Adaptable Software Architectures and Task Synthesis for UAVs

Howard Foster, Jeff Magee, Jeff Kramer, Sebastian Uchitel
Imperial College London
180 Queens Gate, London, SW7 2AZ. United Kingdom

Abstract

In this paper we outline a framework for an adaptable architecture in which services are provided by components and components are interconnected to support more complex services. The use of component modes and architecture constraints on modes and structure is presented as a basis for permitting the safe update of systems while deployed. The goal is an adaptable software architecture that can be easily modified while in the operational field without the need to return to a service depot and more challengingly that permits update while the system is deployed on a mission. We propose a componentised architecture that together with explicit architectural constraints, self-assembly and self-healing will provide the flexibility necessary to satisfy the requirements for dynamic adaptability in UAVs.

Keywords : UAVs, Autonomous Systems, Control Architectures, Self-Managing, Synthesis

Introduction

The accelerated development of UAVs (Unmanned Autonomous Vehicles) marks a major step-change in the evolution of both civilian and military vehicle operations. The potential for change in everything from strategic and tactical applications to manufacturing processes for the vehicles is only now becoming understood and accepted. Central to the success of the UAV concept are reliable, robust and cost-effective electronic systems. In a similar way to that of early information system architectures, early prototypes for UAVs were based upon a closed, bespoke, architecture with components defined and built linked directly for a mode of operation (such as in surveillance) yet there is now a need for these vehicles to operate in varied scenarios and with open and distributed system environments. In this paper we propose flexible software control architectures for autonomous systems and explore the adoption of adaptable, self-managed architectures, with the support of synthesised task controls from goals and software component behaviour models. Two complementary approaches are considered, firstly the direct derivation of task state machines by goal decomposition and secondly, the synthesis of task state machines from sets of scenarios expressed as sequence diagrams. The approaches are complementary in that tasks derived from scenarios can be verified with respect to properties derived from goals and conversely, goals can be validated to show that they encompass the required scenarios.

Software Architectures for Autonomous Systems

In this section, we briefly recount the development of the control architectures for autonomous robots.

Sense-Plan-Act

Perhaps the earliest robotic autonomous architecture was the “Sense-Plan-Act” (SPA) [1] architecture which defined a control system for autonomous mobile robots decomposing functional elements into a sensing system, a planning system and an execution system. The job of the sensing system is to translate sensor input (usually audio or visual) into a world model. The planner element used as input

this world model and a defined goal, and then generated a plan to achieve the goal. The execution system took this plan and performed the necessary operations on the robot to achieve the goal of the plan (Figure 1). An extended form of the SPA architecture, coined "Deliberate Reasoning" emerged whereby plans were also based upon previous knowledge (or previously gathered world data).

Figure 1: Sense-Plan-Act Architecture

A series of works using this basis were also reported in [2-4], with the attempts to resemble a closer representation of planner decision making to that of human decisions and reactions.

Subsumption

In the 1990s, the "subsumption" architectural view took a significant alternative view to those prior to that date. The subsumption architecture is a parallel and distributed computation formalism for connecting sensors to actuators in robots [5]. Brooks argues that it is better to view a methodology of creating autonomous control architectures from the viewpoint of our own existence, in that we started without reasoning about complex environments and assumed a simple world and gradually built upon this with greater knowledge. The core of the architecture is defined as "an event-dispatch architecture which waits in parallel for a number of different events and when one happens branches to the designated state (Figure 2). However, the architecture has the short coming that it is tightly-coupled between control layers. If a change occurs in an upper layer then it is likely that the whole vehicle would have to be redesigned [6]. In other words, the software is built to handle a single task.

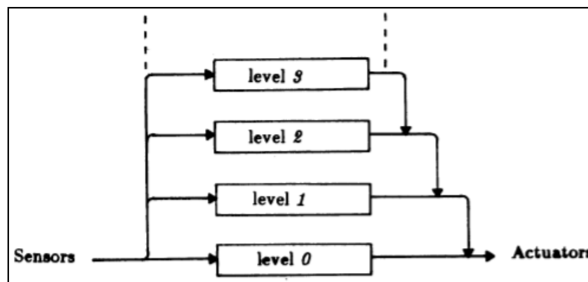

Figure 2: Subsumption Architecture

Multi-Layer and Reactive Control

To solve the problems of the subsumption architecture, Multi-layer architectures (MLAs) were proposed in the late 90's. MLAs organize algorithms according to whether they contain no state, contain state reflecting memories about the past, or contain state reflecting predictions about the future. However, this layered model alone is not sufficient to capture the flexibility required for autonomous systems that operate in multiple modes and which can be given different and complex tasks to perform. A recent proposal has addressed these issues of operating in different modes and switching between these modes and flexibly accommodating different tasks. This is an event-driven, service-based architecture for autonomous systems known as Reactive Concentric Control (RCC) [7] (Figure 3).

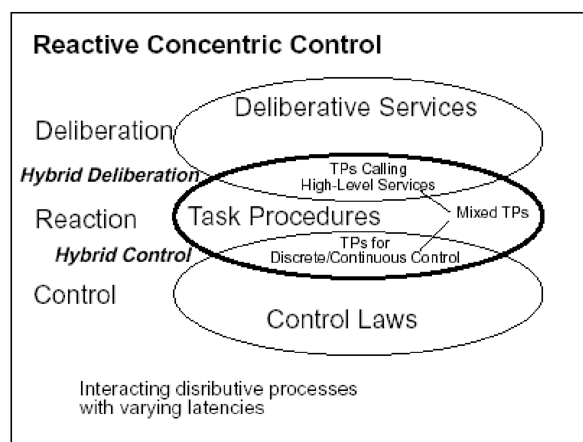

Figure 3: Reactive Concentric Control Architecture

Adaptable Software Architectures

A service is a function or set of functions that can be described by an interface that describes how to access that service without revealing how that service is implemented. Services are implemented by components that may be composed of more primitive components. A component can be a complex piece of software such as a path planner, or something less complex such as a signal processing algorithm or a data repository. In [7] the authors present a set of components that implement the services used in an aerial (in this case a helicopter) UAV, as illustrated in Figure 4.

Self-Assembly

A requirement for adaptable architecture software architectures is that it must easily accommodate the addition of new capabilities. For example, suppose we wished to add a *Grapple* capability to a UUV (Unmanned Underwater Vehicle) such that it could attach itself to objects such as the containers described in the Vignette 7A Autonomous Scenario. In addition to adding new tasks to activate and control the grappling device, it would be necessary to add new components that encapsulated the software drivers to the physical device and for it to be able to switch modes.

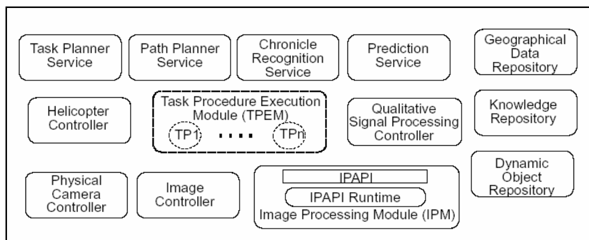

Figure 4: Services in an Aerial UAV Control Architecture

System Modes

A characteristic of autonomous vehicles is that they operate in many different modes. Using the example previously, the helicopter UAV operates in landing, take off, navigation, hovering etc. modes. A novel aspect of the approach we propose in this report is exposing and exploiting mode status at the architectural level. We believe this will be useful in controlling resource usage and in verifying the correctness of systems. In a later section of the paper, we propose a configuration of UAVs that involves UUVs operating in a slave mode in which they are attached to a container and their motor speed is controlled remotely by another UAV. The architecture of the UUV in this slave mode is depicted in Figure 5 with the components that are inactive in the mode greyed out.

Self-Healing

An extension to self-assembly is the notion of self-healing in which a system attempts to repair its architecture in response to a disturbance, such as component failure. When the failed component are removed, the system attempts to assemble itself into a configuration that satisfies overall architectural constraints. We have previously investigated this in the context of self-organising systems in which we were concerned primarily with structural constraints. In the context of UAVs, we plan to look at self-healing in the context of both structural and modal constraints. That is to look at the explicit specification of constraints that express permissible combinations of component modes and transitions between these modes such that self-healing may cause transition to a degraded mode.

Task Synthesis

In the previous section, we have outlined an approach for adaptable architectures that flexibly support the services required for a UAV. In this section, we address the problem of how to orchestrate these services to accomplish a task required of the UAV using goal orientation.

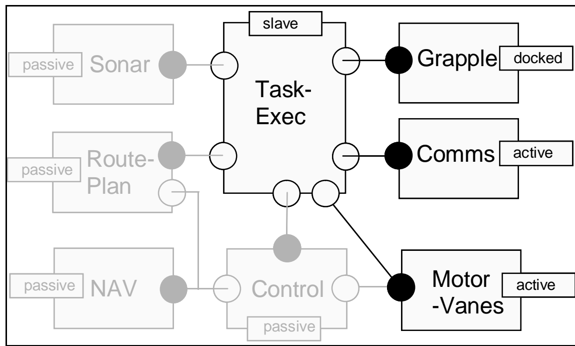

Figure 5: A UUV Software Architecture

Goal Orientation

Goal orientation is a paradigm for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting and modifying software requirements [8,9]. *Goals* are prescriptive statements of intent whose satisfaction requires the cooperation of *agents* (or active components) in the software and its environment. Goals may refer to functional or non-functional concerns and range from high-level, strategic concerns (such as "avoid explosion" for the safety control mechanism of a nuclear power plant) to low-level, technical ones (such as "safety injection overridden when block switch is on and pressure is less than 'Permit'"). We wish to use the approach of goals and goal refinement to arrive at tasks required of a UAV. We are in particular interested in mission specific tasks which are not designed in, but which need to be generated quickly from users' intentions. Goal hierarchies may be given a formal semantics in temporal logic such as has been done in KAOS [10]. These semantics need not be visible to users, however they are essential if tasks are to be generated directly from goals. In KAOS, this task generation is termed goal "operationalisation". The objective is to bridge the gap between the high-level goal desired by a user and a task which can be executed using the set of services provided by the UUV architecture. We would like this bridge to be as automated as possible and we intend to investigate work on generating goal decomposition trees given the high-level goal and the set of actions that agents can perform [10]. An example goal hierarchy for the UUV architecture is illustrated in Figure 6.

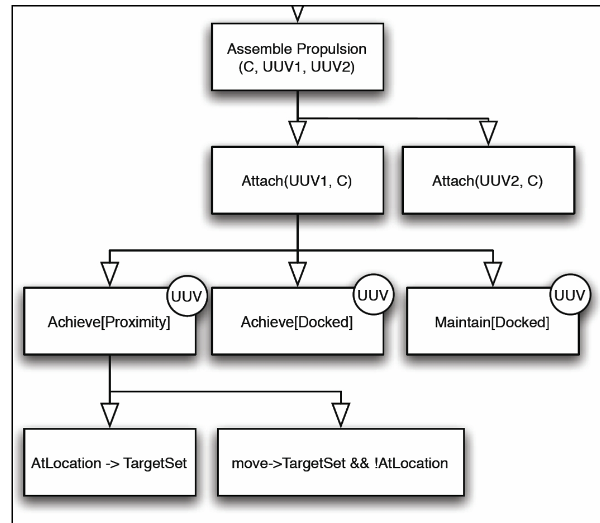

Figure 6: Goal Decomposition for Assemble Propulsion

Scenario-based Synthesis

In some cases, it is simpler for the user to specify directly and imperatively what is required rather than leaving the set of actions executed to some automated interpretation of a higher-level goal. Rather than leave it to the user to construct a state machine, it is usually more intuitive for the user to specify a set of scenarios in the form of Message Sequence Charts (MSC) that capture the allowed sequences of actions. These sequences can then be combined to form one or more state machines [11]. The example in Figure 7 gives two MSCs which can be used to generate a task to use the Grapple service of the previous section to achieve docking. Scenario a) is the situation where the UUV has achieved proximity with the container and subsequently the UUV attaches to the container using the grapple. Scenario b) is the situation where the UUV is still moving towards proximity and consequently the command to dock fails. These two scenarios can be used to generate the task state machine (LTS) depicted in Figure 8. We have of greatly simplified the docking

task for the purposes of exposition. In general, tasks generated will be too large for manual inspection as above – however our LTSA tool can deal with systems with more than 10^7 states. An approach that automates analysis and verification is needed.

Analysis and Simulation

To facilitate analysis of behaviour models, and subsequently the executable tasks derived from these models, we use the LTSA tool (Figure 9) to analyse the models for safety, liveness and temporal logic properties.

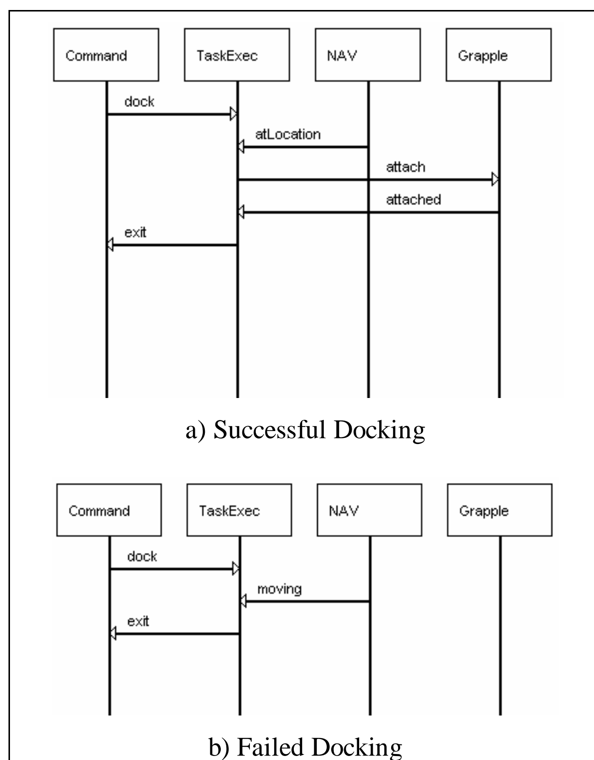

Figure 7: UUV Docking Message Sequence Charts

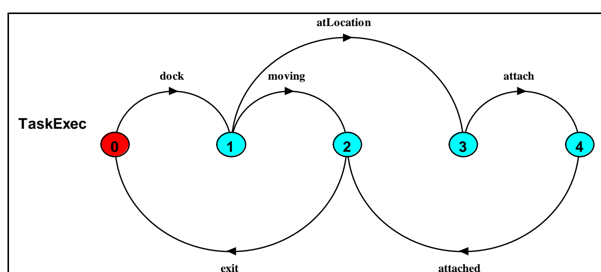

Figure 8: State Machine (LTS) for Docking Task

We can check that models synthesized from MSCs do not violate goals and conversely that models synthesized from goals accept the required scenarios. Analysis of models can determine whether the user has specified appropriate scenarios for the initial requirements and whether any additional scenarios implied by the architecture are acceptable in the autonomous system. More specifically, validation and verification consists of comparing and observing states of these transition systems. Checks can be made on the models with respect to desirable general global properties such as absence of deadlock and liveness (using formal model-checking techniques). Feedback to the user can be in the form of UML style MSCs.

The aim is to hide the underlying Labelled Transition System (LTS) representations and let the user view only the behaviour in an accessible way. Goals and scenarios are complementary both in their application to task synthesis but also in validation and verification. This approach has previously been successfully applied to Web service systems in which the analogue of UAV tasks is BPEL/simulations and business processes [12]. Our approach facilitates the use of models to drive animations and simulations, and it is our intention to exploit the same opportunity for UAVs.

Application to DTC Vignettes

The Harbour Reconnaissance Vignette has the requirement that multiple autonomous vehicles must work together to achieve the objective – that of moving container containing toxic materials out of the harbour and landing them on a near by beach. We focus on the goal of moving containers using multiple co-ordinated UAV assets rather than either the surveying the harbour or planning the scheduled and routing for container movement. We postulate a solution to container movement using one Unmanned Surface Vehicle (USV) and two Unmanned Underwater Vehicles (UUV).

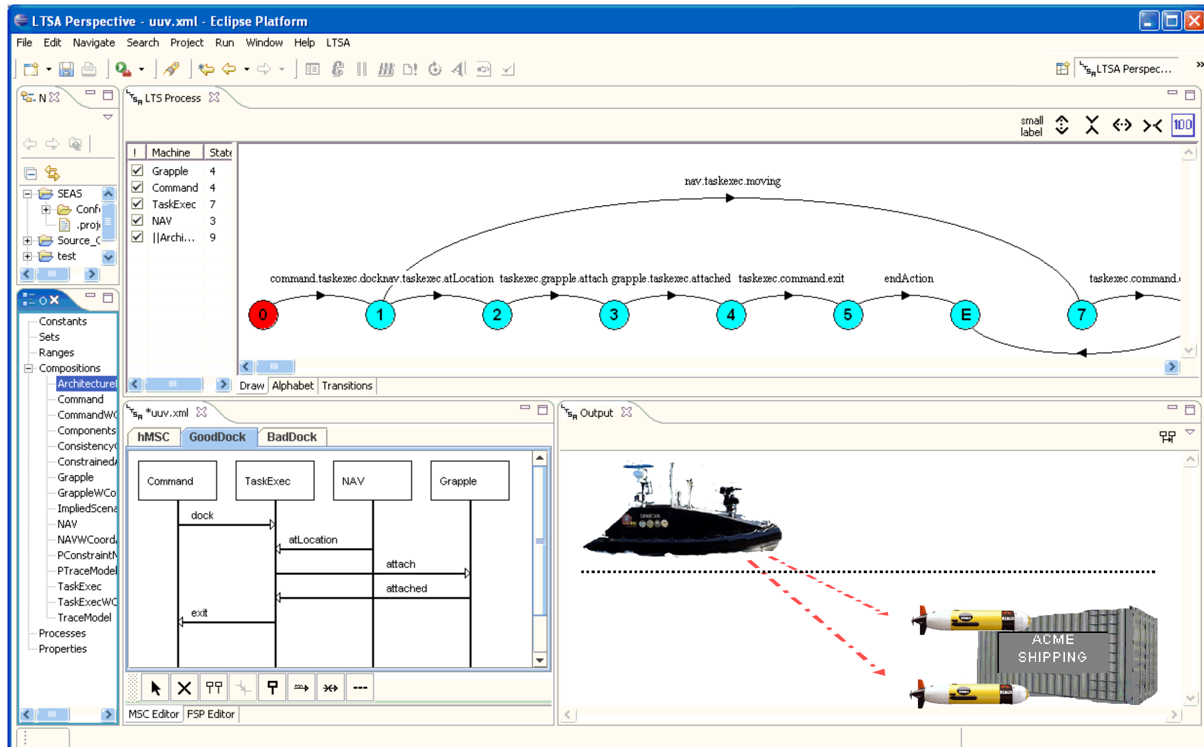

Figure 9: LTSA Tool with MSC, LTS and Simulation Views

Using our approach described in the previous sections of this paper, a goal tree divides assembling the propulsion of a container from the assembly of the control of a movable container entity which consists of the USV and the two attached UUVs. We suggest a solution in which a new control component is loaded dynamically into the USV which uses the two USVs in slave mode. This new Moveable Container Controller (MCC) encapsulates the problem of moving the container. It gets heading and speed information for the container from the USV radar system and the desired heading and speed from a higher level task running in the USV task execution component. It uses the UUVs in slave mode to affect heading and speed changes.

There are of course a number of variants to the scheme we have suggested here. For example, the USV might contribute its propulsive power by pushing the container. However, this might obscure its radar. Alternatively, if the USV is capable of producing track information for multiple objects, we could instantiate multiple MCCs on a single USV and have it control multiple containers rather than a single one. If we assume that UUVs are plentiful, we could attach more than two to each container. We have suggested this particular solution as it highlights the advantage of adaptable architectures in achieving the close coupling of UAV assets. We intend to focus on this aspect in the next phase of our work.

Conclusions

The paper has presented a framework for an adaptable software architecture in which services are provided by components and components are interconnected to support more complex services. The use of component modes and architecture constraints on modes and structure is presented as a basis for permitting the safe update of systems while deployed. The goal is an adaptable software architecture that can be easily modified while in the field without the need to return to a service depot and more challengingly that permits update

while the system is deployed on a mission. We have proposed a componentised architecture that together with explicit architectural constraints, self-assembly and self-healing will provide the flexibility necessary to satisfy the requirements for dynamic adaptability in UAVs. The approach was illustrated with a software architecture for the RAMIS-like Unmanned Underwater Vehicle (UUV) that has been suggested for Vignette 7A.

References

[1] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," IEEE Journal on Robotics and Automation, vol. RA-2, 1986.

[2] P. E. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," presented at International Joint Conference on Artificial Intelligence, Milan, Italy, 1987.

[3] R. J. Firby, "An Investigation Into Reactive Planning in Complex Domains," presented at International Joint Conference on Artificial Intelligence (IJCAI), Milan, Italy, 1987.

[4] D. Payton, "An Architecture for Reflexive Autonomous Vehicle Control," presented at International Conference on Robotics and Automation (ICRA), 1986.

[5] R. A. Brooks, "How to build complete creatures rather than isolated cognitive simulators," in Architectures for Intelligence, K. VanLehn, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1991, pp. 225-239.

[6] R. Hartley and F. Pipitone, "Experiments with the Subsumption Architecture," presented at International Conference on Robotics and Automation (ICRA), 1991.

[7] P. Doherty, P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, and B. Wingman, "A Distributed Architecture for Autonomous Unmanned Aerial Vehicle Experimentation," presented at 7th International Symposium on Distributed Robotic Autonomous Systems (DARS2004), Toulouse, France, 2004.

[8] A. v. Lamsweerde, "Handling Obstacles in Goal-Oriented Requirements Engineering," IEEE Transactions on Software Engineering, Special Issue on Exception Handling, 2000.

[9] A. v. Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour - Invited Minitutorial," presented at 5th Intl. Symp. Requirements Engineering, Toronto, Canada, 2001.

[10] R. Darimont, E. Delor, P. Massonet, and A. v. Lamsweerde, "GRAIL/KAOS: An Environment For Goal-Driven Requirements Engineering," presented at 19th International Conference on Software Engineering (ICSE'97), Boston, MA, 1997.

[11] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, "System Architecture: the Context for Scenario-based Model Synthesis," presented at ACM International Symposium on Foundations of Software Engineering (FSE'04), Newport Beach, CA, USA, 2004.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Tool Support for Model-Based Engineering of Web Service Compositions," presented at 3rd IEEE International Conference on Web Services (ICWS2005), Orlando, FL, 2005.

Acknowledgements

The work reported in this paper was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.