Peer-Reviewed Technical Communication

Automated Fault Diagnosis for an Autonomous Underwater Vehicle

Richard Dearden and Juhan Ernits

***Abstract*—This paper reports our results in using a discrete fault diagnosis system Livingstone 2 (L2), onboard an autonomous underwater vehicle (AUV) Autosub 6000. Due to the difficulty of communicating between an AUV and its operators, AUVs can benefit particularly from increased autonomy, of which fault diagnosis is a part. However, they are also restricted in their power consumption. We show that a discrete diagnosis system can detect and identify a number of faults that would threaten the health of an AUV, while also being sufficiently lightweight computationally to be deployed onboard the vehicle. Since AUVs also often have their missions designed just before deployment in response to data from previous missions, a diagnosis system that monitors the software as well as the hardware of the system is also very useful. We show how a software diagnosis model can be built automatically that can be integrated with the hardware model to diagnose the complete system. We show empirically that on Autosub 6000 this allows us to diagnose real vehicle faults that could potentially lead to the loss of the vehicle.**

***Index Terms*—Automatic fault diagnosis, autonomous underwater vehicle (AUV), autonomy.**

I. Introduction

THE use of autonomous underwater vehicles (AUVs) has significantly increased in recent years as their value has been demonstrated in applications as diverse as oil-field operations and oceanography. As the capabilities of these vehicles are improving, the missions are becoming longer, riskier, and more complex, and the amount of human intervention is decreasing. This in turn means that improving the reliability of the vehicle becomes more and more important. More reliable hardware and control systems, and designing missions to have less risk [1] will help significantly with this, but significant gains can also be made through onboard diagnosis of these vehicles to detect faults when they occur and take appropriate actions to mitigate their effects. The environment these vehicles operate in, including extreme pressure, the corrosive effects of seawater, and the risk of damage due to waves while on the surface and collisions during launch and later, is too harsh to expect even the best designed components to operate flawlessly over repeated missions.

The potential benefits of fault detection on an AUV are illustrated by the work of Griffiths and Brito [1] on predicting the risk of loss of a vehicle. Their approach uses a Bayesian network (BN) to predict loss likelihood for a mission under sea ice. The BN probabilities are based on historical data of failures of individual components along with expert judgement of how likely particular failures are to cause loss of vehicle under different mission conditions. By inputing the observed values of variables such as percentage ice cover, ice thickness, etc., these are combined with the historical and expert data to compute an estimate of the probability that the planned mission will lead to the loss of the vehicle.

Fig. 1 shows the BN from Griffiths and Brito [1]. Circles represent variables that influence the likelihood of vehicle loss and arrows represent dependencies between the variables. For example, the arrow between *AUV susceptibility* and *probability of loss* indicates that the probability of loss of the vehicle depends on how susceptible the vehicle is to failures, so changing one will change the probability of the other. Intuitively, there are three ways to decrease the likelihood of vehicle loss, and these are reflected in the BN.

- Make the mission less risky. This corresponds to changing the probabilities in *vessel characteristics* (using a more capable vessel), or *ice concentration* and *ice thickness*.
- Use more reliable components on the vehicle that fail less frequently. This corresponds to changing the probability of the top right node, *risk reference case*, which indicates the risk of vehicle failures.
- Mitigate the effects of any failures, by changing the probabilities of *AUV susceptibility* (how robust the vehicle is to failures and environmental conditions) given *risk reference case* and the *probability of loss* given *AUV susceptibility*.

Onboard fault diagnosis is intended to reduce the likelihood of vehicle loss by the third approach. It achieves this by detecting and identifying faults as they occur. Although we do not discuss it in this paper, the next step would then be to take appropriate actions to prevent the faults that have been identified from leading to vehicle loss.

The task of the diagnosis system is to determine the state of the vehicle as it is running, based on *telemetry* from internal sensors of the system, and any commands sent to the system or its components. Typically system state consists of a discrete operating *mode*, plus the values of various continuous parameters of the system. However, as we discuss below, in this work, we use a discrete diagnosis algorithm, to only track the discrete part of

Manuscript received February 09, 2009; revised May 08, 2012; accepted October 23, 2012. Date of publication January 29, 2013; date of current version July 10, 2013. This work was supported by the U.K. Natural Environment Research Council under Grant NE/F01256X/1, "Automated Diagnosis for Fault Detection, Identification and Recovery in Autosub6000."

Guest Editor: M. Seto.

R. Dearden is with the School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, U.K. (e-mail: rwd@cs.bham.ac.uk).

J. Ernits is with the Department of Computer Science, Tallinn University of Technology, Tallinn 12618, Estonia (e-mail: juhan.ernits@ttu.ee).

Digital Object Identifier 10.1109/JOE.2012.2227540

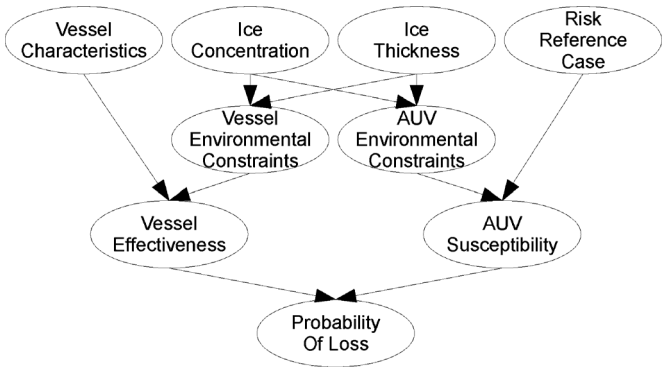

Fig. 1. A Bayesian network for estimating the risk of vehicle loss in an under-ice mission (adapted from [1]).

the state. Often it is convenient to distinguish normal operating states, referred to as *nominal modes*, from fault modes. Note that the nominal mode may change during a mission depending on which commands are sent to the system.

This paper describes our experiences using the Livingstone 2 (L2) diagnosis engine on an AUV. L2 is a discrete, consistency-based diagnosis system, meaning that it contains a discrete model of the system it is diagnosing, and discovers faults by finding inconsistencies between predictions made by the model and telemetry from the system. We chose L2 for a number of reasons. It has previously been deployed on an Earth observing satellite [2], and thus has been designed to be lightweight in terms of computational requirements and engineered to the standards expected for space flight software. The fact that L2 is an entirely discrete engine not only significantly reduces its computational requirements, but also makes model building much simpler. It also means the models are composable, so that models of new vehicles or vehicle configurations can be constructed more easily and quickly by plugging together component models from a library. This makes it a natural choice to apply to AUVs, which in a number of ways (computational restrictions, control architectures, etc.) resemble the spacecraft that L2 was originally designed to diagnose. Consistency-based diagnosis, as well as related work on diagnosis in AUVs, is described in Section II.

To use a discrete diagnosis system such as L2, we generally need to translate the telemetry from the system—typically continuous measurements such as voltages or actuator positions—into the discrete inputs, called *observations*, needed by the diagnosis engine. This is accomplished by what are known as *monitors*. In designing a diagnosis system, ideally the monitors should be as simple as possible, preferably taking a single measurement as input and producing a single discrete output. For example, telemetry from a voltage sensor might be used by a monitor that applies some smoothing to eliminate transient spikes, and then applies various thresholds to output a discrete value from the set *{zero, low, nominal, high}*.

We applied L2 to the Autosub 6000 [3] AUV operated by the U.K. National Oceanography Centre. Autosub 6000 is designed to operate for up to 48 h at a time at depths of up to 6000 m to collect science data from the deep ocean. During more than 400 previous scientific missions, Autosub 6000 and its predecessors have suffered both near losses and one actual loss. In two cases, the AUV has been recovered with a remotely operated underwater vehicle (ROV) at significant expense. In one case, the Autosub2 AUV was permanently lost under an ice shelf in the Antarctic. There are numerous cases of missions that have had to be aborted but where recovery was possible by the operations team and the attending support ship. This means that not only do we have large amounts of real logged data to test the diagnosis algorithm on, but we also have data that includes actual faults that endangered the vehicle. Section III contains a more detailed description of the Autosub as well as a categorization of the faults that have occurred during previous missions. We describe the L2 model of the Autosub hardware in Section IV.

In practice, many subsystems on a vehicle such as an AUV will have their own fault protection schemes built in. For example, on Autosub 6000, there are ground fault detectors in a number of components. The strength of L2 is in system-level diagnosis—it is able to integrate data from multiple sensors in different subsystems and reason about the root cause of failures. Frequently, when a fault occurs, it triggers a cascade of secondary faults (for example, a battery failure or short circuit may lead to power shortages which lead to other components shutting down or intermittently failing). The model allows L2 to determine that all the apparent faulty behavior can be explained by the single root cause so that an appropriate response to that one fault (for example, switching off a subsystem to reduce power demand) can be taken.

As well as monitoring the health of the hardware components of the AUV, we are also interested in monitoring the execution of the mission. This is important as it allows us to check that the mission is being executed correctly, and also because different parts of the mission will depend on different components, so knowing what will happen in future parts of the mission allows us to decide whether to abort the mission when a fault occurs, or whether the mission can continue. For the Autosub, in common with many AUVs, the mission is defined in a file called the *mission script* which is uploaded to the vehicle at the start of every mission. To monitor the execution of this, and also because the mission script contains information about the behavior of the vehicle that is not available in the logs, we generate a diagnosis model of the mission script automatically from the script itself. This *mission script model* is then combined with the prebuilt model of the vehicle itself. The mission script model and its generation are described in Section V.

Mitigating the effects of faults by identifying them—in some cases before they cause any serious problems—and choosing appropriate responses is only one useful outcome of automatic fault diagnosis in systems such as Autosub 6000. In addition, a diagnosis system running offboard (on the support ship) that uses transmitted telemetry can aid mission controllers in identifying unexpected behavior and enable them to take appropriate responses. Finally, diagnosis can reduce turnaround times between missions by pinpointing the subsystems that need to be replaced or at least examined to ensure they are working normally.

In Section V, we discuss the use of diagnosis both *onboard* the vehicle (where the diagnosis software is running in real time

on the AUV during a mission) but also *offboard*, where the software is still running in real time, but is using telemetry transmitted from the AUV to the support ship acoustically. This has the advantage that it allows the AUV's operators to monitor the behavior of the vehicle as it carries out the mission, and potentially to trigger recovery actions or abort the mission if something goes wrong. However, since the offboard diagnosis system only has access to a limited subset of variables, the diagnoses that can be made offboard are quite restricted compared with onboard. Some critical faults can be identified, albeit with a significant delay compared with onboard detection. We report our results in Section VI. Finally, in Section VII, we draw conclusions from this work and sketch our future directions.

II. Consistency-Based Diagnosis

Traditionally, automatic diagnosis has been performed using expert or rule-based systems [4]. These approaches use knowledge from human experts along with data about symptoms to perform diagnosis. However, these approaches rely on the quality and completeness of the expert data, meaning that the expert must enumerate and reason about the effects of every plausible combination of faults. The faults that occur in any real-world system will include some that are rare, multiple faults, as well as complex interactions between subsystems that may mask faults or cause their symptoms to appear in unexpected places. Thus, these expert data are hard to collect and the set of rules becomes large and complex, and hence very difficult to manage.

Model-based diagnosis (MBD) is an alternative to expert systems that has been successfully applied in a number of domains. MBD requires models to be built of both the nominal and fault behaviors of the system; these models are used at runtime by the diagnosis engine to estimate the current state of the system. MBD can roughly be divided into three categories: analytical, knowledge based, and stochastic.

The analytical approach is typically used in the design of control systems where the models are constructed from first principles [5]. While being precise, it is targeted at problems fairly close to the hardware and constructing the models is very labor intensive.

The more recent stochastic approach typically involves using a bank of Kalman filters [6] or a particle filter [7], [8] to track multiple possible states that the system might be in simultaneously. These approaches are computationally expensive and usually require very high fidelity models. On systems such as AUVs where onboard computation is limited and power is often the limiting factor for mission duration, they may require significantly more computational power than may be practical.

The knowledge-based approach uses qualitative models of the system to detect and diagnose faults. One of the most successful approaches for fault identification and recovery has been consistency-based diagnosis [9]. This approach uses a model of the physical behavior of the system. Diagnosis proceeds by comparing the predicted system behavior according to the model with the observed system behavior and using various kinds of inference to explain any discrepancies. A number of systems have been developed based on this approach, including GDE [9], Livingstone [10], Hyde [11], and Lydia [12]. All of these are broadly similar in capability, although Hyde, for example, allows a richer set of models, including hybrid discrete-continuous system models. The system we are using is L2, because it is freely available and it has already been used in similar applications. Discrete consistency-based approaches have the significant advantage that their computational requirements are generally low, and only increase when faults occur: if the system is nominal, it is sufficient to simulate it in the model and check that the model matches the observed system.

A. Livingstone 2

L2 [10], [13] is an MBD system that uses a constraint solver to perform consistency checks between the predicted and observed system behaviors. Livingstone has been successfully used on two spacecraft [2], [14] as well as a number of testbed systems.[1] As with most MBD systems, L2 models are split into models of individual components which are then composed to build an overall system model. This has the twin advantages of increasing reuse of models since components are often common between systems, and allowing the effects of multiple faults to be modeled without explicitly enumerating all possible sets of faults: the effects of multiple faults naturally arise from the effects of single faults and the connections between the components in the model. To achieve this *compositionality*, L2 allows only discrete variables within components. Components are connected together to build a system by specifying constraints on the values of these variables. For example, a battery component may contain a variable CurrentOut that represents the current being produced by the battery, with possible values zero, low, nominal, and high. Similarly, a circuit breaker (CB) component may contain a CurrentIn variable, and the system is constructed by declaring constraints such as Battery.CurrentOut = CB.CurrentIn.

As well as containing variables, components also have an internal state, which corresponds to the modes of operation of the component, some of which may be nominal and the rest fault modes. For example, our simple CB component may include On, Off, tripped, blown, and other fault modes. Changes in state are governed by transitions between modes, some triggered by commands such as SwitchOn, which causes a transition to the On mode if the CB is currently in the Off mode, and others triggered by faults. The constraints on variables may change depending on the mode, so, for example, in the Off mode, CurrentOut = zero, while in the On mode, CurrentOut = CurrentIn. Thus, a component model can be thought of as a finite state machine with constraint sets for each state. Fig. 2 shows this simple CB model both graphically and in the Java-like model programming language (JMPL) used by L2. In the graphical representation, nominal states are shown as squares, with fault states shown as octagons. Commanded transitions are solid arrows with their commands in italics, while fault transitions are dashed. The Unknown Fault state can be transitioned to from anywhere, and provides no constraints on the system.

[1]L2 is freely available from the National Aeronautics and Space Administration (NASA). However, potential users of the system may wish to contact the authors of this paper as they have significantly updated the version NASA makes available, as well as developed a number of tools to aid model building.

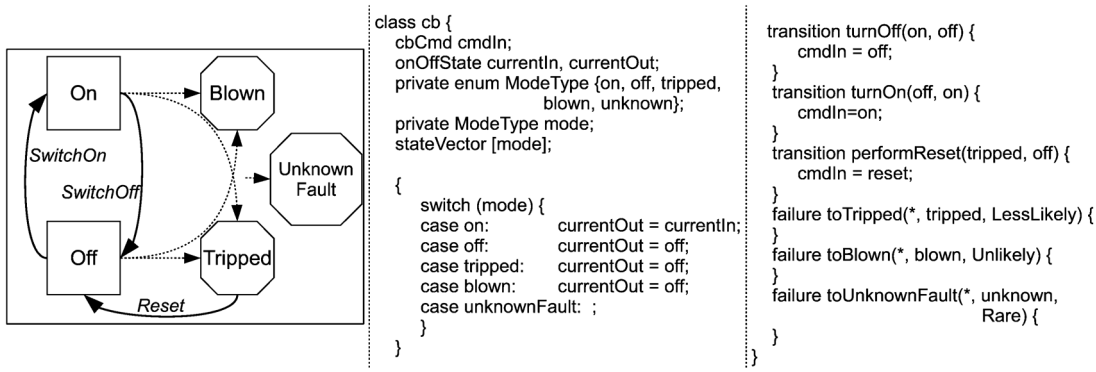

Fig. 2. A circuit breaker model in L2. On the left is the pictorial representation of the state machine, while the corresponding code in L2's JMPL language is on the right.

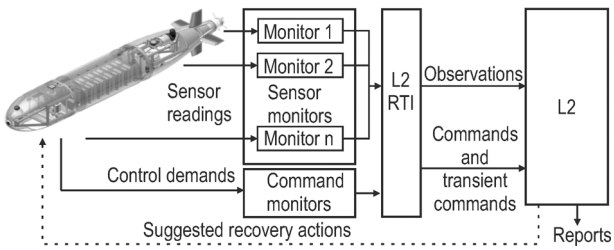

Fig. 3. Architecture of the real-time diagnosis system with L2.

In normal operation, L2 intercepts all commands sent to the system and uses them to update the modes of components in its model. The set of component modes is used to predict the behavior of the system, which is compared with the observations from the real system. This is done by adding constraints corresponding to the values of the observed system variables. As long as these are consistent with the current set of constraints governing the system, L2 reports that the system is nominal, and nothing else happens. In the event that the observations lead to an inconsistency with the constraints in the model, L2 then searches for a transition or transitions that the model could have taken to leave it in a state consistent with the observations. This search is done using an algorithm called *conflict-directed best first search* [15]. Essentially, the variables that produce the inconsistency are used to identify the components that could be involved, and the fault transitions in the finite state machines for those components are searched to find transitions to modes consistent with the observations. If a transition is found that leads to a mode consistent with the observations, then this is a candidate hypothesis that can explain the fault. L2 allows multiple candidates to be maintained to represent the fact that there may be multiple possible explanations for a particular symptom. Subsequent observations can then be used to eliminate incorrect candidates.

All the above assumes that the observations of the system are discrete so they can work with discrete models and constraints. In practice, almost all the observations we get from real systems are continuous, so these must somehow be converted into discrete values, a task performed by what are called *monitors*. This is one of the significant challenges in using consistency-based MBD systems such as L2, as building the monitors is rarely trivial as thresholds have to be set, smoothing may need to be applied, and short-term fluctuations known as *transients* need to be dealt with. In some cases, the need to discretize may prevent MBD from working at all. From a model engineering point of view, the key is to make the monitors as simple as possible: if determining the discrete value of some variable requires examining the values of four other variables, then effectively one is performing the diagnosis task in the monitor.

The overall interface between the physical system and L2 is shown in Fig. 3. Sensor data from the system are run through the monitors to generate discrete observations, and these, together with commands, are passed into L2 through the real-time interface of L2 [16]. L2 then updates its model to reflect any commanded transitions, and then checks for consistency. If an inconsistency is found, it uses conflict-directed best first search to update its internal model to one consistent with the system.

B. Diagnosis in AUVs

There has been a long history of work on fault detection and fault-tolerant control for AUVs. As with most vehicles, Autosub 6000 and its predecessors have the simplest form of fault protection, an emergency abort system [17] that is triggered by a variety of events on the vehicle such as critical subsystems being unresponsive, or the vehicle exceeding its maximum depth limit. The conditions that trigger the abort system have been constructed manually based on previously seen behaviors: if a new fault that endangers the vehicle is observed during operations, additional conditions will often be added to the system. This results in a fault protection system that is complex and difficult to maintain [18].

A survey of fault detection and fault tolerance schemes for AUVs and ROVs has been presented in [19]. Many of these are model-based approaches like the one described in this paper, but tend to be only for restricted subsystems, for example, the actuators in [20]. The models in that case are continuous, rather than the discrete ones L2 uses, and faults consist of changes to the parameters of the system. The relationship between these kinds of approaches and the one we take is that L2 provides system-level fault detection, rather than component-level fault detection.

The only previous system-level diagnosis approach we are aware of that has been applied in AUVs is the approach by Hamilton *et al.* [18]. Their approach is more of an overall architecture for diagnosis, including rule-based diagnosis as well as model-based diagnosis. They use what they call a *watcher*

which checks if any rules have been triggered as well as looks for discrepancies between the telemetry and the model. Their combination of multiple diagnosers means they use a more *ad hoc* approach to determining the likely faults: essentially, they count the number of times a particular component is implicated by one of the diagosers to decide which component may be faulty. They also distinguish component and subcomponent diagnosis, performing component-level diagnosis first, and then searching in the two most implicated components for subcomponents that might explain the fault. This contrasts with L2, where the use of a single model-based approach means that conflict-directed search can be used to find the n most likely fault candidates in a more systematic way.

Like L2, the approach in [18] can use existing component fault detectors such as those described in [20]. The difference is that the approach in [18] uses them as simply another component of the diagnosis engine, while L2 uses them as monitors, providing discrete estimates of the component state as inputs to the diagnosis engine. Although we do not evaluate such a system here, a diagnosis system which uses the continuous models of [20] to diagnose small subsystems, and a discrete approach such as L2 to combine these local diagnoses together at the system level may be the best solution in terms of keeping the computational load of diagnosis manageable while maximizing what can be diagnosed without the maintainability challenge of a hand-generated fault protection scheme.

Although it is not a diagnosis system in itself, but rather a way to evaluate the fault tolerance of a vehicle, the approach in [21] is related from a modeling perspective in that it uses the same vehicle as this work. Rather than the component-based models used in model-based diagnosis, Ezekiel *et al.* take a top–down approach to modeling, from a general Matlab model of the AUV to discretized agents where faults are then automatically injected and consequences analyzed. It is an interesting research question beyond the scope of this paper how to combine that approach with ours.

III. AUTOSUB 6000

Autosub 6000 is the latest in the Autosub series of AUVs developed by the U.K. National Oceanographic Centre (NOC). It is a self-controlled underwater robot with a length of 5.5 m, a width of 0.9 m, and a dry mass of two metric tons. As were its predecessors (Autosubs 1, 2, and 3), the AUV is designed to carry out solo science missions deep in the ocean, often under ice, and potentially so far from its support ship that it is out of range of any communications. The support ship and crew can, following the AUV's deployment and "in the spirit of true AUV autonomy," freely proceed with other ocean science work until it returns.

In an individual mission, Autosub 6000 can travel distances of up to 1000 km (depending on its speed) and can dive to depths of 6000 m. It has a configurable payload capacity of 0.5 m^3, allowing the inclusion of various pieces of scientific equipment. The design and capabilities of Autosub 6000 allow it to be used for a wide variety of oceanographic tasks. In the past, successful tasks have included water column sampling, measuring temperature and salinity, photographing and mapping the seafloor, and carrying out chemical analysis.

Apart from science instruments, Autosub 6000 is also equipped with a variety of other sensors including compass, depth gauge, an inertial navigation system, and Doppler velocity log for dead reckoning, as well as internal sensors for detecting leaks, short circuits to ground, battery voltage and current, etc. The vehicle has four actuators, consisting of the propeller, the stern plane, and the rudder, and an abort-weight release mechanism which when activated makes the vehicle significantly more buoyant so it returns to the surface.

Autosub missions are specified by mission scripts. These files contain a sequential list of actions that must be carried out by the AUV to complete the objectives of a mission. The configuration of the AUV throughout a mission is partly dynamic in that mission scripts specify behaviors which in turn require the AUV to modify components (e.g., turning the rudder), however certain aspects of the AUV's configuration are statically defined at the start of a mission. These configurations include such things as safety limits and events that trigger "mission abort." These latter, fixed configurations are provided to the sub in configuration files.

The control and navigation of the Autosub is managed in a distributed fashion by several interacting processing nodes on an internal control network. Each node is responsible for some subset of Autosub behavior; some manage relatively simple tasks such as reading a single sensor while others take responsibility for more complex work. Mission scripts are executed by the mission control node. This moves through the list of required actions as their triggering events occur and sends appropriate commands to other control nodes which enforce the actions.

Other than the mission control node, there are three major behavior-controlling nodes. These are the *depth control node*, the *position control node*, and the *motor/propulsion control node*, which, respectively, manage the AUV's vertical position (depth), geographical position, and speed. These control nodes follow the instructions of the mission control node; when the mission control node broadcasts a command indicating that the AUV should, for example, maintain a depth of 200 m, the depth control node does what is necessary to enforce this request. Fig. 4 shows the control nodes in the Autosub control system and the data flow between them. Mission control sends commands to these nodes in the form of demands; they in turn pass around data to calculate events such as GOTDEPTH, which indicate a demand has been met.

Each of the key control nodes can be in one of several control modes; these modes dictate the way in which the node works and interprets demand arguments, the computations it carries out, and the way in which it controls components. For example, the depth control node has three possible control modes: *stern plane control*, *depth control*, and *altitude control*. In the latter two modes, the controller is instructed to maintain a given depth (below the surface) or altitude (above the seafloor) and, therefore, has to carry out some computation to determine what control to apply to match the actual depth/altitude with the demand. In contrast, in stern plane control mode, the controller is given an exact angle to which to set the stern plane.

A. Historical Faults

The Autosub series of AUVs have been used in more than 400 missions. However, in the course of these missions, the

TABLE I
OBSERVED FAILURES IN AUTOSUB 6000 AND ITS PREDECESSORS. THE "OCCURRENCES ELSEWHERE" COLUMN REFERS TO FAULTS ON PREDECESSORS OF AUTOSUB 6000. A DASH IN THIS COLUMN INDICATES THAT WE HAVE NOT FOUND ANY EXAMPLES

Component	Failure	Occurrences on Autosub 6000	Occurrences elsewhere
Stern plane actuator	Not responding to commands	1	-
	Actuator feedback potentiometer failed	1	Yes
	Intermittent drive faults	1	-
	Actuator stuck	no	several
	Actuator offset from zero	no	several
	Leak in actuator vessel	1	-
	Pressure related actuator malfunction	1	-
Rudder actuator	Failed	no	1
Propulsion system	Gradual loss of propeller speed	no	2
	Entanglements (mostly with recovery lines)	no	several
Emergency Abort	Failure to abort	no	One found pre-dive
Sensors	GPS lost initialisation	1	-
	GPS Antenna failed	7	-
	Poor navigational accuracy	2	-
	Collision avoidance software fault	1	-
	ADCP spontaneous restarts	2	-
Acoustic modem	Reduced sensitivity	2	-
Argos and WIFI links	Antenna failure	2	-
Batteries	Not switching on	several	Different battery type
	Not sourcing current	2	
	Failed to charge	1	
	Discharge fuses blown	1	
Other power system	Short due to cable failure	several	-
	Failed power switch	1	-
Science instruments	No sonar data recorded	1	1
	No CTD data recorded	No	At least one
Recovery lines	Lines entangles in propeller	No	Several
	Lines dragging in water, prevent diving	2	Several

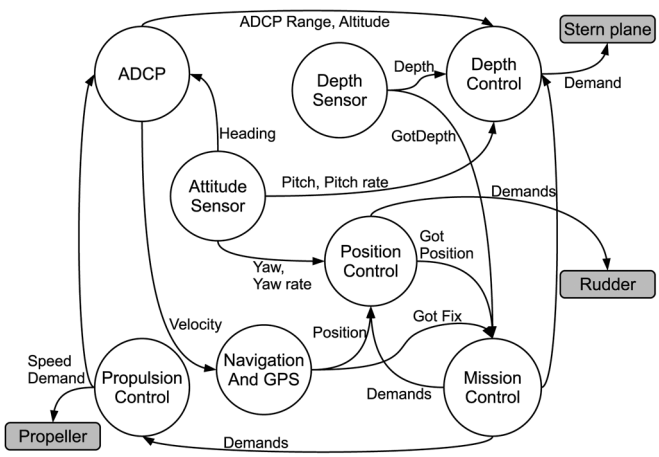

Fig. 4. A simplified architecture of Autosub 6000 showing how demands, data, and events are passed between control nodes. Adapted from [17].

project team has encountered a large number of physical faults and other problems which have resulted in several mission failures, several cases of damage to the AUV, several occurrences of equipment loss, and, in one case, the actual irretrievable loss of Autosub 2 under the Fimbul Ice Shelf in Antarctica. It is clearly undesirable and inconvenient for Autosub to fail; if it does not fulfill its mission objective, then a potentially important science goal remains unachieved and a great deal of time is wasted on planning the mission, traveling to and from the deployment location, and conducting the failing mission itself. Autosub damage and loss is also very expensive. Autosub 6000 is worth around $1 million while the lost Autosub 2 at the time of its loss was worth around $1.5 million.

Table I lists a representative set of faults that have actually occurred on Autosub 6000 or its predecessors.

IV. LIVINGSTONE 2 ON AUTOSUB

In Section III, we looked at what has gone wrong in previous missions. Before using L2 to diagnose the Autosub, we used these data to identify critical faults that put the system at risk. For example, the failed power switch fault recorded in Table I, while it led to an abort and the premature end of the mission, did not put the vehicle at risk. On the other hand, the stern plane actuator failures on both Autosub 6000 and Autosub 3 directly endangered the vehicle by potentially leading to collision with the seabed. Thus, we identified depth control as a priority subsystem for diagnosis. In addition to depth control, the other most critical subsystems for modeling were the stern plane (itself a component of depth control) and the rudder control systems. We also chose to model the batteries, largely because a significant number of faults were observed. However, this subsystem is very challenging to model with a discrete diagnosis system, so it also provides a useful evaluation of where L2 does not perform well for AUV diagnosis.

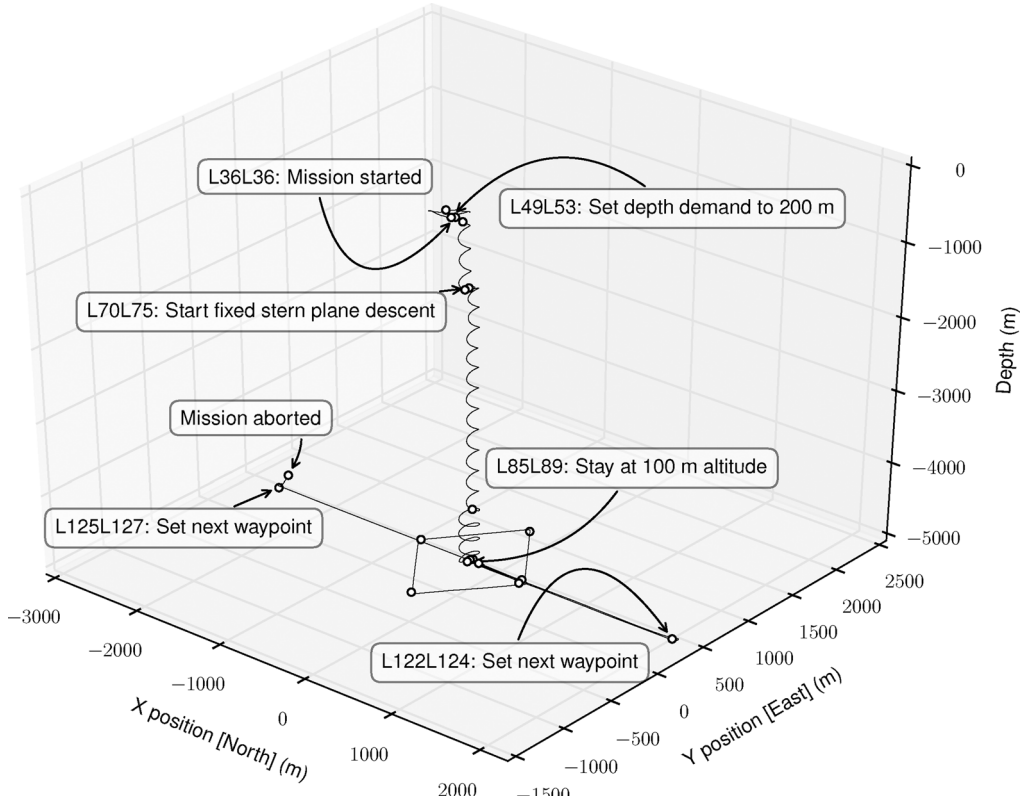

Fig. 5. Three-dimensional navigation plot of Autosub6000 mission M012. The AUV spiralled to depth, executed a 1-km side box at 4714-m depth, and started a lawn mower pattern survey. The vehicle returned to the surface after a failure in the stern plane actuator led to it exceeding the maximum depth limit for the mission and aborting. Labels indicate the corresponding lines in the mission script.

A. Mitigations of Faults in Autosub 6000

Detecting faults do not increase the reliability and robustness of a system; this can only be achieved by actually taking action based on the diagnoses. L2 achieves this in the same way as it identifies faults: it searches for commanded transitions from the fault mode it finds itself in back into a nominal operating mode. On systems with redundancy built in, such as rocket fuel systems, this typically involves opening and closing valves to allow the redundant pathway to supply fuel to the motor rather than using the faulty one. On the Autosub, there is very little redundancy so for many faults there is nothing that can be done to achieve nominal operation, but rather the challenge is to ensure the safety of the vehicle, so the correct mitigation if there is any risk of a seabed collision is to drop the abort weights and return to the surface. Of course, in an overhead environment such as operations under an ice sheet, this may not be a safe mitigation either, and more sophisticated strategies may be required.

B. Construction of the Diagnosis Model

Autosub6000 is well documented and we have access to recorded data from a number of previous missions that capture almost all the telemetry needed for diagnosis. However, due to the architecture of the onboard communication network, many commands in the mission script are not captured by the logger. Thus, these will need to be inferred from other information.

Fig. 5 shows a navigation track of Autosub 6000 during an otherwise typical mission until an abort occurred due to a hardware failure. A mission is split into individual phases. In this case, after the mission started, the mission control software demanded the AUV to dive to 200-m depth, then started fixed stern plane descent to 1000 m, and then to 4000 m. After cautiously approaching the target depth of 4714 m, the AUV switched over to altitude following mode and performed a box pattern 100 m above the seafloor, immediately followed by a lawn mower pattern survey, which was interrupted by the fault. Each small circle in Fig. 5 denotes a change in behavior in the mission script; the label shows the lines of code in the mission script that define the new behavior. Because the actual commands are not available to the diagnosis engine, we use these mission script lines as a proxy for them. The details of this are in Section V.

Fig. 6 shows the diagnosis models we have built for Autosub 6000. To illustrate the model in more detail we describe the depth control component and its relation to the mission control and stern plane components, as illustrated by Fig. 6(a). The depth control component has three nominal modes, DEPTH, ALTITUDE, and STERNPLANE, plus a single fault mode, SOFTWAREFAULT. Nominal mode transitions are triggered by commands in the mission script, represented in Fig. 6(a) by the link between the VERTICAL MODE variables in the mission script and depth control components. When VERTICAL MODE = DEPTH, the vehicle is being controlled by specifying an absolute depth the vehicle should be trying to achieve. In this case, we expect that the stern plane will be adjusted to move the vehicle to that depth (if the required depth is shallower than the current depth, the vehicle needs to climb, so the stern

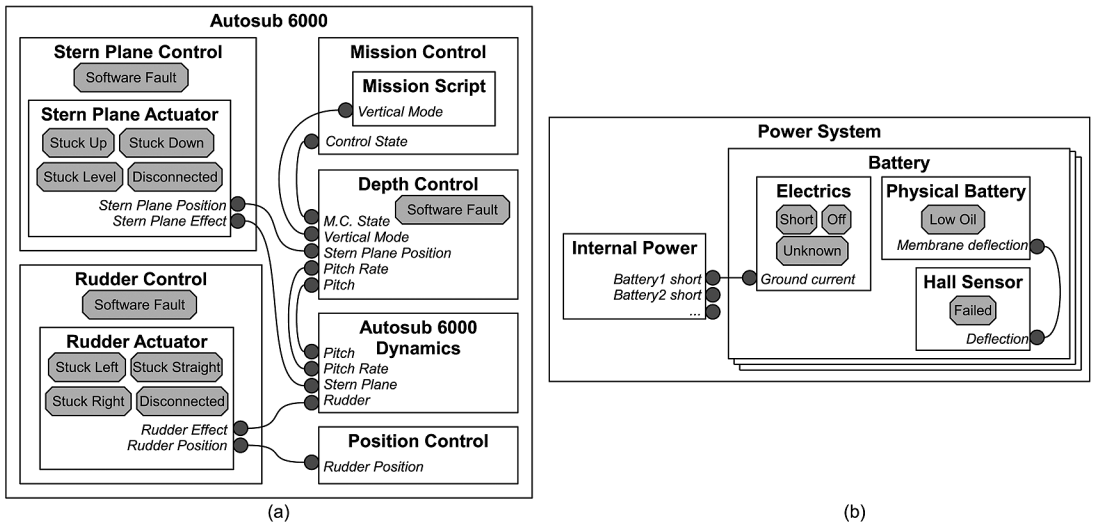

Fig. 6. A graphical representation of the Autosub 6000 diagnosis models. Boxes represent components in the model. Each component has one or more nominal modes (not shown) and fault modes (gray octagons). To show the connections between components, the figure also lists (in italics) the variables involved in global constraints in the model. For clarity, transitions between component modes and component variables not involved in global constraints are not shown. (a) The main control model. (b) The battery model. In this model, the nested boxes under the battery component represent the fact that there are as many of this component in the model as there are batteries installed in the vehicle.

plane will point up), and any other behavior indicates a fault. Similarly, VERTICAL MODE = ALTITUDE indicates that control is in terms of altitudes above the seafloor, and VERTICAL MODE = STERNPLANE indicates that the stern plane angle is being commanded directly. In Fig. 5, the initial dive (labeled L49L53) is in DEPTH mode; there is a subsequent switch to STERNPLANE mode at L70L75, and then at L85L89 to ALTITUDE mode.

As we said, in a situation where VERTICAL MODE = DEPTH and the commanded depth is shallower than the current vehicle depth, the stern plane should be pointing up to cause the vehicle to climb. The connection in Fig. 6(a) between STERNPLANEPOSITION in the depth control and stern plane actuator components indicates that these two should be equal. If the stern plane does not point up in this situation, the diagnosis engine will consider faults that might explain this situation. The most likely of these are either the software fault in the depth control component (in which case there is no constraint on the position of the stern plane), or any of the faults apart from STUCKUP in the stern plane actuator component, for example, STUCK DOWN, which specifies that the stern plane is always in the down position no matter what command is sent. A similar process can be used to infer the pitch of the vehicle using the relationship between the variables in the depth control and vehicle dynamics components.

Fig. 6(b) shows the battery model. In this model, we have found it convenient to divide a single physical battery into three subcomponents, representing the electrical behavior of the battery, the physical behavior, and the Hall-effect sensor, which is used to detect pressurization problems with the battery. Autosub 6000's batteries are oil filled, with a flexible membrane to mitigate the effects of pressure. The Hall-effect sensor detects deflection of this membrane which indicates leakage of battery oil. This is captured by the LOWOIL fault in the physical battery model. Thus, if the Hall sensor is on, the diagnosis engine finds that either the battery has low oil or the sensor has failed. Fig. 6(b) also includes a model of the power system that integrates an additional sensor that measures any motor ground faults. Since we only have a single sensor for this, it is impossible to localize which battery has a ground fault.

C. Monitors

As shown in Fig. 3, the interface between the vehicle and L2 is achieved by the use of monitors. The monitors are responsible for deriving the discrete finite domain values that are used in the diagnosis model from the telemetry of the vehicle.

We chose to integrate the L2 system development into the Eclipse IDE,[2] and, thus, it was most convenient to build monitors in the Java programming language. The monitors for the depth control diagnosis system are made up of 11 main classes, monitors, 22 helper classes and enumerations, an automatically generated enumeration corresponding to line ranges in the mission script, and a mission-specific configuration file mirroring some relevant mission-specific configuration for the diagnosis system. The code in the monitors is not complex, but adjusting various parameters, for example, hysteresis levels of the stern plane angle when moving between various angles of deflection, proved to be quite time consuming. The monitors depend on 17 parameters that are reused between campaigns. The 11 monitors follow 30 different variables that have been recorded by the logger node and update 23 discrete values in the L2 model.

D. Onboard and Offboard Diagnosis

Running third party software onboard an expensive vehicle requires extensive preparation. Therefore, let us look at the alternative of splitting the diagnosis system between the vehicle and the support ship. We call this approach *offboard diagnosis* because the data monitors and the real-time interface run on the AUV and communicate discretized values to the support ship where the actual diagnosis algorithm runs. As the acoustic link

[2]The tools, some data samples, and additional information can be obtained from http://www.cs.bham.ac.uk/go/afda.

has very modest bandwidth, 80 B every 30 s, it is impossible to transfer, e.g., acoustic Doppler current profiler (ADCP) data that produces readings at 2-Hz rate over to the support ship. Based on the discretizations of actual missions, we can show that it is possible to run the diagnosis algorithm on the support vessel by modifying the data sent over the acoustic link to contain data relevant to the diagnosis system.

In fact, we have built the diagnosis system in such a way that it is able to detect faults even if intermediate frames of data have been lost. Although there are 30 variables that constitute observations, most of these variables have three to four valued domains that can each be encoded in 2 B. A simple calculation shows that such information can be sent over the communication link within the order of 10 B. If we add the requirement that some variables (like the current line of the mission script) require more space, it is still possible to encode a complete frame of diagnosis data into far less than 80 B.

E. Diagnosis System and Hardware Reconfiguration

The AUV is reconfigured to a larger or lesser extent before each mission. Some of such reconfigurations involve fitting or updating some science equipment, resulting in changed power load levels. These require changes to the monitors, but typically not to the L2 model unless a component for the new equipment is being added to the model. However, other changes alter the nominal behavior of the vehicle, and in these cases the model may need substantial changes. For example, in 2009, a collision avoidance sonar and appropriate software support was developed for Autosub 6000 [22] that enables the vehicle to change its operating mode to flying very close to the seafloor, e.g., for the purpose of photography. Adding this subsystem results in false positives being generated whenever the obstacle avoidance is triggered, since the vehicle no longer responds as expected. This would require adding new nominal modes to the depth and position control components.

V. Mission Scripts

The mission scripts used in Autosub 6000 [23] consist of a sequence of commanded behaviors known as *when blocks*, each of which defines the AUV's behavior for a fragment of the mission. The argument of a *when* statement is a set of triggering events. When a triggering event occurs, for example, a *StartCommandReceived* or *GotDepth*, the body of the *when* block is executed and the demands specified in the body, such as that the motor power should be 300 W, are activated. The script waits at the next *when* block until the triggering condition is satisfied. An example is given in Fig. 7 which shows three *when* blocks representing the start of a mission. In the first *when* block, the stern plane is fixed at $-20°$, the rudder at 5°, and the vehicle is commanded to descend to 1000-m depth. The second *when* block is triggered when the target depth is reached, at which point the vehicle is commanded to an altitude of 100 m off the seafloor and is told to track from one waypoint to another. The third *when* block is similar, and is triggered when the second waypoint is reached or when the timer runs out. In addition to the main sequence of states, each mission script may have up to two separate *termination sequences* that get activated upon abnormal termination of the mission.

```
 0  when( Start )
 1    MotorPower( 300W),
 2    SetElementTimer( 0:0:18:0),
 3    RudderAngle(5),
 4    SetDepthThreshold( 1000m),
 5    SPlaneAngle( -20);

 6  when( GotDepth )
 7    MotorPower( 250W),
 8    Altitude( 100m),
 9    TrackP( StartWayPoint1, EndWayPoint1),
10    SetElementTimer( 0:1:2:30);

11  when( GotPosition, ElementTimeout)
12    TrackP( StartWayPoint2, EndWayPoint2),
13    SetElementTimer( 0:1:2:30);
```

Fig. 7. A sample fragment of an Autosub 6000 mission script.

Since the mission script defines what the vehicle should actually be doing at each step in a mission, it is important for it to be included in the diagnosis model so deviations from this expected behavior can be detected and diagnosed. As we will see in Section VI, this is crucial to detecting real faults. The approach we take is to generate a diagnosis model of the mission script automatically from the script itself. Automatic generation is desirable since mission scripts are frequently written in the short space of time between missions to respond to data collected, and it is unreasonable to expect the mission designer to also create a diagnosis model at this time. The automatically generated mission model integrates with the hand-built hardware model of the AUV through constraint variables just as in the battery and CB example in Section IV. Mission script commands generate constraints such as MISSION SCRIPT.VERTICAL MODE = DEPTH which in turn ensures DEPTH CONTROL.VERTICAL MODE = DEPTH (we use X.Y to denote the Y variable of component X), and hence (if the system is working correctly) constrains the behavior of the stern plane as described in Section IV.

Generating a model of the expected mission also has additional advantages as many checks can be made on the script to ensure its correctness. At the same time as generating a diagnosis model to be integrated with the hardware model, we also produce an expected depth profile for the mission, and check to see if the mission is executable given the safety and performance characteristics of the AUV. For example, we check under what conditions the maximum depth would be exceeded, that no actuator limits are exceeded, and that waypoints are reachable in the time given.

A. Generating the Mission Script Model

We can think of the mission script as of a sequence of guarded update rules that update the values of demands active in each state. To formalize the semantics of mission scripts, we use the notion of *model programs* as defined in [24]. The definition of a model program is given in Definition 1.

Definition 1: A model program is a tuple $P = (\Sigma, \Gamma, \phi^0, R)$ where:

- Σ is a finite set of variables called *state variables*;

- Γ is a finite set of *action symbols*;
- ϕ^0 is a formula called the *initial state condition*;
- R is a collection $\{R_f\}_{f\in\Gamma}$ of action rules $R_f = (\gamma, U, X)$, where:
  - γ is a formula called the guard of f;
  - U is an update rule $\{x := t_x\}_{x\in\Sigma_f}$ for some $\Sigma_f \subset \Sigma$, U is called the update rule of f;
  - X is a set of variables, disjoint from Σ, called choice variables of f; each $\chi \in X$ is associated with a formula $\exists x\phi[x]$, called the range condition of χ, denoted by $\chi^{\exists x\phi[x]}$.

To illustrate how the formalization works we use the example mission script in Fig. 7. The script contains seven different kinds of demands: *MotorPower*, *SetElementTimer*, *RudderAngle*, *SetDepthThreshold*, *SPlaneAngle*, *Altitude*, and *TrackP*. In the formalization, these variables are all contained in Σ. The initial values of the demands correspond to the rule ϕ^0. The first *when* block denoted by action symbol $w_1 \in \Gamma$ sets five demands, thus we can think of it as action w_1 with an update rule U_1 which assigns values to five variables. The next *when* block corresponds to w_2 and U_2 and sets four demands (resetting some of the previously set demands). Update rules U_3 of action w_3 set two demands. Action guards $\gamma_1, \gamma_2, \gamma_3$, respectively, need to ensure that the *when* blocks can only be executed in sequence and are triggered by appropriate events. For example, γ_2 is defined to be $e \in$ *{GotDepth}* where $e \in X$ is the event argument of action w_2.

Because of the sequential nature of mission scripts, we add a variable pc (the program counter) to Σ that tracks the progress of the mission script, and appropriate action guards to ensure that w_2 can only be executed after w_1, w_3 after w_2, etc. pc is incremented after each *when* block is executed and is of an enumerated type where the possible values correspond to the ranges of line numbers in each *when* block of the mission script; {L0L5, L6L10, L11L13} in our example. For example, γ_2 becomes $\in$ *{GotDepth}* $e \wedge pc =$ L6L10 and the modified update rule of w_1 becomes $U_1 \cup$ *{*$pc :=$ L6L10*}*.

While the above is sufficient for tracking the values of demands, the actual assignments to certain demands have side effects in the mission control system. The behavior of the depth control system of Autosub 6000 was outlined in Section III. The mode of the depth control system is changed by assignments to the *SPlane*, *Depth*, *Altitude*, and *Surface* demands. Thus, when translating mission scripts to model programs, we add a depth control mode variable dcm to Σ and, if appropriate update rules U_i of action w_i contain any of the four listed depth control demands, we add an appropriate assignment $dcm :=$*{*$x \mid x \in$ *{SPlane, Depth, Altitude, Surface}}* to the update rules of w_i.

Thus, the set of state variables Σ becomes $D \cup \{pc, dcm\}$ where D is the set of all demand variables of the AUV.

It is possible to distinguish triggering events in the *when* blocks that take time, e.g., it takes time to reach a depth of 1000 m from the surface and it takes time to reach a waypoint. Thus, we can perform some transient analysis on the mission scripts and add information about transient states to the monitor that sends commands to the mission script model. Such transients are modeled as transient variables that are kept active by the monitor for a time constant that is specified in the vehicle-specific configuration, 3000 ms in the case of Autosub 6000. Such an approach allows us to activate constraints that should hold during a transition from, e.g., FIXED STERN PLANE to DEPTH control mode. The steady-state constraints of the model are activated only after the transient has expired.

In addition to generating the diagnosis model, because Autosub 6000 only logs output from the various system components, not input, we also need to automatically generate monitors to provide relevant information to various modeled components. For example, when the AUV is in altitude control mode (maintaining a distance above the seafloor), the actual demanded altitudes are not available in the logger, where the diagnosis engine resides. For this reason, we automatically generate a monitor that translates line numbers in the mission script into altitude demands so the depth control model can use them.

B. Mission Script Model for Onboard Diagnosis

Fig. 8 shows the automatically generated L2 model of a mission script for onboard diagnosis. The *MissionLineCmd* enumeration is a finite domain representation of the program counter pc that tracks the state of the mission script. For example, L5L8 represents the fact that the AUV is executing the *when* block on lines 5–8 of the mission script. The mission script component has a state variable of type *MissionLineCmd* that is used as the command input to sequentially transition through the states. Each state of the mission script corresponds to a set of constraints which are specified in the *case* clauses of the *mode* switch statement. For example, the *when* block on lines 0–4 specifies that the depth control mode is FIXED STERN PLANE.

Given a model program representation of a mission script m the rules for generating the onboard mission script model can be summarized as follows.

- The values of pc assigned in each state are converted into a *MissionLineCmd* enumeration adding the value *noCommand* and *noConstraints*. The enumeration is used as the range of commands passed to the component.
- The values of pc assigned in each state are converted into a *ModeType* enumeration adding the value *initial* and *noConstraints*. A variable *mode* of this type is used for tracking the state of the mission script component.
- For every action w_i, given the set of variables D_m that are selected to be included in the model, add a constraint to the corresponding state (in the case clause) of the mission script model that sets up the relationship between the model variable and an abstracted value of the demand.
- For every state of the mission script add a transition to the subsequent state triggered by the corresponding command.

C. Mission Script Model for Offboard Diagnosis

Underwater operations provide a huge challenge for communication. All data exchanged between the AUV and the support ship must be sent acoustically. While the acoustic link provides communication up to the range of 7 km, its throughput is limited to 80-B packets that are sent every 20–30 s. The probability of

```
/**
* Automatically generated enumeration
* of states in the mission script.
*/

enum MissionLineCmd {noCommand,
  noConstraints,
  L0L4,
  L5L8,
  ...
  L253L255,
  L256L258
};

/**
* Automatically generated mission
* script component.
*/
class MissionScript {
  MissionLineCmd missionLineCmd;
  Modetype mode;
  MotorPower motorPowerOut;
  VerticalMode verticalModeOut;

  enum Modetype {initial,
         noConstraints,
         L0L4,
         L5L8,
         ...
         L253L255,
         L256L258
  };

  stateVector [mode];
  {
    switch (mode) {
    case initial:
      //no constraints
    case L0L4:
      verticalModeOut = SPLANE;
    case L5L8:
...
    case L45L48:
      motorPowerOut = POSITIVE;
      verticalModeOut = DEPTH;
...
    case L85L89:
      motorPowerOut = POSITIVE;
      verticalModeOut = ALTITUDE;
...
    case L247L250:
      motorPowerOut = POSITIVE;
      verticalModeOut = SURFACE;
    case L251L252:
      motorPowerOut = ZERO;
      verticalModeOut = SURFACE;
    }
}

transition L0L4(initial, L0L4) {
  missionLineCmd = L0L4;
}
transition L5L8(L0L4, L5L8) {
  missionLineCmd = L5L8;
}
...
transition L247L250(L243L246, L247L250) {
  missionLineCmd = L247L250;
}
transition L251L252(L247L250, L251L252) {
  missionLineCmd = L251L252;
}
}
```

Fig. 8. Automatically generated model of the mission script for onboard diagnosis.

packet loss increases proportionally with distance. This means that only a very small subset of the onboard telemetry can be communicated to the surface, and packet loss is very frequent.

As the operators have the capability to send a message to the AUV to abort the mission, they would like to have diagnostic capabilities based on the acoustic data, so, in addition to the onboard diagnosis, we have also built a diagnosis system based on this telemetry. However, this provides a number of additional difficulties due to the likelihood of dropped data and due to the infrequent reporting of each variable. In particular, commands are even less likely to be reported by the telemetry, so, as in the case of the example above, a model of the mission script is needed to allow the diagnosis engine to fill in the missing parts of the vehicle's behavior. In addition, since many of the monitors compute values such as derivatives of measured values (for example, computing the rate the vehicle is descending at), and these derivatives cannot be computed from infrequent telemetry data, ideally the monitors (everything left of the real-time interface in Fig. 3) should still run onboard the vehicle while the diagnosis engine itself runs on the support ship. The outputs of the monitors are all discretized, so they can be communicated to the surface using very low bandwidth, as was shown in Section IV-D. Since monitors only contain code for translating data from the concrete values to the abstract domain, running that code requires very little of central processing unit (CPU) resources. Due to the nature of the code, the confidence in such performance guarantees can easily be established by program analysis methods.

With such modification to the monitors, we can use exactly the same models of the hardware for offboard diagnosis as for onboard, taking advantage of the ability of L2 to predict unobserved values. However, the low rates of communication necessitate a small change in the model of the mission script.

The mission script model for offboard diagnosis is generated in the same way as for onboard diagnosis. The differences are in the transitions as due to the long gaps between receiving mission

```
class MissionScript {
  ...
  transition L0L4(*, L0L4) {
    missionLineCmd = L0L4;
  }
  transition L5L8(*, L5L8) {
    missionLineCmd = L5L8;
  }
  ...
  transition L247L250(*, L247L250) {
    missionLineCmd = L247L250;
  }
  transition L251L252(*, L251L252) {
    missionLineCmd = L251L252;
  }
  transition toNoConstraints(*, noConstraints) {
    missionLineCmd = noConstraints;
  }
}
```

Fig. 9. Differences in the automatically generated model of the mission script for offboard diagnosis compared to the onboard model.

script progress messages, and, due to potential message loss, it is not guaranteed that every mission script state will be seen in the telemetry. Thus, as shown in Fig. 9, transitions to a mission script state are allowed from any previous state, rather than only from the immediately previous one. To accommodate situations where the telemetry does not allow the diagnosis algorithm to tell to which mission script state the current set of readings corresponds, we add an additional transient state *noConstraints* where the values of the variables controlled by the mission script (*verticalModeOut* and *motorPowerOut* for the depth control node, for example) are unconstrained.

VI. Experiments

To demonstrate the effectiveness of model-based diagnosis for AUVs, we ran L2 on logged data from 15 actual Autosub 6000 missions of significant duration. These missions included four with various battery faults (not switching on; not providing current; only partly charged; battery internal pressure sensor failures), mission M12 that includes a stern plane actuator feedback potentiometer failure, and mission M38 where the stern plane actuator had an intermittent fault where it stuck pointing up. The data comprised a total of more than 50 h of AUV operations. The model we used included batteries, the mission script and mission control node, the stern plane, the depth control node, and the rudder. We did not model other systems for these experiments due to a combination of lack of time, lack of fault data, and lack of sensors. The complete model contained 109 variables and approximately 1000 lines of JMPL code, over half of which were generated automatically from the mission script.

Because of the differences in behavior, and because the two models are essentially independent, we describe the results of the control system model [Fig. 6(a)] separately from the power system model [Fig. 6(b)].

For the control system model, only four faults were detected in the 50 h of data. These are described in detail below.

A. Mission M12

Mission M12 is the one illustrated in Fig. 5. In this mission, the potentiometer measuring the stern plane position failed, resulting in the stern plane sticking in the down position. At the time, the vehicle was conducting a scientific survey, maintaining an altitude of 100 m. The failure resulted in the vehicle diving, but fortunately the maximum allowable vehicle depth was exceeded before Autosub hit the bottom, resulting in an abort and the vehicle returning to the surface. If the maximum depth had been deeper than the seabed at that point in the mission, the vehicle would have collided with the seabed and would not have dropped its weights. This is very similar to an event that led to a vehicle loss for an earlier Autosub vehicle, which had to be retrieved using an ROV.

The diagnosis system detected a stern plane fault in mission M12 22 173 s after mission start, approximately 10 s after a human expert can identify the fault in the data and 58 s before the abort weights were dropped.

Fig. 10 shows Autosub 6000 log data and L2 model variables for 100 s of mission M12 covering the time of the fault. The top four graphs show demanded and actual values for depth, altitude, pitch, and stern plane angle. The bottom six graphs show the values of six of the variables in the model: the difference between the demanded and actual depth and altitude, vehicle pitch, the stern plane actuator transient (this prevents any inference on the effect of the stern plane angle immediately after it has been changed), the stern plane position, and the stern plane effect on vehicle dynamics (this is UNKNOWN when the transient is active). The fault occurs around time 22 160, and the detection, shown by the vertical dashed line, is at time 22 173. The other vertical lines show the times at which the diagnosis engine performs a diagnosis step to check that the state is still consistent (these are the only times significant computation is being performed).

The fourth graph shows that at the time of the fault the stern plane demand is changing quite rapidly between negative and positive values. This results in the stern plane actuator transient being triggered whenever the demand changes sign. The result of this is that diagnosis of the effects of the stern plane is impossible until the value settles down for long enough for the transient to end (this corresponds to the stern plane staying at one setting for long enough that it will have an effect on the vehicle). Once this occurs, L2 observes that although the stern plane position should be causing the vehicle to climb (due to the depth demand being shallower than the current depth), the vehicle is pitching down. This produces two fault candidates, the stern plane actuator being stuck down, or the stern plane actuator being disconnected. The first of these is preferred since it is a higher probability fault.

B. Mission M38

In mission M38, the stern plane actuator intermittently gets stuck in the “up” position, causing the vehicle to climb. It appears the fault is depth dependent, so the vehicle rises, then starts to dive again, and so on. The diagnosis system detected a stern plane stuck up fault at second 50 727, approximately 5 s after it first occurred in the data.

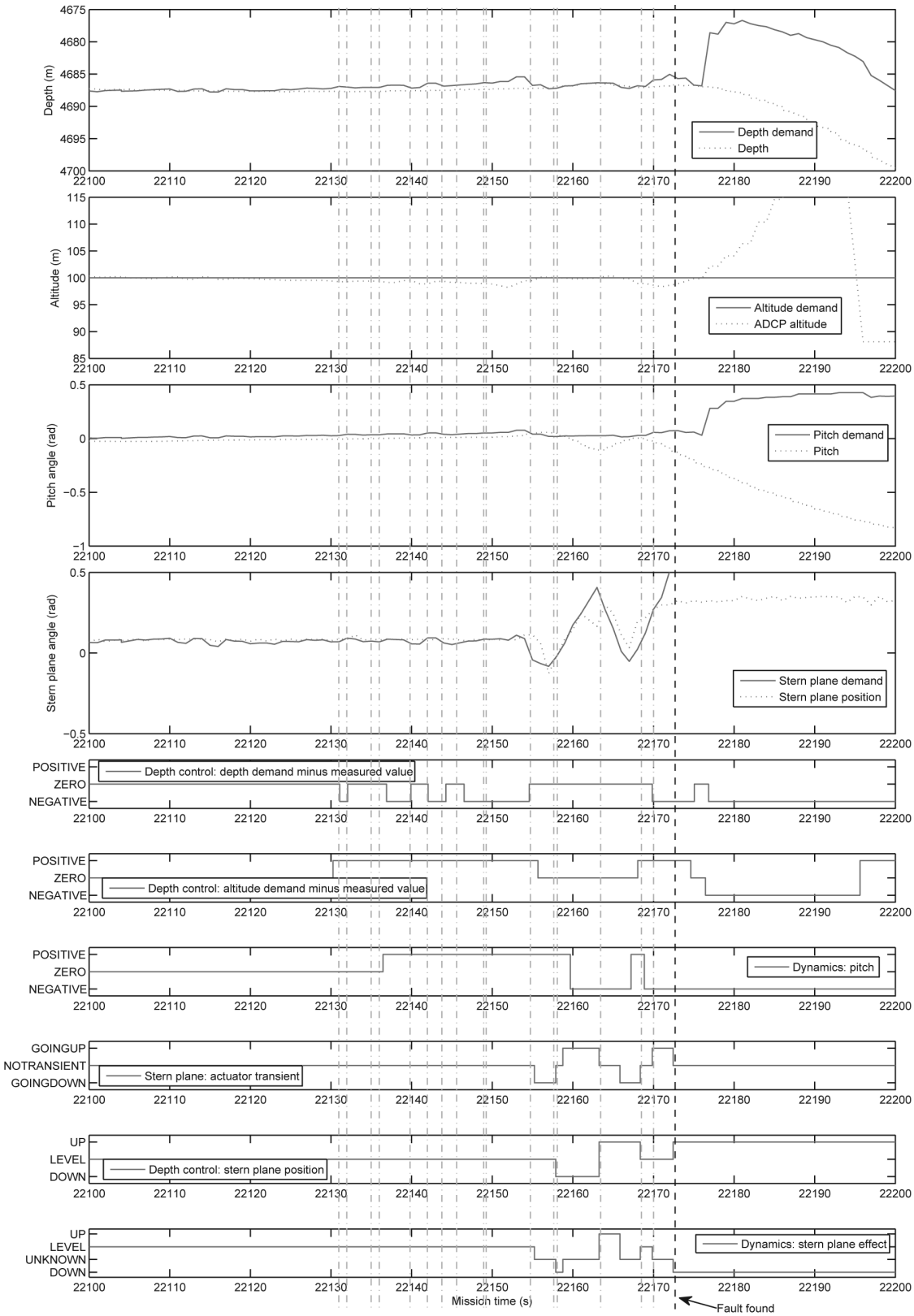

Fig. 10. Data from the time when the fault occurred in Autosub 6000 in mission M12. Here the vehicle is in altitude mode so the actual depth should match the demanded depth. At time 22 163, the potentiometer measuring stern plane position fails and the vehicle begins to dive.

C. Mission M10

Running the diagnosis system on an early trial mission of Autosub 6000, the system returned a stern plane fault at second 16 450. While we initially considered it to be a false positive, a closer inspection of the data revealed that the diagnosis of a possible stern plane problem was due to the vehicle oscillating around its horizontal axis, i.e., rocking from side to side while

TABLE II
THE FUNCTIONALITY OF L2 ON AUTOSUB 6000 COMPARED TO PREVIOUS LIVINGSTONE EXPERIMENTS

Functionality	Remote Agent (L1)	PITEX (L2)	L2 on EO-1	L2 on Autosub6000
Craft hardware in the loop	Yes	No	Yes	No
Multiple hypotheses	No	Yes	Yes	Yes
Multiple hypotheses with backtracking	No	Yes	Yes	Yes
Diagnosis during transients	No	Yes	Yes	Yes
Separation of Code and Model	Yes	No	Yes	Yes
Number of diagnostic scenarios	2	24	17	15 (10^2)
Long-term autonomous (on-board) operation	No	No	Yes	No
Automated diagnosis component generation	No	No	No	Yes
On-board and off-board diagnosis with L2	No	No	No	Yes

turning. Such behavior also caused oscillations in the vertical direction that was flagged as not nominal by the diagnosis system. As the diagnosis model currently does not have an appropriate fault model, it reported the best guess based on modeled faults.

D. Mission M11

The story with mission M11 is similar to M10: a stern plane fault was triggered during a period when the depth demand was constant, but the vehicle was rocking up and down, and was detected by the diagnosis system at second 20 006. As such behavior does not occur in later missions, it is likely that the problem was fixed by tuning the parameters of the control system of Autosub 6000 by the Autosub team.

E. Diagnosis of Battery Faults

The battery model presents a case where L2 does not perform particularly well. We have a number of missions where a variety of battery problems occurred and have tested the model on all of them. The model successfully diagnosed a problem with a Hall-effect sensor used to detect pressure problems with the batteries in a number of missions, although it could not determine if this was due to a sensor failure or a real fault. It also successfully detected a battery that failed to turn on in one mission. However, for short circuit to ground faults, it was unable to determine which battery produced the fault, so it produced a candidate set with a fault for each battery (it seems that this is in fact undiagnosable with the sensors available), and for the other battery faults in Table I (batteries not completely charged, not sourcing current or discharge fuses blown), while L2 was able to detect the faults, it could not identify them and returned an unknown fault in all cases. In the case of one mission, it also produced a false positive where it identified a battery fault on the basis of a transient variation in measured current. These problems are due to the complexity of the continuous data: thresholds on nominal current and voltage had to be made sufficiently wide to accommodate the large amount of normal variation due to changes in load without generating false positives. However, the diagnosis engine was then unable to distinguish between different kinds of faults.

One of the arguments for using L2 is that it is optimized to run using very limited computational resources. We performed our experiments on a laptop computer with a mobile 2.4-GHz Intel Core 2 Duo processor. We split the experiments up into two parts, first running the monitors on the raw logs and second, running the diagnoser on the discretized scenario. The total CPU time for the diagnosis system can be regarded as the sum of the two. Extracting the discretized values from 66 MB of raw binary log data recorded during a 24-h window that contained mission M012 with monitors written in Java took 15 s. Running the diagnoser on the mission data until the fault at second 22 173 (6 h, 10 min) since the mission started took 100 s. That included 1464 calls to the consistency engine. Based on the experiments with multiple missions, the average time per call to the consistency engine was 68 ms. These numbers demonstrate that on this processor diagnosis can be performed in real time with moderate CPU overhead.

VII. CONCLUSION AND FUTURE WORK

This paper has described our experience implementing a model-based diagnosis system on an AUV, Autosub 6000, and in particular the approach we took to integrating hardware and software diagnosis by automatically building a diagnosis model of the mission script the vehicle executes during a mission, and integrating that model with the hand-coded hardware models. We showed experimentally that the combination of the two models allowed us to diagnose faults that could not be detected with a hardware model alone, although this was to some extent because useful data about system modes was not available to the diagnosis algorithm due to the architecture of the AUV's control system. We also showed that in the equivalent of over two days of continuous operation on real data from the vehicle, all the known faults that occurred were successfully detected, and only one false positive detection occurred, and that was a fault in a single battery so it would have had no effect on the overall mission. Importantly, this can be achieved with only a small computational cost.

Table II extends the summary of application of L2 for space applications [16] with the summary of the experiments with Autosub 6000. As the table shows, although we did not get the opportunity to run L2 live on the Autosub, we have extended previous applications in terms of generating diagnosis components automatically and performing diagnosis using telemetry sent from the vehicle as well as directly on the vehicle data.

There are a few limitations of the system as it stands which we are currently investigating solutions for. First, for computational reasons, Livingstone greatly restricts the transitions allowable in the state machine models of components. In particular, it requires that nominal transitions be commanded; they cannot occur due to other events such as interactions with other components. Since the operating mode of many components is changed by commands from mission control (for instance,

setting the depth control system in stern plane control mode), this prevented us from modeling these operating modes as diagnosis model modes; instead we used statements in a single model. This is a rather inelegant solution, but more importantly, it makes it much harder to diagnose the control network on the vehicle in a sensible way. By removing the restriction on transitions and allowing one component to "command" another in the diagnosis model we can significantly improve the coverage of the diagnosis system. However, we are still investigating what the computational overhead will be.

An important property of any diagnosis system is its coverage—what it can and cannot diagnose. Currently, since we have been largely using real logged vehicle data, we have not investigated this question for our system in any systematic way. There is a tool for Livingstone called Livingstone Pathfinder [25] that uses simulation to check that Livingstone is providing correct diagnoses. Unfortunately, the tool is not publicly available so we have been unable to apply it to our models. Obviously, measuring the coverage of a diagnosis system is important in evaluating its effectiveness and assuring system operators that the diagnoses will be correct. We are currently evaluating a number of approaches for doing this.

Finally, we note that there are a number of subsystems and faults in an AUV that discrete diagnosis systems will find very challenging to diagnose. For example, distinguishing a case where the vehicle is moving against a strong current from one where the propeller is entangled or otherwise impaired is very difficult as it is easy for the discretization of the variables to obscure the effects of these situations on the relationship between motor power and vehicle speed. As we noted above, we have similar challenges when diagnosing the batteries. For this reason, we are investigating ways to do hybrid diagnosis without the high computational costs of existing approaches by doing as much of the diagnosis as possible in the discrete domain, but allowing continuous quantities where there is no way to perform diagnosis without them. Possible solutions include combining L2 with handcrafted specialized fault detection approaches such as [20], as we suggested in the introduction, to combine consistency-based and stochastic diagnosers as in HyDe [11], or new approaches such as [26], which do hybrid diagnosis efficiently using more powerful constraint solvers.

References

[1] G. Griffiths and M. Brito, "Predicting risk in missions under sea ice with autonomous underwater vehicles," in *Proc. IEEE/OES Autonom. Underwater Veh.*, Oct. 2008, DOI: 10.1109/AUV.2008.5290536.

[2] S. C. Hayden, A. J. Sweet, and S. Shulman, "Lessons learned in the Livingstone 2 on earth observing one flight experiment," in *Proc. 1st Intell. Syst. Tech. Conf. Amer. Inst. Aeronaut. Astronaut.*, 2004, pp. 1–15.

[3] S. McPhail, "Autosub6000: A deep diving long range AUV," *J. Bionic Eng.*, vol. 6, no. 1, pp. 55–62, 2009.

[4] S. G. Tzafestas, , R. J. Patton, P. M. Frank, and R. N. Clark, Eds., "System fault diagnosis using the knowledge-based methodology," in *Fault Diagnosis in Dynamic Systems, Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1998, ch. 2–4.

[5] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki, *Diagnosis and Fault-Tolerant Control*. Secaucus, NJ: Springer-Verlag, 2006, ch. 1–4.

[6] M. W. Hofbaur and B. C. Williams, "Mode estimation of probabilistic hybrid systems," in *Proc . Int. Conf. Hybrid Syst., Comput. Control*, 2002, pp. 253–266.

[7] R. Dearden and D. Clancy, "Particle filters for real-time fault detection in planetary rovers," in *Proc. 13th Int. Workshop Principles of Diagnosis*, Semmering, Austria, 2002, pp. 1–8.

[8] N. de Freitas, R. Dearden, F. Hutter, R. Morales-Menendez, J. Mutch, and D. Poole, "Diagnosis by a waiter and a Mars explorer," *Proc. IEEE*, vol. 92, no. 3, pp. 455–468, Mar. 2004.

[9] J. de Kleer and B. Williams, "Diagnosing multiple faults," *Artif. Intell.*, vol. 32, no. 1, pp. 97–130, 1987.

[10] B. C. Williams and P. P. Nayak, "A model-based approach to reactive self-configuring systems," in *Proc. 13th Nat. Conf. Artif. Intell.*, 1996, pp. 971–978.

[11] S. Narasimhan and L. Brownston, "Hyde—A general framework for stochastic and hybrid model-based diagnosis," in *Proc. 18th Int. Workshop Principles of Diagnosis*, 2007, pp. 162–169.

[12] A. Feldman, "Approximation algorithms for model-based diagnosis," Ph.D. dissertation, Comput. Sci. Dept., Technische Universitiet, Delft, The Netherlands, 2010.

[13] J. Kurien and P. P. Nayak, "Back to the future for consistency-based trajectory tracking," in *Proc. 17th Nat. Conf. Artif. Intell.*, 2000, pp. 370–377.

[14] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams, "Remote agent: To boldly go where no AI system has gone before," *Artif. Intell.*, vol. 103, no. 1–2, pp. 5–47, 1998.

[15] B. C. Williams and R. J. Ragno, "Conflict-directed a* and its role in model-based embedded systems," *Discrete Appl. Math.*, vol. 155, no. 12, pp. 1562–1595, 2007.

[16] S. C. Hayden, A. J. Sweet, S. E. Christa, D. Tran, and S. Shulman, "Advanced diagnostic system on earth observing one," in *Proc. AIAA Space Conf. Exhibit*, San Diego, CA, Sep. 28–30, 2004, Paper 2004-6108.

[17] S. D. McPhail and M. Pebody, "Navigation and control of an autonomous underwater vehicle using a distributed, networked control architecture," *Int. J. Soc. Underwater Technol.*, vol. 23, pp. 19–30, 1998.

[18] K. Hamilton, D. M. Lane, K. E. Brown, J. Evans, and N. K. Taylor, "An integrated diagnostic architecture for autonomous underwater vehicles: Research articles," *J. Field Robot.*, vol. 24, no. 6, pp. 497–526, Jun. 2007.

[19] G. Antonelli, "Fault detection/tolerance strategies for AUVs and ROVs," in *Underwater Robots*, ser. Springer Tracts in Advanced Robotics, 2nd ed. Berlin, Germany: Springer-Verlag, 2006, vol. 2, pp. 79–91.

[20] A. Alessandri, T. Hawkinson, A. J. Healey, and G. Veruggio, "Robust model-based fault diagnosis for unmanned underwater vehicles using sliding mode-observers," in *Proc. 11th Int. Symp. Unmanned Untethered Submersible Technol. Autonom. Undersea Syst. Inst.*, 1999, pp. 22–25.

[21] J. Ezekiel, A. Lomuscio, L. Molnar, and S. M. Veres, "Verifying fault tolerance and self-diagnosability of an autonomous underwater vehicle," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, T. Walsh, Ed., 2011, pp. 1659–1664.

[22] S. D. McPhail, M. Furlong, and M. Pebody, "Low-altitude terrain following and collision avoidance in a flight-class autonomous underwater vehicle," *Proc. Inst. Mech. Eng. M/J. Eng. Maritime Environ.*, vol. 224, pp. 279–292, 2010.

[23] M. Pebody, "The contribution of scripted command sequences and low level control behaviours to autonomous underwater vehicle control systems and their impact on reliability and mission success," in *Proc. OCEANS Eur. Conf.*, Jun. 2007, DOI: 10.1109/OCEANSE.2007.4302383.

[24] M. Veanes, N. Bjørner, Y. Gurevich, and W. Schulte, "Symbolic bounded model checking of abstract state machines," *Int. J. Softw. Inf.*, vol. 3, no. 2–3, pp. 149–170, 2009.

[25] A. Lindsey and C. Pecheur, "Simulation-based verification of autonomous controllers via Livingstone PathFinder," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds. Berlin, Germany: Springer-Verlag, 2004, vol. 2988, pp. 357–371.

[26] J. Ernits and R. Dearden, "Towards diagnosis modulo theories," in *Proc. 21st Workshop Principles of Diagnosis*, Murnau, Germany, 2011, pp. 1–8.

Richard Dearden received the Ph.D. degree in Markov decision process planning from the University of British Columbia, Vancouver, BC, Canada, in 2000.

He is a Reader in Intelligent Robotics in the School of Computer Science, University of Birmingham, Edgbaston, Birmingham, U.K., where he has been since 2005. He works in the broad area of reasoning under uncertainty, and in particular on planning, execution, and fault diagnosis, all applied to autonomous robots. Before that, he spent five years leading the Model-based Diagnosis and Recovery group at NASA Ames Research Center, Moffett Field, CA, where he worked primarily on Mars rover diagnosis and planning.

Juhan Ernits received the Ph.D. degree in model checking from the Tallinn University of Technology, Tallinn, Estonia, in 2007.

From 2008 to 2011, he worked as a Research Fellow in the School of Computer Science, University of Birmingham, Edgbaston, Birmingham, U.K., specializing in automated fault diagnosis. Currently, he is with the Tallinn University of Technology. His research interests include model-based diagnosis and testing, state–space reduction methods for search with applications in model checking and classical planning, and rigorous software engineering methods for producing reliable software for robots.