Verifying Fault Tolerance and Self-Diagnosability of an Autonomous Underwater Vehicle*

Jonathan Ezekiel and Alessio Lomuscio
Imperial College London, UK
{jezekiel,alessio}@doc.ic.ac.uk

Levente Molnar and Sandor Veres
University of Southampton, UK
{L.Molnar,S.M.Veres}@soton.ac.uk

Miles Pebody
National Oceanography Centre, UK M.Pebody@noc.soton.ac.uk

Abstract

We report the results obtained during the verification of Autosub6000, an autonomous underwater vehicle used for deep oceanic exploration. Our starting point is the Simulink/Matlab engineering model of the submarine, which is discretised by a compiler into a representation suitable for model checking. We assess the ability of the vehicle to function under degraded conditions by injecting faults automatically into the discretised model. The resulting system is analysed by means of the model checker MCMAS, and conclusions are drawn on the system's ability to withstand faults and to perform self-diagnosis and recovery. We present lessons learnt from this and suggest a general method for verifying autonomous vehicles.

1 Introduction

The Autosub6000 [McPhail, 2009] is an autonomous underwater vehicle (AUV) aimed at scientific oceanographic exploration. With an autonomy of up to 8.6 days, and an operating range of 1000km, the Autosub6000 performs Oceanographic seafloor surveys at depths of several thousand meters. The missions Autosub6000 has successfully completed includes 24 hours under sea ice in the Arctic, and a 30km run under an Antarctic ice shelf [McPhail, 2009]. During these missions the vehicle operated beyond communication range of the mother ship, and without hope of rescue if anything went wrong with the vehicle.

AUVs are very costly; their reliability is a key concern. Diagnosing and repairing faults, as well as recovering the vehicle in case of problems can be extremely expensive (easily in excess of $2,000 per hour [Chance, 2003]), when possible at all. Thus, there is a pressing need for key characteristics of these vehicles to be correct. For example, if the vehicle is too deep in the water we need to be certain that the control module will not keep the vehicle diving and endanger it. It is therefore appealing to verify the core AUV behaviours systematically before operation.

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

Studying an AUV such as the Autosub6000 is inherently difficult as it contains several subsystems that communicate with each other, each *autonomously* responsible for different parts of the system, that were not necessarily designed to work together. To analyse their overall behaviour, given the heterogeneous nature of the components, their characteristics, as well as the overall goal of the AUV, it is natural to adopt a *multi-agent* based stance [Wooldridge, 2000] when reasoning about it.

Indeed, significant inroads have been made by the multi-agent systems (MAS) community in terms of specification languages (epistemic logic, ATL/cooperation logics, bounded rationality, etc.) to denote the properties a MAS may have to satisfy. More recently, methodologies and toolkits have been put forward to verify automatically behaviours of MAS, typically by model checking [Clarke *et al.*, 1999]. Model checkers for MAS have been released [Gammie and van der Meyden, 2004; Lomuscio *et al.*, 2009; Niewiadomski *et al.*, 2004] for the community to use. While much of this effort has made a theoretical contribution to the problem of verifying MAS, the approach is largely untested in any reasonably large and realistic scenario. The aim of this paper is to present a methodology built upon existing work and apply it to verify key fault tolerance properties of a real and existing autonomous system, the Autosub6000.

Our starting point is the recent development of a general integrity and fault assessment system (IFAS) for complex autonomous engineering systems [Molnar and Veres, 2009a; 2009b]. The IFAS is intended as a general toolkit to assist system engineers in designing reliable AUVs. In the IFAS approach, a formal model is defined for the AUV by means of Simulink/Matlab descriptions. These are commonplace in engineering and used extensively in the industry. The Simulink model is then automatically refined by means of a discrete abstraction of the hybrid system model, thereby converting into a MAS formalism. This formalism can then be used with automated verification tools such as model checking to verify that the system satisfies safety requirements

However, the methodology above is not generally sufficient when reasoning about AUVs, as in this domain we are typically interested to reason about the operation of an AUV *under degraded conditions*. These aspects have recently been addressed by an approach that combines fault injection [Iyer, 1995] with model checking to verify the correctness of vari-

ous aspects of fault tolerance in MAS [Ezekiel and Lomuscio, 2009]. In contrast to ad-hoc modelling of faulty behaviour, in this approach faults can be automatically injected into a model of a correctly behaving system to create a mutated model which exhibits both correct and faulty behaviour. Temporal-epistemic specifications [Fagin *et al.*, 1995] can then be verified to analyse the correct and faulty behaviour of agents in the mutated model, as well as the knowledge that agents have about the behaviour. This allows for the seamless verification of fault tolerance, recovery from faults, and diagnosability, i.e., whether an unobservable fault can be accurately diagnosed from the observable events of the system [Sampath *et al.*, 1995], which is difficult to specify without epistemic operators in the language. The high level of usability offered by the automatic nature of both the fault injection and the model checking process makes the approach particularly attractive to non-experts in verification such as system engineers [Bozzano and Villafiorita, 2007]. However, until now this approach has never been applied to truly autonomous systems, such as robotic vehicles.

In this paper we suggest a methodology that combines fault-injection with IFAS and use it to verify the Autosub6000 AUV [McPhail and Pebody, 1998]. We find the exercise useful not only in terms of the actual results we obtained that pertain to the AUV in question, but, more generally, because we believe that the methodology we put forward here can largely be adopted when assessing other autonomous systems. We present the background to the methodology in the next Section, the methodology in Section 3, the Autosub6000 AUV in Section 4, and the results in Section 5. In Section 6 we discuss the related work and conclude.

2 Background

Model checking [Clarke *et al.*, 1999] is a widely adopted technique for systems verification. The system S considered for verification is represented by a logical model M_S which encodes all behaviours of the system as computational traces. A specification of a property P is expressed by means of a logical formula φ_P. The model checker establishes whether or not M_S satisfies φ_P (formally, $M_S \models \varphi_P$). The satisfaction relation is implemented as an automatic decision procedure, making model checking attractive for the purpose of verification [Clarke *et al.*, 1999]. In the case of MAS, φ_P is often expressed by using a number of rich modal logics including temporal and epistemic logics [Wooldridge, 2000].

2.1 Interpreted Systems and MCMAS

We assume familiarity with the interpreted systems model as presented in [Fagin *et al.*, 1995]. This models a MAS by taking a collection of local states, actions, protocols, local transitions... The model has been used in several key scenarios including robotics, web services, etc.

As a specification language we take the temporal-epistemic logic CTLK, defined by:

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

In the grammar above $p \in AP$ is an atomic proposition; EX is a temporal operator expressing that there exists a next state in which φ holds; AG expresses that in all runs φ holds globally; $E(\varphi U \psi)$ is a temporal operator expressing that there exists a run in which φ holds until ψ holds; $K_i\varphi$ expresses that *agent i knows* φ; $E_\Gamma\varphi$ expresses that *everybody in group* Γ *knows* φ; $C_\Gamma\varphi$ expresses that *it is common knowledge in group* Γ *that* φ. This syntax, as well as other temporal modalities, can be interpreted on interpreted systems as standard. We refer to [Lomuscio *et al.*, 2009] for details.

MCMAS [Lomuscio *et al.*, 2009] provides ISPL as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulas as specifications of the system. ISPL programs are closely related to interpreted systems; specifically each ISPL program describes an interpreted system. MCMAS supports the verification for all formulas in the language above.

2.2 Injecting Faults into MAS Programs

The first step of a combined fault injection and model checking approach [Bozzano and Villafiorita, 2007; Ezekiel and Lomuscio, 2009] involves *mutating* a model of correct system behaviour by *automatically* injecting faulty behaviour into it. This provides a general technique to automate the process of creating system models containing faulty behaviour in comparison to the lengthy process of manually introducing the behaviour. The output of a mutation step is a model containing correct and faulty behaviour.

We summarise the mutation technique for interpreted systems presented in [Ezekiel and Lomuscio, 2009]. Any agent A of the system can be mutated into a faulty agent A^{F*} ($F*$ denotes mutated) that includes faulty behaviour. A fault injection agent FI implements the timing characteristics of the fault. The faulty behaviour is triggered in the faulty agent whenever an $inject$ action is performed by FI and the correct behaviour is preserved in the faulty agent when the $inject$ action is not performed. A number of timing options can be selected for FI: injecting *constantly*, *randomly*, *after a random start*, *until a random stop*, and *after and until an action has occurred* [Ezekiel and Lomuscio, 2009]. The local states, actions, protocol, and evolution function of the fault injection agent a defined according to these options, which can be combined to create complex timing characteristics of the fault. A mutated set of initial states I^{F*} stipulates that the local state of FI is set to either $notfaulty$ which persists throughout the system run, or to a state in which faults may be injected into the system by FI in the future according to the timing options. The faulty behaviour in the faulty agent A^{F*} is introduced using a number of *mutation rules* that determine how the evolution function t_A is mutated to $t_{A^{F*}}$. These can be applied to mutate the model to represent desired faulty behaviour, e.g., by inverting a particular state of an agent.

A mutated valuation function V^{F*} relates atomic propositions to the local states of each fault injection agent. This can be used to reason about the correct and faulty behaviours of the mutated interpreted system IS^{F*}. For each fault $j \in \{1, \ldots, m\}$ the mutated set of atomic propositions AP^{F*} extends with the propositions $faulty_j$, $injected_j$, $injecting_j$, $stopped_j$. This allows for reasoning about the *persistence* of a fault, i.e., the continued existence or occurrence of a fault during a system run. The proposition $faulty_j$

represents that a fault can be injected during the system run; $injected_j$ expresses that a fault is injected at the current clock tick (a global state describing the system at a particular instant of time); $injecting_j$ denotes that a fault can be injected at the current clock tick; $stopped_j$ describes that a fault has been injected and can not be injected at the current clock tick.

Once a mutated model IS^{F*} has been obtained, both the correct and faulty behaviours of the system can be analysed. For this purpose a library of *specification patterns* pertaining to fault tolerance, recoverability, and diagnosability was defined in [Ezekiel and Lomuscio, 2009].

2.3 IFAS

The IFAS process [Molnar and Veres, 2009b] is intended as a methodology to carry out fault and integrity assessment of complex autonomous systems in order to provide an assurance of their reliability. It consists of the following stages: 1) A formal model is defined for the autonomous system to be investigated; 2) The model is refined for adapting it to a mainstream model checker by means of a discrete abstraction of the hybrid system model and a conversion of the design into a MAS formalism; 3) The requirements of the system are analysed, hence asserting the properties that the design must satisfy; 4) A model checker is used to verify whether these requirements are satisfied or whether bugs exist; 5) If the verification stage shows that some requirements are not satisfied, then counter-examples are produced and the design process is resumed to refine the model based on the counter-examples.

An industry-standard format for embedded systems (Stateflow/Simulink) is used to represent the formal model. Simulink blocks provide the ability to model the continuous dynamics corresponding to the discrete states. Stateflow charts allow the creation of discrete state transition systems based on hierarchical state machines. Testing and simulation can be performed on the autonomous system using these tools. However, this is insufficient to verify the correct operation of the system under degraded conditions.

A compiler automatically translates the discrete state transition system from Stateflow into ISPL [Molnar and Veres, 2009b]. MCMAS is used for verification, providing counter-examples if the requirements of the system are not satisfied. By analysing this resulting model we can infer which properties the system satisfies during correct executions, but also when certain faults arise. Specifically, we are interested in assessing the robustness of the system to faults, its capacity to diagnose them and to recover from them.

Until now, the IFAS process has been demonstrated on illustrative examples to highlight how liveness properties can be verified. In this paper we apply the IFAS process to the Autosub6000 AUV. This required the integration of fault injection with the IFAS process, which we now present.

3 Methodology

The methodology we propose here extends the IFAS process [Molnar and Veres, 2009b], and is presented as a waterfall model in Fig. 1. The user techniques and approaches, that can be used at each stage of the process, are highlighted. Fault injection is integrated into the process at the "Requirements Analysis and Formulae Definition" stage. This allowed

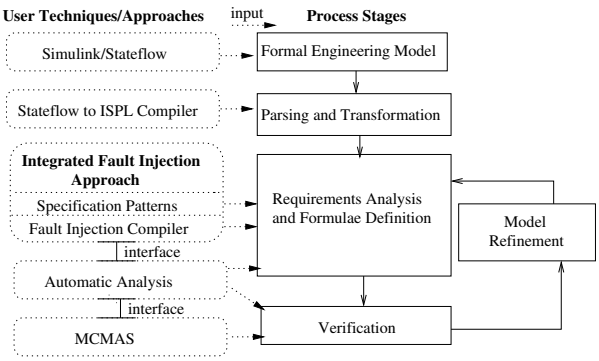

Figure 1: Waterfall model of the IFAS process.

for verification of the correct operation of the Autosub6000 under degraded conditions. Applying IFAS to complex real system allowed us to extensively study the process from a system engineering perspective.

The formal engineering model was defined by analysing the existing Autosub6000 design in detail and consulting with the engineers that designed the AUV. The engineering model was extensively simulated using the Stateflow/Simulink and Matlab tools which provided a preliminary assurance of the validity of the model. Minor oversights during the analysis process meant that bugs were uncovered in the model during verification. These were corrected during refinement, providing further assurances as to the validity of the model. We expect that this exercise of uncovering bugs is similar to one that a system engineer would experience when assessing a new system design. This makes our lessons learnt at the verification stage of the process particularly relevant. Furthermore, the assurance of the validity of the model reinforced confidence that the results from applying the methodology (see Sec. 5) provided a genuine assessment of the reliability of Autosub6000 operating under degraded conditions.

To introduce faulty behaviour into the discretised model, we used a compiler for fault injection available for public use [Ezekiel and Lomuscio, 2009]. The compiler allowed faults to be injected into an ISPL program using a graphical user interface to facilitate fault definition, including the mutation rules, and timing options of the fault. Following fault definition, the compiler output a mutated ISPL program including the definition of the fault injection agents, the mutated faulty behaviour, and the atomic propositions for reasoning about the faults. This allowed for a high level of automation compared to introducing faulty behaviour manually.

The mutated system was visualised by reversing the compilation process in [Molnar and Veres, 2009b] to convert the mutated ISPL program into Stateflow. This allowed for visual inspection of the faulty behaviour, increasing our confidence that the faults had been chosen and defined correctly. A graphical depiction of the mutated system in a familiar format assists engineers in understanding the complex behaviour.

Specification patterns from [Ezekiel and Lomuscio, 2009] were used to analyse fault tolerant requirements of the system. This gave us a template for defining formulae for verifying fault tolerance, diagnosability, and recoverability requirements. This facilitated an easier way to analyse requirements than by crafting specifications individually.

MCMAS was used at the verification stage to verify these specifications and provide counter-examples when the requirements of the system were not satisfied. When counter-examples were produced, we used the Stateflow representation of the system, along with the counter-example to understand the problematic behaviour. When the system was refined, the previously defined faults were automatically injected into the refined model, and the same specifications were applied to the mutated refined model. In this sense, both system behaviour and mutated faulty behaviour can be altered without having to re-write specifications.

A further benefit of a combined fault injection and model checking approach is that graphical artifacts such as fault trees and diagnosability analysis can be produced by using the fault definitions to automatically generate specification that analyse the system, and by interpreting the results from interfacing with MCMAS to verify these specifications against the mutated model [Ezekiel and Lomuscio, 2010]. Although we have not tested this approach on the Autosub system yet, this facility is available in the fault injection compiler.

The methodology we put forward in this section significantly enhances the power of IFAS to assist system engineers in designing reliable autonomous systems, by integrating fault injection into the process. The application of the process to the Autosub6000 has allowed us to describe its usage in this section from the perspective of its suitability to real complex autonomous systems. In the next section we describe the ISPL definition of the vehicle, generated from the parsing and transformation stage of the IFAS process.

4 Modelling of Autosub6000

The Autosub6000 AUV [McPhail, 2009] is engineered to operate in hazardous conditions using a number of reliable subsystems. Autonomous subsystem nodes are distributed throughout the vehicle and carry out tasks such as guidance/mission control, control of position, depth and forward speed, navigation, actuator control, battery/power system monitoring and communication. Communication between these subsystems takes place using a modular, distributed and networked control architecture [McPhail and Pebody, 1998].

The formal model of the Autosub6000 was discretised during the IFAS process, in this same manner as the discretisation process described in [Molnar and Veres, 2009b]. A finite state transition system describes the functionality of each network control node. Translating this discretised model using the Stateflow to ISPL compiler resulted in 27 agents representing the communicating processes autonomously governing the AUV, its environment, and a human observing the AUV from the mother ship. The Stateflow/Simulink model file consists of 15000 lines and is 450KB in size. Here, we summarise some of the key agents relating to our study.

A *MScriptEx* agent models the state machine functionality of the mission control process. Mission events send instructions to the control, actuator and sensor payload systems. Alternative mission ending scenarios are indicated by "mission termination exception" states. An *Echosounder* agent models an echosounder that detects objects on the vehicle's path. A *HorizAvoidStrat* agent models the multi-state process developed for the collision and obstacle avoidance for the horizontal control dimension. When the vehicle encounters an obstacle, it takes a 180 degree turn, retreats along its path to a safe retreat distance, and tries a new track parallel to the original track. Agents *SPMeter, RPArray, RTLength* are used during the horizontal avoidance process in order to navigate to a safe retreat distance from the obstacle. An *AvoidTimer* agent monitors the length of time of the avoidance process since it became active and an *RTCounter* agent counts of the number of times the AUV retreats during the horizontal avoidance process. A *PowerNode* agent models the onboard battery and if the battery voltage falls below a preconfigured set level the mission control state machine is notified. An *EmergencyAbort* models the functionality of the emergency abort node. If the mission has continued for too long, or the vehicle is too deep, the emergency abort system is notified. A *HomingSystem* agent models the vehicle's onboard homing system which guides the vehicle towards the ship's position when a homing signal is received. An *Environment* agent represents the hazardous physical environment that the vehicle operates in. A *Human* agent represents the human operator observing the AUV from the support ship during its operation. The operator can instruct an acoustic beacon to be lowered from the support ship to trigger the vehicle's onboard homing system.

We now turn our attention to verifying various properties of fault tolerance in the discretised Autosub6000 system.

5 Verification Results

The fault injection compiler from [Ezekiel and Lomuscio, 2009] was used to create thirteen carefully selected faults that can potentially be encountered during a real mission. Each fault was chosen in consultation with the engineers so that important properties of the mission control state machine, the homing system, the emergency abort system, and the avoidance strategy could be verified. The faults were automatically injected into the discretised Autosub6000 ISPL model to create a mutated ISPL model for input into MCMAS. We provide a brief description of some of the faults in Table 1.

A total of 85 specifications were written in consultation with the engineers to assess key fault tolerance properties of the AUV. Fault tolerance, recoverability, and diagnosability specifications were described using the specifications patterns in [Ezekiel and Lomuscio, 2009]. MCMAS verified all of the specifications in just under 2 hours using approximately 203MB of memory on a 3.2GHz processor with 2GB of memory, where the number of reachable states is approximately 4.7×10^{10} out of a possible 6.2×10^{27}. Here we simply focus on some of the interesting specifications and the results obtained. We use the atoms in Table 1, and as a naming convention, we use $_f$ to indicate the $faulty$ persistence of a fault; $_i$ to indicate the $injecting$ persistence; $_s$ to indicate the $stopped$ persistence; and $_{is}$ to indicate the disjunction of the $injecting$ persistence and $stopped$ persistence.

To verify that the mission control state machine enters the first mission termination exception $mte1$ when the power is low, and the vehicle does not appear faulty, we used the following specification.

$$AG((\neg VAF_f \wedge PLs) \rightarrow AF(mte1))$$

Table 1: Description of the fault injection agents.

VehicleAppearsFaulty (VAF) : Human observes physically that the AUV appears to be faulty
PowerLow (PL) : AUV onboard battery power is low
AvoidTimerFail (ATF) : Obstacle avoidance process has continued for too long
RetryTrackAvoidanceFail (RTA): AUV has attempted to retreat too many times during obstacle avoidance
RPAAvoidanceFail (RPAAF) : There are no demands to be sent to the position control during obstacle avoidance
Horiz.Avoid.Fail (HAF) : Horizontal avoidance strategy fails to avoid an obstacle
EchosounderDetects (ED) : The forward looking echosounder has detected an object on the vehicle's path
MinIceRange (MIR) : Reached the "minimum ice clearance" ' configuration limit
MinAltRange (MAR) : Reached the "minimum altitude" configuration limit

This specification states that in all paths in which the vehicle does not appear faulty and a power low fault has occurred, $mte1$ is reached. MCMAS reported this specification as true. The mission termination is therefore correctly entered when the power is low.

The following specification expresses that the first mission termination exception is a state of the system in which the mission control node can diagnose that the power is low or one of the avoidance failures has occurred.

$$AG(mte1 \rightarrow (K_{MScriptEx}(ATF_{is} \vee PL_{is} \vee RPAAF_{is} \vee RTA_{is} \vee HAF_{is}) \wedge \neg K_{MScriptEx}(ATF_{is}) \wedge \neg K_{MScriptEx}(PL_{is}) \wedge \neg K_{MScriptEx}(RPAAF_{is}) \wedge \neg K_{MScriptEx}(RTA_{is}) \wedge \neg K_{MScriptEx}(HAF_{is}))$$

This specification states that whenever $mte1$ is reached, the $MScriptEx$ agent knows about the occurrence (i.e., the fault has occurred or is occurring) of either a power low or obstacle avoidance failure, but does not know about the specific occurrence of either of these faults. MCMAS verified this specification as true which testifies to the ability of the mission control state machine to diagnose the occurrence of low power or obstacle avoidance failures.

The following specification encodes that when the forward looking echosounder diagnoses the detection of the object on the vehicles path, the knowledge of the fault is propagated to the horizontal avoidance strategy.

$$\neg E(\neg K_{Echosounder}(ED_{is})\, U\, (K_{Echosounder}(ED_{is}) \wedge \neg AF(K_{HorizAvoidStrat}(ED_{is} \vee (MIR_{is} \wedge MAR_{is}))))$$

This specification states that there is no path in which the $Echosounder$ agent comes to know about the echosounder detecting the fault without the $HorizAvoidStrat$ agent always coming to know that an obstacle needs to be avoided due to the detection of the object on the vehicles path or the reaching of the "minimum ice clearance" and "minimum altitude" configuration limits. MCMAS reported this specification as true. Thus, the forward looking echosounder propagates its knowledge of the detection of the object on the vehicles path to the horizontal avoidance strategy.

The following specification encodes that when the horizontal avoidance strategy diagnoses the detection of the object on the vehicles path or the reaching of the "minimum ice clearance" and "minimum altitude" configuration limits, the knowledge of that one of these faults has occurred is propagated to the obstacle avoidance timer.

$$\neg E(\neg K_{HorizAvoidStrat}(ED_{is} \vee (MIR_{is} \wedge MAR_{is}))\, U\, (K_{HorizAvoidStrat}(ED_{is} \vee (MIR_{is} \wedge MAR_{is})) \wedge \neg AF(K_{AvoidTimer}(ED_{is} \vee (MIR_{is} \wedge MAR_{is}))))$$

This specification states that there is no path in which the $HorizAvoidStrat$ agent comes to know about the echosounder detecting fault or the minimum ice clearance and minimum altitude configuration limits faults without the $AvoidTimer$ agent always coming to know these faults. MCMAS verified this specification as true. Thus, the horizontal avoidance strategy propagates its knowledge of the detection of the object on the vehicles path or the reaching of the configuration limits to the obstacle avoidance timer.

The following specification expresses that when horizontal avoidance strategy diagnoses the detection of the object on the vehicles path or the reaching of the "minimum ice clearance" and "minimum altitude" configuration limits, the knowledge of that one of these faults has occurred is propagated to a group of obstacle avoidance agents.

$$\neg E(\neg K_{HorizAvoidStrat}(ED_{is} \vee (MIR_{is} \wedge MAR_{is}))\, U\, (K_{HorizAvoidStrat}(ED_{is} \vee (MIR_{is} \wedge MAR_{is})) \wedge \neg AF(EK_{ObstAvoidGroup}(ED_{is} \vee (MIR_{is} \wedge MAR_{is}))))$$

This specification states that there is no path in which the $HorizAvoidStrat$ agent comes to know about the echosounder detecting the faults, or the minimum ice clearance and minimum altitude configuration limits faults, without everybody in the group $ObstAvoidGroup$ (comprising of the $AvoidTimer$, $RTCounter$, $SPMeter$, and $RTLength$ agents) always coming to know about the occurrence of these faults. MCMAS verified this specification as true. Thus, the horizontal control dimension avoidance strategy propagates its knowledge of the detection of the object on the vehicles path or the reaching of the configuration limits to the obstacle avoidance agents that require it.

The verification results illustrate how our methodology can provide a useful assessment of reliability of the Autosub6000 AUV under degraded operational conditions. In particular, establishing an assurance that: in faulty scenarios the vehicle can terminate its mission correctly; faults are suitably diagnosed; and the knowledge of obstacles is appropriately propagated amongst the subsystems of the AUV.

6 Related Work and Conclusions

Previous work on combining fault injection with model checking [Bozzano and Villafiorita, 2007; Ezekiel and Lomuscio, 2009] has been applied to reactive systems [Bozzano and Villafiorita, 2007] and MAS [Ezekiel and Lomuscio, 2009]. In [Bozzano and Villafiorita, 2007] an integrated tool for injecting faults into a system model defined in NuSMV [Cimatti *et al.*, 1999] is applied to verify safety-critical avionics systems. The tool automatically mutates the

NuSMV code using a library of failure modes. Temporal specifications are used to verify fault tolerance and diagnosability is not considered.

In [Ezekiel and Lomuscio, 2009] a combined fault injection and model checking approach is applied to MAS, using temporal-epistemic specifications to verify fault tolerance, recoverability and diagnosability. However, the evaluation of the approach is limited to network protocols.

Previous work on verifying AUVs [O'Connor *et al.*, 2006] presents a systematic method of verification for an AUV as a hybrid system. The AUV is modelled as a reactive system. In this approach the UPPAAL [Behrmann *et al.*, 2006] temporal logic model checker is used to verify safety and liveness properties of the AUV model. Properties of fault tolerance, recoverability and diagnosability are not considered.

In this paper we presented a methodology that addresses fault tolerance, recoverability and diagnosability through a combination of fault injection and IFAS. These new methods have been applied to verify a real autonomous system, the Autosub6000 AUV. We presented results from verifying various properties of a discretised model of Autosub6000, which was mutated by injecting faults of real degraded scenarios.

The results from this study are useful to system engineers working on the design of AUVs and similar complex autonomous engineering systems. The key contribution of our work is the integration of the fault injection approach into IFAS, which provides a methodology for system engineers to assess and refine their designs to meet stringent operational requirements of reliability. We learned several important lessons from applying this methodology to Autosub6000: the ability to automate the definition of faulty scenarios using the fault injection compiler and the ability to examine these scenarios in Stateflow, give the methodology significant merits in terms of its usability for system engineers.The expression of reliability requirements can be made easily through specification patterns and re-applied automatically when the system is refined. Scenarios in which requirements are not met can be investigated visually and comprehensively using a combination of counter-examples and Stateflow. These aspects of the methodology provide a highly usable and automated way, in which the reliability of real complex autonomous systems can be assessed.

Furthermore, artifacts such as fault trees that analyse reliability requirements can be automatically produced using our approach [Ezekiel and Lomuscio, 2010]. We leave this investigation to future work.

References

[Behrmann *et al.*, 2006] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings of QEST'06*, pages 125–126. IEEE, 2006.

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

[Chance, 2003] T.S. Chance. AUV surveys - extending our reach, 24000 km later. In *Proceedings of UUST'03*. AUSI, 2003.

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

[Clarke *et al.*, 1999] E.M. Clarke, O.Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

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

[Ezekiel and Lomuscio, 2010] J. Ezekiel and A. Lomuscio. A methodology for automatic diagnosability analysis. In *Proceedings of ICFEM'10*. To appear, 2010.

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

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

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

[Lomuscio *et al.*, 2009] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.

[McPhail and Pebody, 1998] S.D. McPhail and M. Pebody. Navigation and control of an autonomous underwater vehicle using a distributed, networked, control architecture. *Society for Underwater Technology*, 23(12):19–30, 1998.

[McPhail, 2009] S. McPhail. Autosub6000: A deep diving long range AUV. *Bionic Engineering*, 6(1):55–62, 2009.

[Molnar and Veres, 2009a] L. Molnar and S. M. Veres. System verification of autonomous underwater vehicles by model checking. In *Proceedings of Oceans'09*, pages 1–10. IEEE, 2009.

[Molnar and Veres, 2009b] L. Molnar and S. M. Veres. Verification of autonomous underwater vehicles using formal logic. In *Proceedings of ECC'09*, pages 1–6. EUCA, 2009.

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

[O'Connor *et al.*, 2006] M. O'Connor, S. Tangirala, R. Kumar, S. Bhattacharyya, M. Sznaier, and L.E. Holloway. A bottom-up approach to verification of hybrid model-based hierarchical controllers with application to underwater vehicles. In *Proceedings of ACC'06*, page 6 pp. IEEE, 2006.

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

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