Diagnosis of Autosub 6000 using Automatically Generated Software Models

Juhan Ernits [1], **Richard Dearden** [1], **Miles Pebody** [2], **and James Guggenheim** [1]

[1] *University of Birmingham, Edgbaston, B15 2TT, United Kingdom*
{*j.ernits,rwd,j.guggenheim*}*@cs.bham.ac.uk*

[2] *National Oceanography Centre, Southampton, SO14 3ZH, UK*
miles.pebody@noc.soton.ac.uk

ABSTRACT

Modern systems frequently consist of a complex mixture of hardware and software. Model-based diagnosis typically assumes that the effects of the software can be summarised by the commands sent to the hardware and thus the software can be left out of the model. In our effort to build a diagnosis system for an autonomous underwater vehicle (AUV) we have an example where this is not the case—commands sent to the hardware are not all available to the diagnosis system for a variety of reasons. In addition, the software controlling the AUV, the *mission script* is frequently completely changed from one mission to the next. Taking advantage of the fact that the mission script has a relatively simple structure that does not include loops we show that a diagnosis model of the mission script can be generated automatically that integrates with the model of the physical hardware. We show that this model allows us to diagnose faults that cannot be detected from the hardware model alone.

1 INTRODUCTION

Autosub 6000 (McPhail, 2009) is an autonomous underwater vehicle (AUV) designed to operate for days at a time at depths of up to 6000m to collect science data from the deep ocean. This is an extremely challenging task as the vehicle must execute missions in an unknown environment with minimal intervention from human operators. 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 at significant expense. In one case the Autosub 2 AUV was permanently lost 17km under the 200m thick Fimbul Ice Shelf in the Antarctic (Strutt, 2006). 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.

Based on the experience of operating the Autosub AUVs a project to apply automated diagnosis and recovery methods for Autosub 6000 was initiated with the primary focus being on the detection of faults that may result in collisions with the seabed. Collision with the seabed is undesirable because it has been demonstrated to have been one of the primary causes of vehicle loss. The project aims to provide both onboard diagnosis and, using telemetry broadcast acoustically to the support ship, off-board diagnosis when the ship is in range of the AUV. The approach we are taking is to use the Livingstone 2 (L2) diagnosis engine (Williams and Nayak, 1996; Kurien and Nayak, 2000) on Autosub. L2 is a discrete, model-based diagnosis system that is compositional, allowing models of individual components to be plugged together relatively easily to build larger models. We describe L2 in more detail in Section 3, although the version we use is essentially identical to (Kurien and Nayak, 2000; Hayden *et al.*, 2004a). Consistency-based diagnosis systems such as Livingstone are a great advantage in this application in that we can use essentially the same models both on- and off-board even though only a fraction of the data is available off-board.

Autosub 6000 provides a configurable payload space that can accommodate a range of sensor equipment that may be changed between missions. While a typical mission may last from 12 to 36 hours, the turnaround time between missions may be much shorter, with data analysis, changes to the hardware and planning for the new mission all accomplished in the space of a few hours while the batteries charge. This effort culminates in the writing of a new *mission script* that specifies the behaviour of the vehicle on the next mission. The fact that Autosub is frequently reconfigured between missions means that it is fundamentally different from the space missions for which L2 has previously been used. This leads to an important challenge: How can we update the diagnosis model quickly and correctly between missions?

In many applications of diagnosis only the hardware involved in the system needs to be modelled as the behaviour of any software is assumed to be captured by the commands sent to the hardware. This can lead to problems in diagnosing the overall health of the system and in particular in recovery as it assumes the software

is correct. However, it is particularly a problem on Autosub for two reasons: First, as we said above, the mission script is generally written between missions so the likelihood of errors in the software is greatly increased; second, the architecture of the system means that commands aren't available to the diagnosis system at all. For the on-board diagnosis, commands to components aren't broadcast on the network so can't be seen by the diagnosis engine, while in off-board diagnosis the low bandwidth and high rate of packet loss means that even if we did have access to the commands, not every command may be received on the support ship. Our solution to these problems is to automatically generate a model of the mission script before it is downloaded to the vehicle. The idea is that the line in the mission script that is being executed can be used to generate the commands sent to the relevant hardware systems, and any changes in the hardware configuration can be modelled by adding constraints into the overall system model. Fortunately, the current generation of Autosub mission scripts has a relatively simple structure, so generating a finite state machine that represents all the possible execution paths through the mission script is relatively simple.

The situation where not all commands are visible to the diagnosis system is a common one. It was also encountered in developing the diagnosis system for the Earth Observing One spacecraft (Hayden *et al.*, 2004b). The problem was overcome there by a workaround which made internal command sequences available to the diagnosis system.

The approach of building a diagnosis model of a program automatically is related to that of (Mateis *et al.*, 2000) and its successors, although our program models are much simpler, and theirs are of standalone pieces of software. Using the state of the control software to improve diagnosis is similar to ideas of software-extended diagnosis presented in (Mikaelian *et al.*, 2005). The novelty here is in combining these two ideas, with automatically generated models of the software being integrated into a hand-built model of the AUV hardware, allowing us to detect faults that cannot be detected by either part alone.

Mission failures on Autosub are generally caused by two classes of problem: hardware faults and errors in the mission script. The second of these are particularly likely in AUV operations due to the short turnaround time between missions. An important additional benefit of automatically generating a diagnosis model of the script is that various checks can be performed to try to reduce human errors in creating the mission script. These checks include making sure that constraints such as bounds on the maximum operational depth of the vehicle are not exceeded, but also calculating an expected profile of the mission to show that for example it can reach its waypoints in the time available.

In the next section we briefly describe the architecture of Autosub 6000 and describe two scenarios, one nominal and one including a fault, that demonstrate the difficulty of diagnosis without a model of the mission script. Section 3 then describes a fragment of the hardware model for the vehicle, concentrating on the depth control system as that is the most critical for vehicle safety. In Section 4 we describe the mission scripts in detail and the structure of the diagnosis model that is generated from them for on-board and off-board diagnosis. In Section 5 we describe experiments to evaluate the usefulness of the approach.

2 ARCHITECTURE OF AUTOSUB 6000

Autosub 6000's architecture consists of a network of components including science instruments, control surfaces and motors, all controlled by a single mission control component that executes the mission script and distributes the commands in it to the various components. Sensor data from each component is transmitted over the network to mission control and also to the logger, which supplies them to the diagnosis engine and makes them available for acoustic transmission to the surface. However, the commands from the mission control component are not available to the logger. The state of the mission control component is accessible only via the line number in the mission script that is currently being executed.

The vehicle only has four actuators, the propeller, the releases for the abort weights, the rudder (controlling the yaw, i.e. turning to the left and right) and the stern-plane (controlling the pitch, i.e. diving and climbing[1]). Since a major objective of the project is to prevent collisions with the seabed, and this is largely governed by the stern-plane, we will concentrate on that component here. The stern-plane is usually operated in one of the following modes:

- In DEPTH FOLLOWING mode the AUV flies at a depth set by the depth demand parameter. It uses the stern plane autonomously to control its depth to achieve the demand.
- The ALTITUDE FOLLOWING mode is a special case of the depth following mode where the depth demand is set to observed depth of the seafloor minus altitude.
- In SURFACING mode the AUV's goal is to emerge at the surface. It is also a special case of depth following mode.
- The FIXED STERN-PLANE mode is used for efficient diving and ascent by commanding the stern-plane to a particular angle. The vehicle maintains that angle until another command is given.

Similar to the DEPTH FOLLOWING and FIXED STERN-PLANE modes there are corresponding POSITION CONTROL and RUDDER CONTROL modes for the rudder.

One consequence of the control scheme is shown in Figure 1. In the top graph we see a normal descent profile for the AUV. The vehicle is initially commanded via DEPTH FOLLOWING demand to descend to 200m. Subsequently it is commanded via FIXED STERN-PLANE demand to descend in a spiral to 1000m, at which it halts to check its status and await a start command from the surface before descending again. Despite the fact that the depth demand remains at 200m, the vehicle descends because depth demands are ignored in FIXED STERN-PLANE descent.

[1] The roll of the vehicle is passively controlled by the centre of buoyancy being higher than the centre of gravity.

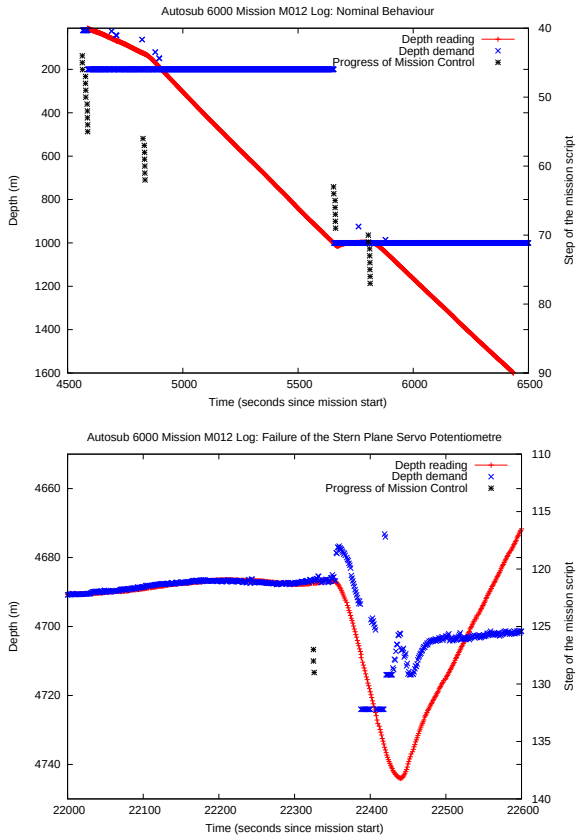

Figure 1: Nominal (above) and faulty (below) behaviour of Autosub 6000 in mission M012.

In the second graph we have an example of a fault occurring. Here the AUV is in ALTITUDE FOLLOWING mode when (at approximately time 22350) the potentiometer measuring the stern-plane angle fails. This causes the stern-plane to stick down and the vehicle to dive. To recover from the dive, mission control sets the depth demand higher, but the vehicle does not respond. Eventually, at time 22420, the vehicle's maximum depth is exceeded and the abort weights are dropped. Fortunately this occurs before the vehicle hits the seabed, which could have led to the vehicle being lost.

The key point illustrated by these two scenarios is that without knowing if the vehicle is in DEPTH FOLLOWING or FIXED STERN-PLANE mode we cannot tell if the behaviour is nominal. In both cases, the commanded depth is shallower than the vehicle depth and yet the vehicle dives. In the first case, the AUV is in FIXED STERN-PLANE mode so this is nominal behaviour, while in the second it is in DEPTH FOLLOWING mode and there is a fault. Given that these commands occur internally in the mission control node and are not logged, if we wish to detect this fault we need to be able to infer what the current mode is.

Further analysis of potential faults and corresponding mitigations based on faults that have occurred in previous missions of Autosub 6000 and its predecessors is given in (Ernits *et al.*, 2010).

3 DIAGNOSIS MODEL OF THE DEPTH CONTROL SYSTEM

We use Livingstone 2 (L2) (Hayden *et al.*, 2004b; Williams and Nayak, 1996) as our diagnosis engine. L2 is a discrete model-based diagnosis tool developed at NASA Ames Research Center primarily for spacecraft diagnosis. L2 combines a discrete event model of the system with a constraint solver that operates on the system variables and detects inconsistencies between the inferred state of the model and the observations of the actual system. One can think of an L2 model of a component as a finite state machine with states corresponding to nominal and fault modes of the system and transitions between them corresponding to commanded mode changes or faults occurring. Each state is associated with a set of constraints on the component variables that characterise the behaviour of the system in that mode. Because all the variables in L2 models are discrete, the constraints are over corresponding domains of discrete variables.

An L2 model is built from a hierarchical collection of component models where the interactions between components are represented by placing additional constraints on their variables in their parent component. For example, in the Autosub 6000 model we have components for the depth control system, the power system, various instruments, etc. In addition we have the automatically generated component representing the mission script. All of these are contained within the Autosub component where the connections between variables of different subcomponents are specified.

We illustrate the modelling effort by the example based on Figure 1. For the sake of brevity we restrict ourselves to the depth control system.

Depth control diagnosis is based around just four variables: the change in the demanded depth (either *decreased*, *increased*, or *constant*), the change in the actual depth (*decreased*, *increased*, or *constant*), the difference between the demanded and measured depth (*positive*, *negative* or *zero*), and motor power (*positive*, *negative* or *zero*). The available depth control modes were described in Section 2. For each mode we give a brief description of how the depth control component variables should behave in that mode.

- In DEPTH FOLLOWING and ALTITUDE FOLLOWING modes predicting the behaviour of the AUV qualitatively is rather difficult due to transients when the depth demand changes frequently. Thus in our model the only constraint in this mode is that if the demanded depth stays constant, then the difference between the demanded and measured depth should be decreasing or constant. In other words, if the demand is constant, the vehicle should not move away from the demand.
- In the FIXED STERN-PLANE mode the demanded depth is ignored so the constraints on depth are based on the commanded stern-plane angle rather than on demand.
- In SURFACING mode the measured depth should never increase.

In addition to these nominal modes, the system also has a number of fault modes. The relevant one for our scenario is called STERN-PLANE-STUCK-DOWN

which is what results when the potentiometer in the stern-plane fails (and also could be caused by a failure in the stern-plane motor, or a physically jammed actuator). In this fault mode the depth will increase no matter what the control mode as long as the propeller is operating and the abort weights have not been dropped.

As the description above indicates, the operation mode of the depth control system determines the variables which are relevant. However, as we said in Section 2, the mode commands are not sent to the logger and hence are not available to the diagnosis system. We solve this problem by connecting the depth control system model to the automatically generated mission script model (which we will describe in the next section). To do this we specify in the model of the complete AUV that if the network is nominal then the depth control operating mode must be equal to the mission script operating mode. Of course, if there were a fault in the on-board network that meant commands were not being delivered, this constraint would no longer hold. We have omitted modelling network faults in the current illustrative example for the sake of brevity.

The relevant sections of the static part of the Livingstone 2 model of the depth control diagnosis model of Autosub 6000 are given in Figure 2. Although the model is simplified, it is able to detect the stern plane stuck down fault in a mission where the stern plane feedback potentiometer failed. A graphical representation of the model is given in Figure 3.

4 DIAGNOSIS MODELS OF MISSION SCRIPTS

Mission scripts used in Autosub 6000 (Pebody, 2007) consist of a sequence of *when blocks*, each of which defines the AUV's behaviour for a fragment of the mission. An example is given in Figure 4 with three when blocks representing the start of a 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 300W, are activated. The script blocks at the next *when* block until the triggering condition is satisfied. 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.

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 formalise the semantics of mission scripts, we use the notion of *model programs* as defined in (Veanes *et al.*, 2009). 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*;*

```
enum MotorPower {NEGATIVE, ZERO, POSITIVE};
enum VerticalMode {SPLANE,DEPTH,TODEPTH,
   ALTITUDE,SURFACE};

class Autosub6000 {
  MissionControl missionControl;
  DepthControl depthControl;
  {
    depthControl.vertModeIn =
        missionControl.missionScript.verticalModeOut;
    depthControl.motorPowerIn =
        missionControl.missionScript.motorPowerOut;
  }
}

enum DepthDemDifference{DECREASED,CONSTANT,INCREASED};
enum DepthDifference {POSITIVE, ZERO, NEGATIVE};

class DepthControl {
  VerticalMode vertModeIn;
  MotorPower motorPowerIn;
  DepthDemDifference depthDemandDifference;
  DepthDifference depthDemandMinusMeasured;
  DepthDifference altitudeDemandMinusMeasured;
  DepthDifference deltaDepth;
  enum ModeType {nom,sPlaneStuckDown,
    sPlaneStuckLevel, sPlaneStuckUp, unknownFault};
  ModeType mode;
  stateVector [mode];
  {
   switch (mode) {
    case nom: // nominal
     if (motorPowerIn = POSITIVE) {
      if (vertModeIn=DEPTH || vertModeIn=ALTITUDE) {
       if (depthDemandDifference = CONSTANT) {
         ((depthDemandMinusMeasured = POSITIVE) |
          (depthDemandMinusMeasured = ZERO));
       }
       if (verticalModeIn = ALTITUDE ) {
        if (altitudeDemandMinusMeasured=POSITIVE) {
          ((deltaDepth = NEGATIVE) |
           (deltaDepth = ZERO)) ;
       }
      }
     }
    case sPlaneStuckDown:
        if (motorPowerIn = POSITIVE) {
          deltaDepth = POSITIVE;
    }
    case sPlaneStuckLevel:
        if (motorPowerIn = POSITIVE) {
          deltaDepth = ZERO;
    }
    case sPlaneStuckUp:
        if (motorPowerIn = POSITIVE) {
          deltaDepth = NEGATIVE;
    }
    case unknownFault:
       // no constraints
    }
  }
  failure toSPStuckD(nom,sPlaneStuckDown,unlikely) {}
  failure toSPStuckL(nom,sPlaneStuckLevel,unlikely) {}
  failure toSPStuckU(nom,sPlaneStuckUp,unlikely) {}
  failure toUnknownFault(*, unknownFault, rare) {}
}
enum MissionControlState{MCS_RUNNING,MCS_ABORTED};

class MissionControl {
  MissionScript missionScript;
  MissionControlState missionControlStateObs;
  {}
}
```

Figure 2: Simplified depth control diagnosis model of Autosub 6000.

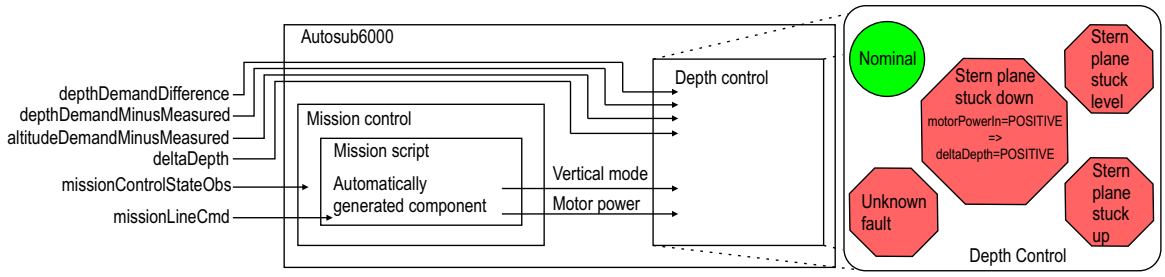

Figure 3: A graphical representation of the diagnosis model given in Figure 2. The overall structure of the model is given in the centre, with commands and observed variables on the left and the finite state machine model of the the depth control component on the right. The octagonal states in the depth control component correspond to fault states that can be transitioned to from any other state.

```
 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);
```

Figure 4: A sample fragment of an Autosub 6000 mission script.

- *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 formalisation works, let us look at the mission script in Figure 4. The script contains 7 different kinds of demands: *MotorPower*, *SetElementTimer*, *RudderAngle*, *SetDepthThreshold*, *SPlaneAngle*, *Altitude* and *TrackP*. In the formalisation, 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 triggered by appropriate events. γ_2 can be thought of as $e \in \{GotDepth\}$ 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 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 a simple program counter that is incremented after each when block is executed. pc is of an enumerated type with the value range specific to the corresponding line ranges of each particular mission script, $\{L0L5, L6L10, L11L13\}$ in the current case. For example, γ_2 becomes $e \in \{GotDepth\} \wedge pc = L6L10$ and modified update rules of w_1 become $U_1 \cup \{pc := L6L10\}$.

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 behaviour of the depth control system of Autosub 6000 was outlined in Section 3. 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 $dcm := \{x | x \in \{SPlane, Depth, Altitude, Surface\}\}$ 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 1000m from the surface and it takes time to reach a waypoint. Thus we can perform some transient analysis on the mission scripts and add transient states to the mission script model. In fact, we need only one transient state that is denoted by *TODEPTH*. *TODEPTH* is used to designate a depth mode where the AUV has just transitioned from some significantly different depth following mode to a new depth or altitude following mode. The constraints of the model are activated only after the AUV has proceeded from the transient step to the next step in the mission script.

Mission script model for onboard diagnosis

Figure 5 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 compo-

```
/**
* Automatically generated enumeration denoting
* the 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 = TODEPTH;
      ...
      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;
  }
}
```

Figure 5: Automatically generated model of the mission script for onboard diangosis.

nent 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 summarised as follows:

- The values of pc assigned in each state are converted into a MissionLineCmd enumeration adding the value *noCommand*. 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*. 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.

Mission script model for off-board 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 7km, its throughput is limited to 80 byte packets that are sent every 20-30 seconds. The probability of 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 on-board 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 behaviour. We can use exactly the same models for off-board diagnosis as for on-board, 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 off-board 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 script progress messages and due to potential message loss, it is not guaranteed that every mission script state will be seen in

```
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;
  }
}
```

Figure 6: Differences in the automatically generated model of the mission script for off-board diagnosis compared to the onboard model.

the telemetry. Thus, as shown in Figure 6, transitions to a mission script state are allowed from any previous state, rather than only from the immediately previous one. To accommodate situations where it cannot tell to which mission script state the current set of readings corresponds, we add an additional transient state *noConstraints* where the values of *verticalModeOut* and *motorPowerOut* are unconstrained.

Interfaces of the mission script component

The automatically generated mission script component has two interfaces: the interface to the monitor which listens to current mission line indications and the interface to the other components in the diagnosis model. The interface to the other components of the diagnosis model is defined in the *Autosub6000* component in Figure 2. Thus the requirement of the model generator is to only include values that are included in the corresponding domain types. The *MissionLineCmd* enumeration in mission script models also needs a corresponding monitor implementation which we generate automatically alongside the diagnosis model component.

5 RESULTS

To test the efficacy of the automatically generated mission script models we ran the diagnosis engine on logged data from eight actual Autosub 6000 missions, including the mission that includes the nominal and potentiometer fault data shown in Figure 1. The data comprised a total of approximately 85 hours of AUV operations. For comparison, we ran the diagnosis model without the mission script model included. In all cases the models generated no false positive fault detections. However, in the case of the potentiometer fault scenario, the model that did not include the mission script component failed to detect the fault, while the model with the mission script component detected it at mission time 22409 seconds since mission start, approximately 54 seconds after evidence of the fault first appears in the data and eight seconds before the abort weights were dropped because the vehicle exceeded the maximum allowed depth.

We also ran the off-board telemetry on the same scenario using simulated telemetry generated from the logs, and without any packet losses. Telemetry messages were generated every 20 seconds, so each of the five different sets of data would appear once in 100 seconds. In that case, the fault was detected at mission time 22469, one minute later than the on-board diagnosis would detect it, and in this case 52 seconds after the abort weights were dropped. This delay was largely due to the fact that the mission line only appears in one of the five telemetry messages, and the fault couldn't be detected until a mission line had been received.

While detecting the fault only eight seconds before the abort weights were dropped (or a minute after) may seem somewhat unspectacular, the abort weights were dropped only because the vehicle exceeded its maximum permitted depth. Had the vehicle not already been very close to its maximum depth the abort weights would have been released much later. In particular, 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.

6 CONCLUSIONS AND FUTURE WORK

We demonstrate that by generating diagnosis models of the program being executed we are able to provide diagnosis in situations where not all commands that are relevant to diagnosis get logged or are directly available to the diagnoser. While ideally, for the sake of reliability, we would like to have direct access to the commands, this approach enables us to work on existing logs and on the current system without requiring any changes to the software or hardware. We solved the problem of unobservable demands by translating the corresponding software in an automated way into the diagnosis model. In addition we show that slightly different versions of these automatically generated models can be used with broadcast telemetry to provide an effective off-board diagnosis system that is robust to packet loss and noise. This is an extremely valuable feature for Autosub 6000 as it lets the mission controllers see what is happening on-board the AUV in much greater detail than was previously possible and intervene if necessary. In both cases, the automatically generated models are used to constrain the behaviour of the system more than the hardware models alone can. This makes fault detection easier as there are fewer ways to explain away anomalous sensor data.

Statistics collected over the hundreds of Autosub missions carried out to date show that configuration and mission script errors constitute a significant cause of unsuccessful or partially successful missions and catching those faults before launch is valuable. The mission script checking that is carried out alongside the generation of the models has already been shown to reduce these errors.

When it comes to recovery from faults, the AUV is handicapped by its lack of redundancy in actuators and science instruments. However, we are currently looking at this problem, and in particular at decisions about

whether to abort a mission or carry on with reduced functionality if a fault occurs. The mission script models are very useful for this analysis as they allow the effects of a fault on future mission operations to be determined. One example from a recent AUV mission is a case where some batteries turned off during the mission due to a fault. As there is some redundancy in batteries for shorter missions and given that we can roughly estimate remaining energy requirements, it is possible to decide whether to continue on to the next leg of the mission or not.

Mission script models have to be automatically generated for vehicles such as Autosub that perform multiple missions with short turnaround times. It is unrealistic to expect the mission script planner to write a diagnosis model alongside the mission script, and there simply isn't time for a diagnosis expert to build a model once the script is written. Fortunately, in the case of Autosub the scripts have very simple semantics. A future challenge is to extend the approach to more complex script languages. It may even be that if on-board diagnosis becomes an important component of future autonomous vehicles, the scripting language will be constrained by the requirement that diagnosis models can be automatically generated.

ACKNOWLEDGEMENTS

This work was supported by the UK's Natural Environment Research Council SOFI grant "Automated Fault Detection for Autosub 6000". Thanks to The National Oceanographic Centre and in particular the Ocean Technology Division for their help with modelling Autosub 6000.

REFERENCES

(Ernits *et al.*, 2010) Juhan Ernits, Richard Dearden, and Miles Pebody. Automatic fault detection and execution monitoring for AUV missions. In *Autonomous Underwater Vehicles, 2010. AUV 2010. IEEE/OES*, pages 1–9, Sept. 2010.

(Hayden *et al.*, 2004a) Sandra C. Hayden, Adam J. Sweet, Scott E. Christa, Daniel Tran, and Seth Shulman. Advanced diagnostic system on Earth Observing One, 2004. AIAA paper 2004-6108. AIAA Space Conference and Exhibit, 28-30 Sep., San Diego, CA, United States, 14pp.

(Hayden *et al.*, 2004b) Sandra C. Hayden, Adam J. Sweet, and Seth Shulman. Lessons learned in the Livingstone 2 on Earth Observing One flight experiment. In *Proc. AIAA 1st Intelligent Systems Tech. Conf., Am. Inst. Aeronautics and Astronautics*, pages 1–15, 2004.

(Kurien and Nayak, 2000) James Kurien and P. Pandurang Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 370–377. AAAI Press, 2000.

(Mateis *et al.*, 2000) Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java programs for diagnosis. In Werner Horn, editor, *ECAI*, pages 171–175. IOS Press, 2000.

(McPhail, 2009) Stephen McPhail. Autosub6000: A deep diving long range AUV. *Journal of Bionic Engineering*, 6(1):55 – 62, 2009.

(Mikaelian *et al.*, 2005) Tsoline Mikaelian, Brian C. Williams, and Martin Sachenbacher. Model-based monitoring and diagnosis of systems with software-extended behavior. In *AAAI'05: Proceedings of the 20th National Conference on Artificial Intelligence*, pages 327–333. AAAI Press, 2005.

(Pebody, 2007) Miles 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. *OCEANS 2007 - Europe*, pages 1–5, June 2007.

(Strutt, 2006) J.E. Strutt. Report of the inquiry into the loss of Autosub2 under the Fimbulisen, 2006. National Oceanography Centre Southampton Research and Consultancy Report, 12, 39pp.

(Veanes *et al.*, 2009) Margus Veanes, Nikolai Bjørner, Yuri Gurevich, and Wolfram Schulte. Symbolic bounded model checking of abstract state machines. *International Journal of Software Informatics*, 3(2-3):149–170, 2009.

(Williams and Nayak, 1996) Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *AAAI/IAAI-96, Vol. 2*, pages 971–978, 1996.