightarrow L_E$. 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 $S \subseteq L_1 \times \dots \times L_n \times L_E$, and $Act \subseteq Act_1 \times \dots \times Act_n \times Act_E$. 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:

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

The syntactical constructs and the semantic model that are presented in [13] are adopted for the interpretation of temporal-epistemic formulae in interpreted systems. Specifically, we consider the following syntax defining our specification language:

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

In the grammar above $p \in AP$ is an atomic proposition; EX 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* , $E_\Gamma\varphi$ expresses that *everybody in group Γ knows φ* , $C_\Gamma\varphi$ expresses 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* [9].

Any interpreted system is associated with a model $M_{IS} = (W, R_t, \sim_1, \dots, \sim_n, L)$ 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 [9]). The labelling relation $L \subseteq AP \times W$ can easily be defined in terms of the valuation relation V .

Formulae can be interpreted in M_{IS} in a standard way [5], [9], [14] as follows. Let $\pi = (w_0, w_1, \dots)$ 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 $(M, w) \models \varphi$ 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.

$$\begin{aligned} (M, w) \models p & \quad \text{iff } (p, w) \in L, \\ (M, w) \models \neg\varphi & \quad \text{iff } M \not\models \varphi, \\ (M, w) \models \varphi_1 \vee \varphi_2 & \quad \text{iff either } M \models \varphi_1 \text{ or } M \models \varphi_2, \\ (M, w) \models EX\varphi & \quad \text{iff there exists a path } \pi \text{ such that} \\ & \quad \pi(0) = w, \text{ and } (M, \pi(1)) \models \varphi, \\ (M, w) \models AG\varphi & \quad \text{iff for all paths we have that} \\ & \quad \pi(0) = w, \text{ and } (M, \pi(i)) \models \varphi, \\ & \quad \text{for all } i \geq 0, \\ (M, w) \models E(\varphi U \psi) & \quad \text{iff there exists a path } \pi \text{ such that} \\ & \quad \pi(0) = w, \text{ and there exists } k \geq 0 \\ & \quad \text{and } (M, \pi(j)) \models \varphi \text{ such that} \\ & \quad (M, \pi(k)) \models \psi \text{ and } (M, \pi(j)) \models \varphi \\ & \quad \text{for all } 0 \leq j < k, \\ (M, w) \models K_i\varphi & \quad \text{iff for all } w' \in W, w \sim_i w' \text{ implies} \\ & \quad (M, w') \models \varphi, \\ (M, w) \models E_\Gamma\varphi & \quad \text{iff for all } w' \in W R_\Gamma^E(w, w') \text{ implies} \\ & \quad (M, w') \models \varphi, \end{aligned}$$

$(M, w) \models C_\Gamma \varphi$ 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
 $(M, w') \models \varphi$.

In the definition above, the relation R_Γ^E is defined as the union of the epistemic relations for the agents in Γ : $R_\Gamma^E = \bigcup_{i \in \Gamma} \sim_i$; the relation R_Γ^D is defined as the intersection of the epistemic relations for the agents in Γ : $R_\Gamma^D = \bigcap_{i \in \Gamma} \sim_i$; the relation R_Γ^C is the transitive closure of R_Γ^E .

We say that a formula φ is true in the model and we write $M \models \varphi$ if $(M, w) \models \varphi$ for all $w \in W$. Similarly to [9], we say that a formula φ is true in an interpreted system IS , and we write $IS \models \varphi$, if $M \models \varphi$. A formula is true in an interpreted system if it is true in the associated Kripke model.

MCMAS [13] provides ISPL 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 an interpreted system. MCMAS supports the verification for all formulas in the language above.

B. Fault injection into MAS programs

Model checking has traditionally been applied to provide assurances about the *correct* behaviour of the system in terms of temporal specifications. However, injecting faulty behaviour allows the analysis of both the *correct and faulty* behaviours of systems by means of model checking [1]–[3], [8], [11].

To inject faulty behaviour into an interpreted system, a general way to mutate any agent A into a faulty agent A^{F*} was defined in [8]. In this approach a fault injection agent (FI) determines the conditions under which faulty behaviour occurs in the faulty agent. 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 the *notinject* action is performed by FI .

The occurrence of the *notinject* and *inject* actions are handled by FI 's protocol $P_{FI}(nofault) = \{notinject\}$, $P_{FI}(fault) = \{notinject, inject\}$. The local states of the fault injection agent $L_{FI} = \{nofault, fault\}$ indicate whether a fault is *ever* injected during a run of the system. This is set as an initial state of the system and persists during the run. In other words, faults are injected *randomly* into the faulty agent whenever FI is in a faulty state.

Atomic propositions can be used to reason about the correct and faulty behaviours of the mutated interpreted system IS^{F*} where the mutated set of atomic propositions $AP^{F*} = AP \cup \{faulty, injected\}$. The proposition *faulty* is

Table I
TRANSITION RELATION MUTATION RULES FOR A^{F*} .

Trans	Fault	Target State	Transition Condition
t_A	N/A	$state = x$	$*_{tc}$
$t_{A^{F*}}$	any	$state = x$	$*_{tc} \ \& \ Act_{FI} = notinject$
$t_{A^{F*}}$	random	$state = x \ \ !x$	$*_{tc} \ \& \ Act_{FI} = inject$
$t_{A^{F*}}$	invert	$state = !x$	$*_{tc} \ \& \ Act_{FI} = inject$
$t_{A^{F*}}$	stuck at	$state = state$	$*_{tc} \ \& \ Act_{FI} = inject$

used (together with temporal-epistemic formulas) to reason about system runs where faults are either never injected or randomly injected. Similarly the proposition *injected* is used to reason about system runs where faults are injected at the current tick of the clock.

The evaluation function V is modified to V^{F*} so that $V^{F*}(faulty) = \{g \in G \mid l_{A^{F*}}(g) = faulty\}$ and $V^{F*}(injected)$ whenever the evolution into the current state of the faulty agent required the action *inject* of FI to occur. The set of initial states I is updated so that the initial state of the fault injection agent FI is either *nofault* or *fault* in I^{F*} . The extended faulty system is defined as follows:

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

The evolution function t_A is mutated to contain the desired faulty behaviour in $t_{A^{F*}}$. More formally, Table I gives the precise *mutation rules* for different potential faults. In the table *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 $*_{tc}$ indicates the original transition condition.

The definition of the fault injection agent and the mutation rules for the faulty agent presented above as reported in [8] are limited in that:

- during a system run it is only possible to reason about paths in which faults are either never injected or randomly injected,
- faults can only be injected on states which contain boolean variables.

However, for diagnosability it is necessary to reason about faults which can be identified by a diagnosis mechanism. This implies defining faults with both random and constant persistence, that can begin and stop at certain points of a system run. To inject faults into an interpreted system that models a realistic MAS in which fault diagnosis is present, we need to reason about faults affecting local variables.

III. TOWARDS VERIFYING DIAGNOSABILITY

In the previous section we described the limitations of our existing approach to fault injection which makes it currently unsuitable for reasoning about diagnosability. In this section

Table II
DEFINITION OF LOCAL STATES AND ACTIONS OF FI .

options	L_{FI}	Act_{FI}
default	$\{nofault, fault_i, fault_ni\}$	$\{notinject, inject\}$
$rstt$	$\cup \{w_rstt\}$	$\cup \{start\}$
$astt$	$\cup \{w_astt\}$	
$asto$	$\cup \{stop_ni\}$	
$asto, !rsto$	$\cup \{stop_i\}$	
$rsto$	$\cup \{stop_ni\}$	$\cup \{stop\}$
$rsto, asto$	$\cup \{asto_i\}$	
$rsto, asto, rnd$	$\cup \{asto_ni\}$	

we surpass these limitations by defining extensions to the persistence of the fault injection agent and introduce new transition relation mutation rules on enumerate variables for the faulty agent. We also show how these are incorporated into a fully functional compiler for fault injection.

A. Persistence extensions for the fault injection agent

To allow for varying fault persistence we define a *constant* fault injection agent 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 injection after and before a *start action occurs* ($astt$) and a *stop action occurs* ($asto$), which is an action of any agent. Any of these options can be combined defining different types of faults and determining the local states of the fault injection agent.

Table II 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 ($fault_i, ni$), waiting for a start condition ($w_astt, rstt$), has injected or not injected a fault at the current tick of the clock after a

Table III
DEFINITION OF PROTOCOL OF FI .

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

Table IV
DEFINITION OF TRANSITION RELATION OF FI .

options	Target State	Transition Condition
$astt$	$fault_ni$	$w_astt \& STT$
$astt, rstt$	w_astt	$w_rstt \& Act_{FI} = start$
$!astt, rstt$	$fault_ni$	$w_rstt \& Act_{FI} = start$
$!asto$	$fault_i$	$(fault_i \parallel fault_ni) \& Act_{FI} = inject$
$!asto, rnd$	$fault_ni$	$(fault_i \parallel fault_ni) \& Act_{FI} = notinject$
$asto$	$fault_i$	$(fault_i \parallel fault_ni) \& ACT_{FI} = inject \& !STO$
$asto, rnd$	$fault_ni$	$(fault_i \parallel fault_ni) \& ACT_{FI} = notinject \& !STO$
$asto, !rsto$	$stop_i$	$((fault_i \parallel fault_ni) \& STO) \& Act_{FI} = inject$
	$stop_ni$	$stop_i$
$asto, !rsto, rnd$	$stop_ni$	$((fault_i \parallel fault_ni) \& STO) \& Act_{FI} = notinject$
$asto, rsto, rnd$	$asto_i$	$((fault_i \parallel fault_ni) \& STO) \parallel (asto_i \parallel asto_ni) \& Act_{FI} = inject$
	$asto_ni$	$((fault_i \parallel fault_ni) \& STO) \parallel (asto_i \parallel asto_ni) \& Act_{FI} = notinject$
$asto, rsto, !rnd$	$asto_i$	$(fault_i \parallel fault_ni) \& STO$
$rsto$	$stop_ni$	$Act_{FI} = stop$

Table V
DEFINITION OF INITIAL STATES OF FI .

options	I^{FI*}
default	$\{nofault\}$
$rstt$	$\cup \{w_rstt\}$
$astt, !rstt$	$\cup \{w_astt\}$
$!rstt, !astt$	$\cup \{fault_ni\}$

stop action ($asto_i, ni$), has injected or not injected a fault at the current tick of the clock and will not inject faults in any future tick of the clock ($stop_i, ni$).

The definition of the fault injection agent protocol P_{FI} is given in Table III. For the default fault injection agent we have that $P_{FI}(nofault) = \{notinject\}$, $P_{FI}(fault_i) = \{inject\}$. A set of actions Act_P associated with some of the states is defined dynamically.

The transition relation for the fault injection agent is defined in Table IV, which shows the target state and the transition condition as defined by the options. $STT \subseteq Act_1 \times \dots \times Act_n \times Act_E$ is a set of start actions. $STO \subseteq Act_1 \times \dots \times Act_n \times Act_E$ is a set of stop actions.

The initial states of the fault injection agent are defined in Table V. The initial state is either $nofault$, which persists indefinitely, or a state determined by the selected options.

Finally, we introduce some atomic propositions to reason about faults. Initially we have that the mutated set of atomic propositions $AP^{F*} = AP \cup \{faulty\}$ where $faulty$ is defined to reason about whether faults are ever injected into the

Table VI
DEFINITIONS OF STATES FOR UPDATING THE EVALUATION FUNCTION.

options	IND	options	NIJ
default	$\{fault_i\}$	default	\emptyset
$asto, !rsto$	$\cup \{stop_i\}$	$astt$	$\cup \{w_astt\}$
$asto, rsto$	$\cup \{asto_i\}$	$rstt$	$\cup \{w_rstt\}$
		$asto$	$\cup \{stop_ni\}$
		$rsto$	$\cup \{stop_ni\}$

Table VII
NEW MUTATION RULES FOR A^{F*} .

Variable value replace		
Trans	Target State	Transition Condition
t_A	$var = v_1 \ \& \ *_{ts}$	$*_{tc}$
t_{AF*}	$var = v_1 \ \& \ *_{ts}$	$*_{tc} \ \& \ !(ACT_{FI} = inject)$
t_{AF*}	$var = v_2 \ \& \ *_{ts}$	$*_{tc} \ \& \ ACT_{FI} = inject$
Stuck at select		
Trans	Target State	Transition Condition
t_A	$var = v_x \ \& \ *_{ts}$	$var = v_1 \ \& \ *_{tc}$
t_{AF*}	$var = v_x \ \& \ *_{ts}$	$var = v_1 \ \& \ *_{tc} \ \& \ !(ACT_{FI} = inject)$
t_{AF*}	$var = v_1 \ \& \ *_{ts}$	$var = v_1 \ \& \ *_{tc} \ \& \ ACT_{FI} = inject$
Variable transition		
Trans	Target State	Transition Condition
t_A	$*_{ts}$	$*_{tc}$
t_{AF*}	$*_{ts}$	$*_{tc} \ \& \ !(ACT_{FI} = inject)$
t_{AF*}	$var = v_1$	$ACT_{FI} = inject$

system. The corresponding evaluation function V is updated so that $V^{F*}(faulty) = \{g \in G \mid l_{AF*}(g) \neq nofault\}$. Similarly we have that $AP^{F*} = AP^{F*} \cup \{injected\}$. The corresponding evaluation function is updated according to IND a set of states in which faults are injected into the system at the current tick of the clock as defined in Table VI, so that $V^{F*}(injected) = \{g \in G \mid l_{AF*}(g) \in IND\}$. For random faults if any start or stop option is selected we have that $AP^{F*} = AP^{F*} \cup \{injecting\}$. The corresponding evaluation function is updated according to NIJ a set of states in which faults cannot be injected into the system at the current tick of the clock as defined in Table VI, so that $V^{F*}(injecting) = \{g \in G \mid l_{AF*}(g) \notin NIJ\}$. If any stop option is selected we have $AP^{F*} = AP^{F*} \cup \{stopped\}$ and $V^{F*}(stopped) = \{g \in G \mid l_{AF*}(g) = stop_ni\}$.

B. Transitions for the faulty agent

In this subsection we define new mutation rules for the faulty agent that denote the mutated transition relation for states that include *enumerate* variables. The rules are shown in Table VII. 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. In the cases where 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. Note that we use $!(ACT_{FI} = inject)$ to define when a fault is injected rather than $(ACT_{FI} = notinject)$ since the action *start* and *stop* do not cause a fault to be injected.

A *variable value replace* fault defines that the value of an enumerate variable *var* is updated with a value v_2 in t_{AF*} whenever the value of *var* is updated to v_1 in t_A . This fault is useful for defining faulty conditions where some of the correct agent behaviour is skipped.

A *stuck at select* fault defines that the value v_1 of a 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 the faulty behaviour in t_{AF*} preserves $var = v_1$. This allows other values of the variable to change when the agent evolves and $var \neq v_1$. A complete stuck at select rule would consider cases where *var* is absent from the transition condition and $var = v_x$ is in the target state.

A *variable transition* fault defines that a variable *var* is set to the value v_1 in t_{AF*} 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 $ACT_{FI} = inject$. The fault is useful for defining scenarios in which an agent immediately evolves to a state as a result of a fault.

C. A fully functional compiler for fault injection

In the previous two sections we introduced complex faults based upon combining several options and mutation rules. To allow for these faults to be injected into an ISPL program we developed a fully functional compiler to manage these faults during the injection process. The compiler is available for public use. [7]

The toolkit takes an ISPL program as input and provides a GUI which allows the user to inject faults into the program and output a mutated program. 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 the persistence options previously introduced.

The ISPL code is mutated by applying the mutation rules 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_{AF*} . The mutation is performed using string find and string remove functions. The pseudo-code below illustrates how the transition relation t_A of an agent A is mutated for a variable value. In the pseudo-code LHS returns a string containing the target state and RHS return 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 resultant string.

```

For each line L in tA
  If Find(LHS(L), var + "=" + v1)
    tA* += LHS(L) + "if" + RHS(L) + "and"
        + FI + ".Action = inject\n";
    tA* += var + "=" + v2 + "and"
        + Remove(LHS(L), var + "=" + v1)
        + "if" + RHS(L) + "and !"
        + FI + "Action = inject\n";
  else
    tA* += LHS(L) + "if" + RHS(L) + "\n";

```

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

IV. REASONING ABOUT DIAGNOSABILITY

So far we have described the automatic fault injection approach that can be applied to a MAS model IS to produce a mutated model IS^{F*} . In this section we define a number of *diagnosability specification patterns*. These are temporal-epistemic formulas that can be used to reason about the knowledge of injected faults for verifying diagnosability.

Diagnosability is informally 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* [17]. In the context of MAS, we say that the *knowledge a fault is* the property that determines whether a fault is diagnosable. As we show below we can formally verify this high-level expressive property.

For our specification patterns we define a diagnosis property of the system as Δ and express the persistence of an injected fault as Θ which we define as:

$$\Theta ::= \Theta \vee \Theta | \Theta \wedge \Theta | \text{faulty} | \text{injecting} | \text{stopped} | \text{injected}$$

where *faulty*, *injecting*, *stopped*, and *injected* are atomic propositions that relate to the faults persistence.

Consider the following formula:

$$AG(\Delta \rightarrow K_i(\Theta)) \quad (1)$$

This formula states that whenever a diagnosis property of the system Δ occurs, agent i knows that Θ . This 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.

In the case where a diagnosis is made for different types of faults using the same mechanism, consider the following formula:

$$AG((\Theta_1 \wedge \Theta_2 \wedge \Delta) \rightarrow (K_i(\Theta_1 \vee \Theta_2) \wedge \neg K_i(\Theta_1) \wedge \neg K_i(\Theta_2))) \quad (2)$$

This formula states that whenever Θ_1 and Θ_2 and a diagnosis property of the system Δ occurs, agent i knows that the either Θ_1 or Θ_2 but does not know specifically whether Θ_1

or Θ_2 . 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.

The previous two formulas refer to the knowledge of faults with respect to an observable diagnosis property of the system. We now consider knowledge of faults in relation to the occurrence of faults, which is not necessarily an observable property of the system. The following formula can be used to reason about the diagnosability of a fault after the first time the fault has occurred without explicitly referencing a diagnosis property of the system.

$$\neg E(\neg \Theta U (\Theta \wedge \neg AF(K_i(\Theta)))) \quad (3)$$

This formula states that there is no path in which at some point Θ becomes true and at which point it is not true that at some point in the future agent i knows Θ . The formula specifies the ability of agent i to diagnose faults correctly in relation to the first occurrence of a fault. This specification is suitable for situations in which fault diagnosis leads to resolution of the fault that implies no further diagnosis of the fault is necessary. We leave situations in which diagnosis of repeating faults is required for future work.

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:

$$\neg E(\neg \Theta U \Theta \wedge \neg AF(D_\Gamma(\Theta))) \quad (4)$$

This formula states that there is no path in which at some point Θ and a which point it is not true that at some point in the future it is distributed knowledge in group Γ that Θ . The formula specifies the ability of a group of agents to diagnose faults correctly. Similar specifications can also be defined for reasoning about whether everybody knows about faults and common knowledge of faults.

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

$$\neg E(\neg K_i(\Theta) U (K_i(\Theta) \wedge \neg AF(D_\Gamma(\Theta)))) \quad (5)$$

This formula states that there is no path in which at some point agent i comes to know Θ and at which point it is not true that at some point in the future it is distributed knowledge in group Γ that Θ . This formula specifies the propagation of the knowledge of faults to a group of agents. We consider this formula to be the most interesting of or specification patterns. Similar specifications can also be defined determining whether distributed knowledge of the fault leads to everybody knowing about the fault, and whether everybody knowing about the fault leads to common knowledge of the fault.

We consider these specification patterns to be of interest since they provide an insight into the correctness of diagnosability, however, other variations exist. The formulas can be extended where required, for example we can extend Formulas 3, 4, and 5, to reason about the diagnosis of a range of faults in a similar manner to Formula 2.

V. THE IEEE 802.5 TOKEN RING 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. 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 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* which 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 token. 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 when 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 enters into a 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 disconnect from the network.

A. ISPL implementation

We implemented the protocol above to study its resilience under faulty behaviour. In the ISPL implementation the environment agent encodes the hub, abstracts from the timer details of the individual nodes, and manages the token passing between nodes. A node agent represents a node of the network; we implement as many node agents as nodes there are on the ring. Each node agent is named $N[x]$ where $[x]$ is the number of the node. A token agent contains the token data; an associated token bit agent determines whether the token has been inspected.

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}
Amon boolean
Bfail 1...MAXF
Brec 1...MAXR

```

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, *Amon* defines whether the node is an active monitor, and *Bfail* and *Brec* are the number of tokens that have been sent unsuccessfully, and received to indicate a fault on the node.

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 nodes 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 for a token type $[t]$ using an action $send_ [t]$.

Table VIII
INJECTED FAULTS ON THE TOKEN RING PROTOCOL.

Fault	Agent	Fault type	Persistence
$sN2ns$	$N2$	State replace Istatus $v1 = Send$ $v2 = Wait$	$rstt, astt, asto$ $STT = \{N1.send_D\}$ $STO = \{N1.send_C\}$
$hN3ns$	$N3$	State replace Istatus $v1 = Send$ $v2 = Wait$	$rstt, astt$ $STT = \{N2.send_D\}$
$hN4nr$	$N4$	Stuck at select Istatus $v1 = Wait$	$rstt, astt$ $STT = \{N3.send_D\}$
$hN6ns$	$N6$	State replace Istatus $v1 = Send$ $v2 = Wait$	$rstt, astt$ $STT = \{N5.send_D\}$

Table IX
PROPOSITIONS FOR THE TOKEN RING PROTOCOL.

Proposition	Condition
$hard_i$	$hN3ns_i \vee hN4nr_i \vee hN6ns_i$
$soft_i$	$sN2ns_i$
any_i	$hard_i \vee soft_i$
$N3_d$	$N3.Istatus = Disconnect$
$N3_{rd}$	$N3.Bfail = MAXF \vee N3.Brec = MAXR$
$N4_{sb}$	$N3.RStatus = Beaconing \wedge N3.Bfail = 0$
$N1_{am}^{to}$	$N1.Istatus = TimeO \wedge N1.Amon = true$
$N1_{-am}^{to}$	$N1.Istatus = TimeO \wedge N1.Amon = false$
all_f	$hN3ns_f \wedge hN4nr_f \wedge hN6ns_f \wedge sN2ns_f$

VI. VERIFYING THE DIAGNOSABILITY OF THE TOKEN RING PROTOCOL

To verify diagnosability in the token ring protocol we used a model with 6 nodes $N1 \dots N6$. We injected different faults into several node agents in order to determine the ability of the active and standby monitors to diagnose faults. In the model 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.

The injected faults we experimented with are shown in Table VIII 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:

- $sN2ns$: 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).

A *soft* fault is one in which the ring recovers without entering the beaconing process; a *hard* fault prevents tokens from circulating until the faulty node is removed.

For this example we define that the ring does not enter a state where non-faulty nodes become disconnected. To

Table X
SPECIFICATIONS FOR THE TOKEN RING PROTOCOL.

A	$AG(N1_{-am}^{to} \rightarrow K_{N1}(hN6ns_f))$	T
B	$AG((all_f \wedge N1_{am}^{to}) \rightarrow (K_{N1}(any_i) \wedge \neg K_{N1}(hard_i) \wedge \neg K_{N1}(soft_i)))$	T
C	$\neg E(\neg sN2ns_i \ U \ (sN2ns_i \ \wedge \ \neg AF(K_{N1}(any_i))))$	T
D	$\neg E((\neg hN3ns_i \ \wedge \ \neg N3_d) \ U \ ((hN3ns_i \ \wedge \ \neg N3_d) \ \wedge \ \neg AF(K_{N1}(any_i))))$	T
E	$\neg E(\neg hN4nr_i \ U \ (hN4nr_i \ \wedge \ \neg AF(K_{N1}(any_i))))$	T
F	$\neg E(\neg hN6ns_i \ U \ (hN6ns_i \ \wedge \ \neg AF(K_{N1}(any_i))))$	T
G	$\neg E(\neg sN2ns_i \ U \ (sN2ns_i \ \wedge \ \neg AF(D_{ALL}(any_i))))$	T
H	$\neg E((\neg hN3ns_i \ \wedge \ \neg N3_d) \ U \ ((hN3ns_i \ \wedge \ \neg N3_d) \ \wedge \ \neg AF(D_{ALL}(any_i))))$	T
I	$\neg E(\neg hN4nr_i \ U \ (hN4nr_i \ \wedge \ \neg AF(D_{ALL}(any_i))))$	T
J	$\neg E(\neg hN6ns_i \ U \ (hN6ns_i \ \wedge \ \neg AF(D_{ALL}(any_i))))$	T
K	$\neg E(\neg sN2ns_i \ U \ (sN2ns_i \ \wedge \ \neg AF(E_{ALL}(any_i))))$	F
L	$\neg E((\neg hN3ns_i \ \wedge \ \neg N3_d) \ U \ ((hN3ns_i \ \wedge \ \neg N3_d) \ \wedge \ \neg AF(E_{ALL}(any_i))))$	F
M	$\neg E(\neg hN4nr_i \ U \ (hN4nr_i \ \wedge \ \neg AF(E_{ALL}(any_i))))$	F
N	$\neg E(\neg hN6ns_i \ U \ (hN6ns_i \ \wedge \ \neg AF(E_{ALL}(any_i))))$	F
O	$\neg E(\neg K_{N1}(any_i) \ U \ (K_{N1}(any_i) \ \wedge \ \neg AF(D_{ALL}(any_i))))$	T
P	$\neg E(\neg D_{ALL}(any_i) \ U \ (D_{ALL}(any_i) \ \wedge \ \neg AF(E_{ALL}(any_i))))$	F
Q	$AG((all_f \ \wedge \ N4_{sb}) \rightarrow (K_{N4}(hN3ns_i \ \wedge \ hN4nr_i) \ \wedge \ \neg K_{N4}(hN3ns_i) \ \wedge \ \neg K_{N4}(hN4nr_i)))$	T
R	$AG((all_f \ \wedge \ N3_{rd}) \rightarrow (K_{N3}(K_{N4}(hN3ns_i \ \vee \ hN4nr_i) \ \wedge \ \neg K_{N4}(hN3ns_i) \ \wedge \ \neg K_{N4}(hN4nr_i))))$	T
S	$\neg E((\neg hN3ns_i \ \wedge \ \neg N3_d) \ U \ ((hN3ns_i \ \wedge \ \neg N3_d) \ \wedge \ \neg AF(K_{N4}(hN3ns_i \ \vee \ hN4nr_i))))$	T
T	$\neg E(\neg hN4nr_i \ U \ (hN4nr_i \ \wedge \ \neg AF(K_{N4}(hN3ns_i \ \vee \ hN4nr_i))))$	T
U	$\neg E(\neg K_{N4}(hN3ns_i \ \vee \ hN4nr_i) \ U \ (K_{N4}(hN3ns_i \ \vee \ hN4nr_i) \ \wedge \ \neg AF(D_{ALL}(hN3ns_i \ \vee \ hN4nr_i))))$	T
V	$\neg E(\neg D_{ALL}((hN3ns_i \ \vee \ hN4nr_i)) \ U \ (D_{ALL}(hN3ns_i \ \vee \ hN4nr_i) \ \wedge \ \neg AF(E_{ALL}(hN3ns_i \ \vee \ hN4nr_i))))$	F
W	$\neg E(\neg D_{ALL}((hN3ns_i \ \vee \ hN4nr_i)) \ U \ (D_{ALL}((hN3ns_i \ \vee \ hN4nr_i)) \ \wedge \ \neg AF(E_{ALL}(hard_i))))$	T
X	$\neg E(\neg E_{ALL}(hard_i) \ U \ (E_{ALL}(hard_i) \ \wedge \ \neg AF(C_{ALL}(hard_i))))$	F

achieve this the start actions are set so that faults must occur at different times from each other. To distinguish between soft and hard faults, the stop action for the soft fault $sN2ns$ is set so that it stops injecting when 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 occurs for the fault.

To reason about the injected faults, we define a number of atomic propositions which can be found in Table IX where *Proposition* is the name of the atomic proposition and *Condition* is the condition in which the atomic proposition is true. For the naming convention *hard* indicates any hard fault; *soft* indicates a soft fault; *all* indicates all faults; *any* indicates any fault; f is the *faulty* persistence; i

is the *injecting* persistence; d indicates that a node is disconnected; sb indicates that a node has started sending beacons; rd indicates that a node will disconnect after receiving a beaconing token; to indicates a node has timed out; am indicates that the node is the active monitor.

The specifications defined to reason about diagnosability in the token ring protocol are given in Table X, along with the truth value that MCMAS returns for each specification. We also performed preliminary verification on the protocol to ensure that: 1) all the faults can enter a start state; 2) node 1 is the only active monitor; 3) if there is no active monitor eventually node 1 becomes the active monitor; 4) nodes 1 and 4 can reach a timeout state; 5) nodes 3, 4, and 6 are the only nodes that can disconnect. These represent additional properties that we expect the system to satisfy. MCMAS verifies all the specifications in approximately 14 hours using a 3.2GHz processor and approximately 57MB of memory, where the number of reachable states is approximately 2.3×10^5 out of a possible 1.4×10^{13} .

To verify the diagnosability of the diagnosis properties used for the monitoring process, Specification A states that whenever node 1 is not an active monitor and enters a timeout state, it knows that there is a hard fault on node 6. Specification B states that whenever all faults occur in a run of the system, if node 1 is an active monitor and enters a timeout state, it knows that there is a fault occurring on the ring, but does not know if it is a soft or hard fault. This provides an insight into the ability of active and standby monitors to determine different faults on the ring. A standby monitor can detect a fault on its nearest upstream neighbour and an active monitor can determine any fault on the ring.

To verify the diagnosability of the token ring without referring to a diagnosis property, Specifications C-F state that when a fault begins injecting always at some point in the future node 1 (the active monitor) knows that a fault has occurred. Since $hN3ns$ can occur when node 3 is disconnected, for our specifications we are only interested the first occurrence of $hN3ns$ when node 3 is not disconnected. Specifications G-J represent that when a fault begins injecting it always becomes distributed knowledge amongst all (*ALL*) nodes that that a fault has occurred. However, Specifications K-N verify that it is not the case that all of the nodes know that a fault has occurred after one has begun injecting. Similarly, Specifications O and P verify that the when node 1 first comes to know a fault has occurred it always eventually becomes distributed knowledge that a fault has occurred, but not all of the nodes always eventually come to know that a fault has occurred. This is because 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 resolution of soft faults.

To verify the diagnosability of the beaconing process, Specification Q encodes that whenever all faults occur in a

run of the system, if node 4 enters a state where it has begun sending beacons, 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 R states that whenever all faults occur in a run of the system, if node 3 intends to disconnect, it knows that node 4 knows that either node 3 is 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 S and T state that whenever node 3 first stops sending tokens or node 4 first stops receiving tokens, at some point in the future, node 4 knows that one of these faults has occurred. Specification U states that whenever node 4 first comes to know that node 3 is not sending tokens or node 4 is not receiving tokens, it always eventually becomes distributed knowledge that one of these faults has occurred. Further to this, Specification V states that whenever it first becomes distributed knowledge that one of these faults has occurred all of the nodes always eventually come to know that one of these faults has occurred. Similarly, Specification W states that whenever it first becomes distributed knowledge that one of these faults has occurred all of the nodes always eventually come to know that a hard fault has occurred. Specification X states that whenever all of the nodes first come to know that a hard fault has occurred it always eventually becomes common knowledge amongst all of the nodes that a hard fault has occurred.

The results from verifying these specifications using MCMAS allows us to determine a number of diagnosability properties of the token ring protocol as follows; 1) A standby monitor can diagnose a fault on its nearest upstream neighbour and an active monitor can diagnose any fault on the ring; 2) The knowledge of the occurrence of a fault is propagated by the active monitor to become distributed knowledge on the ring. 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.

VII. RELATED WORK

The majority of the previous work on combining fault injection with model checking [1]–[3], [11] is limited to

model checkers that use temporal logic to reason about properties of the system and are not suitable for MAS. Moreover, the approaches do not deal with diagnosability, and are primarily concerned with the properties of safety, fault tolerance, and recoverability. 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 [11].

In previous work on verifying diagnosability, the model checker NuSMV has been applied to verify diagnosability in a model based diagnosis system [4]. A model of the correct and faulty behaviour is constructed using a tool that translates the diagnosis system into a NuSMV input model. Temporal specifications are used to verify diagnosability, and distributed diagnosis is not present.

Previous work on injecting faults into MAS [8] provides a limited proof of concept command line fault injection compiler that injects statically persistent random faults into boolean states. The verification is limited to recoverability and fault tolerant properties, and is carried out on a simple example of the bit-transmission protocol. In this paper, we extended this work to allow for complex varying persistence faults to be injected for states comprising of enumerate variables and developed a fully functional graphical compiler to inject the faults automatically. Epistemic specifications were defined to reason about the knowledge of faults for verifying diagnosability, and used to verify diagnosability in the widely employed token ring protocol in which several agents containing rich functionality are present.

VIII. CONCLUSION

In this paper we presented an automated approach to verifying the property of diagnosability in MAS. We regard this work as a practically useful for ensuring accurate fault identification in MAS, which is important when diagnosis is used to achieve fault tolerance. The compiler we developed to inject complex faults provides a powerful and flexible tool for mutating a correctly behaving model into a faulty one. This automates the difficult and time consuming step of hand modelling faulty behaviour. The epistemic specifications we defined are suitable for verifying diagnosability for MAS in which both individual agent and system wide diagnosis of faults is present. The practical aspect of our tool has been demonstrated by using the compiler and epistemic specifications to verify diagnosability in the token ring protocol, which includes distributed diagnosis.

In future work we intend to use our approach to verify diagnosability in autonomous vehicle control systems and pass our fault injection compiler on to engineers working on the design of these systems. We envision the extension of the compiler to allow for user defined mutation rules and automatic generation of diagnosability specifications. Finally, we intend to investigate techniques to minimise

any negative impact our approach has on the memory consumption and time efficiency of the verification process.

ACKNOWLEDGEMENT

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

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. Villaforita. 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] A. Cimatti, C. Pecheur, and R. Cavada. Formal verification of diagnosability via symbolic model checking. In *Proceedings of IJCAI'03*, pages 363–369. Morgan Kaufmann, 2003.
- [5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, 1999.
- [6] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
- [7] J. Ezekiel and A. Lomuscio. MCMAS fault injection compiler: <http://www.doc.ic.ac.uk/~jezekiel/ficompiler.html>.
- [8] J. Ezekiel and A. Lomuscio. Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In *Proceedings of AAMAS'09*, pages 113–120. IFAAMAS, 2009.
- [9] R. Fagin, J. Y. Halpern, M. Y. Vardi, and Y. Moses. *Reasoning about knowledge*. MIT Press, Cambridge, 1995.
- [10] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.
- [11] 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.
- [12] M. Kalech and G. A. Kaminka. On the design of social diagnosis algorithms for multi-agent teams. In *Proceedings of IJCAI'03*, pages 370–375. Morgan Kaufmann, 2003.
- [13] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.
- [14] A. Lomuscio and M. J. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.
- [15] P. T. R. Micalizio and G. Torta. On-line monitoring and diagnosis of multi-agent systems: a model based approach. In *Proceedings of ECAI'04*, pages 848–852. IOS Press, 2004.
- [16] N. Roos, A. ten Teije, and C. Witteveen. A protocol for multi-agent diagnosis with spatially distributed knowledge. In *Proceedings of AAMAS'03*, pages 655–661. ACM, 2003.
- [17] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.
- [18] M. J. Wooldridge. *Reasoning about Rational Agents*. MIT Press, Cambridge, 2000.