Guided construction of testing scenarios for autonomous underwater vehicles using the augmented-reality framework and JavaBeans

B C Davis* and D M Lane
Ocean Systems Laboratory, Department of Electrical, Electronic and Computer Engineering, Heriot–Watt University, Riccarton, Edinburgh, UK

The manuscript was received on 27 November 2009 and was accepted after revision for publication on 10 March 2010.
DOI: 10.1243/14750902JEME195

1 INTRODUCTION

System integration and validation of embedded technologies have always been challenges, particularly in the case of autonomous underwater vehicles (AUVs). The inaccessibility of the remote environment and also the cost of field operations have been the main obstacles to the maturity and evolution of underwater technologies. Additionally, the analysis of embedded technologies is hampered by data processing and analysis time lags due to low-bandwidth data communications with the underwater platform. For the developer and operator, this makes real-world monitoring and testing challenging as they are unable to react quickly or in real time to the remote platform stimuli.

This paper discusses the different testing techniques useful for unmanned underwater vehicles (UUVs) and gives example applications where necessary. Later sections digress into more detail about a new novel framework called the augmented-reality framework (ARF) and its applications to improving pre-real-world testing facilities for UUVs. To begin with, more background is given on current testing techniques and their uses. Initially some background details are also given about AUVs.

An AUV [1] is a type of UUV. The difference between AUVs and remotely operated vehicles (ROVs) is that AUVs employ intelligence, such as...
sensing and automatic decision making, allowing them to perform tasks autonomously, while ROVs are controlled remotely by a human with communications running down a tether. AUVs can operate for long periods of time without communication with an operator as they run a predefined mission plan. An operator can design missions for multiple AUVs and monitor their progress in parallel. ROVs require at least one pilot per ROV to control them continuously. The cost of using AUVs should be drastically reduced compared with the cost of ROVs, provided that the AUV technology is mature enough to execute the task as well as an ROV. AUVs have no tether or physical connection with surface vessels and therefore are free to move without restriction around or inside complex structures. AUVs can be smaller and have lower-powered thrusters than ROVs do because they do not have to drag a tether behind them. Tethers can be thousands of metres in length for deep-sea missions and consequently very heavy. In general, AUVs require less infrastructure than ROVs; i.e. ROVs usually require a large ship and crew to operate, and these are not required with an AUV because it is easier to deploy and recover.

In general, autonomous vehicles [2] can go where humans cannot or do not want to; in more relaxed terms they are suited to doing ‘the dull, the dirty, and the dangerous’. One of the main driving forces behind AUV development is automating potentially tedious tasks which take a long time to perform manually and therefore incur large expenses. These can include oceanographic surveys, oil or gas pipeline inspection, cable inspection, and clearing of underwater mine fields. These tasks can be monotonous for humans and can also require expensive ROV pilot skills. AUVs are well suited to labour-intensive or repetitive tasks and can perform their jobs more quickly and with higher accuracy than humans can. The ability to venture into hostile or contaminated environments is a feature which makes AUVs particularly useful and cost efficient.

AUVs highlight a more specific problem. Underwater vehicles are expensive because they have to cope with the incredibly high pressures of the deepest oceans (the pressure increases by 1 atm every 10 m). The underwater environment itself is both hazardous and inaccessible, which increases the costs of operations owing to the necessary safety precautions. Therefore the cost of real-world testing, the later phase of the testing cycle, is particularly expensive in the case of UUVs. When this is coupled with poor communications with the remote platform (owing to slow acoustic methods), debugging becomes very difficult and time consuming. This incurs huge expenses or, more likely, places large constraints on the amount of real-world testing that can be feasibly achieved. It is paramount for environments which are hazardous and/or inaccessible, such as sea, air, and space, that large amounts of unnecessary real-world testing are avoided at all costs. Ideally, mixed-reality testing facilities should be available for pre-real-world testing of the platform. However, because of the expense of creating specific virtual-reality testing facilities themselves, adequate pre-real-world tests are not always carried out. This leads to failed projects crippled by costs or, worse, a system which is unreliable because of inadequate testing.

Different testing mechanisms can be used to keep real-world testing to a minimum. Hardware-in-the-loop (HIL) testing, hybrid simulation (HS), and pure simulation (PS) are common pre-real-world testing methods. In most cases, all these techniques will require a virtual world for the AUV [3]. However, the testing harness created is usually very specific to the platform. This creates a problem when the user requires testing of multiple heterogeneous platforms in heterogeneous environments. Normally this requires many specific test harnesses, but creating these is often time consuming and expensive. Therefore, large amounts of integration tests are left until real-world trials, which is less than ideal.

Real-world testing is not always feasible owing to the high cost involved. It would be beneficial to test the systems in a laboratory first. One method of doing this is via PS of data for each of the platform’s systems. This is not a very realistic scenario as it does not test the actual system as a whole and only focuses on individual systems within a vehicle. The problem with PS alone is that system integration errors can go undetected until later stages of development, since this is when different modules will be tested working together. This can lead to problems later in the testing cycle, by which time they are harder to detect and more costly to rectify. Therefore, as many tests as possible should be carried out in a laboratory. A thorough testing cycle for a remote platform would include HIL, HS, and PS testing scenarios. For example, an intuitive testing harness for HIL or HS testing would include the following:

(a) a three-dimensional (3D) virtual world with customizable geometry and terrain allowing operator observation;
(b) a sensor simulation suite providing exteroceptive sensor data which mimics the real-world data interpreted by higher-level systems;
(c) a distributed communication protocol to allow swapping of real for simulated systems running in different locations.

Thorough testing of the remote platform is usually left until later stages of development because creating a test harness for every platform can be complicated and costly. Therefore, when considering a testing harness, it is important that it is reconfigurable and very generic in order to accommodate all required testing scenarios. The ability to extend the testing harness to use specialized modules is important so that it can be used to test specialized systems. Therefore a dynamic extendable testing framework is required that allows the user to create modules in order to produce the testing scenario quickly and easily for their intended platform or environment.

2 TAXONOMY OF TESTING METHODS

The reality–virtuality continuum described by Milgram et al. [4], shown in Fig. 1, depicts the continuum from reality to virtual reality and all the hybrid stages in between. The hybrid stages between real and virtual are known as augmented reality [5] and augmented virtuality. The hybrid reality concepts are built upon by the ideas of HIL and HS. Figure 1 shows how the different types of testing conform to the different types of mixed reality in the continuum. There are four different testing types.

1. PS [6] testing of a platform’s modules is performed on an individual basis before being integrated onto the platform with other modules.

2. HIL [7] testing of the real integrated platform is carried out in a laboratory environment. Exteroceptive sensors such as sonar or video, which interact with the intended environment, may have to be simulated to fool the robot into thinking it is in the real world. This is very useful for integration testing as the entire system can be tested as a whole, allowing any system integration errors to be detected in advance of real-world trials.

3. HS [6, 8] testing of the platform in its intended environment is done in conjunction with some simulated sensors driven from a virtual environment. For example, virtual objects can be added to the real world and the exteroceptive sensor data altered so that the robot thinks that something in the sensor data set is real. This type of system is used if some higher-level modules are not yet reliable enough to be trusted to behave as intended using real data. Consequently, fictitious data are used instead, augmented with the real data, and input to the higher-level systems. Thus, if a mistake is made, it does not damage the platform. An example of this is discussed in section 4.2.

4. Real-world testing is the last stage of testing. When all systems are trusted, the platform is ready for testing in the intended environment. All implementation errors should have been fixed in the previous stages; otherwise this stage is very costly. For this stage to be as useful as possible the system designers and programmers need to have reliable intuitive feedback, in a virtual environment, about what the platform is doing; otherwise problems can be very hard to see and diagnose.

The ARF provides functionality across all stages of the continuum, allowing virtually any testing sce-
nario to be realized. For this reason it is referred to as a mixed-reality framework.

In the case of augmented reality, simulated data are added to the real-world perception of some entity. For example, sonar data on an AUV could be altered so that they contain fictitious objects, i.e. objects which are not present in the real world, but which are present in the virtual world. This can be used to test the higher-level systems of an AUV such as obstacle detection (see the obstacle detection and avoidance example in section 4.2). A virtual world is used to generate synthetic sensor data which are then mixed with the real-world data. The virtual world has to be kept in precise synchronization with the real world. This is commonly known in augmented reality as the registration problem. The accuracy of registration is dependent on the accuracy of the position and navigation systems on board the platform. Registration is a well-known problem with underwater vehicles when trying to match different sensor data sets to one another for visualization. Accurate registration is paramount for displaying the virtual objects in the correct position in the simulated sensor data.

Augmented virtuality is the opposite of augmented reality; i.e. instead of being from a robot’s or person’s perspective it is from the virtual world’s perspective; the virtual world is augmented with real-world data. For example, real data collected by an AUV’s sensors are rendered in real time in the virtual world in order to recreate the real world in virtual reality. This can be used for online monitoring and operator training [6]. This allows an AUV or ROV operator to see how the platform is situated in the remote environment, thus increasing situational awareness.

In HS the platform operates in the real environment in conjunction with some sensors being simulated in real time by a synchronized virtual environment. Similar to augmented reality, the virtual environment is kept in synchronization using position data transmitted from the remote platform. Thus simulated sensors are attached to the virtual platform and moved around in synchronization with the real platform. Simulated sensors collect data from the virtual world and transmit the data back to the real systems on the remote platform. The real systems then interpret these data as if they were real. It is important that simulated data are very similar to the real data so that the higher-level systems cannot distinguish between the two. In summary, the real platform’s perception of the real environment is being augmented with virtual data. Hence HS is inherently augmented reality. An example of a real scenario where augmented-reality testing procedures are useful is in obstacle detection and avoidance in the underwater environment by an AUV (see the obstacle detection and avoidance example in section 4.2).

HIL is another type of mixed-reality testing technique. This type of testing allows the platform to be tested in a laboratory instead of in its intended environment. This is achieved by simulating all required exteroceptive sensors using a virtual environment. Virtual sensor data are then sent to the real platform’s systems in order to fool them. In essence this is simply virtual reality for robots. Concurrently, the outputs of the higher-level systems that receive the simulated data can be relayed back and displayed in the virtual environment for operator feedback. This can help to show the system developer that the robot is interpreting the simulated sensor data correctly. HIL requires that all sensors and systems that interact directly with the virtual environment are simulated. Vehicle navigation systems are a good example since these use exteroceptive sensors, actuators, and motors to determine the position. Using simulated sensors means that the developer can specify exactly the data that will be fed into the systems being tested. This is complicated to achieve reliably in the real environment as there are too many external factors which cannot be easily controlled. Augmenting the virtual environment with feedback data from the platform for observation means that the HIL method can be augmented virtuality as well as merely virtual reality for the platform.

Consequently, HIL and HS testing are both deemed to be mixed-reality concepts; thus any testing architecture for creating the testing facilities should provide all types of mixed-reality capability and should be inherently distributed in nature.

3 THE ARF

The problem is not providing testing facilities as such, but rather being able to create them in a timely manner so that the costs do not outweigh the benefits. Any architecture for creating mixed-reality testing scenarios should be easily configurable, extendable, and unrestrictive so that it is feasible to create the testing facilities rather than to perform more expensive and less efficient real-world tests. In essence, creating testing facilities requires a short-term payout for a long-term gain. Long-term gains
are only applicable if the facilities are extendable and reconfigurable for different tasks.

The ARF is a component-based architecture which provides a framework of components that are specifically designed to facilitate the rapid construction of mixed-reality testing facilities. The ARF provides a generic extendable architecture based on JavaBeans and Java3D by Sun Microsystems, makes use of visual programming and introspection techniques to infer information about components, and consequently provides users with help via guided construction for creating testing scenarios. This allows rapid prototyping of multiple testing combinations allowing virtual-reality scenarios to be realized quickly for a multitude of different applications.

There are other architectures which provide HIL, HS, and PS capabilities such as Neptune [6]. However, they focus only on testing and do not provide the extendability and low-level architecture that allow easy extension by utilizing users' own components. This is where the ARF provides enhanced capabilities since it uses visual programming to provide a more intuitive interface that allows both quick scenario creation and configurations to be changed quickly and easily by abstracting the user from modifying the configuration files directly.

### 3.1 Visual programming

A subject that has been touched upon is that of visual programming or guided construction. One of the main performance-inhibiting problems for software programmers and designers is whether or not they know exactly how all the software modules that they require work, and knowing how to use them. More often than not, computer software modules are poorly documented and do not provide example implementations of how to use them. This is particularly the case when projects are on a tight schedule with little capital backing them, since there simply is not the time nor the money to spend on creating nicely commented and documented code. This problem is self-perpetuating since, each time a badly documented module is required, the programmer spends so much time figuring out how to use the module that there is less time to document his or her own code.

Poor documentation is merely one aspect which decreases productivity when it comes to developing software modules. Another problem for the programmer is knowing which modules are available and their functionality. Quite often package names and module names are not sufficient for the programmer to determine a module's functionality. Consequently, the programmer has to trawl through the application programming interface (API) specification for the entire programming library to find out whether or not it is useful for the purpose required. Documentation may be poor or nonexistent; even if it does exist, it can be time consuming to find out exactly what to do to use the module, because no sensible examples are given. Thus, most of the time spent by the programmer is not spent actually programming. Conversely, when the functionality of a module is known exactly, a programmer can create a program to use it with great speed. Therefore, any architecture which reduces the amount of time spent by the programmer looking at documentation enables the programmer to finish the task more quickly.

This combination of problems means that programmers spend much time 're-inventing the wheel' since existing modules are hard to locate, poorly documented, or impossible to use. This problem is rife when it comes to producing virtual environments and simulated modules for testing robots, especially AUVs. This is usually because environments are quickly 'hacked up' to fulfil one purpose without considering the many other potential usages. Monolithic programming approaches then make reconfiguration and extension almost impossible. Add-ons can sometimes be 'hacked' into the existing program, but in essence it is still a very inflexible program that will eventually become obsolete because a new usage or platform is required. At this stage it may be too complicated to try to extend the existing program to fulfil the new requirements; therefore, instead, a new program is quickly created with a few differences but is in essence just as inflexible as the first. More time spent making generic modules with basic inputs and basic outputs in a configurable environment permits changes to be made later more quickly and easily. However, when completely new functionality is required, a configurable environment still has to be reprogrammed to incorporate new modules. This can be difficult unless the environment is specifically designed to allow extension.

Visual programming [9] provides a solution for rapid module development and provides some ideas which can be harnessed to provide the basic idea behind a generic architecture for creating virtual environments. Visual programming is the activity of making programs through spatial manipulations of visual elements. It is not new and has been around
since the very early 1970s when logic circuits were starting to be designed using computer-aided design (CAD) packages. Visual programming is more intuitive than standard computer programming because visual programming provides more direct communication between human and computer which, given the correct visual queues, makes it easier for the user to understand the relationships between entities, and thus makes connecting components easier. Consider the example of taking ice cube trays out of the freezer and placing one in a drink. The human way of doing this is simply to look to locate the ice cubes, to use the hands to manipulate the ice cube out of the tray, and then to drop it into the drink. Thus any interface which allows the user to work in their natural way is going to make the task quicker. The other option, which is more like computer programming, is to have the human write down every single action required to do this. The visual programming approach might be to manipulate a visual representation of the ice cube trays by dragging and clicking a mouse. Programming is far more clunky and will take much longer, as it is not as intuitive. Therefore, visual programming aims to exploit natural human instincts in order to be as intuitive and effective as possible.

### 3.2 JavaBeans

Visual programming is a good idea; however, owing to its visual nature, it places requirements on how a module is written. This usually requires that the low-level components are programmed in a specially designed language which provides more information to the visual programming interface. This means that visual programming can only be utilized for more specific uses, such as connecting data flows using CAD packages. However, visual programming can become far more powerful if it places nearly zero restrictions on how the low-level components are created, i.e. the programming language used. In order for visual programming to be widely accepted it has somehow to make use of existing software components even if they are not designed to be used in this way. One such method of visual programming exists whereby components only have to implement a few simple 'programming conventions' in order to be able to be used visually. These special software components are called JavaBeans and are based on the Java programming language [10].

JavaBean visual programming tools work on the basis that a Java class object has been programmed adhering to certain coding conventions. Using this assumption the visual programming tool is able to use introspection techniques to infer what the inputs and outputs to a Java class are and then to display these as properties to the user. Because of Java's relatively high-level byte code compilation layer, it is relatively simple for a JavaBean processor to analyse any given class and to produce a set of properties which a user can edit visually, therefore removing the need for the programmer to write code in order to allow the configuration of Java class objects.

JavaBean programming environments currently exist which allow a user to connect and configure JavaBeans to make two-dimensional applications based on a graphical user interface (GUI). The BeanBuilder [11] is one such program which provides the user with an intuitive visual interface for creating software out of JavaBeans. However, this does not provide any extra guidance other than graphical property sheet generation. A virtual environment is needed for mixed-reality testing scenarios, and this cannot be easily provided using the Bean Builder's current platform. However, JavaBeans offer a very flexible base upon which a virtual-environment development tool can be built, since it can easily be extended via JavaBeans and also all the advantages of JavaBeans can be harnessed. Another advantage of using JavaBeans is that scenario configurations can be exported to an XML file for distribution to others and for manual configuration.

### 3.3 Architecture

The ARF provides the ability to execute all testing regimes across the reality continuum. It does this by incorporating the OceanSHELL distributed communication protocol, vehicle dynamics and navigation simulators, sensor simulation, an interactive 3D virtual world, and information display. All spatially distributed components are easily interconnected using message passing via the communication protocol, or directly by method call using the ARF’s visual programming interface based on JavaBeans. The key to the ARF’s HIL and HS capabilities is the flexibility of the communications protocol. Other external communications protocols are easily implemented by extending the event passing currently used by the ARF’s JavaBeans.

The ARF provides a new type of JavaBean which allows the user to create a 3D environment from JavaBeans. It is called a Java3DBean and is based on Java3D and JavaBeans. Java3DBeans inherently have all the functionality of JavaBean objects but with the added advantage that they are Java3D scene graph
nodes. This gives them extra features and functionality such as 3D geometry and behaviours, and they are able to interact with other Java3DBeans within the virtual world. The ARF provides a user interface which extends the JavaBean PropertySheet, allowing Java3DBeans to be configured in the same way. The user is able to construct the 3D environment using Java3DBeans and to decide which data to input or output to or from the real world. This provides unlimited functionality for HIL, HS, and PS testing since any communication protocol can be implemented in JavaBeans and used to communicate to or from the ARF to a remote platform. Mixed-reality techniques can be used to render data visually in order to increase the situational awareness of an operator of a UUV and to provide simulation of systems for testing the remote platform. This increases the rate at which errors are detected, resulting in a more robust system in less time.

3.4 OceanSHELL distributed communications protocol

The obstacle detection and avoidance example (see section 4.2) highlights the need for a location transparent communication system. The ARF requires real modules to be able to be swapped for similar simulated modules without the other systems knowing, having to be informed, or being programmed to allow it. The underlying communication protocol which provides the flexibility needed by the framework is OceanSHELL [12]. OceanSHELL provides distributed communications via User Datagram Protocol packets which allow modules to run anywhere, i.e. it provides module location transparency. Location transparency makes mixed-reality testing straightforward because modules can run either on the remote platform or somewhere else such as a laboratory.

OceanSHELL is a software library implementing a low-overhead architecture for organizing and communicating between distributed processes. OceanSHELL’s low overheads in terms of execution speed, size, and complexity make it eminently suitable for embedded applications. An extension to OceanSHELL, called JavaSHELL, is portable because it runs on Java platforms. Both JavaSHELL and OceanSHELL fully interact, the only difference being that OceanSHELL uses C structures to specify message definitions instead of the XML files which JavaSHELL uses. However, both systems are fully compatible. OceanSHELL is not only platform independent but also language independent, increasing portability.

The ARF allows the OceanSHELL message queues to be dynamically switched and the port number to be changed. This allows information flows to be rerouted by simulated modules in real time. This is ideal for carrying out HS or HIL testing. Figure 2

![Fig. 2 Illustration showing how OceanSHELL provides the backbone for switching between real and simulated (top-side) components for use with HS and HIL testing](image-url)
shows how OceanSHELL is used to communicate between the remote environment and the virtual environment.

### 3.5 The ARF features

The ARF is a configurable and extendable virtual-reality framework of tools for creating mixed-reality environments. It provides sensor simulation, sensor data interpretation, visualization, and operator interaction with the remote platform. The ARF can be extended to use many sensors and data interpreters specific to the needs of the user and target domain. It is also domain independent and can be tailored to the specific needs of the application. The ARF provides modularity and extendability by providing mechanisms to load specific modules created by the user and provides a visual programming interface used to link together the different components. Figure 3 shows the ARF GUI which the user uses to create their virtual environment. The 3D virtual environment is built using the scene graph displayed at the top left of Fig. 3.

The ARF provides many programming libraries which allow a developer to create his or her own components. In addition, the ARF has many components ready for the creation of a tailored virtual environment by the user. The more that the ARF is used, the more the component library will grow, providing increasingly greater flexibility and therefore exponentially reducing scenario creation times.

The ARF framework provides a 3D virtual world which Java3DBeans can use to display data and to sense the virtual environment. The ARF provides many basic components to build virtual environments from. These components can then be configured specifically to work as desired by the user. If the required functionality does not exist, the user can program his or her own components and add them to the ARF component library. For example, a component could be a data listener which listens for certain data messages from some sensor, on some communication protocol (OceanSHELL, serial, etc.), and then displays the data ‘live’ in the virtual environment. The component may literally be an interface to a communications protocol such as OceanSHELL, from which other components can be connected in order to transmit and receive data.

The ARF has the ability to create groups of configured components which perform some specific task or make up some functional unit. This group of components can then be exported as a SuperComponent to the ARF component library for others to use. For example, an AUV SuperComponent could include a 3D model of an AUV, a vehicle dynamics simulator, sonar, and a control input to
the vehicle dynamics (keyboard or joystick). These virtual components can then be substituted for the real AUV systems for use with HIL and HS testing or copied and pasted to provide multiple-vehicle support.

The ARF allows complete scenarios to be loaded and saved so that no work is required to recreate an environment. The ARF has components which provide interfaces to OceanSHELL sensor simulation (sonar and video), and it provides components for interpreting live OceanSHELL traffic and displaying it meaningfully in the virtual world. Figures 7 and 8 given later show a simple sonar simulation using the ARF virtual environment. This sonar can then be output to a real vehicle’s systems for some usage, e.g. obstacle detection.

The ARF provides a collection of special utility components for the user. These components are also provided in the programming API to be extended by the programmer to create his or her own JavaBeans. Java3DBeans are merely extensions to Java3D objects which adhere to the JavaBean programming conventions. The ARF is capable of identifying which Beans are Java3DBeans and therefore knows how to deal with them. The only real differences between Java3DBeans and JavaBeans is that Java3DBeans are added to the 3D virtual-world part of the ARF and that JavaBeans are only added as objects to the ARF BeanBoard (which keeps track of all objects). However, Java3DBeans can still communicate with any other objects in the ARF BeanBoard in the same way as JavaBeans.

In summary, JavaBeans are a collection of conventions which, if adhered to by the programmer, allow a Java class to be dynamically loaded and configured using a graphical interface. The configurations of objects can also be loaded and saved at the click of a mouse button to a simple human-readable XML file.

The ARF provides many utility JavaBeans and Java3DBeans which the user can use directly, or extend. These include the following:

(a) geometric shapes for building scenes;
(b) mesh file loaders for importing VRML, X3D, DXF, and many more 3D file types;
(c) input listeners for controlling 3D objects with input devices (keyboard, mouse, or joystick);
(d) behaviours for making 3D objects carry out tasks such as animations;
(e) camera control for inspecting and following the progress of objects;
(f) OceanSHELL input–output behaviours for rendering real data and for outputting virtual data from simulated sensors;
(g) provision of basic sensors for underwater technologies such as forward-looking sonar,
side-scan sonar, bathymetric sonar, altimeter, inertial measurement unit, and Doppler velocity logger (DVL);
(h) vehicle dynamics models for movement simulation.

4 CURRENT APPLICATIONS OF THE ARF

It is very hard to measure the effectiveness of the ARF in improving the performance of creating testing scenarios. Performance testing (see section 5) alone does not reflect how the ARF is likely to be used and also does not demonstrate the ARF’s flexibility either. Although the potential applications are innumerable, this section describes some representative examples of applications and topics of research that are already gaining benefit from the capabilities provided by the ARF.

4.1 Multiple-vehicle applications

The main objective of the European Project GREX [13] is to create both a conceptual framework and middleware systems to coordinate a swarm of diverse heterogeneous physical objects (underwater vehicles) working in cooperation to achieve a well-defined practical goal (e.g. a search of hydrothermal vents) in an optimized manner (Fig. 5).

In the context of GREX, algorithms for coordinated control are being developed. As these algorithms need to be tested on different vehicle platforms (and for different scenarios), real testing becomes difficult because of the cost of transporting and using vehicles; furthermore, the efficiency and safety of the different control strategies need to be tested. The ARF’s virtual environment provides the ideal test bed; simulations can be run externally and fed into the virtual AUVs, so that the suitability of the different control strategies can be observed. The virtual environment not only serves as an observation platform but also can be used to simulate sensors for finding mines, as used in the DELPHIS multi-agent architecture [14], depicted in Fig. 6.

Other applications of the DELPHIS multi-agent architecture have been demonstrated using the ARF. These include a potential scenario for the Ministry of Defence’s Grand Challenge. This involves both surface and air vehicles working together to find targets, in a village, and to inspect them. DELPHIS was tested using simulated air vehicles, rather than underwater vehicles, which executed a search, classify, and inspection task. The ARF provided the

Fig. 5  Simulated UAVs cooperating and collaborating to complete a mission more efficiently
virtual environment, vehicle simulation, and object
detection sensors required for the identification of
potential threats in the scenario. Figure 5 displays
the virtual environment view of the Grand Challenge
scenario. The top of the screen shows a bird’s eye
observation of the area with the bottom left and right
views following the survey class unmanned aerial
vehicle (UAV) and inspection class UAV respectively.
The red circles represent targets which the survey
class UAV will detect upon coming within range of
the UAV’s object sensor. Information regarding
specific targets of interest is shared between agents
utilizing the DELPHIS system. Vehicles with the
required capabilities can opt to investigate further
the detected objects and to reserve that task from
being executed by another agent. Figure 6 shows
AUVs working together to complete a lawnmower
survey task of the sea bottom. The DELPHIS system
executes this more quickly than a single AUV since
each AUV agent does a different lawnmower leg. The
agents appear to use a divide-and-conquer method;
however, this is achieved completely autonomously
and is not preprogrammed. The AUVs decide which
lawnmower legs to execute based on distance to the
leg, predictions as to what other agents will choose,
and which tasks have already been reserved by other
competing self-interested agents.

Apart from the fact that the ARF is used for
observation purposes, multiple AUVs and UAVs were
simulated which helped tremendously to observe
the behaviours and to test the DELPHIS architecture.
Basic object detection sensors provided a simple but
effective method of outputting data from the virtual
environment to DELPHIS. Detections of certain
types of object meant that new goals were added
to the plan. Generally these would be of the
following form: identify an object with one vehicle,
and then classify that object with another type of
vehicle with the appropriate sensor. Thus some of
the simulated AUVs were only capable of detecting
the objects, while others were capable of inspecting
and classifying those objects.

4.2 Obstacle detection and avoidance
One of the most common problems for unmanned
vehicles is trajectory planning. This is the need to
navigate in unknown environments, trying to reach a
goal or target, while avoiding obstacles. These environ-
ments are expected to be subject to permanent change.
As a consequence, sensors are installed on the vehicle
to provide local information continuously about these
changes. When object detections or modifications are
sensed, the platform is expected to be able to react in
real time and to adapt its trajectory continuously to the
current mission-targeted waypoint.

Testing these kinds of adaptive algorithm requires
driving the vehicle against man-made structures in
order to analyse its response behaviours. This incurs
high collision risks on the platform and clearly
compromises the vehicle’s survivability.

A novel approach to this problem uses the ARF to
remove the collision risk during the development
process. Using HS, the approach uses a set of
simulated sensors to render synthetic acoustic
images from virtually placed obstacles. The algo-

rithms are then debugged on a real platform,
performing avoidance manoeuvres over the virtual
obstacles in a real environment. Figure 2 shows the
required framework components, Fig. 7 shows the
virtual-environment view, and Fig. 8 shows the
resulting simulated sonar of the obstacles. It should
be noted that the top-side simulated components
can be switched on to replace the remote platform’s
real components, therefore achieving HIL or HS
testing. A detailed description of the evaluation and
testing of obstacle avoidance algorithms for AUVs
can be found in references [15] and [16].
4.3 Autonomous tracking for pipeline inspection

Oil companies are increasing their interest in AUV technologies to improve large-field oil availability and, therefore, production. It is known that inspection, repair, and maintenance constitute up to 90 per cent of the related field activity. This inspection is clearly dictated by the vessel’s availability. One analysis of potential cost savings suggests the use of an inspection AUV. The predicted savings of this over traditional methods for inspecting a pipeline network system are up to 30 per cent [17].

Planning and control vehicle payloads, such as the AUTOTRACKER payload [17], can provide such capabilities. However, as mentioned, vessel availability and offshore operation costs make these types of payload a difficult technology to evaluate. The ARF can provide simulated side-scan sonar sensors for synthetically generated pipeline rendering. These capabilities provide a transparent interface for the correct and low-cost debugging of the tracking technologies. Furthermore, potentially complicated scenarios, such as multiple pipeline tracking and junctions of pipes, can be easily created to test the pipe detection and decision algorithms of the AUTOTRACKER system (Fig. 9). This could not easily be tested in the real environment as real-time debugging is not available and the potential for incorrect decision due to confusion about which pipeline to follow is high, resulting therefore in a higher risk of loss of an AUV.

4.4 Nessie III

Nessie III (Fig. 10) is the Ocean Systems Laboratory’s entry to the 2008 SAUC-E. The tasks that the AUV had to execute included the following:

(a) searching for the ground targets (tyres, drop target, or cones);
(b) searching for the midwater targets (orange and green balls);
(c) touching the orange ball;
(d) dropping weights on the bottom target;
(e) surfacing in the surface zone designated on the tangent between two tyres or two cones.

Specific behaviour routines are needed to line up with the drop target for dropping the drop weights and to line up with the orange ball ready for
touching. This is straightforward if the object positions are accurate and vehicle navigation does not drift. However, the object positions are subject to inaccuracies due to sensor accuracy, false detections, and navigational drift. Thus, once an object has been detected (e.g. the orange ball), the mission planner will change its task to touching the orange ball so that the positional drift is minimal. Rather than searching for all objects first and then going back for a closer inspection later, the positions of the detected objects are stored by a world model on Nessie so that, once higher-priority tasks are completed, the AUV goes to approximately where it thinks that the object should be and starts a new

Fig. 9  Left and right side-scan sonar simulation using the ARF

Fig. 10  Nessie tracking the orange ball for real and in simulation
The ARF provides four main features which make it very powerful when being used to create virtual-reality scenarios.

1. Connection guidance is provided for the JavaBean or Java3DBean components (for connecting data flows to and from components).

5 GUIDED CONSTRUCTION MECHANISMS OF THE ARF

The ARF provides guided construction and help to the user by analysing how JavaBean classes in its repository can be connected together and configured. This information can then be relayed to the user in a visual way so that, when the user creates a component, ‘options’ are given as to which other components can be connected to the component as well as providing useful graphical editors for configuring the component. All this information is inferred about the JavaBean class without any specific editors or extra information being added by the programmer. This is the nature of JavaBean coding conventions, as they allow information to be extrapolated about a class, provided that the programmer has followed some simple guidelines when programming the class. The JavaBean patterns within the Java class are then analysed by ARF and turned into an easy-to-use GUI.

The ARF provides four main features which make it very powerful when being used to create virtual-reality scenarios.
2. Graphical property sheet generation allows rapid and simple configuration of the parameters of a JavaBean. For example, a Sonar JavaBean may have values for range, resolution, rate of rotation, field of view, etc. These can all be turned into graphical editors directly from the JavaBean properties, thus allowing simple reconfiguration by an inexperienced user.

3. SuperComponent export and import allow the user to export and import certain groups of components, their connections, and their configurations, which are useful and which are needed in different scenarios time and time again. This allows specific configurations of simulated sensors and data flows to be easily added to many different scenarios in minimal time.

4. A scenario can be easily saved and loaded to and from human-readable and highly portable XML representation.

Figures 11 and 12 show examples of the mechanism that the ARF uses to extract the information from the Java class to provide help and guidance to the user.

Many of the time savings attributable to these enhancements are described and measured in the next section.

6 PERFORMANCE TESTING

The use cases demonstrate the ARF’s many different uses, and how extendable it is because of the JavaBean component-based architecture. The increase in performance arising from using the ARF to create testing scenarios while taking into account its performance optimizations, such as guided construction, rapid configuration, and large code reuse, is unknown. This can be quantified by comparing it with the standard approach of programming a test environment by hand. Since the ARF uses JavaBean classes, it is relatively straightforward to compare the speed at which scenarios can be created using the ARF against scenarios created using the same JavaBean classes but programmed by hand.

For a fair test, the exact same JavaBean classes which are used in the ARF will be used by the programmers. The programmers will use the NetBeans integrated development environment, as this provides help when programming by showing the user-relevant documentation of methods for classes. Also a full API specification will be available to programmers. The main task is to create a small virtual environment and to connect the components together to simulate a basic AUV. The exact same components should be connected via programming using NetBeans. Measures of performance are gained from statistical feedback from the test participants as well as the tasks being timed. To make the test more useful, all test participants will be novices in using the ARF and will have only seen a brief video demonstration of how to use it, although they are experts in programming. Other performance indicators, such as the amount of help and documentation required, are also logged. Figure 13 clearly shows the increase in performance of the ARF compared with NetBeans. Number 12 on the graph shows the average values for all test participants; this equates to, on average, the fact that the ARF is over twice as fast as NetBeans. Generally, the ARF would be expected to be even faster; however, because users were unfamiliar with the ARF, they took longer to perform certain tasks. The times given above were to create a simple scenario with an AUV. Because of the ARF’s SuperComponent creation tools, the users were able first to save the AUV configuration, next to add a clone to the same environment, and then to adjust the properties so that it was slightly different. The average time taken by the users to do this in the ARF was 47 s. Furthermore, the average time taken to make an alteration and to save as a new configuration using the ARF was only 37 s compared with 77 s in NetBeans. The ability to reuse large sections of code easily via the use of SuperComponents supports the ‘create once, use many times’ ideal.

All these small increases in performance lead to greater increases in performance over a longer period of sustained use of the ARF. The availability of such testing facilities helps the platform being developed to become more mature within a shorter time frame. Different vehicle configurations can be easily created, allowing the testing of potentially new vehicle configurations to see whether they are viable and thus helping to develop the optimal solution of vehicle for a specific task.

7 CONCLUSIONS AND FURTHER WORK

The testing carried out by the ARF performance tests highlights the features of the ARF that are difficult to prove with only the use cases. The most important points to draw from the results discussion is that the ARF provides an extendable architecture which is generic enough to provide the capabilities of all the different usages. In essence, the ARF is a ‘one size fits all’ architecture which provides generality on different granular levels, giving flexibility for both pro-
Fig. 11  (a) Illustration showing how a table of graphical property editors is created to represent all the properties (JavaBean patterns) contained within a JavaBean in the ARF. Each property of a JavaBean which adheres to JavaBean coding conventions can be described as a property descriptor. This property descriptor describes the programming type of the property. The ARF can match that type to a graphical property editor (if one exists in the ARF’s property editor repository or if one has been provided with the JavaBean and registered with the property editor repository); it can aid the user in connecting this property to a JavaBean of appropriate type already created; or it can suggest JavaBeans which can be created which fit the type (linking component data flows). This information is then displayed in a useful manner to the user. Thus a property editor, or a JavaBean link editor, is added to the graphical property sheet for every property descriptor of that JavaBean. (b) The ARF provides information on which properties can be connected to other JavaBeans within the ARF by looking for JavaBean matches on the BeanBoard (which keeps track of all JavaBeans in a scenario) or in the JavaBean repository (which holds information on all available JavaBeans). This information is then displayed to the user so that they can quickly connect component data flows.
grammers and high-end users. The second most important feature that the ARF performance tests highlighted is the considerable performance improvement over conventional methods of scenario creation. Furthermore, the trade-off of using the ARF is minimal because it is built on existing JavaBeans technology and so the programmer has little work to do to interface with the ARF. The reconfigurability of the ARF is fast because of the use of simple project creation, SuperComponents, and guided construction, which all help to increase programmer efficiency.

In addition to faster scenario creation, the ARF can help to detect errors which are otherwise undetectable in real-world testing. The flexibility of the ARF allows one scenario to be reconfigured time and

---

Fig. 12 Diagram showing an example of the constituent parts of the ARF. The BeanBoard (displayed at the top right) holds all the information on JavaBeans currently configured and added to a scenario. This information can be exported to an XML file (shown in the blue dashed box) for use by others. A subset of the JavaBeans, shown in the red dashed box, can also be exported as a SuperComponent for use in other scenarios. The guided construction mechanism is displayed in the pink dashed box and this helps the user to create, connect, and configure the JavaBeans to form the representation shown on the component board (BeanBoard). Examples of the levels for each stage of a guided construction pattern are shown in the pink dashed box. Obviously, in theory, guidance could go on for ever, in some cases linking component data flows in a loop-like fashion.
time again for use in different levels of mixed-reality testing to aid discovery of new problems and to help to fix them in a timely manner. This means that a better, more mature, more reliable platform is able to be developed more quickly than without the ARF.

The analysis of the performance of the ARF provides the fundamental evidence for why the ARF is so powerful for higher-level applications, such as HIL testing scenarios. The many small time savings which the ARF provides propagate to massive time savings in more complex projects. Scenarios no longer need to be programmed without making use of any previous work; they can simply be extended, have new components bolted on, and merged with functionality from completely different projects through the use of SuperComponents. As the ARF’s Java3DBean and JavaBean component base grows, the more quickly scenarios can be created with ever more useful SuperComponents. The runaway effect of having a large, ever-increasing component base means that, before long, the design of new components will hardly ever be necessary. Further work is being carried out to increase the ARF’s component base and to expand the usages of the ARF into domains and fields other than subsea.

© Authors 2010

REFERENCES


8 Choi, S. K. and Yuh, J. A virtual collaborative world simulator for underwater robots using multidimensional, synthetic environment. In Proceedings of...


