A methodology for automatic diagnosability analysis

Jonathan Ezekiel and Alessio Lomuscio

Department of Computing, Imperial College London, UK
{jezekiel,alessio}@doc.ic.ac.uk

Abstract. We present an algorithm based on temporal-epistemic model checking combined with fault injection to analyse automatically the diagnosability of faults by agents in the system. We describe an implementation built on the multi-agent systems model checker MCMAS and a dedicated compiler for injecting faults into an MCMAS program. A diagnosability report is generated by the implementation which can be utilised at an early stage of fault tolerant multi-agent system design to ensure accurate fault diagnosis. We demonstrate the practical usefulness of the algorithm by performing automatic diagnosability analysis on a model of the IEEE 802.5 token ring LAN protocol which employs fault diagnosis mechanisms to achieve fault tolerance.

1 Introduction

Distributed fault tolerant systems are notoriously difficult to understand and design due to their high level of complexity [10]. A potential way to manage this complexity is offered by the multi-agent systems (MAS) paradigm [25] in which agents, representing processes of a distributed system, autonomously interact with one another, engaging in communication, co-ordination, negotiation, etc. Moreover, a number of fault tolerant MAS architectures have been designed which utilise strategies such as agents *diagnosing* faults so that they can communicate and co-ordinate to recover from them (see e.g., [18]).

As a design paradigm, MAS has many applications including, but not limited to distributed control systems (DCS) (see e.g., [20]). Within the general area of DCS it is known that safe design is a major industrial concern since DCS are becoming increasingly complex and involved in many safety-critical applications [6]. However, practical approaches towards *verifying* fault tolerant MAS are required to certify that they conform to the stringent requirement of operating correctly under degraded conditions [1].

Recently, an approach combining fault injection [16] with model checking [8] has been used to verify the correctness of fault tolerance mechanisms in reactive systems [2, 4, 5, 17] and MAS [12, 13]. In contrast to ad-hoc modelling of faulty behaviour, in this approach faults can be automatically injected into a model of a correctly behaving system to create a mutated model which exhibits both correct and faulty behaviour. Temporal-epistemic specifications [14] can then be verified

to analyse the correct and faulty behaviour of agents in the mutated model, as well as the knowledge that agents have about the behaviour. This allows for the verification of fault tolerance, recovery from faults, and *diagnosability*, i.e., whether an unobservable fault can be accurately diagnosed from the observable events of the system [22].

The high level of usability offered by the automatic nature of both the fault injection and the model checking process makes the approach particularly attractive to non-experts in verification [4]. Another advantage is the ability to use the model checker to generate automatically artifacts that analyse the impact of the injected faults such as fault trees [23]. To date, these artifacts have been generated by using temporal formulas to describe which component failures occur as a result of the injected faults [3, 4]. However, artifacts relating to diagnosability have yet to be suggested in the literature.

In this paper we show how a diagnosability artifact can be generated from a combined fault injection and temporal-epistemic model checking [15, 19, 21] approach by presenting a methodology for automatic *diagnosability analysis*. The analysis is used to provide the user with a report on the diagnosis of each injected fault by every agent in the system. We consider this to be part of a practical approach towards verifying fault tolerant MAS that can be used at an early stage of system design to ensure that agents in the system accurately diagnose faults.

We implement these ideas by integrating the algorithm we propose with the model checker MCMAS [19] and an existing fault injection compiler that injects automatically faults into a model for input into MCMAS [12, 13]. We describe a framework in which powerful modules which generate fault analysis artifacts for fault tolerant MAS can be integrated with MCMAS and the fault injection compiler. To highlight the practical usefulness of the algorithm from a user perspective we use the implementation to perform automatic diagnosability analysis on a model of the IEEE 802.5 token ring LAN protocol from [13] which utilises distributed diagnosis mechanisms to achieve fault tolerance.

The rest of the paper is structured as follows. In Section 2 we provide the background on model checking, interpreted systems, MCMAS, fault injection, and artifact generation. In Section 3 we present the algorithm for diagnosability analysis. In Section 4 we describe a framework for integrating fault analysis modules with MCMAS and the fault injection compiler, which we use to implement the algorithm for diagnosability analysis. In Section 5 we show how the diagnosability analysis module is applied to the token ring protocol. In Section 6 we discuss the related work and in Section 7 we conclude and put forward future work.

2 Background

Model checking [8] is a widely adopted technique for systems verification. The system considered for verification S is represented by a logical model M_S which encodes the behaviour of the system as computational traces. 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 an automatic decision procedure, making model checking attractive for the purpose of verification [8]. In the case of MAS φ_P is often expressed by using a number of rich modal logics including temporal, ATL, and epistemic logics [25]. Particularly relevant to diagnosability is temporal-epistemic logic, which can be used to reason about the *knowledge* of the agents over time.

2.1 Interpreted systems and MCMAS

We summarise the key points of the formalism used by following the presentation given in [12]. Interpreted systems [14] are a popular semantics for temporal-epistemic logic. Each agent $i \in \{1, \ldots, n\}$ 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 $P_i : L_i \to 2^{Act_i}$, 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 $g = (l_1, \ldots, l_n, l_E) \in L_1 \times \ldots \times L_n \times L_E$ where $l_i \in L_i$ for each i and $l_E \in L_E$, 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 $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \to L_i$, 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 $t_E : L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \to L_E$. It is assumed that in every state agents evolve simultaneously. The evolution of the global states of the system is described by a function $t : S \times Act \to S$, 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 by $t(g,a) = g'$ 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 a global state 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 $V : S \to 2^{AP}$. We define an *interpreted system* as the tuple:

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

The syntactical constructs and the semantic model that are presented in [19] 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 [14].

IS is associated with a model $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible 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 [14]). 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 [14] 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 $(M, w) \models \varphi$ to represent that a formula φ is true at a world w in a Kripke model M, associated with an interpreted system IS. Satisfaction is defined as follows.

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

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 [14], we say that a formula φ is true in an interpreted system IS, and we write $IS \models \varphi$, if $M_{IS} \models \varphi$. *A formula is true in an interpreted system if it is true in the associated Kripke model.*

MCMAS [19] 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. MCMAS supports the verification for all formulas in the language above. The structure of an ISPL program allows the local states to be defined using *boolean*, *bounded integer*, and *enumeration* variables.

2.2 Fault injection into MAS programs

The first step of a combined fault injection and model checking approach [2, 4, 5, 13, 17] involves *mutating* a model of correct system behaviour by injecting

faulty behaviour into it. The output of the mutation step is a model containing correct and faulty behaviour.

We summarise the mutation technique for interpreted systems defined in [12, 13]. Any agent A of the system can be mutated into a faulty agent A^{F*} which includes the faulty behaviour that results from a fault occurring. For each fault an additional fault injection agent FI implements the timing characteristics of the fault. The faulty behaviour is triggered in the faulty agent whenever an *inject* action is performed by FI. Conversely, the correct behaviour is preserved in the faulty agent whenever the *inject* action is not performed by FI.

The faulty behaviour in the faulty agent A^{F*} is introduced using a number of *mutation rules* which determine how the evolution function t_A is mutated to $t_{A^{F*}}$. A *variable value replace* fault defines that the value of a variable var is updated with a value v_2 in $t_{A^{F*}}$ whenever the value of var is updated to a value 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 faulty behaviour in $t_{A^{F*}}$ preserves $var = v_1$. This fault can be used to define behaviour in which a component becomes stuck in particular state. Further rules are defined in [12].

The occurrence of the *inject* action is determined by the behaviour of the fault injection agent FI. A number of timing options can be selected for FI: injecting *constantly*, *randomly*, *after a random start*, *until a random stop*, and *after and until an action has occurred* [12]. The local states, actions, protocol, and evolution function of the fault injection agent are defined according to these options, which can be combined to create complex timing characteristics of the fault. A mutated set of initial states I^{F*} stipulates that the local state of FI is set to either $notfaulty$ which persists throughout the system run, or to a state in which faults may be injected into the system by FI in the future according to the timing options.

A mutated valuation function V^{F*} relates atomic propositions to the local states of each fault injection agent. This can be used to reason about the correct and faulty behaviours of the mutated interpreted system IS^{F*}. For each fault $j \in \{1, \ldots, m\}$ the mutated set of atomic propositions AP^{F*} extends with the propositions $faulty_j$, $injected_j$, $injecting_j$, $stopped_j$. $faulty_j$ represents that a fault can be injected during the system run; $injected_j$ expresses that a fault is injected at the current tick of the clock (i.e., a global state describing the system at a particular instant of time); $injecting_j$ denotes that a fault can be injected at the current tick of the clock; $stopped_j$ describes that a fault has been injected but can no longer be injected at the current tick of the clock. The extended faulty system is defined as:

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

Once a mutated model IS^{F*} has been obtained, both the correct and faulty behaviours of the system can be analysed. A library of *specification patterns* pertaining to fault tolerance, recoverability, and diagnosability defined in [12, 13]

can be used to reason about properties of the system in relation to the taxonomy of dependable computing given in in [1]. Such properties include whether the fault becomes an error that is propagated internally amongst the agents of the system, whether it can be diagnosed, recovered from, or further propagated to the service interface to cause a failure. In this case, the *system boundary* is the shared actions between the agents of the system and the environment. Thus, we can verify properties of the system that affect the system boundary under faulty behaviour to determine whether the fault becomes a failure.

We highlight two specification patterns that are relevant to this paper. To reason about fault tolerance for a property ϕ we can analyse [13]:

$$AG((\neg faulty_j \wedge faulty_k) \rightarrow \phi) \tag{1}$$

This formula states that ϕ always holds whenever fault j is never injected into the model and fault k may be injected into the model. The formula specifies the ability of the system to tolerate faults, in this instance, fault k.

Usually diagnosability is informally defined by saying that *a fault is diagnosable if some observations after the occurrence of the fault can correctly identify it* [22]. A fault is diagnosable if any agent of the system *knows* about it at some point after its occurrence. We can express this as [12]:

$$\neg E(\neg injected_j \; U \; (injected_j \wedge \neg AF(K_i(injecting_j \vee stopped_j)))) \tag{2}$$

This formula states that there is no path in which at some point fault j is injected and from that time it is not true that at some point in the future agent i knows fault j can be or has been injected. In other words the formula specifies the ability of agent i to diagnose fault j correctly after j has first been injected.

2.3 Generating fault analysis artifacts

A particular advantage of using an approach based on model checking and fault injection is the ability to generate automatically artifacts such as fault trees [23]. A fault tree is a graphical representation which is constructed by identifying a *minimal cut set* of events that can cause a system malfunction to occur. This malfunction is also known as a top level event (TLE). The fault tree displays these events connected by logic gates to highlight their relation to each other.

The process of identifying this minimal cut set can be automated by combining fault injection and model checking [3]. The function $h : \{f, \neg f, \epsilon\}^m \rightarrow \{T, F\}$ represents the *cut set* for m injected faults where f corresponds to the fault being injected, $\neg f$ corresponds to the fault not being injected, and ϵ corresponds to the fault being either injected or not injected. For example, $h(\{f_1, \epsilon, \neg f_3\}) = T$ represents that the TLE occurs when: fault 1 is injected, fault 2 may or may not be injected, and fault 3 is not injected. In other words a TLE occurs when fault 1 is injected and fault 3 is not irrespective of fault 2. The cut set can be created automatically by using a model checker to verify a number of specifications against a mutated model in order to determine whether a TLE occurs for each

fault injection combination of the cut set. Formula 1 is an example of a suitable formula that can be used to define specifications that create the cut set.

Once the cut set has been created the *prime implicants* of the cut set, i.e., the minimal set of events relevant to the occurrence of the TLE can be determined. For example, a cut set $f_1 \vee (f_1 \wedge f_3)$ describes that when fault 1 is injected, or when fault 1 is injected and fault 3 is injected, the TLE occurs. In this cut set f_1 is a prime implicant representing the minimal cut set, since f_1 is sufficient for all occurrences of the TLE. To identify a minimal cut set, many prime implicant algorithms exist, for further discussion see [3].

Fault trees are useful for studying the impact of faults, however, they provide limited insight into the cause of system malfunctions. At an early stage of design it is desirable to analyse diagnosability so that system malfunctions do not occur as a result of fault tolerance mechanisms diagnosing faults inaccurately.

3 Diagnosability analysis

In this section we apply the general ideas for artifact generation presented in the previous section in order to analyse diagnosability. We achieve this by employing a diagnosability formula to generate a diagnosability cut set and by defining an algorithm to identity a minimal cut set. Unlike fault trees which communicate the impact of faults, there are no pre-defined graphical representations of diagnosability. Thus, any diagnosability specification pattern used for analysis must be carefully selected in order to generate a cut set of meaningful events. As a starting point for producing diagnosability artifacts we choose Formula 2. The formula is used to determine the diagnosis of faults by each of the non fault-injection agents in the system.

In the following we define the diagnosability cut set. For each non fault-injection agent $i \in \{1, \ldots, n\}$ and each fault $j \in \{1, \ldots, m\}$, given a *diagnosis group*, i.e., a group of faults for diagnosis $DG \in 2^{\{1,\ldots,m\}}$, the cut set which determines that a diagnosis group can be diagnosed by an agent after a fault has first been injected into the system is described by the function $D : \{1, \ldots, n\} \times \{1, \ldots, m\} \times 2^{\{1,\ldots,m\}} \rightarrow \{T, F\}$, where T indicates that the diagnosis can be made. For example $D(i, j, \{j, k\})$ represents whether agent i diagnoses that either fault j or fault k has occurred after fault j is first injected into the system. We can define this function where $ijst_k = (injecting_k \vee stopped_k)$ using Formula 2 as follows :

$$D(i, j, DG) = T \text{ iff } j \in DG \text{ and}$$
$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\textstyle\bigvee_{k \in DG} ijst_k))))$$

In other words the diagnosability cut set function evaluates to true if fault j is in the diagnosis group DG and the mutated interpreted system IS^{F*} is satisfied by a formula describing that agent i diagnoses the faults in DG after j has first been injected.

Due to the way that the mutated initial states are defined in IS^{F*} there is at least one path in IS^{F*} in which at some point fault j is injected and along

that path any fault $k \in \{1, \ldots, m\} \setminus \{j\}$ is never injected. Agent i cannot know about fault k along a path in which k is never injected. Thus if:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DG} ijst_k))))$$

it follows that:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DG} ijst_k) \wedge \neg K_i(\bigvee_{k \in DG \setminus \{j\}} ijst_k))))$$

The formula above specifies in addition to Formula 2 that agent i does not know that any of the faults in DG other than j can be or has been injected into the system. Given that this specification is implied by Formula 2 it is implicitly included in the definition of the diagnosability cut set function.

For each fault injected into the system the minimal cut set must contain the *most specific diagnosis* of that fault by each of the agents, i.e., if an agent can diagnose a fault that has been injected into the system, then it is meaningful to identify the smallest set of faults that the injected fault is part of, that can be diagnosed by the agent. For example, in a network protocol it may be necessary to diagnose a specific severe fault in order to reconfigure a router to bypass a workstation. Conversely, it may suffice to diagnose that any of a number of possible intermittent faults have occurred so that a message can be resent.

The diagnosability cut set function D does not determine the most specific diagnosis. This is because if DG can be diagnosed by an agent, then according to the conjunction of faults in the definition of Formula 2, $DG \cup DH \in 2^{\{1,\ldots,m\}}$ can be diagnosed by the agent. For example, given three faults j, k, and l, if $D(i, j, \{j, k\}) = T$, then $D(i, j, \{j, k, l\}) = T$, but only $\{j, k\}$ should be included in the minimal cut set. Thus, we wish to identify a minimal cut set $MCS_{ij} \subset 2^{\{1,\ldots,m\}}$ containing the most specific diagnosis of fault j by agent i. To identify the minimal cut set we only need to consider conjunctions. Instead of employing a prime implicant algorithm that considers disjunctions we define a restriction to the minimal cut set as follows:

$$DH \in 2^{\{1,\ldots,m\}} \notin MCS_{ij} \text{ iff } D(i, j, DG \in 2^{\{1,\ldots,m\}}) = T \text{ and } DG \subset DH$$

In other words for every diagnosis group in the minimal cut set there can not be any subsets of that diagnosis group in the minimal cut set.

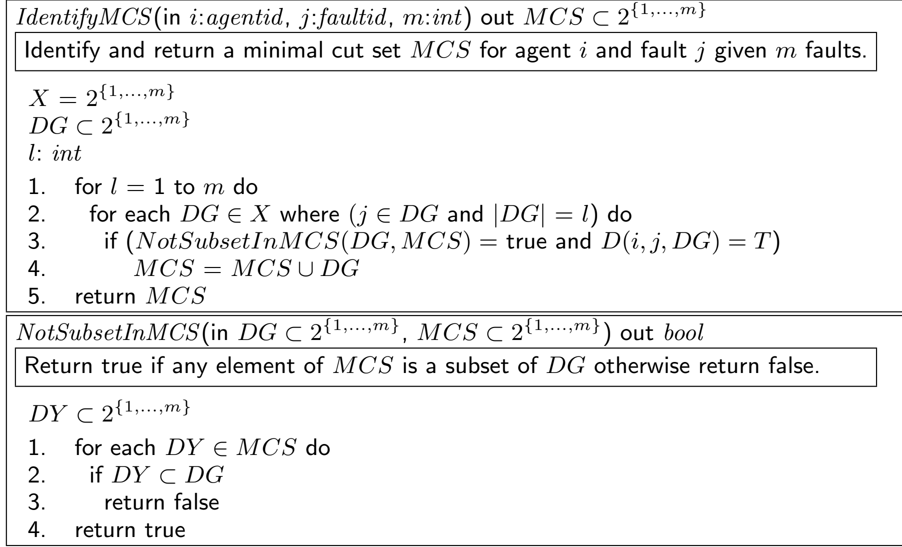
If we apply this restriction to the minimal cut set then for every diagnosis group $DY \in MCS_{ij}$ we have that for every diagnosis group $DZ \subset DY$, $D(i, j, DZ) = F$. It follows that:

$$IS^{F*} \not\models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(\bigvee_{DZ}^{DY} K_i(\bigvee_{k \in DZ} ijst_k))))$$

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(\bigwedge_{DZ}^{DY} \neg K_i(\bigvee_{k \in DZ} ijst_k))))$$

hence:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DY} ijst_k) \bigwedge_{DZ}^{DY} \neg K_i(\bigvee_{k \in DZ} ijst_k))))$$



```
IdentifyMCS(in i:agentid, j:faultid, m:int) out MCS ⊂ 2^{1,...,m}
  Identify and return a minimal cut set MCS for agent i and fault j given m faults.

  X = 2^{1,...,m}
  DG ⊂ 2^{1,...,m}
  l: int
  1.  for l = 1 to m do
  2.    for each DG ∈ X where (j ∈ DG and |DG| = l) do
  3.      if (NotSubsetInMCS(DG, MCS) = true and D(i, j, DG) = T)
  4.          MCS = MCS ∪ DG
  5.  return MCS

NotSubsetInMCS(in DG ⊂ 2^{1,...,m}, MCS ⊂ 2^{1,...,m}) out bool
  Return true if any element of MCS is a subset of DG otherwise return false.

  DY ⊂ 2^{1,...,m}
  1.  for each DY ∈ MCS do
  2.    if DY ⊂ DG
  3.      return false
  4.  return true
```

Fig. 1. Psuedo-code for the diagnosability analysis algorithm.

where $\bigwedge_{DZ}^{DY}$ and $\bigvee_{DZ}^{DY}$ are the conjunctions and disjunctions of all subsets of indices $DZ \subset DY$ respectively. The restriction defined for the minimal cut set determines that every diagnosis group in the minimal cut set is verified by the strong specification above. This specifies, in addition to Formula 2, that agent i does not know that any combination of the faults in every diagnosis group $DZ \subset DY$ other than j can be or has been injected into the system.

Based on the cut set function D and the minimal cut set restriction, we defined a sequential algorithm that performs diagnosability analysis to identify a minimal cut set for each agent as shown in Figure 1. A description for both the functions in the algorithm is as follows:

IdentifyMCS: identifies a minimal cut set for agent i and fault j which are passed as arguments to $IdentifyMCS$. Lines 1-2 define the iteration through all the possible diagnosis groups that include fault j, in ascending order of size. The iteration order ensures that the restriction function only needs to be applied once to each diagnosis group. Line 3 checks that a diagnosis group DG does not have any subsets in the minimal cut set and whether the faults in DG can be diagnosed for agent i and fault j. If both these conditions are met then the diagnosis group is added to the minimal cut set. Line 6 returns the minimal cut set for agent i and fault j.

NotSubsetInMCS: applies the restriction to the minimal cut set by checking to see whether any subsets of a diagnosis group DG passed as an argument to $NotSubsetInMCS$ are subsets of any diagnosis groups in the minimal cut set MCS passed as an argument to $NotSubsetInMCS$. Lines 1-3 iterate through all the diagnosis groups DY in MCS and $NotSubsetInMCS$ returns false if any

diagnosis group DY is a subset of DG. Line 4 is reached if no diagnosis group DY in MCS is a subset of DG, at which point $NotSubsetInMCS$ returns true.

Example: Given a system with four faults j, k, l, and m, in which agent i diagnoses that either faults j or k have occurred, or faults j or l or m have occurred, after fault j has been injected, a call to $IdentifyMCS(i, j, 4)$ performs four iterations of the for loop in line 2. *Iteration 1*: Diagnosis groups containing one element that include fault j: $\{j\}$. Since MCS is empty $NotSubsetInMCS$ always returns true. The function $D(i, j, \{j\})$ returns false and $\{j\}$ is not added to MCS. $MCS = \emptyset$. *Iteration 2*: Diagnosis groups which contain two elements that include fault j: $\{j, k\}, \{j, l\}, \{j, m\}$. Since MCS is empty $NotSubsetInMCS$ always returns true. $D(i, j, \{j, k\})$ returns true and $\{j, k\}$ is added to MCS. $D(i, j, \{j, l\})$ and $D(i, j, \{j, m\})$ return false. $MCS = \{\{j, k\}\}$. *Iteration 3*: Diagnosis groups which contain three elements that include fault j: $\{j, k, l\}$, $\{j, k, m\}$, $\{j, l, m\}$. Since $MCS = \{\{j, k\}\}$ $NotSubsetInMCS$ returns false for $\{j, k, l\}$ and $\{j, k, m\}$. $D(i, j, \{j, l, m\})$ returns true and $\{j, l, m\}$ is added to MCS. $MCS = \{\{j, k\}, \{j, l, m\}\}$. *Iteration 4*: Diagnosis groups containing four elements that include fault j: $\{j, k, l, m\}$. Since $MCS = \{\{j, k\}, \{j, l, m\}\}$ $NotSubsetInMCS$ returns false for $\{j, k, l, m\}$. $MCS = \{\{j, k\}, \{j, l, m\}\}$.

The algorithm is suitable for systems in which faults are resolved after they are diagnosed and future occurrences of the fault have no further impact on the system. Repeating faults can be analysed by substituting Formula 2 with a formula that reasons about the diagnosability of a fault after each injection of the fault. A more dynamic analysis would consider the case where one agent may diagnose faults if another agent is not available for diagnosis, i.e., the disjunction of agents diagnosing the disjunction of faults. Another case to consider is the conjunction of faults, for situations in which different faults injected into the system are diagnosed as a single fault. Other diagnosability specification patterns from [12] can also be applied to reason about possible diagnosis, group diagnosis, and the propagation of the knowledge of faults through the system. These extensions seem feasible but would require a different definition of the cut set function and minimal cut set restriction.

The implementation of the diagnosability analysis algorithm requires an interface to a temporal-epistemic logic model checker, the definitions of faults and agents, and the mutated interpreted system model. In particular, function D must use the model checker to verify specifications on the mutated model. We describe how these requirements are met by a fault analysis module interface which is defined as part of a framework for implementing fault analysis algorithms as modules in the next section.

4 A framework for integrating fault analysis modules

As part of a practical approach towards verifying fault tolerant MAS we constructed a framework for integrating fault analysis modules with the model checker MCMAS and a compiler for injecting automatically faults into a MCMAS program [12]. Doing so provides a manner in which powerful fault analysis

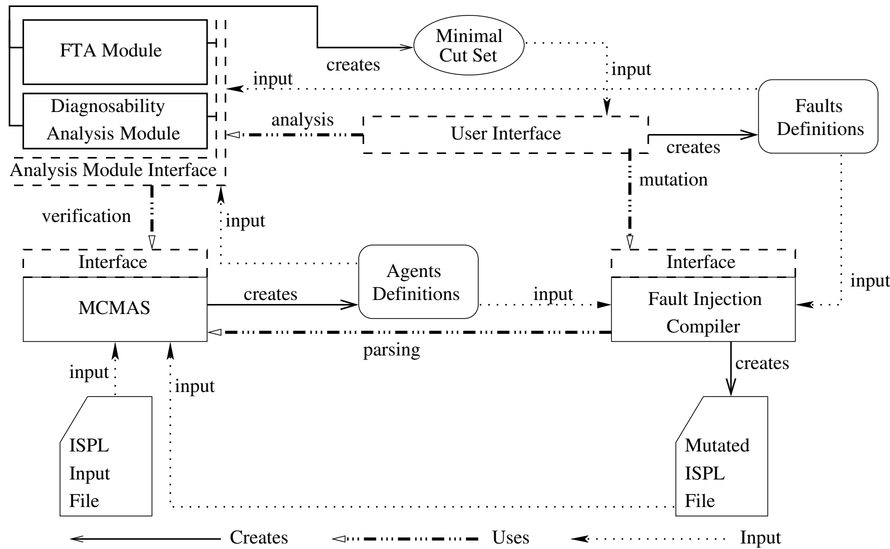


Fig. 2. Integrating fault analysis modules using the analysis module interface.

modules based on algorithms such as the one presented in Section 3 can be implemented. The high level of automation achieved by fault analysis modules makes them particularly usable for non-experts in verification who are working on the design of fault tolerant MAS.

We outline the framework in Figure 2. Any analysis module can be inserted into the framework via the *analysis module interface* which provides the module with an interface from which it is invoked by the user, the definitions of the agents and faults which can be used to construct formulas to be used for analysis, an interface to MCMAS to verify formulas on a mutated ISPL program, and a method to display to the user a minimal cut set in a graphical or report format. We further describe the framework as follows.

User interaction: the user interface allows the user to define the faults which are to be injected into an ISPL program; save a file containing the mutated ISPL program which includes the injected faults; and analyse automatically the behaviour of the model defined by the mutated ISPL program by using any available fault analysis module. The process of defining faults includes their name, mutation rules, timing options, etc. If automatic analysis of faults is required then the compiler interface is used to create a file containing the mutated ISPL program for analysis, and the fault analysis module selected by the user is initiated though the analysis module interface to create a minimal cut set which is passed back from the analysis module interface to be displayed in either a graphical or report format.

MCMAS: the MCMAS interface is used to perform the parsing of an ISPL file and perform the verification of specifications against a mutated ISPL program. The parsing procedure results in the creation of agents definitions including their name, protocol, transition relation, actions, etc. Passing a formula to the MCMAS interface instructs MCMAS to perform verification of the specification against the mutated ISPL program and the result of the verification is passed back through the MCMAS interface.

Fault injection compiler: the compiler interface is used to request the mutation of an ISPL program. MCMAS is used to parse the file in which the ISPL program containing correct behaviour is defined in order to create the agents definitions. The agents definitions and faults definitions are utilised by the compiler to save a file consisting of the mutated ISPL program which contains both correct and faulty behaviour including the injected faults.

Automatic fault analysis: the analysis module interface is used to invoke any available fault analysis module to perform analysis on a mutated ISPL program that has been created using the agents definitions and faults definitions. The module identifies a minimal cut set as a result of the analysis. The agents definitions and faults definitions are utilised to construct the formulas which are used for the analysis. MCMAS is used to verify these specifications against the mutated ISPL program. The minimal cut set identified as a result of the analysis is passed back through the interface so that it can be displayed to the user.

The framework is implemented in C++ (for linux operating systems) so that modules can be easily integrated into the framework in an object oriented manner. We implemented into the framework an FTA module which displays fault trees in a graphical format, and a diagnosability analysis module based on the algorithm in Section 3 which displays a report on the diagnosability of faults by agents in the system. The framework and these modules are included as part of the fault injection compiler which is available for public use [11].

5 Automatic diagnosability analysis of the IEEE 802.5 token ring protocol

The IEEE 802.5 token ring protocol is a widely used local area network (LAN) protocol in which network nodes are logically organised in a ring. The data circulates in one direction around the ring via a token passed from node to node. The network is logically defined as a ring topology and physically defined as a star topology. Fault tolerance is achieved by physically disconnecting faulty nodes and re-establishing the logical ring to bypass them. The protocol employs a distributed diagnosis mechanism to diagnose faulty nodes for disconnection.

Tokens containing fault information are sent around the ring when a fault occurs, which allows nodes to diagnose faults. One node on the ring is designated as an *active monitor* which diagnoses and resolves *soft faults*, i.e., faults that can be resolved without requiring a node to be disconnected. The other nodes are designated as *standby monitors* which diagnose *hard faults* on themselves and

Diagnosability Analysis

Fault injected for diagnosis : sN2ns

Agent	Fault Diagnosis
Node_1	hN6ns or hN4nr or hN3ns or sN2ns

Fault injected for diagnosis : hN3ns

Agent	Fault Diagnosis
Node_1	hN6ns or hN4nr or hN3ns
Node_2	hN6ns or hN4nr or hN3ns
Node_3	hN4nr or hN3ns
Node_4	hN4nr or hN3ns
Node_5	hN6ns or hN4nr or hN3ns
Node_6	hN6ns or hN4nr or hN3ns

Fault injected for diagnosis : hN4nr

Agent	Fault Diagnosis
Node_1	hN6ns or hN4nr or hN3ns
Node_2	hN6ns or hN4nr or hN3ns
Node_3	hN4nr or hN3ns
Node_4	hN4nr
Node_5	hN4nr
Node_6	hN6ns or hN4nr or hN3ns

Fault injected for diagnosis : hN6ns

Agent	Fault Diagnosis
Node_1	hN6ns
Node_2	hN6ns or hN4nr or hN3ns
Node_3	hN6ns or hN4nr or hN3ns
Node_4	hN6ns or hN4nr or hN3ns
Node_5	hN6ns or hN4nr or hN3ns
Node_6	hN6ns

Fig. 3. A diagnosability analysis report on the token ring model.

their nearest upstream neighbour i.e., faults that are resolved by disconnecting and bypassing nodes. Once a fault is diagnosed, information can be propagated around the ring to inform nodes that a fault has been diagnosed.

To illustrate the practical application of our diagnosability analysis module we use the implementation of the token ring protocol in ISPL from [12], which we refer the reader to for in-depth details of the implementation and the faults we injected into the model. The model contains 6 nodes labelled $N1, \ldots, N6$ with Node 1 designated as the active monitor; the token circulates clockwise from $N1$ to $N6$. Several different faults were chosen to be injected into different nodes of the model so that the diagnosability of the active and standby monitors could be analysed. Once a fault has been resolved it has no further impact on the protocol. We used the *variable value replace* and *stuck at select* mutation rules combined with various timing options to define the following faults for injection:

$sN2ns$: 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).

Once the faults had been defined for injection, automatic diagnosability analysis was selected to be performed on the mutated token ring model containing correct and faulty behaviour. The diagnosability analysis took approximately 22 minutes to complete on an Intel Pentium 2.5GHz Dual Core E5200 processor using approximately 57MB of memory. The number of reachable states was approximately 2.3×10^5 out of a possible 1.4×10^{13}.

The diagnosability report displayed to the user which shows the results from the analysis is illustrated in Figure 3. The report displays the minimal cut sets identified for each injected fault and agent combination. For each injected fault

a list is shown of the most specific diagnosis of that fault (under the heading *Fault Diagnosis*) that each agent can make.

The report allows us to determine a number of diagnosability properties of the token ring protocol as follows; Firstly, when the soft fault sN2ns is injected it is diagnosed by the active monitor as a possible soft or hard fault, thus, the active monitor uses the same mechanism to diagnose all faults; Secondly, only the active monitor can diagnose a soft fault; Thirdly, when a hard fault is injected, all monitors can diagnose the occurrence of a hard fault that has affected themselves or their nearest upstream neighbour; Finally, all monitors can diagnose the occurrence of a hard fault on the ring.

The diagnosability properties we described are critical to achieving fault tolerance in the token ring protocol. Soft faults need to be diagnosed by the active monitor for resolution, hard faults are resolved by the standby monitor if the fault has occurred on itself or its nearest upstream neighbours, and all other monitors need to know that a hard fault has occurred so that they can propagate information about the fault around the ring.

Without having to write any specifications, and in the absence of reasoning about fault resolution mechanisms, we have demonstrated that the token ring protocol accurately diagnoses faults. This illustrates how automatic diagnosability analysis can be usefully applied during the early design stages of fault tolerant MAS.

6 Related work

The majority of the previous work on combining fault injection with model checking [2, 3, 4, 5, 17] is limited to temporal logic model checking. Moreover, the approaches are primarily concerned with verifying properties of fault tolerance and do not analyse diagnosability. A platform in which analysis artifacts including an FTA module are integrated with an automatic fault injection tool and a temporal logic model checker is described in [4]. The general approach to producing fault analysis artifacts in our paper is similar to the implementation of the automatic FTA algorithm in [3].

The earlier work on combining fault injection with temporal-epistemic model checking [12, 13] implements a compiler for injecting faults automatically into an ISPL model, and defines a number of formulas for verifying fault tolerance, recoverability, and diagnosability. Specifications are hand written to analyse diagnosability and automatic diagnosability analysis is not considered.

The previous work on analysing diagnosability [7, 9, 22, 24] considers discrete event systems [22, 24], and model based diagnosis systems [7, 9]. The main focus of the work in [9, 22, 24] is the formalisation of the diagnosability problem and less attention is given to the practicality of the proposed algorithms for analysis. A practical approach to verifying diagnosability using temporal logic model checking is given in [7]. In this approach, a coupled twin model of the diagnosis system must be constructed so that diagnosability can be expressed as a temporal specification. This implies that the modelling component of the technique

is significantly more difficult in comparison to injecting automatically faulty behaviour. The practicality of this approach is not examined for distributed systems.

7 Conclusions

In this paper we presented a methodology for automatic diagnosability analysis based on fault injection and temporal-epistemic model checking. We implemented an algorithm to analyse automatically the diagnosability of faults by agents in a system. As part of the implementation we defined a framework for integrating fault analysis modules with the model checker MCMAS and a fault injection compiler. We demonstrated the practical usefulness of our approach by analysing automatically diagnosability in a model of the token ring protocol which utilises distributed diagnosis to achieve fault tolerance.

We regard our methodology as an important step in the development of practical tools for verifying fault tolerant MAS. Our framework can be used to build powerful automatic fault analysis modules which are user friendly. These are particularly useful for non-experts in verification who are working on the design of fault tolerant MAS. The analysis can be employed at an early stage of design with a high level of automation. These aspects of our work encourage a unified design and verification approach for fault tolerant MAS.

In future work we intend to implement diagnosability analysis modules that analyse group diagnosis, and the propagation of the knowledge of faults through the system. Our analysis modules will be applied to autonomous vehicle control systems and we aim to provide a powerful analysis tool for engineers working on the design of these systems.

Acknowledgement

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

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

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

[3] M. Bozzano and A. Villafiorita. Integrating fault tree analysis with event ordering information. In *Proceedings of ESREL'03*, pages 247–254. Lisse: Swets & Zeitlinger, 2003.

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

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

[6] P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *Proceedings of SAFECOMP'01*, volume 2187 of *LNCS*, pages 215–226. Springer, 2001.

[7] 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.

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

[9] L. Console, C. Picardi, and M. Ribaudo. Diagnosis and diagnosability analysis using PEPA. In *Proceedings of ECAI'00*, pages 131–135. IOS Press, 2000.

[10] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

[11] J. Ezekiel and A. Lomuscio. MCMAS fault injection compiler project page: http://www.doc.ic.ac.uk/∼jezekiel/ficompiler.html.

[12] J. Ezekiel and A. Lomuscio. An automated approach to verifying diagnosability in multi-agent systems. In *Proceedings of SEFM'09*, pages 51–60. IEEE, 2009.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] S. Mannor and J. S. Shamma. Multi-agent learning for engineers. *Artificial Intelligence*, 171(7):417–422, 2007.

[21] 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.

[22] 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.

[23] W. Vesley, F. Goldberg, N. Roberts, and D. Haasl. Fault tree handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Comission, 1981.

[24] Y. Wang, T.-S. Yoo, and S. Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(2):233–263, 2007.

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