Working Paper Series
ISSN 1177-777X

VERIFICATION OF THE DIAGNOSABILITY OF DISCRETE-EVENT SYSTEMS IN WATERS

Nicholas McGrath

Working Paper: 04/2018
21 November 2018

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Abstract

The task of detecting faults and reacting to them appropriately is a crucial aspect of a building a stable system. If a fault cannot be directly observed, its occurrence must be inferred from what can be observed. In the realm of discrete-event systems, the property of diagnosability has been defined, and several algorithms for testing diagnosability have been presented. Diagnosability relates to the ability for all possible faults to be correctly detected or inferred within a finite amount of time from their occurrence.

WATERS (Waikato Analysis Toolkit for Events in Reactive Systems) is a software toolkit for the creation and analysis of discrete-event systems. In this report pre-existing existing algorithms for the verification of diagnosability and the implementation of one of these algorithms into WATERS are discussed. The created implementation can verify the diagnosability of a discrete-event system in polynomial time with respect to its state space and provide a counterexample if the discrete-event system was found to be not diagnosable.

Acknowledgements

I wish to acknowledge my supervisor Robi Malik for the countless conversations that were crucial in helping me understand the concepts discussed in this report and the helpful review of this report in all stages of its creation. I also acknowledge Shyan Duncan for her expertise in grammar and sentence structure that helped in making this report both readable and understandable.

Table of Contents

List of Figures

1. Introduction

Discrete-event systems (DESs) model the behaviour of automated systems so they may be analysed algorithmically to ensure the system will conform to its specifications. A DES contains a set of states, each representing a unique state the modelled system can be in, and a set of events that represent discrete-events that may occur within the system and cause its state to change.

Detecting faults in an automated system and reacting to them in a suitable manner is a crucial requirement for a stable system. As the complexity of these systems increases, the task of ensuring all faults can be correctly identified in a timely manner becomes increasingly difficult, so an algorithmic approach was required. Diagnosability as a property of a DES was first introduced in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995) along with a method for determining the diagnosability of a DES. Diagnosability as defined, relates to the ability for all events that represent a fault in the system to be detected in a finite amount of time. If all possible fault events in a system can be directly observed via some sensor, then the system is inherently diagnosable. However, if there are fault events that cannot be directly observed, for the system to be diagnosable, it must be possible to infer the occurrence of those fault events from the events occurring that can be observed. Using the algorithm presented in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995) as a basis, additional algorithms that can verify the diagnosability of a DES were presented in (Jiang, Huang, Chandra, & Kumar, 2001) and (Yoo & Lafortune, 2002). These algorithms all revolve around the creation of additional automata from the DES, which can then be more easily analysed for structures that indicate the diagnosability of the DES. Although the automata they create, and method they are created by, differ in each of these algorithms, they all rely on the underlying theory defined in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995).

Waikato Analysis Toolkit for Events in Reactive Systems (WATERS) is a software toolkit for the creation and analysis of discrete-event systems, produced by the University of Waikato and Chalmers University of Technology (Åkesson, Fabian, Flordal, & Malik, 2006). WATERS consists of a graphical interface that allows for a DES to be easily created and edited and a library of checks that can verify if the created DES has certain properties. WATERS does not have the ability to verify the diagnosability of a DES. The aim of this project is to implement a check into waters that can determine the diagnosability of a given DES. To achieve this goal the following steps were performed. Pre-existing algorithms for the verification of the diagnosability of a DES were researched. The algorithm most suitable for implementation into WATERS was identified, along with any additional algorithms needed to complete the verification process. A set of DESs with known diagnosability were created in WATERS to serve as test cases. Finally, the chosen algorithm was implemented into WATERS and tested against the test cases to ensure it performs as expected. The implemented verification process can correctly determine the diagnosability of a DES in polynomial-time with respect to the state-space of the DES.

This report begins in section 2 by defining the terms and concepts used in the following sections to discuss the algorithms that verify diagnosability and the implementation of one of the algorithms. A description of each algorithm considered is given in section 3, and their benefits and limitations are discussed in relation to the requirements of their intended use. In section 4, the algorithm that was chosen for implementation is then discussed in in more practical terms as to how it will be used to verify diagnosability. Any further analysis of the data produced by the chosen algorithm that requires an additional algorithm is identified, and a suitable algorithm is found and discussed in section 5. Once a complete method for the verification of diagnosability has been outlined its implementation into WATERS is discussed in section 6. Finally, in section 7, the test cases created to ensure the implementation correctly verifies the diagnosability of a DES, are provided.

2. Preliminaries

2.1 System Model

A Discrete-event system (DES) is a state-based representation of a real-world system. DES's have a finite set of states, each state within the set represents a unique state that the real-world system can be in. A DES is in only one of these states at any given time, starting in a defined initial state, and only changes state in response to a discrete event that can occur within the modelled system. As with the states, all events possible within the modelled system are represented in a finite set. In addition to the state and event sets, a set of possible transitions is also defined. A transition consists of a state that the DES must be in for the transition to apply, an event that is possible from that state, and the state the DES will transition to after the occurrence of the event. A given DES, called G, is formally defined with a finite state machine model $G = (X, \Sigma, \delta, x_0)$ where

- X is the finite set of states;
- Σ is the finite set of events;
- $\delta \subseteq X \times \Sigma \times X$ is the finite set of transitions;
- $x_0 \in X$ is the initial state.

Any state $x \in X$ is reachable from x_0 with a combination of events from Σ via the transitions defined by δ. Σ^* is the set of all event sequences or traces which includes ϵ the zero-length trace. The language $L(G) \subseteq \Sigma^*$ accepted by G is the subset of Σ^* consisting of all traces of events possible within G. For a trace s and an event σ, $\sigma \in s$ means that the event σ is contained at some point within s.

2.1.1 Example DES

Figure 2.1 shows an example DES model of a simple machine that is part of an assembly line in a factory. The state set X for this machine consists of three states: IDLE, WORKING, and DOWN. The initial state x_0 is IDLE. The event set Σ consists of the four possible events *start*, *finish*, *break* and *repair*. The set of transitions δ is then defined so that there exists a transition defining the target state reached for every event possible out of each state.

X = {IDLE, WORKING, DOWN}

Σ = {*start*, *finish*, *break*, *repair*}

δ = {(IDLE, *start*, WORKING), (WORKING, *finish*, IDLE), (WORKING, *break, DOWN*), (DOWN, *repair*, IDLE)}

x_0 = IDLE

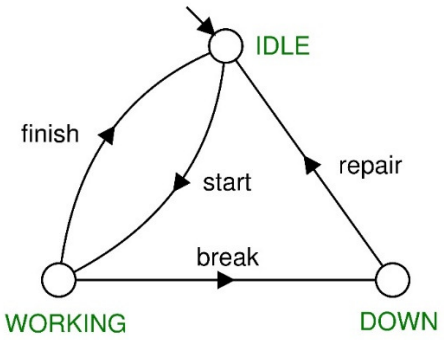

Figure 2.1 Simple Machine

2.2 Partially-Observed Discrete-Event Systems

For most systems, detecting an event and changing state to react to it is a trivial task, simply incorporate a sensor that can detect an event when it happens and use its results. However, it is sometimes not possible for an event to be directly measured, or the system is too large to have a dedicated sensor for every possible event. The DES models of such systems are referred to as partially-observed DES's because only a portion of their event set can be directly observed. Σ, the set of events possible within a partially-observed DES, is therefore partitioned into two subsets; Σ_o the set observable events, those events whose occurrence can be directly detected by the system; and Σ_{uo} the set of unobservable events, which cannot be detected.

2.3 Diagnosability

Problems can arise if a system is partially-observed because failure to detect and properly handle certain events can potentially lead to a total system failure or at least prevent the system from completing a task. Consequently, some mechanism is needed that can infer the occurrence of unobservable-events, using only the information available which is the occurrence of observable-events. This is possible because, although an unobservable event cannot be directly measured, its occurrence can affect what observable events follow it. For example, a printer that cannot detect if its paper tray is empty but can detect a failure to print and all other faults like no toner, can assume it has run out of paper if it fails to print and no other faults are detected.

This brings us to the property of diagnosability which relates to the ability for unobservable-events within a partially-observed DES to be inferred from the trace of observable events. A precise language-based definition of diagnosability is given in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). Essentially a DES is diagnosable if, following any event $\sigma \in \Sigma_{uo}$, there is a combination of observable events, within a finite delay, unique enough to infer from it that σ has occurred. This definition, however, can be relaxed because most unobservable events are of no consequence to the system. If an unobservable event does not prevent the system from completing its task and therefore does not require the system to adjust in response to it, it does not need to be diagnosed. We can then make a subset of unobservable events called fault-events, which consists of only those unobservable events that require a reaction from the system. It is also possible that multiple fault-events require identical reactions from the system. In this case we can make further partitions of the set of fault-events called fault types or fault classes which are groups of fault-events that require the same remedy. With these definitions a DES is said to be diagnosable if it is possible to diagnose, within a finite delay, the fault class of any fault-event that has occurred using the combination of observable events that follow it. Diagnosability is only interesting as a property of partially-observed systems because if all events are observable then the system is inherently diagnosable.

2.3.1 Examples of Diagnosability

Figure 2.2 shows a DES that is not diagnosable and Figure 2.3 shows the same DES that has been modified to become diagnosable. For both systems events *a*, *b* and *c* are observable and the event FAULT which is the fault-event to be diagnosed is unobservable.

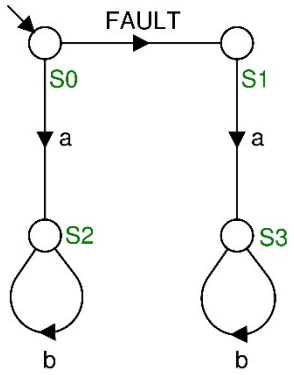

Figure 2.2 Not-Diagnosable DES

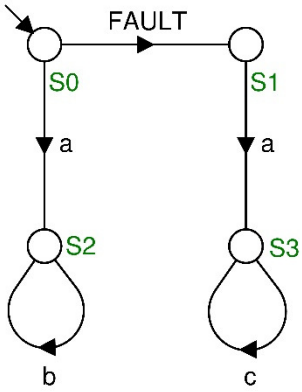

Figure 2.3 Diagnosable DES

In Figure 2.2 the DES starts in the state S0 where the fault event may or may not occur, because the fault event is unobservable there is no way of knowing if the system is in S0 or S1. If event *a* then occurs, the DES knows to change state, but because *a* is possible from S0 and S1, it is now unsure if it should be in S2 or S3. From S2 or S3 the only event possible is *b* with no change of state so the system will never know what state it is in. The combination of observable-events seen will always be event *a* followed by some number of event *b* irrespective of if the fault-event occurs or not, therefore the DES is not diagnosable because there is no unique combination of observable-events that follow the fault-event.

In Figure 2.3 the self-loops on state S2 and S3 happen on different events, this change makes the system diagnosable. Like in Figure 2.2 the system is unsure about if it should be in S2 or S3 after event *a* because the fault event cannot be detected. However, once either event *b* or event *c* occurs, there is only one possible state the system can be in and the occurrence of the fault event can then be inferred.

2.4 Diagnosers

Most algorithms that deal with the diagnosis of a partially-observed DES, revolve around building an automaton called a diagnoser or a similar automaton. A diagnoser is an observer which is a type of automaton that is run in parallel with a system, transitioning on the same event information that the system receives. The diagnoser keeps track of all the possible states the system could be in based off the record of observable events, which are stored as a set attached to each diagnoser state. Each of these potential system states within a diagnoser state has labels indicating the fault class of failure events that have must have occurred for the system to reach that state.

2.4.1 Offline Verification of Diagnosability

Before a diagnoser can be used to diagnose a partially-observed systems fault events, the diagnosability of the system must be verified. Verification of diagnosability is done offline during the design process of the system and shows whether it is possible to diagnose the fault class of all fault events. A diagnoser or similar automaton is built from the system and then analysed for certain structures that indicate the system is not diagnosable, this can be done much more easily than analysing the system directly. Online diagnosis can then be performed but only if the system is found to be diagnosable.

2.4.2 Online Diagnosis

After diagnosability is verified offline and the system is implemented, online diagnosis is performed. Online diagnosis is the process of diagnosing fault-events as the system runs. To achieve this, a diagnoser is run in parallel to the system where the diagnoser changes state when an observable event occurs in the system. The labels attached to the current state of the diagnoser at any given time indicate the fault-class of any fault-events that have occurred. This fault information can then be used to notify the system of a fault-event so the appropriate response can be taken.

3. Available Algorithms

Several algorithms have been proposed for the verification of diagnosability, three of them will be discussed in this section. The process they use for verification is briefly explained along with their benefits, limitations, and suitability for implementation into WATERS.

3.1 Diagnosability as a Property of a Diagnoser

The first method for the construction of a diagnoser from a DES, that can be used to perform online diagnostics was given in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). Additionally, for the first time diagnosability is defined in terms of a diagnoser which allows for the diagnoser to be analysed offline for certain structures that indicate the diagnosability of the DES.

For this algorithm to work some, assumptions about the given DES G must be made:

- The language L(G) must be live or for every state x ∈ X there must be a transition out of that state defined in δ
- There does not exist in G any cycle of unobservable events

Each state of the diagnoser consists of a set of state label pairs. The states are possible states that G could be in given the history of observable events. Attached to each of these possible states is a set of labels that indicate the fault status of G if it is in that state. The set of fault labels that can be assigned to a state of G within a diagnoser state is defined as

$$\Delta_f = \{F_1, F_2, \dots, F_m\}$$

where m is the number of fault classes. The inclusion of F_i in a set of failure labels within a given diagnoser state indicates a fault event from the fault class F_i has occurred at some point. The complete set of possible labels for a state label pair within a diagnoser state is

$$\Delta = \{N\} \cup 2^{\{\Delta_f \cup \{A\}\}}$$

Here the label N is normal, indicating that no fault events could have occurred, and A is ambiguous, if a fault label is labelled ambiguous it is possible that the fault has occurred, but it is also possible that it hasn't. For example, if a diagnoser is in a state that has the

state label pairs {(4, {N}), (8, {F_1}), (10, {A F_2})} it shows that G could be in one of three potential states, it can be in state 4 and no fault has occurred, it can be in state 8 and a fault of the class F_1 has occurred, or it can be in state 10 and a fault of the class F_2 might have occurred. The diagnoser only transitions when an observable-event occurs within G, because of this each diagnoser state only considers those states of G that can be reached via an observable event as possible states G could be in. The set of possible states of G that the diagnoser can consider is defined as

$$X_o = \{x_0\} \cup \{x \in X: x \text{ has an observable event into it}\}$$

This allows the complete set of possible diagnoser states to be defined as

$$Q_o = 2^{X_o \times \Delta}$$

The diagnoser is given the model $G_d = (Q_d, \Sigma_o, \delta_d, q_0)$. q_0 the initial state of the diagnoser is $\{(x_0, \{N\})\}$ where x_0 is the initial state of G. The state space Q_d of the diagnoser is the subset of Q_o reachable from q_0 under δ_d the diagnoser's transition function. States from Q_d take the form $\{(x_1, L_1), \ldots, (x_n, L_n)\}$ where x_i is any state from G that has an observable event into it and $L_i \in \Delta$. The paper goes on to define a label propagation function, label correction function and a range function. These functions are in turn used to define the transition function of the diagnoser in a way so each successive state $q \in Q_d$ starting from q_0 is generated and updated with the new potential states of G, each carrying with it the fault information from the previous diagnoser state and updated with any new fault information.

Necessary and sufficient conditions for diagnosability are then given as properties of the diagnoser. A given diagnoser state $q \in Q_d$ is said to be F_i-certain if, for all state label pairs $(x, L) \in q$ it holds that $F_i \in L$. A diagnoser state $q \in Q_d$ is said to be F_i-uncertain if there exists $(x, L_x), (y, L_y) \in q$ where it holds that $F_i \in L_x$ and $F_i \notin L_y$. An F_i-indeterminate cycle in a diagnoser is a cycle composed of exclusively F_i-uncertain states. An F_i-indeterminate cycle corresponds at least two possible cycles in G all of which have an identical trace of observable events. At least one of these loops follow the occurrence of an F_i failure event and at least one does not. If a cycle exits in G, it is possible that G will stay within the cycle for an infinite amount of time. It can therefore be inferred that if an F_i-indeterminate cycle exists within the diagnoser, G can be in a state where we are uncertain about the occurrence of an F_i fault event for a potentially infinite amount of

time. For diagnosability to hold true for G, G_d must therefore not contain any F_i-indeterminate cycles.

While this is a viable algorithm for the verification of diagnosability, and once the diagnoser is created diagnosability it can be verified in polynomial-time, the method outlined for diagnoser construction has exponential-time complexity with respect to the state space of G. WATERS is used for the creation and testing of DES's, it is only necessary for WATERS to verify diagnosability, so the construction of a diagnoser that can perform online diagnostics can be avoided if another method of verification is used. In addition to this an example is given in the paper of a system G whose diagnoser G_d contains an F_i-indeterminate cycle that has no corresponding cycle following an F_i fault event. Such cases can be handled by computing the strict composition of G and G_d and analysing that instead, but this only adds another step in an already slow algorithm.

3.2 Verification of Diagnosability in Polynomial-Time

After the introduction of the idea that diagnosability can be verified by analysing the structure of a diagnoser in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995), a more purpose-built algorithm for verifying diagnosability was proposed in (Jiang, Huang, Chandra, & Kumar, 2001). This algorithm avoids the building of a diagnoser and the time complexity issues that come with it. This algorithm instead builds a simpler automaton that can still be analysed in the same way to verify diagnosability that can be built in polynomial-time but cannot be used for online diagnosis.

Before the algorithm is given the following are defined. M the observation mask where $M: \Sigma \rightarrow \Sigma_o \cup \{\epsilon\}$ and $M(\epsilon) = \epsilon$. When applied to any trace $s \in \Sigma^*$ and event $\sigma \in \Sigma$ which is a continuation of s, $M(s\sigma) = M(s)M(\sigma)$ where $M(\sigma) = \epsilon$ if $\sigma \in \Sigma_{uo}$ or $M(\sigma) = \sigma$ if $\sigma \in \Sigma_o$. If M is applied to a trace of events $s \in L(G)$ it removes any unobservable events from s so it consists only of observable events. $\boldsymbol{\mathcal{F}} = \{F_1, F_2, \ldots, F_m\}$ is the set of fault-classes and $\psi: \Sigma \rightarrow \boldsymbol{\mathcal{F}} \cup \{\emptyset\}$ is the failure assignment function. The failure assignment function is defined for an event $\sigma \in \Sigma$ as being $\psi(\sigma) = \emptyset$ if $\sigma \notin F_i$ for any $i = \{1, 2, \ldots, m\}$ or $\psi(\sigma) = F_i$ if $\sigma \in F_i$. This algorithm also requires the same assumptions about the system G to be made

as in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995), that is, the language L(G) is live and there does not exist in G any cycle of unobservable events. A method is then outlined that uses M and ψ to obtain a non-deterministic finite state machine $G_o = (X_o, \Sigma_o, \delta_o, x_0^o)$ that accepts the language $L(G_o) = M(L(G))$. States from X_o the state set of G_o take the form {(x,f)} where x is any state from G that can be reached with an observable event, and f is the set of fault-classes that any fault-events along the path from x_0 to x belong to. The event set of G_o is Σ_o which is the set of observable events of G. The set of possible transitions for G_o is $\delta_o \subseteq X_o \times \Sigma_o \times X_o$ and the initial state x_0^o is $(x_0, \emptyset)$. Once G_o is obtained the strict composition of G_o with itself is computed which is referred to as G_d where $G_d = (G_o || G_o)$.

Each state of G_d consists of two states of G_o, both of which contain a state that G could potentially be in given the trace of observable events. Each state of G_d is therefore limited to only two potential states of G as opposed to the diagnoser from the previous algorithm which has no limit. One would initially assume that, because any diagnoser state with more than two potential states of G must correspond to multiple states in G_d to convey the same information, G_d would have more states than the diagnoser for the same system. This is not the case however, if the two states contained within a state of G_d are among a different set of potential states G at another point, that state of G_d can be reused where a new state would be required in a diagnoser. After G_d is created it can then be analysed to verify the diagnosability of G in a similar manner to a diagnoser. Following the same logic as for F_i-indeterminate cycle in a diagnoser, if a cycle exists in G_d where, for every state in the cycle $x_d = ((x_1, f_1), (x_2, f_2))$ and any $F_i \in \boldsymbol{\mathcal{F}}$ it holds that $F_i \in f_1$ and $F_i \notin f_2$, then G is not diagnosable.

This algorithm is more efficient with respect to time than the previous algorithm for producing an automaton that can be analysed to verify diagnosability. However even though for most systems G_d is smaller than a diagnoser, due to the way fault labels are accumulated, as more fault-events occur along a trace the set of fault labels attached to all successive states grows larger and larger. This has the potential to reduce the maximum size of a system that can be run on a given machine by using more memory than is necessary.

3.3 An Alternative Polynomial-Time Solution

Not long after the algorithm given in (Jiang, Huang, Chandra, & Kumar, 2001) was published, a similar algorithm is given in (Yoo & Lafortune, 2002). The algorithm revolves around the construction of automata called verifiers which also require polynomial-time with respect to the state space of G to be built. A new verifier is constructed for each fault-class F_i which is referred to as the F_i-verifier. Each F_i-verifier is individually analysed to verify that G is diagnosable with respect to F_i.

Like the two previous algorithms, this algorithm requires some assumptions about the system G to be made. That is, the language L(G) is live and there does not exist in G any cycle of unobservable events. Like G_d from the algorithm presented in (Jiang, Huang, Chandra, & Kumar, 2001), an F_i-verifier tracks possible states G may be in based off the history of observable events, each verifier state containing two of the possible states of G, each with an attached fault label. The set of possible labels for an F_i-verifier is simply $L_i = \{N, F_i\}$ where N is normal meaning no fault from the class F_i has occurred, and F_i means an F_i fault event has occurred. The verifier for an F_i fault class is defined as $V_{Fi} = (X_{VFi}, \Sigma, \delta_{VFi}, x_0^{VFi})$. The initial state of V_{Fi} is $x_0^{VFi} = (x_0, N, x_0, N)$ where x_0 is the initial state of G. The complete set of possible states of V_{Fi} is $X_{VFi} = (X \times L_i \times X \times L_i)$, where X is the state set of G. However, only the states of X_{VFi} reachable from x_0^{VFi} under δ_{VFi}, the transition function of V_{Fi}, are considered. The F_i-verifier is constructed by generating successor states of x_0^{VFi} which in turn have their successors generated repeating until no new states are found. Successors of a given verifier state $x_v = (x_1, L_1, x_2, L_2)$ are generated by using the non-deterministic transition function $\delta_{VFi}(x_v, \sigma)$ for every σ possible out of states x_1 and x_2 given in δ. As stated earlier only the reachable states of X_{VFi} are considered. Each of the successor states generated by $\delta_{VFi}(x_v, \sigma)$ require from δ one or both target states x_1' and x_2' reached from x_1 and x_2 via σ. If a required δ entry does not exist, the new state is not reachable and is discarded. Additionally, if G is non-deterministic all possible values of x_1' and x_2' given in δ must be considered.

The successor states generated by $\delta_{VFi}(x_v, \sigma)$, where $x_v = (x_1, L_1, x_2, L_2)$ and σ is an event of G, are defined as follows;

If $\sigma \in F_i$ then the successor states generated by $\delta_{VFi}(x_v, \sigma)$ are:

$((x_1'), F_i, x_2, L_2)$ only if $(x_1, \sigma, x_1') \in \delta$

$(x_1, L_1, (x_2'), F_i)$ only if $(x_2, \sigma, x_2') \in \delta$

$((x_1'), F_i, (x_2'), F_i)$ only if $(x_1, \sigma, x_1') \in \delta$ and $(x_2, \sigma, x_2') \in \delta$

If $\sigma \in \Sigma_{uo} \backslash F_i$ then successor states generated by $\delta_{VFi}(x_v, \sigma)$ are:

$((x_1'), L_1, x_2, L_2)$ only if $(x_1, \sigma, x_1') \in \delta$

$(x_1, L_1, (x_2'), L_2)$ only if $(x_2, \sigma, x_2') \in \delta$

$((x_1'), L_1, (x_2'), L_2)$ only if $(x_1, \sigma, x_1') \in \delta$ and $(x_2, \sigma, x_2') \in \delta$

If $\sigma \in \Sigma_o$ then the successor state generated by $\delta_{VFi}(x_v, \sigma)$ is:

$((x_1'), L_1, (x_2'), L_2)$ only if $(x_1, \sigma, x_1') \in \delta$ and $(x_2, \sigma, x_2') \in \delta$

In other words, to obtain all successors of a state $x_v = (x_1, L_1, x_2, L_2)$ from a verifier for the fault class F_i, the verifier transition function $\delta_{VFi}(x_v, \sigma)$ is applied to x_v with all events σ possible out of x_1 or x_2 defined in δ. δ_{VFi} generates successor states based off the requirement that each verifier state consists of two states that G can be in based off the record of observable events. If $\sigma \in \Sigma_o$, for a valid verifier state to be reached from x_v via σ both x_1 and x_2 must have a transition on σ defined in δ. If this is the case the successor state that can be reached from x_v via σ is $x_v' = ((x_1'), L_1, (x_2'), L_2)$ where x_1' and x_2' are the states reached from x_1 and x_2 via σ, and the labels L_1 and L_2 are carried through from x_v to x_v' because the fault status has not changed. If $\sigma \in \Sigma_{uo}$ then the occurrence of σ does not change the trace of observable events therefore x_1, x_2 or both can transition on σ allowing for potentially three valid verifier states to be reached $((x_1'), L_1, x_2, L_2)$, $(x_1, L_1, (x_2'), L_2)$ and $((x_1'), L_1, (x_2'), L_2)$. Finally, if $\sigma \in F_i$ like for a regular unobservable event there are potentially three verifier states that can be reached, depending on if x_1, x_2 or both can transition on σ. If σ is an F_i fault event the three valid states that might be reached are $((x_1'), F_i, x_2, L_2)$, $(x_1, L_1, (x_2'), F_i)$ and $((x_1'), F_i, (x_2'), F_i)$ where the fault label is set to F_i for any of the states of G that were reached via σ.

Once created the verifier is analysed in a similar way to the automata produced by the previous algorithms. The system G is diagnosable with respect to the fault class F_i if V_{Fi} does not contain any F_i-confused cycles. An F_i-confused cycle is a cycle within V_{Fi} consisting only of F_i-confused states $x_v = (x_1, L_1, x_2, L_2)$ where $L_1 \neq L_2$. An F_i-confused cycle prevents G from being diagnosable via the same logic as for an F_i-indeterminate cycle in a diagnoser from (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995).

While the construction method is different, and a verifier only indicates the occurrence of a fault event from a single fault class, a verifier is very similar to G_d from (Jiang, Huang, Chandra, & Kumar, 2001). Like G_d, a verifier benefits from the size reduction that comes with limiting the number of potential system states stored within each state, however by constructing a new verifier for each fault class the size of the labels attached to each verifier is also limited. This further reduces the memory requirements because once a verifier is constructed and analysed it can be discarded to make space for the next verifier.

3.4 Summary

The three algorithms for the verification of diagnosability presented in this section are all based around the creation of an automaton or automata from the DES in question. These automata are analysed for certain structures that indicate the diagnosability of the DES. The automata produced, and their method used for their creation differ in each algorithm, but they all rely on the underlying theory defined in (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). By defining diagnosability in terms of a diagnoser, the possibility for simpler automata to be created and analysed for diagnosability, using the same logic as for a diagnoser, was established. These simpler automata cannot be used for online diagnosis like a diagnoser but can be built much more efficiently than a diagnoser if online diagnosis is not required. Of these algorithms, the algorithm given in (Yoo & Lafortune, 2002) was identified as the most applicable for implementation in WATERS, for reasons that will be discussed in the next section.

4. Chosen Algorithm

The algorithm for the verification of the diagnosability of a DES that was chosen for implementation into WATERS, is the algorithm described in (Yoo & Lafortune, 2002). In this section the reasons for this algorithm being chosen will be discussed along with a more practical description of how the algorithm will be used to verify diagnosability.

4.1 Reasons for Choice

A DES created and tested in WATERS is potentially very large with a high number of states, because of this it is desirable to select an algorithm that allows for the largest possible DES to be verified for diagnosability on a given machine. The maximum size of a DES that can be verified is set by the amount of memory available to store the automaton that will be built and analysed. If the automaton created by a given algorithm is smaller than that of another algorithm it allows for larger DES to be tested on the same machine. The verifiers outlined in (Yoo & Lafortune, 2002) are the smallest automata that can be used to verify diagnosability out of the algorithms considered. This reduction in size is due to the following two factors. Each verifier state only considers two possible states the original DES G can be in, based off the record of observable events. If G could possibly be in more than two states at a given time it is represented with multiple verifier states, each with a unique pair from the set of current potential states of G. This allows for verifier states to be reused if the set of potential states of G at another time is different but has states in common with a previous set of potential states. The other factor is that a separate verifier is built for each fault class, because of this, not only is the potential for very long fault labels in each verifier state removed, each fault label has only two possible values, N or F_i, so it can be encoded as a single bit. In addition to the benefits from verifiers being compact their method for creation can be implemented as a single method that takes a verifier state as input and outputs its successor states. A verifier is simply created by calling that method first on the verifiers initial state and then again on any new states generated, repeating until no new states are found.

4.2 Verifier Creation

Verifying the diagnosability of a given DES G is achieved by creating a new verifier for each fault class F_i called the F_i-verifier. A verifier is a non-deterministic automaton that tracks all possible states G can be in based off observable events. Any point in G where an unobservable event can occur increases the number of possible states G can be in and if any observable event occurs it may reduce the number of possible states. A state from an F_i-verifier consists of two state-label pairs, each state is a possible state of G, and its corresponding label shows if a fault event from F_i occurred along the path to that state. If G has more than two possible states at a given point, the possibilities are represented with multiple verifier states, one for each unique combination of two states from the total set of possible states. Each F_i-verifier is analysed once it is created to find any F_i-confused cycles, if no F_i-confused cycles exist in the F_i-verifier then G is diagnosable with respect to the fault event F_i. G can only be said to be diagnosable if the F_i-verifier for every fault class is found to be free of F_i-confused cycles.

To construct an F_i-verifier G, the verifier's initial state $x_0^{VFi} = \{(x_0, N), (x_0, N)\}$ with x_0 from G, must first be created and added to the verifiers state set X_{VFi}. The successors of x_0^{VFi} are then found by applying the verifier transition function δ_{VFi} defined in section 3.3 on x_0^{VFi} with all events σ possible out of x_0. The resulting states generated by δ_{VFi} take the form $x_v = (x_1, L_1, x_2, L_2)$, they are first checked to see if they already exist within X_{VFi}, if they do not then they are new states and are added to X_{VFi}. Each of the new states that were added to X_{VFi} then have their successors generated by applying δ_{VFi} to them with all events σ possible out of x_1 or x_2, again all new states found are added to X_{VFi}. This process of generating the successors of new states repeats until no new states are found and the verifier is complete.

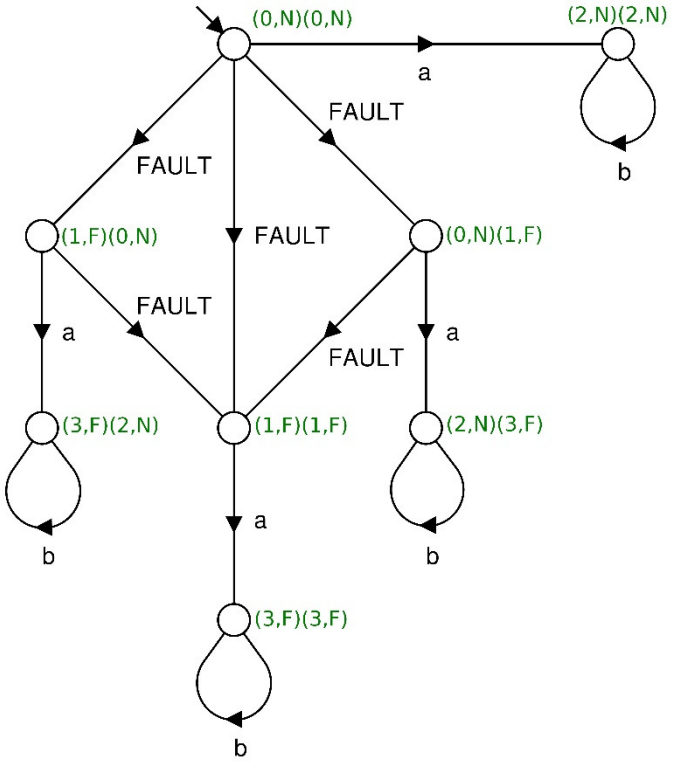

Figure 4.1 Full Verifier

Figure 4.1 shows the full verifier for the fault event "FAULT" from the not diagnosable DES given in Figure 2.2. The verifier shows that the DES is not diagnosable with the F_i-confused loops at state {(2,N), (3,F)} and {(3,F), (2,N)}. While verifiers are smaller than the automata generated by the other algorithms discussed in section 3, this verifier is significantly larger than the original DES with twice the number of states. Upon closer inspection we can see that it contains a considerable amount of redundant information that is not necessary and can be removed. The states {(0,N), (1,F)} and {(1,F), (0,N)} and also {(2,N), (3,F)} and {(3,F), (F,3)} are mirror images of each other, only one state from each of these pairs are needed to convey the same information. This mirroring can be handled by ordering the two state label pairs when a verifier state is generated, the first of the mirrored pair generated will be considered new and added to the state set. When its counterpart is created, and its state label pairs are re-ordered it is identified as already existing and discarded. The direct transition from {(0,N), (0,N)} to {(1,F), (1,F)} is also not necessary because it can be reached via {(0,N), (1,F)}. Due to this δ_{VFi} can be simplified by removing the possible case where x_1 and x_2 can both transition on an unobservable event. Finally, the state {(1,F), (1,F)} and all its successor states are not

needed at all, this is because once a state receives two F_i fault labels and becomes F_i-certain that can never change and, because of how fault labels are carried to successor states, all successor states will also be F_i-certain. If all successor states of an F_i-certain state will also be F_i-certain an F_i-confused cycle can never be reached from it. So, if a new verifier state found is determined to be F_i-certain it can be discarded because no further states of interest can be reached from it.

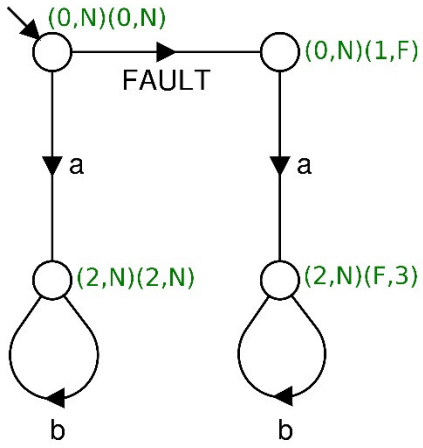

Figure 4.2 Verifier with no Redundancy

Figure 4.2 shows the verifier for the fault event "FAULT" from the DES given in Figure 2.2 that is generated and had all redundant states discarded. It is half the size of the full verifier but still contains all relevant information for the verification of diagnosability. The verifier still contains an F_i-confused cycle on at state {(2,N), (3,F)} that shows the original DES is not diagnosable.

4.3 F_i-Confused Cycles

An F_i-confused cycle in an F_i-verifier is a cycle consisting of only F_i-confused states, an F_i-confused state being any state $x_v = (x_1, L_1, x_2, L_2)$ where $L_1 \neq L_2$. If the F_i-verifier for a DES G contains an F_i-confused cycle, it corresponds to two cycles within G, one following an F_i fault event and the other not. The traces leading to these two cycles in G, and the cycles themselves, are identical when only viewing the observable events and are thus

identical from G's perspective. G can transition into these cycles but there is no way to know which of the two cycles it should be in, and therefore because only one of these cycles are reached via an F_i fault event there is no way to know if the fault event has occurred. There may be a possible trace of events out of one of these two cycles with unique observable events that allows the occurrence of the F_i fault event to be diagnosed. However, it is also possible that G stays within these cycles, in a state were the occurrence of an F_i-fault event is not known. Therefore, it cannot be guaranteed that the F_i-fault event will be diagnosed within a finite amount of time. Diagnosability requires the fault class of a fault event that has occurred to be diagnosed in a finite amount of time, due to this constraint if the F_i-verifier for G contains an F_i-confused cycle then G is not diagnosable with respect F_i.

While the method for constructing a verifier is outlined in (Yoo & Lafortune, 2002) along with the definition of an F_i-confused cycles, no method for finding F_i-confused cycles within verifiers given, so some algorithm that can do this is needed. The algorithm required only needs to be able to identify cycles within the verifier, once identified, a cycle can be then checked to see if it is an F_i-confused cycle. If a verifier state receives an F_i label, it is carried through to all successor states and cannot be removed. Because of this, if a successor of an F_i-confused state receives another F_i label and becomes F_i-certain there is no possible combination of events that will cause the F_i-certain state to transition back to the parent F_i-confused state. Likewise, if the successor of a verifier state with two N labels receives an F_i label and becomes F_i-confused there is no way for it to transition back to the state with two N labels. Due to this persistence of fault labels, if any state contained within a cycle in an F_i-verifier is F_i-confused, we can be certain that all other states within the cycle are F_i-confused and therefore the cycle is an F_i-confused cycle. So, if an algorithm is implemented that can find cycles within a verifier, the cycles found can easily identified as F_i-confused by checking if any state contained within the cycle is F_i-confused.

5. Tarjan's Algorithm

Tarjan's algorithm for strongly connected components, which was outlined in (Tarjan, 1972), is an algorithm designed for use on a directed graph but has since been used in many different applications. When run on a directed graph it partitions the set of vertices into the graph's strongly connected components or SCC's. An SCC is a subgraph where every vertex within it can reach every other vertex, this is essentially a cycle with the exception that a single vertex can form an SCC without the need for a self-loop. Tarjan's algorithm is the algorithm chosen for the task of finding F_i-confused cycles in verifiers. A verifier is essentially a directed graph, so the algorithm can be implemented and run on a verifier with no modifications, however once an SCC is found in a verifier further work must be done to determine if it is an F_i-confused cycle. Unlike most other algorithms that could be used to find cycles, Tarjan's algorithm only requires the graph to be explored through forward transitions, this is advantageous for reasons that will be discussed in section 6.

5.1 Finding Strongly Connected Components

Tarjan's algorithm works by performing a recursive depth-first search or DFS on the directed graph starting at an arbitrary vertex, exploring as far as it can along a single branch before returning to a vertex with an unexplored successor and continuing the search. Figure 5.1 shows the algorithm, the method `run()` takes as input a directed graph `g` and returns a set of SCC objects which are collections of vertices. Each time the DFS is started on a vertex the algorithm visits every vertex that is reachable from that entry point. Because no single vertex is guaranteed to be able to reach every other vertex in the graph, the DFS may then need to be restarted multiple times on any remaining unvisited vertices for the full graph to be explored. Lines 8-12 iterate over every vertex `v` in `g`, if `v` has not already been visited, another DFS is started with `v` as the entry point. The algorithm and DFS is performed by the method `findSCC()`, it is first called on the entry point and then is recursively called on each unexplored successor as it is discovered. When `findSCC()` is run the vertex `v` that it was given is marked as visited, pushed onto a stack, and assigned an index in the order it was discovered and a lowlink value which is

initially its index. Lines 20-27 then continue the algorithm by iterating over each successor `s` that can be reached from `v`. If `s` has yet been visited then `findSCC()` is called on `s`, when it returns, if the lowlink of `s` is lower than the lowlink of `v` then `s` and `v` are members of the same SCC and the lowlink of `v` is updated. If `s` is currently on the stack, then `s` is the entry point of an SCC that `v` is a member of, so the lowlink of `v` is set to the index of `s`.

The result of this updating of lowlink values is if the search enters an SCC, the entry point or root of the SCC is pushed to the stack, as the search continues through the SCC all other members are pushed to the stack on top of the root and given index and lowlink values larger than that of the root. Once the last member of the SCC is discovered it will be found to have a successor (the root of the SCC) already on the stack, it is then given the index of the root as its lowlink. The index of the root is smaller than the other members lowlink so as the search returns through the SCC, the root's index is propagated back as the lowlink of all members of the SCC. Once the search through all successors of `v` completes, if the lowlink of `v` remains equal to its index then `v` is the root of an SCC and all vertices above `v` in the stack are the other members of that SCC. If `v` is an SCC root, then it and all other members are removed from the stack and stored so the algorithm can continue.

```
1 TarjansAlgorithm{
2     Int i;
3     Set<SCC> sccs;
4     Stack stack;
5
6     Set<SCC> run(Graph g){
7         i = 0;
8         foreach(Vertex v in g){
9             if(!v.visited){
10                findSCC(v);
11            }
12        }
13        return SCCs;
14    }
15
16    void findSCC(Vertex v){
17        stack.push(v);
18        v.lowlink = v.index = i++;
19        v.visited = true;
20        foreach(Vertex s in v.successors){
21            if(!s.visited){
22                findSCC(s);
23                v.lowlink = min(v.lowlink, s.lowlink);
24            }else if(stack.contains(s)){
25                v.lowlink = min(v.lowlink, s.index);
26            }
27        }
28        if(v.lowlink == v.index){
29            SCC scc;
30            Vertex m;
31            do{
32                m = stack.pop();
33                scc.add(m);
34            }while(v != m);
35            sccs.add(scc);
36        }
37    }
38}
```

Figure 5.1 Tarjan's Algorithm

5.2 SCC's and F_i-Confused Cycles

A verifier is essentially a directed graph, so Tarjan's algorithm can be used directly on a verifier to find SCCs. However, an SCC in a verifier is not equivalent to an F_i-confused cycle. If we wish to use this algorithm to find F_i-confused cycles, the steps needed to verify if an SCC is an F_i-confused cycle must be determined. The first step is to decide if an SCC found in a verifier is even a cycle. Any vertex within an SCC can reach every other vertex, because of this, if an SCC in a verifier consists of more than one state it must form a cycle. However, a single vertex can form an SCC by itself because it can inherently reach itself,

if an SCC in a verifier consists of a single state there must be a transition defined that allows the state to transition back to itself for it to be a cycle. When an SCC is discovered, the root and all other members are on top of the stack maintained by the algorithm. If there are no other vertexes above the root on the stack, then the root is the only member of the SCC. Therefore, when an SCC is discovered in a verifier, if there is states above the root in the stack then the SCC is a cycle. If there are no states above the root, the SCC is only a cycle if the root state has a transition to itself. If an SCC in a verifier is determined to be a cycle, it can then be checked if it is an F_i-confused cycle. As discussed in section 4.3, if any state within a confirmed cycle is an F_i-confused state then the cycle is an F_i-confused cycle. Therefore, if the root state of an SCC in a verifier, that has been confirmed to be a cycle, is an F_i-confused state then an F_i-confused cycle has been found.

6. Implementation

A complete method for verifying the diagnosability of a DES has been determined and its implementation can now be discussed. The implementation was written in Java and implements a pre-existing interface for a verifier of a DES property. The interface used takes as input a data structure that represents the DES to be verified and outputs a Boolean result which is true if the property is satisfied. In addition to the Boolean result, if the property is not satisfied, a counterexample can be provided which consists of a trace of events that shows why the property is not satisfied.

6.1 Memory Overhead Restrictions

As mentioned in section 4.1 it is desirable for the implemented algorithm to be able to be run on a DES with a large state set. The main constraint on the size of DES that can be verified is the amount of memory available to store the information needed at any given time. Therefore, if less memory is used to store the relevant information the maximum size of a DES that can be verified of a given machine is increased. The algorithm in (Yoo & Lafortune, 2002) was selected because, of the algorithms considered, the verifiers it creates are the smallest automata that can be used to verify diagnosability. Also, as discussed in section 4.2 certain verifier states can be discarded as they are created without affecting the verification process. This is because they either do not contain any new information in the case of a mirrored state, or no useful information can be reached from them in the case of F_i-certain states. This careful selection of the algorithm and information created by the algorithm helps to ensure only essential information is stored. However, to further reduce the memory overhead that information needs to be stored as efficiently as possible.

A verifier $V_{Fi} = (X_{VFi}, \Sigma, \delta_{VFi}, x_0^{VFi})$ consists of a set of states X_{VFi}, a set of events Σ, the set of verifier transitions δ_{VFi} and an initial state x_0^{VFi}. Σ is the set of events from the DES G being tested and is already stored in the data structure that was given to the algorithm and x_0^{VFi} can be easily created using the initial state of G whenever it is needed so does not need to be stored. X_{VFi} holds verifier states after they are created and is used to check a new verifier state against to see if it already exists. A verifier state takes the form $x_v =$

(x_1, L_1, x_2, L_2) where x_1 and x_2 are states of G, and L_1 and L_2 are labels that have two possible values N or F_i. In WATERS states of G are given a 32bit integer identifier which can be used to represent x_1 and x_2, and since the labels only have two possible values they can each be stored in a single bit. The identifiers for states of G are always a positive value, the most significant bit of each identifier, which is used in Java to denote the sign of the value, is therefore not used. This means the most significant bit of each of the two integers that represent x_1 and x_2 can be used to store their respective labels. Finally, the two 32bit values that represent both state-label pairs can be concatenated together into a single 64bit long that represents an entire verifier state. X_{VFi} can therefore be efficiently stored a hash map of primitive longs to allow for quick look up to see if a verifier state already exists.

This leaves δ_{VFi}, the complete set of possible transitions contained within δ_{VFi} is ($X_{VFi} \times \Sigma \times X_{VFi}$). This complete set combined with the fact the transition function used to generate the transitions is non-deterministic makes it easy to see that δ_{VFi} will likely require a lot more memory to be stored than X_{VFi}. δ_{VFi} will be used by Tarjan's algorithm to traverse the verifier and search for F_i-confused cycles. Tarjan's algorithm only requires a graph be traversed once in the forward direction via a depth-first search, unlike other algorithms that find cycles, which also require a backwards traversal. Because the transitions of δ_{VFi} are only needed for this one-time traversal they do not need to be stored. The transitions can be generated using the transition function as Tarjan's algorithm traverses the verifier and then discarded. This can be implemented easily because the depth-first search in Tarjan's algorithm traverses a graph by exploring the successors of a vertex and the transition function creates transitions by generating successors of a state. Using this method, the state set X_{VFi} is also generated as Tarjan's algorithm explores the verifier. Although X_{VFi} still needs to be stored as it is generated to check if a generated successor state already exists, if a F_i-confused cycle is found, the remaining states of the verifier do not need to be generated. X_{VFi} will therefore, only be stored in its entirety if Tarjan's algorithm explores the entire verifier and finds it to be free of F_i-confused cycles, at which point X_{VFi} can immediately be discarded.

6.2 Generating Successors of a Verifier State in WATERS

A separate F_i-verifier needs to be built and analysed for each fault class F_i of G. A verifier is built by repeatedly generating successor states with the verifier transition function δ_{VFi}, starting at x_0^{VFi} the verifier initial state. Each new state found is added to the set of verifier states and has its successors generated, this repeats until no new states are found. The successors created by δ_{VFi} depend on the transitions defined in δ, the set of transitions for G, and the fault class F_i that the verifier is for. To generate all successor states of a verifier state $x_v = (x_1, L_1, x_2, L_2)$, δ_{VFi} must be applied to x_v with every event possible out of x_1 or x_2 that is defined in δ. WATERS can supply an iterator, which will be referred to as SR, that can iterate over the information in δ. SR can be reset with a state x of G and it will iterate over every event and target state out of x defined in δ, or SR can be reset with a state x and event σ and it will iterate over any target state reached from x via σ. If G is non-deterministic SR will iterate over a single event multiple times giving each target state that can be reached.

Using SR a method, shown in Figure 6.1, can be implemented that takes a verifier state v and a fault class F_i as input and iterates over the events possible out of x_1 or x_2 from v, which it then uses to generate the successor states defined by δ_{VFi}. Any new state reached from v is passed on to a method that handles new verifier states. An SR instance is first reset on line 4 to iterate over every event e possible out of x_1. If e is unobservable then one of two new states are created depending on if e is a member of F_i. However, if e is observable, for a valid verifier state to be reached both x_1 and x_2 must be able to transition on e. Therefore, a second instance of SR must be reset on x_2 and e, only if this second SR produces a target state can a new verifier state be created. Finally, once all events out of x_1 have been iterated over, an SR instance can be reset on x_2. For the events out of x_2 only those that are unobservable produce a new successor state because all observable events that both x_1 and x_2 can transition on have already been considered. The method `newVerifierState()` handles the processing of a newly created verifier state. A new verifier state first has its state label pairs ordered to prevent it being considered a unique state if its mirrored counterpart already exists. The state is then checked if it is an F_i-certain state, if it is it is simply discarded because no F_i-confused cycles can be reached from it. Finally, the state is checked against the set of already

discovered states, if it does not already exist it is a valid new state and is added to the state set.

```
1 findSuccessors(verifierState v, faultClass Fi){
2     SR iterA;
3     SR iterB;
4     iterA.reset(v.x1);
5     while(iterA.advance()){
6         Event e = iterA.getCurrentEvent();
7         State targetA = iterA.getCurrentTargetState();
8         if(e is unobservable){
9             if(e is in Fi){
10                newVerifierState(targetA, new Label(Fi), v.x2, v.L2);
11            }else{
12                newVerifierState(targetA, v.L1, v.x2, v.L2);
13            }
14        }else{
15            iterB.reset(v.x2, e);
16            while(iterB.advance()){
17                State targetB = iterB.getCurrentTargetState();
18                newVerifierState(targetA, v.L1, targetB, v.L2);
19            }
20        }
21    }
22    iterB.reset(v.x2);
23    while(iterB.advance()){
24        Event e = iterB.getCurrentEvent();
25        State targetB = iterB.getCurrentTargetState();
26        if(e is unobservable){
27            if(e is in Fi){
28                newVerifierState(v.x1, v.L1, targetB, new Label(Fi));
29            }else{
30                newVerifierState(v.x1, v.L1, targetB, v.L2);
31            }
32        }
33    }
34}
```

Figure 6.1 Verifier Successor Generation

6.3 Integrating Tarjan's Algorithm

Tarjan's algorithm for strongly connected components was selected to perform the task of finding F_i-confused cycles in an F_i-verifier. Tarjan's algorithm was designed for use on a directed graph and can be used with little modification to find strongly connected components (SCC's) in a verifier. A found SCC can then be easily tested to see if it is a F_i-confused cycle. Although a working solution can be created by using Tarjan's algorithm

as it was designed, the recursive nature of the algorithm produces a risk of creating a stack-overflow if run on a large verifier. It is necessary for the test to be able to verify the diagnosability of as large of a DES as possible. Tarjan's algorithm therefore must be modified to be iterative to allow for a large DES to be safely verified. However, before Tarjan's algorithm is modified to an iterative solution a prototype implementation was created that uses the original recursive algorithm. This prototype was created to provide a proof of concept for, constructing a verifier as Tarjan's algorithm runs, and the process of determining if an SCC is a F_i-confused cycle. Both the recursive prototype, and the final iterative solution, were modelled after a previous implementation of Tarjan's algorithm in WATERS for a conflict check algorithm outlined in (Malik, 2018).

6.3.1 Recursive Prototype

The prototype implementation used Tarjan's algorithm recursively as it was originally designed. It was created to prove that Tarjan's algorithm can be integrated with the process of verifier creation and if it can indeed be used to find F_i-confused cycles. Figure 6.2 shows the structure of the prototype implementation. For simplicity in Figure 6.2 all data associated with a verifier state such as index and lowlink values is shown as properties of a verifier state class. In the actual implementation this data was stored separately from the states.

To verify the diagnosability of a DES G a verifier needs to be created and analysed for each fault class. To do this the fault classes of G must be found, and the verifier initial state, which is the same for every verifier regardless of fault class, must be created. The fault classes are then iterated over, and Tarjan's algorithm is started on the verifier initial state with each fault class by calling the method `explore()`. For its intended use, Tarjan's algorithm potentially needs to be run multiple times because it is started at an arbitrary vertex that is not guaranteed to be able to reach every other vertex. For this use, it is started at the initial state of the verifier which is guaranteed to be able to reach every state. This allows us to run the algorithm a single time on the initial state to explore a whole verifier. There is one exception to this however, if G is non-deterministic then it can have multiple initial states which then result in multiple verifier initial states. If G is non-deterministic then the algorithm must be started multiple times with each verifier

initial state as the start state to ensure the full verifier is explored. If an F_i-confused cycle is found at any point during the exploration of a verifier, `explore()` immediately returns false indicating G is not diagnosable.

Each time `explore()` is called recursively on a verifier state `v`, Tarjan's algorithm operates as normal by assigning an index and lowlink and pushing the state onto the stack. However, before the depth-first search can continue to the successors of `v`, they must be generated because they do not yet exist. The successors of `v` are generated with the process that was discussed in section 6.2. For simplicity in Figure 6.2, this process is represented with a call to a method `getSuccessors()` that takes a verifier state and a fault class as input and returns a set of verifier states that are the successors of the input state. In the actual implementation the process of generating successor states is performed directly in `explore()`. Tarjan's algorithm can then continue as normal, iterating over the set of successors that were found.

Once all successors have been explored, if the lowlink and index of `v` remain equal then it is the root of an SCC and all states above it in the stack are members of that SCC. When the root of an SCC is identified the process of determining if it is a F_i-confused cycle begins. The top state in the stack is popped off, if it is a different state than `v` then the SCC has more than one member and is therefore a cycle. In this case the SCC can be immediately checked if it is an F_i-confused cycle. As discussed in section 4.3, if any verifier state within a confirmed cycle is an F_i-confused state then the cycle is an F_i-confused cycle. An F_i-confused verifier state is any state that has different fault labels in each of its two state-label pairs. If the two fault labels of the root state differ, it is a F_i-confused state, therefore the SCC is an F_i-confused cycle and `explore()` returns false. If the SCC was not found to be an F_i-confused cycle the remaining members of the SCC must be popped off the stack, so the algorithm can continue. If the first node popped off the stack is the same as the root, the root is the only member of the SCC. In this case the root must have a transition to itself for it to be a cycle. To determine this, the process of generating successors is repeated, if the set of successors generated contains the root, then it must have a transition to itself and the SCC is therefore a cycle and can be tested like the previous case to see if it is F_i-confused.

```
1 VerifyDiagnosability{
2     Int i;
3     Stack stack;
4     Set<VerifierState> states;
5
6     Boolean run(DES g){
7         Set<FaultClass> faultClasses = findFaultClasses(g);
8         VerifierState verInit = new VerifierState(g.inital);
9         foreach(FaultClass fc in faultClasses){
10            i = 0;
11            stack.clear()
12            states.clear();
13            states.add(verInit);
14            if(!explore(verInit,fc)){
15                return false;
16            }
17        }
18        return true;
19    }
20
21    Boolean explore(VerifierState v, FaultClass fc){
22        stack.push(v);
23        v.lowlink = v.index = i++;
24        v.visited = true;
25        Set<VerifierState> successors = getSuccessors(v,fc);
26        foreach(VerifierState s in successors){
27            if(!s.visited){
28                if(!explore(s,f)){
29                    return false;
30                }
31                v.lowlink = min(v.lowlink, s.lowlink);
32            }else if(stack.contains(s)){
33                v.lowlink = min(v.lowlink, s.index);
34            }
35        }
36        if(v.lowlink == v.index){
37            VerifierState m = stack.pop();
38            if(m != v){
39                if(m.L1 != m.L2){
40                    return false;
41                }
42                while(m != v){
43                    m = stack.pop();
44                }
45            }else{
46                Set<VerifierState> successors = getSuccessors(m,fc);
47                if(successors.contains(m)){
48                    if(m.L1 != m.l2){
49                        return false;
50                   }
51                }
52            }
53        }
54        return true;
55    }
56}
```

Figure 6.2 Prototype with Recursive Tarjan's Algorithm

6.3.2 Iterative Solution

Once the prototype with Tarjan's algorithm operating recursively was implemented and found to be correctly determining the diagnosability of a DES, it was modified to be iterative. The iterative version of Tarjan's algorithm implemented, was modelled after Algorithm 4 given as an appendix in (Malik, 2018). Algorithm 4 is an algorithm for performing a conflict check in a DES but will work with little modification for this application. The recursion of Tarjan's algorithm is used to keep track of the path taken as it traverses through the graph. For the iterative solution, to simulate recursion, a second stack called the control stack is used, and for distinction the original stack used is now called the component stack. Also, the index assigned to newly discovered states cannot be assigned in the order they are processed because it no longer represents depth-first search order, the index of the state in the component stack is used instead and referred to as its DFS value. As states are processed they are assigned a mode that identifies where in Tarjan's algorithm that state has reached. There are four possible modes.

UNVISITED – States that have not been encountered by the algorithm and do not occur in either stack. This is the mode a confirmed new state has after it is generated as a successor but before it is pushed to the control stack.

OPEN – States that are on the control stack but have not yet been expanded. In the recursive Tarjan's algorithm these states have not yet been encountered and as such they are not on the component stack.

EXPANDED – States that have been expanded or are being expanded by the depth-first search. These states appear in both the control stack and the component stack.

CLOSED – These states have been fully processed and assigned to a SCC. They appear on neither stack.

The iterative implementation is controlled by a procedure called `run()` that takes a DES as input and returns a Boolean result that indicates the diagnosability of the DES. `run()` controls the processing of states in the control stack but all actual processing is

done in the two methods `expand()` and `close()`. The structure of `run()`, `expand()` and `close()` is given in Figure 6.3, Figure 6.4 and Figure 6.5 respectively. In these figures all data associated with a verifier state is accessed as properties of a verifier state class, in the actual implementation this information is stored separately. As in the recursive solution, the `run()` method begins by determining the fault classes of the given DES and creating the initial verifier state. The fault classes are then iterated over where a new verifier is built and analysed for each fault class. For each verifier being made the initial state (or states if G is non-deterministic) has its mode is set to OPEN and it is pushed to the control stack. `run()` then sits in a loop, calling `expand()` or `close()` on the top state of the control stack based off its mode. The loop is only exited when either the verifier has been fully explored and no F_i-confused loops are found, in which case the next verifier is started, or a F_i-confused loop is found, in which case `run()` returns false. A call to `expand()` sets any UNVISITED states found to OPEN and places them on the top of the control stack, the last of which will then be the next state to be expanded. This repeats until an expansion does not encounter any UNVISITED states. The loop then consumes the control stack by removing and closing any EXPANDED states on the top of the stack. This process of expanding any OPEN states and closing any EXPANDED states continues until the control stack is empty. The behaviour of this loop models the behaviour of the recursive depth first search from the original Tarjan's algorithm.

`expand()` roughly performs lines 22-35 from the recursive prototype in Figure 6.2. The state `v` being expanded is pushed to the component stack, it is assigned a lowlink and DFS value, and its successors are then generated and iterated over. Figure 6.4 represents the process of generating successors with a call to the method `getSuccessors()`. In the actual implementation the successors are generated directly in `expand()` and instead of being processed in `expand()` as shown in Figure 6.4, each successor is passed to a separate method when it is created. This separate method processes the single successor it was given by performing lines 6-16 from Figure 6.4. Each successor found is processed based off its mode. If a successor is UNVISITED, then it is changed to OPEN, `v` is recorded as being its parent and it is added to the control stack to be expanded. Because the successor is not actually expanded until the current expansion finishes, the lowlink of `v` cannot be updated here like line 31 of Figure 6.2, and is updated in `close()` instead.

If a successor is EXPANDED however, it is the root of an SCC that `v` is a member of so the lowlink of `v` is updated here. The main difference from the recursive algorithm when processing successor states, is the case where a successor is OPEN, in the recursive algorithm this will be the first time this state is encountered so it needs to be put on the top of the component stack. However, because it is already on the component stack, to prevent duplicates it must first be removed and have its parent updated, before it is added to the top of the stack.

`close()` performs the second half of the recursive algorithm, lines 36 to 54 from Figure 6.2. As in the recursive algorithm if the lowlink of the state being closed remains equal to its DFS value then it is the root of an SCC. An SCC that has been found is tested to see if it is an F_i-confused loop, exactly as in the recursive solution, and if it is, `close()` returns false. The difference here is if `v` is not the root of an SCC it must be part of an SCC whose root is below `v` in the component stack. Because the line 31 from Figure 6.2 that carries the lowlink value up through an SCC couldn't performed in `expand()` it must be done here. This is why the parent of each state is recorded. The lowlink of the parent state of `v` is updated to the minimum value between it and the lowlink of `v`.

```
Boolean run(DES g){
    Set<FaultClass> faultClasses = findFaultClasses(g);
    VerifierState verInit = new VerifierState(g.inital);
    foreach(FaultClass fc in g){
        compStack.clear();
        contStack.clear();
        states.clear();
        verInit.mode = OPEN;
        verInit.parent = verInit;
        states.add(verInit);
        contStack.push(verInit);
        while(!contStack.isEmpty()){
            VerifierState v = contStack.getTop();
            if(v.mode == OPEN){
                v.mode = EXPANDED;
                expand(v,fc);
            }else if(v.mode == EXPANDED){
                contStack.pop();
                if(!close(v,fc)){
                    return false;
                }
            }
        }
    }
    return true;
}
```

Figure 6.3 Iterative Implementation Run

```
expand(VerifierState v, FaultClass fc){
    compStack.push(v);
    v.lowlink = v.dfs = compStack.size;
    Set<VerifierState> successors = getSuccessors(v,fc);
    foreach(VerifierState s in successors){
        if(s.mode = UNVISITED){
            s.mode = OPEN;
            s.parent = v;
            contStack.push(s);
        }else if(s.mode = OPEN){
            contStack.remove(s);
            s.parent = v;
            contStack.push(s);
        }else if(s.mode = EXPANDED){
            v.lowlink = min(v.lowlink, s.dfs);
        }
    }
}
```

Figure 6.4 Iterative Implementation Expand

```
1 Boolean close(VerifierState v, FaultClass fc){
2     if(v.lowlink == v.dfs){
3         VerifierState m = stack.pop();
4         if(m != v){
5             if(m.L1 != m.L2){
6                 return false;
7             }
8             while(m != v){
9                 m = stack.pop();
10            }
11        }else{
12            Set<VerifierState> successors = getSuccessors(m,fc)
13            if(successors.contains(m)){
14                if(m.L1 != m.l2){
15                    return false;
16                }
17            }
18        }
19    }else{
20        v.parent.lowlink = min(v.parent.lowlink, v.lowlink);
21    }
22    return true;
23}
```

Figure 6.5 Iterative Implementation Close

6.3.3 Storing Required Data

An efficient method for encoding verifier states has already been determined. Now that Tarjan's algorithm has been modified into an iterative solution there is a lot of data associated with each verifier state that also needs to be stored efficiently. Helpfully, along with the iterative version of Tarjan's algorithm, the appendix of (Malik, 2018) provides an efficient encoding for this data.

For each verifier state the following needs to be stored. The lowlink and DFS values which are both integer values, a mode which has four possible values, the parent of a state, and the two stacks. The information in each verifier state is represented by a single 64-bit long value. To store integer values associated with a verifier state arrays are used, so an integer index associated with each verifier state is needed. If verifier states are assigned an index in the order they are created, that index can then be used to refer to that state as well as to index an array. Some method is needed to translate an index to a

state or a state to an index, this can be done with an array list of long values and a long to int hash map. The hash map also serves as a state set for the verifier, if the map does not provide an index when given a state then that state is a new state. With state indexes established the component stack can simply be an integer stack. The control stack stores a state index but can also store the index of the parent state. The control stack can then be implemented as an array list of int values with three 32-bit ints per entry, one each for the state and parent indexes and a third for a reference to the entry below it in the stack. This leaves the mode, DFS and lowlink values, all of which can be stored with the addition of a single array list of integers called link that is indexed with a state index. An UNVISITED state can be distinguished by its absence in the verifier state set, OPEN and EXPANDED states can be distinguished by tagging the most significant bit of its corresponding link entry and the link value -1 is reserved to distinguish a CLOSED state. If a state is OPEN and does not yet have a lowlink or DFS value, link instead contains a reference to the entry immediately above this entry in the control stack to facilitate removal of the state from the stack in line 11 Figure 6.4. For EXPANDED states, link contains its lowlink value and its control stack entry contains its DFS value in the place of the index. Because the DFS value is the position of the state in the component stack, it can be used to recover the state's index.

6.4 Counterexample

The verification algorithm presented so far only produces a Boolean result to indicate if the conditions for diagnosability have been met. If a user runs the verification and finds their DES G to be not diagnosable, the task of determining why G is not diagnosable is left up to them. For even a modestly sized DES, this is no trivial task, if it was there would be no need for the verification process at all. To aid the user in the process of determining why G is not diagnosable a counterexample can be provided along with a false result. A counterexample for diagnosability would consist of two traces of events in G, Both traces need to have identical observable events and be infinite in length i.e. end in a loop. One of the traces must contain a fault event from the fault class that was found to be not diagnosable and the other trace must be free of fault events from that fault class. The implementation of the verification algorithm has been carefully designed to use as little memory as possible to increase the maximum size of a DES that can be

verified. Because generating a counterexample is not necessary to the verification process, care must be taken to not use any more memory than was already used.

6.4.1 Depth-First vs Breadth-First Search

If a DES G is found to be not diagnosable with respect to the fault class F_i, an F_i-verifier constructed from it was found to contain an F_i-confused cycle. A verifier tracks two traces of events in G that have identical observable events. An F_i-confused cycle in a verifier combined with the trace that leads to it from the verifier initial state represents two traces in G of infinite length, with identical observable events, where only one contains an event from F_i. These two traces fit the requirements for a valid counterexample to the diagnosability of a DES. Therefore, if the trace leading to a found F_i-confused cycle is recovered, it and the cycle can be deconstructed into the two traces of G to serve as a counterexample. This would provide a valid counterexample. However, because Tarjan's algorithm, the algorithm used to find F_i-confused cycles, is based around a depth-first search, the trace taken to reach the cycle is not guaranteed to be the shortest trace possible. The traces given as a counterexample are intended to be read by a user, so a trace generated by a depth-first search is often unusably long.

To find the shortest possible trace to a known F_i-confused cycle a breadth-first search can be performed on the verifier. However, this also poses a problem. Because the verifier transitions are not stored, the successors of each verifier state are generated as the depth-first search of Tarjan's algorithm runs. If an F_i-confused cycle is found the full verifier is never constructed. It is therefore possible that a DES could be found to be not diagnosable, where there isn't enough available memory to store the full verifier. If a breadth-first search is then used to find the shortest trace to the F_i-confused cycle, it will also require successor states to be generated. If this second search adds more states to the state set as it runs it could potentially fill the available memory and cause the verification to fail when a result has already been found.

As a compromise a breadth-first search can be run that can only consider states that already exist as successor states. This is not guaranteed to find the shortest possible trace to the F_i-confused cycle but will likely produce a much shorter trace than the depth-first

search and in the worst case just produce the same trace as the depth-first search. This shorter path in the verifier that goes to and then through the F_i-confused cycle can then be deconstructed into the two event traces that it represents, which can then be returned as a valid counterexample that will be provided to the user.

6.4.2 Implementing Counterexample generation

To allow for counterexample generation, the method `close()` from Figure 6.5 is modified to store an SCC if it is found to be a F_i-confused cycle. If an F_i-confused cycle is found, `close()` returns the false result to `run()` from Figure 6.3. Before the false result is then returned from `run()` as the result of the verification, a counterexample is generated and stored so it can be accessed by WATERS if the user wishes to use it. The process of generating a counterexample is split into two steps. The shortest path is established, through the pre-existing portion of the verifier to the known F_i-confused cycle, and then through the cycle itself. That path is then deconstructed into the two traces of events from the original DES that serve as the counterexample.

The process of establishing the shortest path to, and then through, F_i-confused cycle is outlined in Figure 6.6. The SCC that was found to be an F_i-confused cycle is provided, and each state in the existing set of verifier states is assigned an index of -1 to indicate it has not been encountered by this process. A breadth-first search, or BFS, is then performed, starting at the verifier initial state, until a state within the provided SCC is found. As each state is processed by the BFS its successors are generated, using the same process of generating successors as in Figure 6.1, however only those successors that already exist in the verifier state set are considered. For each successor found, if it has not already been encountered by the BFS, it is assigned an index in the order it was discovered and added to a queue to be processed. If a successor is contained within the SCC then the path the shortest path to the F_i-confused cycle has been established, that successor is then recorded as being the root of the SCC and the BFS is stopped. A second BFS is then started at the root of the SCC. This search is performed to find the shortest path through the F_i-confused cycle. The states are processed like in the last search, assigning an index to successor states in the order they were discovered, however, only

those successors that are contained within the SCC are considered. The second BFS stops when the root of the SCC is encountered again. Once these two searches have been performed the following is true. If a backwards traversal of the verifier is performed, first through the SCC and then from the SCC back to the verifier initial state. The shortest path that was established by the two searches can be followed in reverse by transitioning to the predecessor with the lowest index that is not -1.

```
1 CounterExample generateCE(Set<VerifierState> scc, FaultClass fc){
2     Queue<VerifierState> bfsQueue;
3     int i = 0;
4     VerifierState sccRoot;
5     verInit.index = i++;
6     bfsQueue.add(verInit);
7     while(!bfsQueue.isEmpty()){
8         VerifierState v = bfsQueue.remove();
9         Set<VerifierState> successors = getExistingSuccessors(v,fc);
10        foreach(VerifierState s in successors){
11            if(s.index == -1){
12                s.index = i++;
13                if(scc.contains(s)){
14                    sccRoot = s;
15                    break;
16                }
17                bfsQueue.add(s);
18            }
19        }
20    }
21    bfsQueue.clear();
22    bfsQueue.add(sccRoot);
23    while(!bfsQueue.isEmpty()){
24        VerifierState v = bfsQueue.remove();
25        Set<VerifierState> successors = getSuccessorsIn(scc,v,fc);
26        foreach(VerifierState s in successors){
27            if(s.index == -1){
28                s.index = i++;
29                if(s == sccRoot){
30                    break;
31                }
32                bfsQueue.add(s);
33            }
34        }
35    }
36    return deconstructTrace(sccRoot,scc,fc);
37}
```

Figure 6.6 Counterexample Establish Path

Now that the shortest path to, and through, the known F_i-confused cycle has been established, it can be deconstructed into the two traces that form the counterexample. Accessing the path involves a backwards traversal of the verifier. To follow the path in reverse, the predecessor of the current state with the lowest index that is not -1 must be transitioned to. Finding this predecessor requires all possible predecessors of a verifier state be generated, a process that has not yet been discussed.

The method for generating predecessors of a verifier state, and selecting the predecessor with the lowest index, used for counterexample creation is outlined in Figure 6.7. When generating successors of a verifier state the iterator SR that was discussed in section 6.2 is used. SR is supplied by WATERS and can iterate over the forward transitions of a DES, WATERS can also supply an iterator which we will call PR that can iterate over backwards transitions. If PR is used instead of SR, the predecessors of a verifier state can be generated with an almost identical process to that given in Figure 6.1 for successor generation. There is only one case that needs to be handled differently. When a successor reached via a fault event from the current fault class is created, the fault label attached to the DES state reached via the fault event is set to F_i (line 10 and line 28 in Figure 6.1). When generating the predecessor of a verifier state that was reached via a fault event, we cannot just set this label back to N because it is possible that the label was already set to F_i from a previous fault event. Therefore, two possible predecessors must be considered, one with the label in question set F_i and the other with the label set to N. When creating the counterexample, the set of predecessors that are considered is restricted to only those that already exist within a given set of states. Before a generated state is added to the set of predecessors, it must be checked against the given set of states. These sets of pre-existing states were compiled by successor generation. When a successor state is created, its state-label pairs are ordered to prevent a mirrored state being considered unique. Therefore, the same must also be done for a generated predecessor to ensure it exists in the state set. This poses a problem, for two contiguous traces in the DES to be obtained from a verifier trace, the state-label pairs corresponding to each of these traces must remain in the same position in every verifier state along the trace. To fulfil this requirement, the ordered counterpart of each predecessor is obtained to check if it exists in the state set, if it does then the original unordered predecessor is added to the set of predecessors. However, before a predecessor added, the event it took to reach the current

state is stored so it can be used in constructing the counterexample. When the complete set of valid predecessors is created, the predecessor with the lowest index that is not -1 can be found and returned. In Figure 6.7 the index of each predecessor is accessed as a property of the state, this should not be possible because the states in the set of predecessors are not the original states that the index was set on. In the actual implementation the indexes are stored separately from the state which allow them to be accessed here.

```
1 VerifierState findPredecessorIn(set<VerifierState> states,
2                                 VerifierState v,
3                                 FaultClass fc){
4     Set<VerifierState> predecessors;
5     VerifierState p1,p2;
6     PR iterA,iterB;
7     iterA.reset(v.x1);
8     while(iterA.advance()){
9         Event e = iterA.getCurrentEvent();
10        State sourceA = iterA.getCurrentSourceState();
11        p1 = p2 = null;
12        if(e is unobservable){
13            if(e is in fc){
14                p1 = new VerifierState(sourceA,new Label(F),v.x2,v.L2);
15                p2 = new VerifierState(sourceA,new Label(N),v.x2,v.L2);
16            }else{
17                p1 = new VerifierState(sourceA,v.L1,v.x2,v.L2);
18            }
19        }else{
20            iterB.reset(v.x2, e);
21            while(iterB.advance()){
22                State sourceB = iterB.getCurrentSourceState();
23                p1 = new VerifierState(sourceA,v.L1,sourceB,v.L2);
24            }
25        }
26        if(p1!=null&&states.contains(p1.getOrdered()){
27            p1.event = e; predecessors.add(p1);
28        }
29        if(p2!=null&&states.contains(p2.getOrdered()){
30            p2.event = e; predecessors.add(p2);
31        }
32    }
33    iterB.reset(v.x2);
34    while(iterB.advance()){
35        Event e = iterB.getCurrentEvent();
36        State sourceB = iterB.getCurrentSourceState();
37        p1 = p2 = null;
38        if(e is unobservable){
39            if(e is in fc){
40                p1 = new VerifierState(v.x1,v.L1,sourceB,new Label(F));
41                p2 = new VerifierState(v.x1,v.L1,sourceB,new Label(N));
42            }else{
43                p1 = new VerifierState(v.x1,v.L1,sourceB,v.L2);
44            }
45        }
46        if(p1!=null&&states.contains(p1.getOrdered()){
47            p1.event = e; predecessors.add(p1);
48        }
49        if(p2!=null&&states.contains(p2.getOrdered()){
50            p2.event = e; predecessors.add(p2);
51        }
52    }
53    VerifierState pred = predecessors.get(0);
54    foreach(VerifierState p in predecessors){
55        if((p.index < pred.index)&&(p.index!=-1)){ pred = p; }
56    }
57    return pred;
58}
```

Figure 6.7 Counterexample Find Predecessor

With a method of generating predecessors defined, the process of deconstructing the path through the verifier into the two event traces for the counterexample can be discussed. This process is controlled by the method outlined in Figure 6.8. The path established in Figure 6.6 is followed in reverse and the two event traces are built backwards step by step. The traces required for the counterexample end in the two loops that are represented by the SCC that forms a F_i-confused cycle. Because the traces are being built in reverse, the process begins by deconstructing the path through the SCC. Starting at the SCC root that was identified in Figure 6.6 a backwards traversal of the SCC is performed until the SCC root is encountered again. At each step, of the possible predecessors of the current state, the predecessor that both, is a member of the SCC, and has the lowest index that is not -1, is found with the method from Figure 6.7. The predecessor obtained, and the current state, form a single transition in the verifier path that is deconstructed with another method that adds the event used to reach the current state from the predecessor to the appropriate trace. Once the path through the SCC has been followed, the backwards traversal continues, this time from the SCC root back to the verifier initial state which has an index of 0. As before, at each step, the predecessor of the current state is found, and the traces are added to. However, this time the predecessors considered are only restricted to those that exist within X_{VFi}, the set of all verifier state that have been generated. When the verifier initial state is reached, the two event traces are complete, they are then reversed because they were constructed backwards, and returned as the counter example that will be presented to the user.

The task of adding the event taken in a single verifier transition to the corresponding event traces in the DES is outlined in Figure 6.9. The predecessor and successor of the transition are provided. The event that the predecessor transitions on to reach the successor is stored in the predecessor. If the event is observable, because a valid verifier transition requires the event to occur in both traces, it can simply be added to both traces. However, if the event is unobservable, it only occurs in one of the traces. Because the state-label pairs of the two states provided were never ordered, the states in each position will always belong to the same trace. Therefore, if the DES state in the same position in the successor state is different to that in the predecessor, the unobservable event must have occurred along the trace corresponding to that position and can be added to it.

As the identified path through the verifier is traversed backwards. All predecessor states generated retain the state-label pair ordering of their successor. This is done to maintain two consistent traces in the DES corresponding to each pair. This reverse traversal is started from the SCC root, so the state-label pair ordering of the SCC root is preserved in all states in the path. The SCC root is the only state contained in the path that wasn't generated as a predecessor and therefore has ordered state-label pairs. The SCC root is a F_i-confused state, meaning one of its state-label pairs has an F_i label and the other has an N label. An F_i label attached to a DES state implies a fault event occurred at some point along the trace of events taken to reach that state. The ordering used for an F_i-confused state puts the state-label pair with the F_i label first. Because the state-label pair with the F_i label is always first in the SCC root and all other states in the verifier path maintain that order. Of the two traces constructed for the counterexample, the trace that corresponds to the first state-label pair position will always contain a fault event and the other trace will not.

```
1 CounterExample deconstructTrace(VerifierState sccRoot,
2                                 Set<VerifierState> scc,
3                                 FaultClass fc){
4     List<Event> faultyTrace;
5     List<Event> nonFaultyTrace;
6     VerifierState cur = sccRoot;
7     VerifierState pred;
8     do{
9         pred = findPredecessorIn(scc,cur,fc);
10        addTraceStep(cur,pred,faultyTrace,nonFaultyTrace);
11        cur = pred;
12    }while(pred != sccRoot);
13
14    while(pred.index != 0){
15        pred = findPredecessorIn(Xvfi,cur,fc);
16        addTraceStep(cur,pred,faultyTrace,nonFaultyTrace);
17        suc = pred;
18    }
19    faultyTrace.reverse();
20    nonFaultyTrace.reverse();
21    return new CounterExample(faultyTrace,nonFaultyTrace);
22}
```

Figure 6.8 Counterexample Deconstruct Trace

```
addTraceStep(VerifierState succ, VerifierState pred,
             List<Event> faultyTrace,
             List<Event> nonFaultyTrace){
    if(pred.event is observable){
        faultyTrace.add(pred.event);
        nonFaultyTrace.add(pred.event);
    }else{
        if(succ.x1 != pred.x1){
            faultyTrace.add(pred.event);
        }else if(succ.x2 != pred.x2){
            nonFaultyTrace.add(pred.event);
        }
    }
}
```

Figure 6.9 Counterexample Add Trace Step

6.5 Complexity

The complexity for the verification process implemented, when applied to a DES G is dependent on three factors, S the number of states in G, I the number of fault classes in G, and E the number of events of G which can be split into E_o the number of observable events, and E_{uo} the number of unobservable events.

The space complexity of the implementation is dependent on the maximum number of verifier states stored for a single verifier. This is the case because although I verifiers are required to be built only one verifier is stored at a time. Each verifier state $x_v = (x_1, L_1, x_2, L_2)$ consists of two states of G, x_1 and x_2, and two labels, L_1 and L_2. Because each label has two possibilities N or F_i, there are 2^2 or 4 possible combinations of labels, when combined with two states of G that gives us $4S^2$ possible verifier states, therefore the algorithm has a space complexity of $O(S^2)$. However, a lot of care went into reducing the linear components of the space complexity, so it seems the shame to not calculate a formula for memory needed to verify diagnosability in the worst-case. There will never be a case where every possible verifier state is stored because F_i-certain states and mirrored states are discarded. Therefore, the number of possible verifier states that could be stored is different for each combination of labels.

- (x_1, N, x_2, N) – $x_1 \leq x_2$. There are $\frac{1}{2}S(S+1)$ possible states stored.
- (x_1, F_i, x_2, N) – S^2 possible states stored
- (x_1, N, x_2, F_i) – No states stored (mirror of previous case)
- (x_1, F_i, x_2, F_i) – No states stored (F_i-certain states)

This means that in the worst-case a stored verifier will contain $\frac{1}{2}S(S+1) + S^2 = 1.5S^2 + 0.5S$ states, this is still a space complexity of $O(S^2)$ but a lot better than $4S^2$. Each verifier state is represented with a 8byte long value which, as discussed in section 6.3.3 is stored as an entry in an array list of 8byte long values and a hash map from a 8byte long to a 4byte int. The hash map is on average half full so to store a single verifier state ((4+8)×2)+8 = 32bytes are needed. The information associated with a verifier is stored as a 4byte int stack entry, a 4byte int array list entry, and three 4byte int entries in another array list. Combined this information requires 4+4+(3×4) = 20bytes, when added to memory required for the state itself, each verifier state requires 52bytes to be stored. This gives us the final equation for the maximum number of bytes needed to verify the diagnosability of a DES with S states, which is $52(1.5S^2 + 0.5S)$.

For the verification process to complete a verifier is built and analysed for each fault class, to analyse a verifier each state must be processed with the methods `expand()` and `close()` from Figure 6.4 and Figure 6.5. `expand()` requires the successors of the current state be generated. The number of possible transitions from a verifier state is E_o+2E_{uo} so in the worst case each state generates $2E$ successors. Each successor is then processed. The removal of one of the successor states from an arbitrary position on the stack on line11 Figure 6.4 at first glance could not be performed in constant time. However, because the position of the stack entry above it was recorded, and each entry has a reference to the entry below itself, this removal can be performed in constant time. All other operations on a successor state or the original state are performed in constant time. For `close()` all operations performed on the state require constant time except for the case where an SCC has only one member and the successors of that member must be generated. Again, in the worst case $2E$ successors are generated. From this analysis the processing of a verifier state has a time complexity of $O(E)$. In the worst case this process must be applied to every verifier state possible, the number of possible verifier

states was calculated in the previous paragraph to be $4S^2$. Therefore, analysing an entire verifier has a time complexity of $O(E \times S^2)$. Finally, a verifier must be built and analysed for each fault class. Therefore, the overall time complexity for the complete verification of the diagnosability of a deterministic DES in the worst case is $O(I \times E \times S^2)$. For a non-deterministic DES, the number successors that can be generated from a single verifier state is $2E \times S^2$. The time complexity for verification of the diagnosability of a non-deterministic DES is therefore $O(I \times E \times S^4)$.

7. Testing

To ensure the implementation correctly determines the diagnosability of a DES, it needed to be run against a set of test cases consisting of DESs with known diagnosability. There was no such set of DESs that had been modelled in WATERS available, so one had to me made. Due to time constraints the set of test cases that could be created was somewhat limited, both in the size of each test case and the total number of test cases. Because of this care needed to be taken to ensure the set of tests created were as diverse as possible. Some of the DESs used as test cases were taken from papers and books that discuss diagnosability and others were made from scratch. Once the prototype implementation was created it was tested to ensure it produces the correct result for every test case. The set of test cases was then used for test-driven development as the prototype was modified into the final implementation. The process of modifying the prototype to the final implementation was split into steps such that after each step the implementation should be in a working state. This allowed for the test cases to be run after each step was completed to ensure it still behaves as expected.

7.1 Diagnosable Test Cases

A diagnosable DES is less interesting than a non-diagnosable DES as a test case but is still important for a diverse set of test cases. The variations of a diagnosable DES that were included in the set of test cases are as follows.

- One possible faulty trace and one possible non-faulty trace that both differ in observable events.
- The faulty trace differs by skipping an observable event that is in the non-faulty trace.
- Multiple possible faulty traces that all differ from the non-faulty trace.
- Multiple possible non-faulty traces that all differ from the faulty trace.
- A fault event possible out of every state along the non-faulty trace.
- Two different fault events contained in the same fault class.

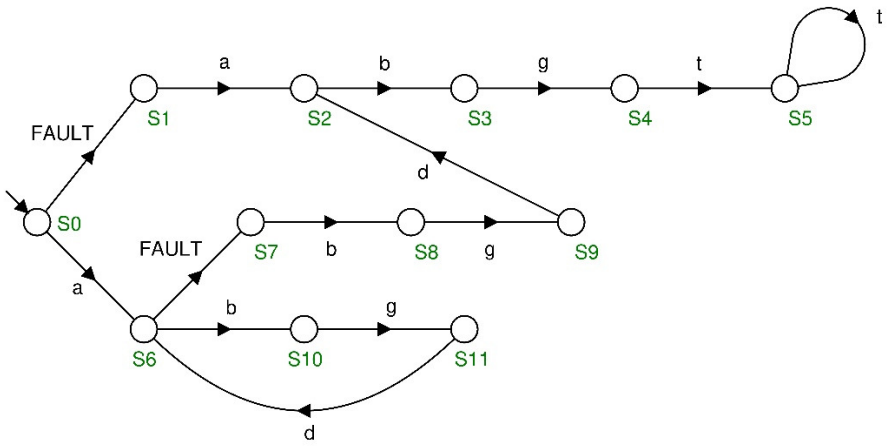

Figure 7.1 Diagnosable Test Case 1

The test case given in Figure 7.1 was taken from (Cassandras & Lafortune, 2009). The event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace of this DES consists of event *a* followed by events *bgc* repeating. The event *FAULT* is possible from state S0 or S6. Every possible trace of events containing the fault has observable events identical to the non-faulty trace up to the point state S4 is reached. When event *t* occurs out of state S4, the faulty trace is no longer identical to the non-faulty trace and the occurrence of event *FAULT* is known.

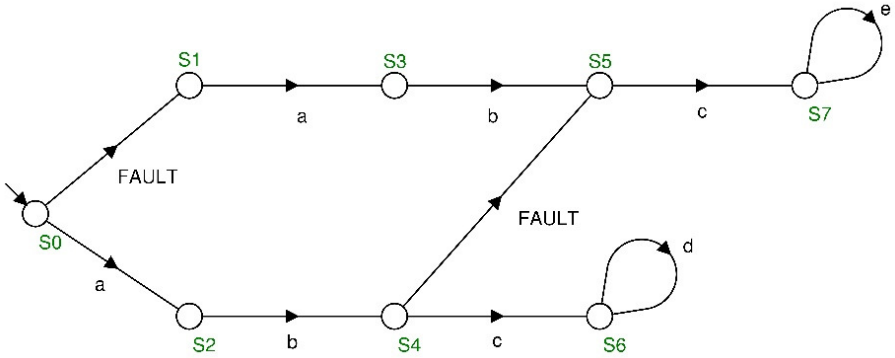

Figure 7.2 Diagnosable Test Case 2

In the test case given in Figure 7.2 the event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of events *abc* followed by event *d* repeating. The fault event can occur from state S0 or S4. If a fault event occurs, the trace of events containing the fault event has identical observable events to the non-faulty trace up until state S7 is reached. When event *e* occurs out of state S7, the faulty trace is no longer identical to the non-faulty trace and the occurrence of event *FAULT* is known.

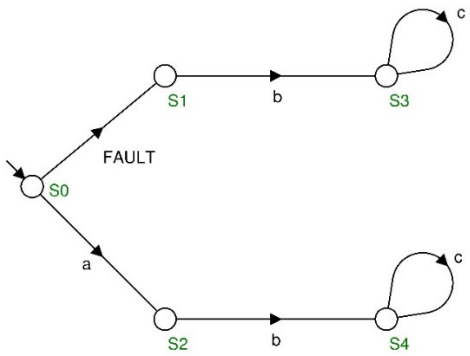

Figure 7.3 Diagnosable Test Case 3

In the test case given in Figure 7.3 event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of events *ab* followed by event *c* repeating. The fault event can occur out of state S0. If the fault event occurs the DES transitions to state S1. From S1, as soon as event *b* occurs without being preceded by event *a,* the occurrence of the event *FAULT* is known.

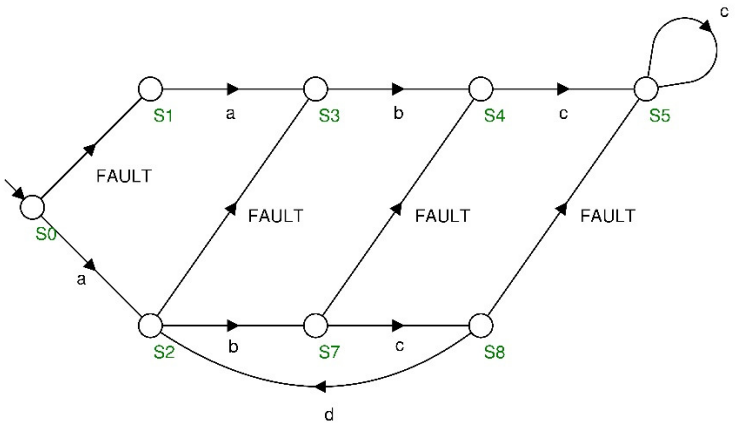

Figure 7.4 Diagnosable Test Case 4

In the test case given in Figure 7.4 event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace of this DES consists of event *a* followed by events *bcd* repeating. The fault event can occur out of any state along the non-faulty trace. Any possible trace of events containing a fault event has identical observable events to the non-faulty trace until state S5 is reached. When event *c* occurs out of state S5 any of the possible faulty traces differ from the non-faulty trace and the occurrence of the event *FAULT* is known.

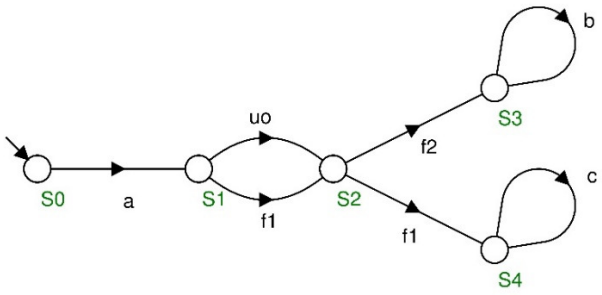

Figure 7.5 Diagnosable Test Case 5

The test case given in Figure 7.5 was taken from (Jiang, Huang, Chandra, & Kumar, 2001). This DES contains two fault events to be diagnosed *f1* and *f2.* Both fault events are members of the same fault class which we will call FC. Event *uo* is an unobservable event and all other events are observable. When the DES reaches the state S1 there are two possibilities, either event *uo* or *f1* can occur, either option transitions to state S2. From S2 either *f1* occurs and S4 is reached, or *f2* occurs and S3 is reached. Up until this point, for all possible traces, the only observable event seen was event *a.* If event *c* occurs the DES must be in state S4. To reach S4, *f1* must have occurred at least once so a fault event from the fault class FC is known to occurred. If event *b* occurs the DES must be in state S3. To reach S3, *f2* must have occurred but it is not known if *f1* occurred. However, because *f1* and *f2* belong to the same fault class FC and it is only necessary to diagnose the fault class of any fault events that occur, the DES is diagnosable.

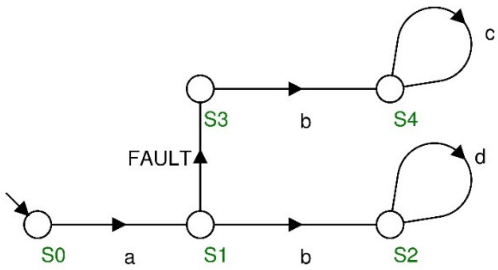

Figure 7.6 Diagnosable Test Case 6

In the test case given in Figure 7.6 event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of events *ab* followed by event *d* repeating. The fault event can occur out of state S1. If the fault event occurs the trace of events containing the fault has observable events identical to the non-faulty trace up to the point state S4 is reached. When event *c* occurs out of state S4 the faulty trace is no longer identical to the non-faulty trace and the occurrence of event *FAULT* is known.

7.2 Non-Diagnosable Test Cases

The following are the test cases consisting of DESs that are not diagnosable. There are more not-diagnosable test cases than diagnosable test cases because there are more unique ways a DES can be not-diagnosable. Also, a counter example is provided if a DES is found to be not-diagnosable. The counterexample gives a better insight into how a result was obtained than just the Boolean result given for a diagnosable DES. By verifying a provided counterexample conforms to the requirements for a counterexample to diagnosability given in section 6.4, we can ensure the algorithm did not merely produce the correct result by chance. To make the test cases diverse as possible the following variations on a not-diagnosable DES were included, some of which have multiple examples.

- The faulty and non-faulty trace differ only in unobservable events.
- Multiple possible faulty traces all with identical observable events to the non-faulty trace.
- Multiple possible non-faulty traces, where at least one has identical observable events to the faulty trace.
- Possible faulty traces that differ in observable events to the non-faulty trace, but at least one possible faulty trace with identical observable events to the non-faulty trace.
- A possible but not required observable event along a faulty trace that would otherwise have identical observable events to the non-faulty trace.
- Multiple fault classes, one diagnosable, the other not-diagnosable.
- Non-deterministic DESs with multiple initial states.

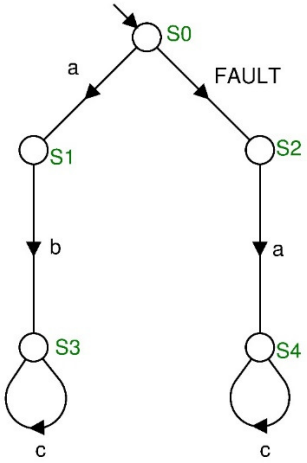

Figure 7.7 Not-Diagnosable Test Case 1

The test case given in Figure 7.7 was taken from (Yoo & Lafortune, 2002). The event *FAULT* is the unobservable fault event to be diagnosed, event *b* is unobservable, and all other events are observable. The non-faulty trace of events consists of events *ab* followed by event *c* repeated. If the fault event occurs out of state S0, the trace of events that follows is event *a* followed by event c repeated. Event *b* in the non-faulty trace is unobservable, therefore the trace of observable events seen by the DES is always event *a* followed by event *c* repeated. Because the DES sees the same trace of observable events regardless of if the fault event occurs the DES is not diagnosable.

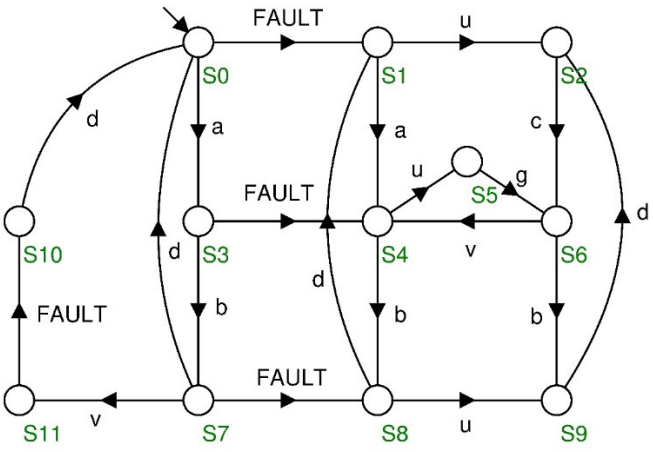

Figure 7.8 Not-Diagnosable Test Case 2

The test case given in Figure 7.8 was taken from (Cassandras & Lafortune, 2009). The event *FAULT* is the unobservable fault event to be diagnosed, events *u* and *v* are unobservable, and all other events are observable. The non-faulty trace of events is *abd* repeated. There are possible traces of events that contain a fault event and have different

observable events to the non-faulty trace that allow for the fault event to be diagnosed. However, the loop from S1 to S4 to S8 and back to S1, results in a possible infinite trace of events that has identical observable events to the non-faulty trace and contains a fault event. Because it cannot be guaranteed that the fault event will be diagnosed in finite time, the DES is not diagnosable.

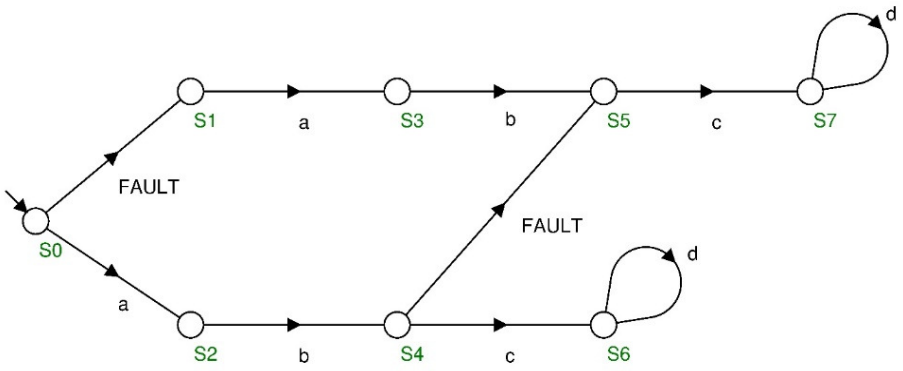

Figure 7.9 Not-Diagnosable Test Case 3

The test case given in Figure 7.9 is the not-diagnosable counterpart to the test case given in Figure 7.2. the event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of events *abc* followed by event *d* repeating. All possible traces of events containing a fault event consist of the observable events *abc* followed by event *d* repeating. Because all traces containing a fault event have identical observable events to the non-faulty trace the DES is not diagnosable.

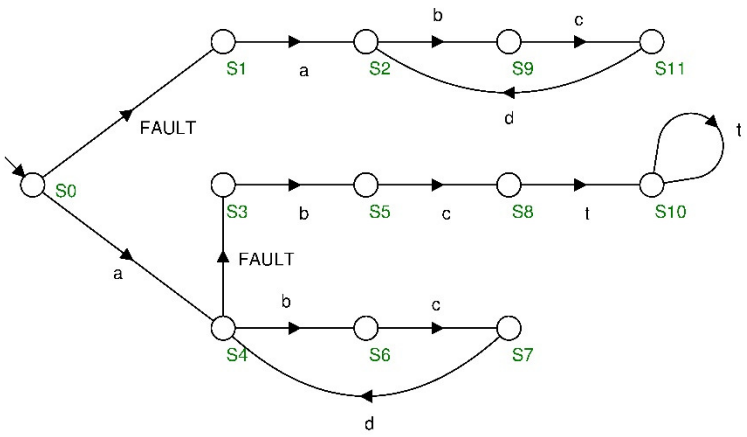

Figure 7.10 Not-Diagnosable Test Case 4

The test case given in Figure 7.10 was taken from (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). The event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of event *a* followed by events *bcd* repeated. If the fault event occurs on state S4 the resulting trace

of events contains different unobservable events to the non-faulty trace and the fault event can be diagnosed. However, if the fault event occurs on state S0 the resulting trace of events does have identical observable events to the non-faulty trace. Therefore, because the fault event cannot always be diagnosed the DES is not diagnosable.

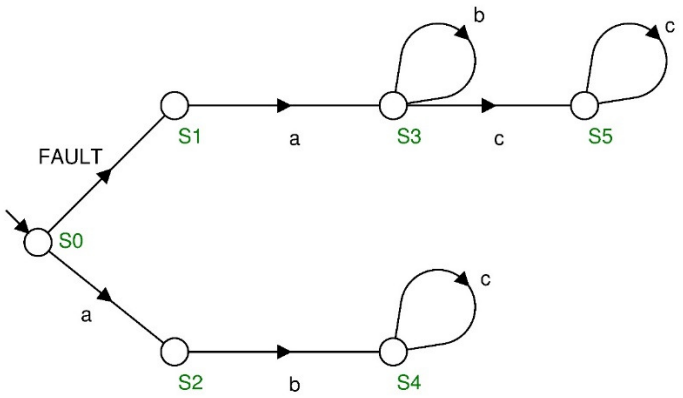

Figure 7.11 Not-Diagnosable Test Case 5

In the test case given in Figure 7.11 the event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. The non-faulty trace consists of events *ab* followed by event *c* repeated. If the fault event occurs in state S0, followed by event *a*, state S1 is reached. From state S1 there are two possibilities. If event *b* occurs from S1, the trace of events will differ in observable events to the non-faulty trace and the fault event can be diagnosed. However, if event *c* occurs from S1 before event *d*, the resulting trace of events will have identical observable events to the non-faulty trace. Therefore, because the fault event cannot always be diagnosed the DES is not diagnosable.

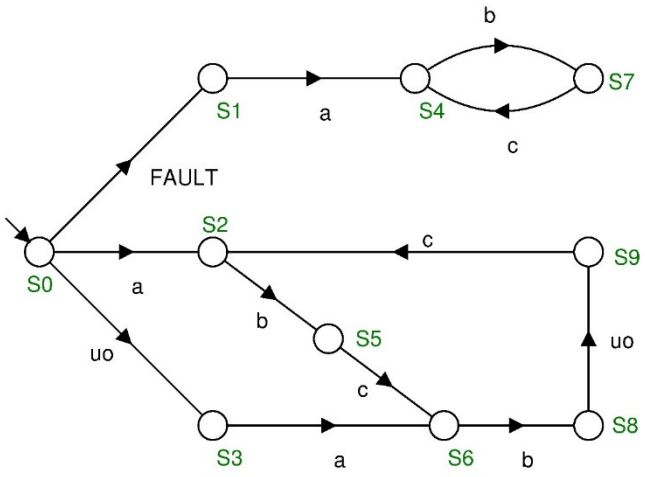

Figure 7.12 Not-Diagnosable Test Case 6

The test case given in Figure 7.12 was taken from (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). The event *FAULT* is the unobservable fault event to be diagnosed, the event *uo* is unobservable, and all other events are observable. There are multiple paths that can be taken, but when only looking at the observable events the non-faulty trace is always event *a* followed by events *bc* repeated. The only possible trace containing a fault event consists of the observable event *a* followed by events *bc* repeated. Because all traces containing a fault event have identical observable events to a non-faulty trace the DES is not diagnosable.

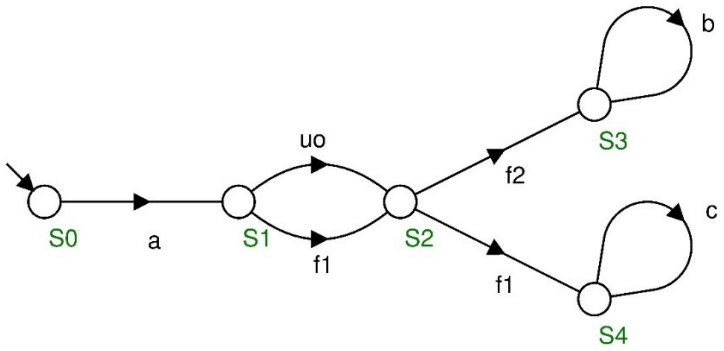

Figure 7.13 Not-Diagnosable Test Case 7

The test case given in Figure 7.14 is the not-diagnosable counterpart to the test case given in Figure 7.5 which was taken from (Jiang, Huang, Chandra, & Kumar, 2001). There are two unobservable fault events to be diagnosed *f1* and *f2* which each belong to a separate fault class, event *uo* is unobservable, and all other events are observable. After event *a* occurs and the DES is in state S1 there are two possible events. Either the fault event *f1* will occur or the event *uo* will occur, both of which cause a transition to S2. From

S2 there are another two possibilities, fault event *f1* can occur transitioning to S4, or fault event *f2* can occur transitioning to S3. Up until this point, for all possible traces, the only observable event seen was event *a.* If event *c* occurs the DES must be in state S4. To reach S4, *f1* needs to have occurred at least once and is therefore diagnosed. However, if event *b* occurs the DES must be in state S3. To reach S3, *f2* must have occurred but it is not known if *f1* or *uo* occurred to reach S2. Once in S3 no further state can be reached and *f1* can never be diagnosed so the DES is not diagnosable with respect to *f1*.

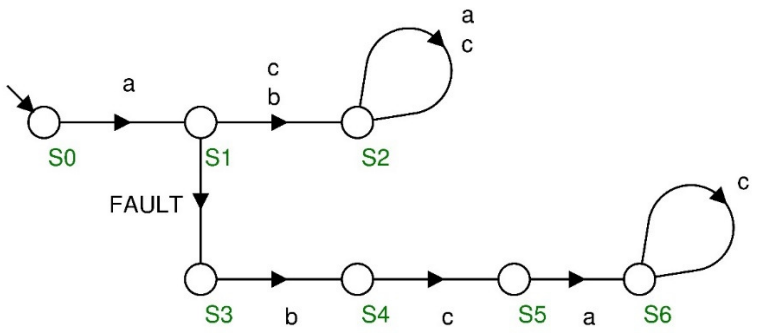

Figure 7.14 Not-Diagnosable Test Case 8

The test case given in Figure 7.14 was taken from (Moreira, Jesus, & Basilio, 2011). The event *FAULT* is the unobservable fault event to be diagnosed and all other events are observable. There are multiple possible non-faulty traces consisting of event *a* followed by either event *c* or *b* which is then followed by either event *a* or *c* repeating. There is one possible trace of events containing a fault event. That trace consists of the observable events *abca* followed by event *c* repeating. Because the trace of events containing a fault event has the same observable events as one of the possible non-faulty traces the DES is not diagnosable.

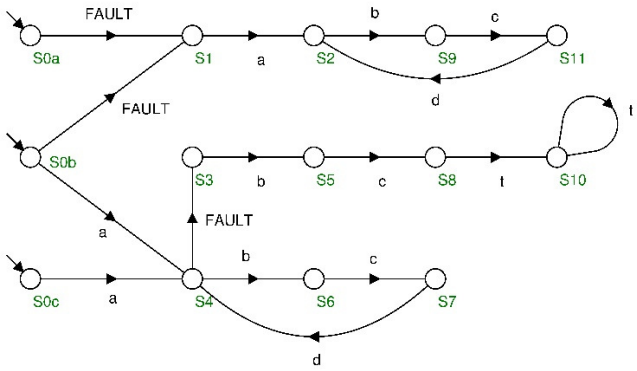

Figure 7.15 Not-Diagnosable Test Case 9

The test case given in Figure 7.15 is a modification of the test case given in Figure 7.10 which was taken from (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995). This test case is not diagnosable for the same reasons as its counterpart. However, this version is a non-deterministic DES with multiple initial states. To handle such a DES the verification process must create multiple verifier initial states to ensure all potential starting states of the DES are considered. This test case was included to ensure this process is performed correctly.

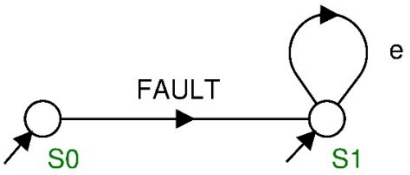

Figure 7.16 Not-Diagnosable Test Case 10

The test case given in Figure 7.16 is another non-deterministic DES with multiple initial states. The event *FAULT* is the unobservable fault event to be diagnosed and event *e* is an observable event. If the DES starts from state S1 the fault event cannot occur, so it is inherently diagnosable. If the DES starts from state S0 the fault event must occur to reach state S1, if event *e* is then seen the fault event had to occur before it and can therefore be diagnosed. However, because the trace of observable events is the same regardless of the start state, and the start state is not known, and the fault event is therefore not diagnosable.

8. Conclusion

In this report, pre-existing algorithms that create automata from a DES, that can then be analysed for structures that indicate the diagnosability of the DES, were discussed. One such algorithm was selected for implementation into WATERS because the automata it creates, known as verifiers, were the most compact automata produced by any of the algorithms considered. In addition to this certain possible verifier states were identified as being not necessary to the verification process and could be discarded to further reduce the memory needed to store a verifier.

Tarjan's algorithm for strongly connected components was identified as a suitable algorithm to analyse each verifier for the described structures, F_i-confused cycles, that indicate the DES they were created from is not diagnosable. Tarjan's algorithm and the chosen algorithm were implemented together into a single process that analyses a verifier as it is created. By integrating Tarjan's algorithm with the process of verifier creation, the need for storing verifier transitions was removed, and the ability for a DES to be found not-diagnosable, without creating an entire verifier, was achieved. The implementation of Tarjan's algorithm, which normally revolves around a recursive depth-first search, was modified to an iterative solution to remove the risk of stack-overflow when verifying the diagnosability of a large DES. Finally, a process for generating a counterexample for a DES that was found to be not-diagnosable was established and implemented.

By careful selection of algorithms, data produced by the algorithms, and encoding for that data, the memory overhead for the implemented verification process was reduced as much as possible. By reducing the memory overhead of the implementation, the maximum size of DES that can be verified for diagnosability on a given machine was increased. The completed test in WATERS can correctly verify the diagnosability of each of the test cases that were created, in polynomial-time with respect to the state-space of the DES, and provides a valid counterexample for those test cases that are not-diagnosable. The test implemented in this report will be made available for users of WATERS in the next release.

8.1 Contribution

The algorithm implemented to produce the verifiers that are analysed to verify diagnosability was a pre-existing algorithm that had been presented in the literature. However, the integration of Tarjan's algorithm with the process of generating verifiers is unique. Because doing so removes the need for storing the verifier transitions, the memory required to store and analyse a verifier is reduced significantly. By reducing the memory overhead for the verification process, the possibility of verifying the diagnosability of large DESs is made available to users with less memory at their disposal. Also, the properties a valid counterexample to diagnosability must have, were defined for the first time along with a process for deconstructing a verifier trace that ends in a F_i-confused cycle into the two traces that form a valid counterexample. Providing a counterexample is extremely useful to a user in determining why their DES is not-diagnosable.

8.2 Limitations

Due to time-constraints the set of test cases that could be created was somewhat limited in both size and diversity. Because it was not possible to test every possible way a DES could be diagnosable or not, there may be some edge cases where the implemented test will provide an incorrect result. However, because the process implemented was based off a pre-existing algorithm, and the set of test cases covered most common possibilities, some level of confidence can be placed on the results the test provides. There are some cases that are known to produce an undefined result because the algorithm implemented makes certain assumptions about the DES being verified. The first of these assumptions is that the language L(G) accepted by the DES is live, that is for every state there is a possible event out of that state. This is a valid assumption to make of a DES and is a property that WATERS can test for and should be tested for before diagnosability. The second assumption is that the DES contains no loops consisting of only unobservable events. This cannot be tested for by WATERS and poses more of a risk of not being true in a DES that is being tested for diagnosability. It should be possible to make modifications to the implementation that allow for this case in the diagnosability verification process. Due to time constraints these modifications were not made but if

they were the implemented test would be more stable in respect to always providing the correct result regardless of the DES it is used on.

9. References

Åkesson, K., Fabian, M., Flordal, H., & Malik, R. (2006). Supremica—an Integrated Environment for Verification, Synthesis and Simulation of Discrete Event Systems. *8th International Workshop on Discrete Event Systems, WODES'06*, (pp. 384–385).

Cassandras, C. G., & Lafortune, S. (2009). *Introduction to Discrete Event Systems.* Springer Science & Business Media.

Jiang, S., Huang, Z., Chandra, V., & Kumar, R. (2001, Aug). A Polynomial Algorithm for Testing Diagnosability of Discrete-Event Systems. *IEEE Transactions on Automatic Control, 46*(8), 1318–1321.

Malik, R. (2018). *The explicit conflict check algorithm implemented in the Waters library.* Working Paper 01/2018, The University of Waikato, Department of Computer Science, Hamilton, New Zealand. Retrieved from https://hdl.handle.net/10289/12069

Moreira, M. V., Jesus, T. C., & Basilio, J. C. (2011, July). Polynomial Time Verification of Decentralized Diagnosability of Discrete Event Systems. *IEEE Transactions on Automatic Control, 56*(7), 1679–1684.

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D. (1995, Sep). Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control, 40*(9), 1555–1575.

Tarjan, R. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing, 1*(2), 146–160.

Yoo, T.-S., & Lafortune, S. (2002, Sep). Polynomial-Time Verification of Diagnosability of Partially Observed Discrete-Event Systems. *IEEE Transactions on Automatic Control, 47*(9), 1491–1495.