Plan-Directed Architectural Change For Autonomous Systems

Daniel Sykes, William Heaven, Jeff Magee, Jeff Kramer
Department of Computing
Imperial College London
{das05, wjh00, j.magee, j.kramer}@imperial.ac.uk

ABSTRACT

Autonomous systems operate in an unpredictable world, where communication with those people responsible for its software architecture may be infrequent or undesirable. If such a system is to continue reliable operation it must be able to derive and initiate adaptations to new circumstances on its own behalf. Much of the previous work on dynamic reconfigurations supposes that the programmer is able to express the possible adaptations before the system is deployed, or at least is able to add new adaptation strategies after deployment. We consider the challenges in providing an autonomous system with the capability to direct its own adaptation, and describe an initial implementation where change in the software architecture of an autonomous system is enacted as a result of executing a reactive plan.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Management, Design, Reliability

Keywords

Self-adaptive, self-healing, software architecture, dynamic reconfiguration, autonomous systems

1. INTRODUCTION

If the goal of highly reliable autonomous systems is to be realised, then the software used to control such systems must itself be reliable and highly adaptable. Furthermore it should be able to cope with failures in its components.

In this context, we consider adaptation as a modification—at runtime—of the configuration of the software components which make up the system. However we do not preclude other forms of adaptation, such as changing component parameters, or changes at the language level. Architectural change has the advantage of permitting widespread, if not total, change, while keeping the consistency and safety issues present at lower levels to a minimum. Thus we are concerned with medium to large-scale adaptations. Much previous work has focused on systems where each configuration is a self-contained and often predefined entity, or where *repair strategies* describe how to change between configurations. However, in an autonomous context, it is not feasible to consider every possible scenario beforehand, and in effect pre-program the system to cope with all circumstances which may require an architectural change.

In order to effect this arbitrary change, there must be mechanisms in place to enable the autonomous system to derive a new configuration. This requires some notion of a goal which drives the selection process. This may take the form of a functional goal whereby components are selected on the basis of what operations they perform. Alternatively, the goal may be implicit in constraints on the configuration, which may describe architectural, functional, or performance-related restrictions.

In our initial work in this area, we have developed a system which permits arbitrary dynamic reconfiguration by exploiting the presence of a reactive plan which determines the system's behaviour. Reactive plans are generated with a planning tool from high-level goals given by the user. The behaviour of the system is defined by the set of condition-action rules given in the plan. These rules indicate what components will be required to execute the plan.

In Section 2 we discuss some existing work in the area of dynamic component configuration before giving an example that motivates our approach in Section 3. Our approach is then outlined in more detail in Section 4. The paper concludes with a discussion and mention of interesting future work in Section 5.

2. RELATED WORK

Many previous authors have described approaches which assume adaptation can be specified and analysed before the system is deployed. Unfortunately, this is not always the case with autonomous systems.

Zhang, Cheng *et al.* [15] apply formal techniques to show how the safety of a transition from one steady-state program (which may be thought of as an architecture) to another can be guaranteed. They assume that the adaptive transitions are specified by the designer which requires a worst case of

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

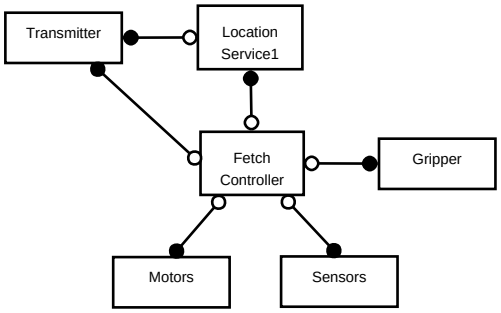

Figure 1: An initial component configuration for the "fetch" task

N^2 transitions for N configurations.

Garlan and Schmerl [5] achieve dynamic change by describing an architectural style [4] for a system and a repair strategy. The repair strategy is a script which modifies the architecture in response to changes in the monitored system properties breaking their associated constraints. Constraints may be on the architecture of the system (as in the usual notion of style) or on the performance of the system. This is a closed-adaptive [10] system since the repairs are specified before deployment. Moreover, this system does not allow architectural change to result from a change in the system's goals.

Dashofy *et al.* [3] use an architectural model and design critics [14] to determine whether a set of changes (an architectural "diff") is safe to apply. They do not directly address when the changes should be applied, but they do allow for an extensible set of repair strategies. Again, these strategies are provided by the user and not derived by the system itself.

Oreizy *et al.* [11] also use an architectural model to ensure changes are valid before they are reflected back into the running implementation. Here descriptions of reconfigurations are provided various parties such as the application vendor.

3. MOTIVATING EXAMPLE

To demonstrate the limitations of current approaches, we consider an example where a mobile autonomous system is deployed and performing a "fetch" operation which requires that it locate, pick up, and return a known object. The software architecture for such a system may resemble that in Figure 1.

The Fetch Controller is responsible for providing operations such as moving to particular locations (while avoiding obstacles), and picking up the object. The Location Service in this case informs the system of its location by communicating with a satellite via the Transmitter.

If at some point during operation, the system's battery no longer has enough power to drive the motors, the system must switch modes in order to use a Beacon component which transmits the system's location in the hope that it will be rescued by another autonomous vehicle, which may have the ability to refuel it. This configuration is shown in

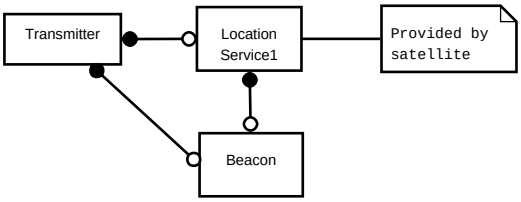

Figure 2: Component configuration following power failure

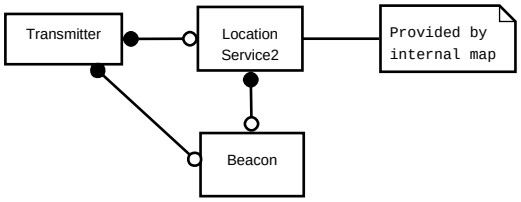

Figure 3: Component configuration following loss of satellite connection

Figure 2.

At this point, the connection to the satellite may be lost (if it moves below the horizon). This prevents the original Location Service (1) from being used, and so the system must find some alternative method for deriving the location. Location Service (2) provides location information based on local information, such as comparing short-range sensor readings to a map of the environment (this may be unreliable). This results in Figure 3. Note in this configuration there is no need for a connection between Location Service (2) and the Transmitter.

One can imagine designing repair strategies for each of these events in isolation, but as the number of possible changes increases it becomes increasingly unlikely that the situation will have been foreseen. Indeed in the worst case $n^m - 1$ repairs must be designed where m is the number of components that can change and n is the number of alternatives.

Hence, we would like to avoid pre-programming repair strategies by having the system derive changes itself.

4. APPROACH

We are experimenting with an approach that derives its own component configurations from *reactive plans* [8]. In the initial planning step, a plan is automatically generated from high-level user goals. This plan describes the behaviour of the system in terms of actions which lead from an initial state to a goal state, without explicit reference to architectural concerns. In particular, there is no correspondence between plan states and configurations. The plan is then submitted to an architecture manager which determines which components are necessary to perform the plan, and instantiates the configuration. The plan interpreter iterates through the rules of the plan to completion, unless a situation is detected which requires reconfiguration or replanning. A brief introduction to reactive plans and their generation is necessary before discussing the derivation of component configurations.

4.1 Generating Reactive Plans

A linear STRIPS-style plan [8] specifies a sequence of actions that are intended to lead from an initial state to a goal state. However, such a plan is not well suited to a non-deterministically changing environment in which a change in the environment may cause an action to lead to a state other than that expected at the time the plan was generated. If this happens, a plan must be regenerated taking into account the changed environment.

A reactive plan, on the other hand, is a plan that accommodates a non-deterministically changing environment by prescribing an action towards a given goal for each state from which that goal is reachable. Execution of such a plan proceeds by determining the current state of the environment, selecting the action prescribed for that state by the plan, performing it and then determining the new state etc. By covering all states from which the goal is reachable, it does not matter if the new state following an action is the "expected" state or not. As long as the goal is reachable from this state, execution of the plan may continue.

In our system, reactive plans are generated using planning-as-model-checking technology [6]. A *domain description* is specified in SMV [13], comprising state predicates and pre- and postcondition constraints on the actions that may be performed. This description is submitted to the Model-Based Planner tool (MBP) [12] along with a specification of the initial state I and a goal G, typically expressed in CTL [2].

The output of MBP is a set of condition-action rules such that each condition corresponds to a state in the environment from which the goal is achievable and each action is an action that may be performed in that state. Formally, this reactive plan is a partial map

$$P : S \to A$$

where S is the set of states in the state space described by the predicates of the domain description and A is the set of actions specified in the domain description. A state $s \in S$ is represented as a set of predicates $\{P_1, P_2, \ldots, P_n\}$.

If a reactive plan P is considered alongside the domain description from which it was generated, it can be represented as a labelled transition system

$$P_{LTS} = \{I, S_P, S_G, T\}$$

where I is the initial state submitted to the planning tool, S_P is the domain of P, $S_G \subseteq S_P$ is the set of states that satisfy G, and $T \subseteq S_P \times A \times S_P$ is a transition relation with transitions labelled with actions in A. T is constructed from P and the domain description so that for all states s in the domain of P there is a state s' such that $(s, a, s') \in T$ if and only if $a = P(s)$. In other words, the transition relation simply picks up the information about what state an action may lead to—which is missing from the information provided by a reactive plan alone—from the postcondition specifications of actions in the domain description.

As a small example, Figure 4 shows a reactive plan for the given domain description. LTS A represents a domain description with start state in the top left corner and goal state

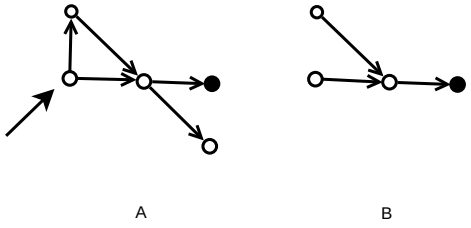

Figure 4: Example plan (B) generated from a domain description (A).

```
...
VAR
object1_location : {loc1, loc2};
rover1_location : {loc1, loc2};
rover1_has : {object1, 0};
rover1_no_power : boolean;
...
INIT
object1_location=loc1 &
rover1_location=loc2 &
rover1_no_power=0 &
rover1_has=0
...
action: {
   rover1_replenish_battery,
   rover1_move_to_loc1,
   rover1_move_to_loc2,
   rover1_pickup,
   rover1_drop,
};
...
ASSIGN next(rover1_location) :=
case
(action = rover1_move_to_loc1) : loc1;
(action = rover1_move_to_loc2) : loc2;
1 : rover1_location;
esac;
...
ASSIGN next(rover1_has) :=
case
(action = rover1_pickup)
   & rover1_location=object1_location : object1;
(action = rover1_drop) : 0;
1 : rover1_has;
esac;
...
-- etc
```

Figure 5: Example domain description fragment input to MBP.

```
-- case 1 (satisfies goal)
(case (and (= object1_location loc2))
   (done))
...
-- case i
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has object1)
      (= rover1_no_power 0))
   (action rover1_move_to_loc2))
...
-- case j
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has item1)
      (= rover1_no_power 1)
   (action rover1_replenish_battery))
...
-- case k
(case (and
      (= object1_location loc1)
      (= rover1_location loc1)
      (= rover1_has 0)
      (= object1_no_power 0))
   (action object2_pickup))
...
-- etc
```

Figure 6: Example reactive plan fragment output by MBP.

in black. LTS B represents a reactive plan which includes all states from which the goal is reachable. Where there are multiple paths to the goal, the shortest is selected. Paths which do not lead to the goal are pruned.

An example domain description, as submitted to MBP, is partially shown in Figure 5. The syntax here is that for the SMV model checker (the back end to MBP). However, the relevant elements of this example are on the whole self-explanatory. The section headed *VAR* list the predicates used to describe the state space. For instance, predicates include *object1_location*, which specifies whether *object1* is in *loc1* or *loc2*. It should be noted that the locations in a domain description are symbolic and are mapped to real locations when the system executes. The section headed *INIT* defines an initial state. Next, the domain description lists the performable actions. Actions are specified through SMV *ASSIGN* blocks, which describe the transitions between states that the system can make. Each block take the form of a case statement. To the left of the colon in each case is the precondition (for technical reasons, actions are treated as part of the precondition) and to the right is the corresponding postcondition. For instance, in the first case of the block describing how the predicate *rover1_location* can evolve, the postcondition for the action *rover1_move_to_loc1* is *rover1_location=loc1*.

This domain description is submitted to MBP along with a goal. Consider, for example, the specified objective for a rover *rover1* to fetch an object *object1* from location *loc1* and bring it to *loc2*. This objective can be captured by a goal stating that in some future state the location of *object1*

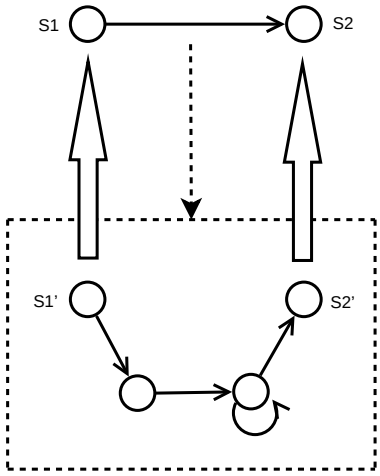

Figure 7: Refinement of an action between states S1 and S2 by a subplan.

is *loc2*. In CTL we capture this as follows:

EF*object1_location=loc2*

(where **EF** may be read "there exists some future state such that .."). Submitting the domain description shown in Figure 5 together with this goal to MBP, we get the plan partially shown in Figure 6. Each case of the plan describes a state from which the goal is reachable and maps that state to an action from the domain description.

As with all model-checking technology, the size of the state space—here determined by the number of predicates in the domain description—becomes a problem in all but the most trivial cases. To address this issue, we organise our domain description into a hierarchy of partial descriptions, generating subplans for each. In this way, each subplan addresses only a part of the overall goal and need only be generated from a partial description of the domain, reducing the number of predicates–and thus size of state space—in each plan generation.

A detailed description of this process is beyond the scope of this paper. However, the core idea is that some of the actions specified in the domain description are "primitive actions" and others are "compound actions". Primitive actions can be performed directly by the system, i.e., it is assumed that they are directly implemented by some component. Compound actions, on the other hand, are abstractions of more complex tasks that require planning. As such, when a plan is being executed and a compound action is encountered, a subplan is generated on the fly for the compound action. The plan is generated with the current state as initial state, postcondition of the compound action as the goal, and a reduced domain description relevant to the performance of the compound action.

Formally, the LTS representing the subplan generated for a compound action is a refinement of the transition representing that action in the original plan. This relationship is depicted in Figure 7, where the transition between states S1 and S2 at the top is refined by the LTS below. The set

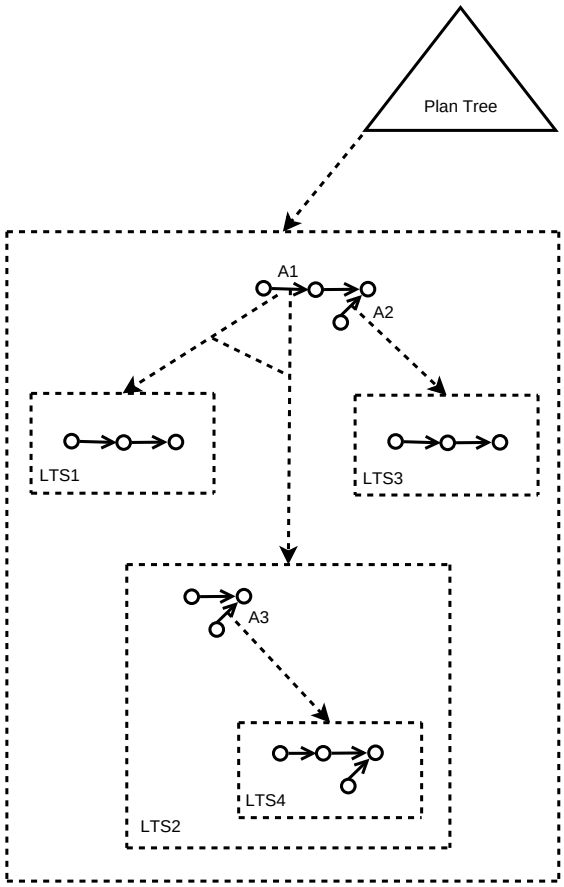

Figure 8: An example plan hierarchy

of predicates describing S1' (resp. S2') implies the set of predicates describing S1 (resp. S2). The dotted arrow and box depict this refinement relationship and will continue to do so in the sequel.

When executing a hierarchical plan of this kind, the system will request the generation of a subplan when a compound action is encountered, execute this subplan, and then jump back to and continue executing the original plan. As such, plan execution can be thought of as resembling depth first traversal of a tree. This is illustrated in Figure 8, which shows an example subtree in the planning hierarchy. It is assumed that the LTS containing actions A1 and A2 is itself a refinement of some transition above it in the plan tree. Here, it can be seen that action A1 has at least two possible refinements, LTS1 and LTS2. Though both are shown here, during execution the planning tool will pick only one alternative at a time and execution will jump to whatever subplan is first chosen. Only if this subplan fails will a request for an alternative be issued by the system. In this case, traversal of the tree would backtrack and execution will jump to LTS2, which in turn contains an action A3 which is refined by LTS4. Again, the dotted lines and boxes depict refinement relationships.

It is possible for execution of a reactive plan to go into a cycle and never reach a goal state. If execution falls into a cycle, we trigger a timeout and have the system request

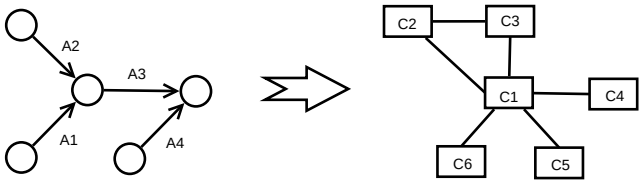

Figure 9: Example component configuration determined by actions of a plan

a different plan. The new plan will typically be generated without the action that caused the cycle, since it is likely that the environment has changed in such a way that the actual effect of this action is no longer accurately modelled by the current domain description.

The new plan will be generated from the point in the hierarchy that the previous plan had been, i.e., unless the plans are at the root of the hierarchy, both new and old plans should be alternative refinements of the same transition in a common parent plan. If no new plan can be generated with the current domain description then the system must backtrack and request a new plan from a node further up the tree.

4.2 Deriving Component Configurations

Since reactive plans are composed of condition-action rules, we are able to use the actions of the plan to derive the functional requirements of the system's architecture. For example, the presence of a move operation in the plan clearly indicates that the configuration must include a component which provides a suitable implementation of this action. We assume that the component responsible for the architectural change (which we call the architecture manager) is aware of the components which provide implementations of actions. For the purposes of deriving component configurations, we do not regard the manager as part of the architecture. Actions may be associated with particular interface types, and the manager selects components which implement the relevant interface. The mapping from actions to interfaces need not be fixed, and could be extended as new components become available.

Given the set of components required for their functionality, the manager can then construct a complete configuration by considering the required interfaces of those components. For example, the component implementing the move operation may require motor and sensor controllers, or a component which provides mapping information. These must also be instantiated and connected to the relevant ports of the action component. In the case where a component is already instantiated, it should be reused. Figure 9 shows a reactive plan and a corresponding architecture. Actions A1 and A3 may be implemented by C1 and actions A2 and A4 may be implemented by C2. The remaining components are found by considering the requirements of C1 and C2.

It may be the case that multiple components provide the same functionality, but have differing non-functional properties. For example, some implementations may require more CPU attention or provide unreliable results. We hope to develop a mechanism whereby the "best" alternative can be

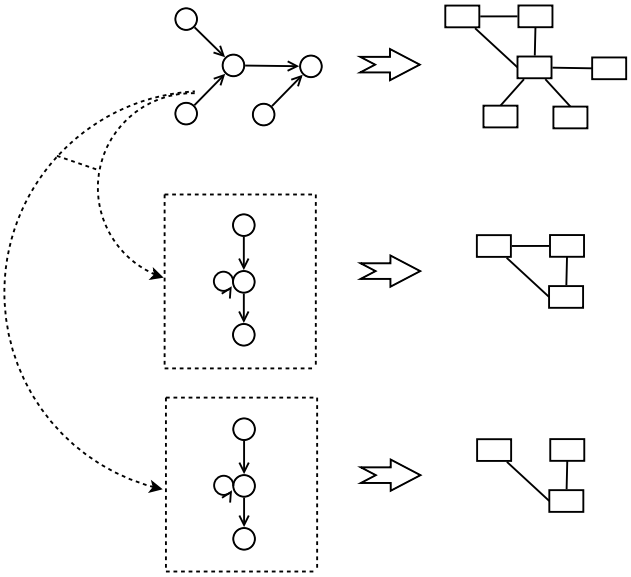

Figure 10: Overview of dynamic changes driven by automatic generation of plans. The two subplans represent alternative refinements of an action in the original LTS. Different component configurations are derived from each alternative.

selected in a given situation.

There is a trade-off to be made in terms of the number of actions a particular component can perform against how often the architecture needs to change. Clearly this depends on particular component implementations and the level of abstraction the plan designer has built into the plans. In our early implementation, components were selected and changed at every step in the plan on the assumption that a component implements only one action. However, this is clearly detrimental to efficiency. Hence, we are moving toward a system whereby a particular plan can be scanned before starting, to construct a configuration that contains components which will implement all the required actions. This is a more natural approach since the architecture is expected to be able to perform the task without changing in the absence of problems.

The exception to this principle is that the architecture may be required to change when a particular abstract action is decomposed into a subplan, since the (concrete) actions contained in the subplan were not known when the parent plan was generated. Furthermore, an abstract action may be decomposed into different subplans in different circumstances. This leads us to the diagram in Figure 10 wherein each plan and subplan has an associated architecture. The top-level plan contains an action which has two possible refinements, resulting in two potential configurations.

We do not employ a verification mechanism for configurations because we regard them as correct in the sense that they must at least provide sufficient functionality to perform the actions of the current plan. Furthermore it is reasonable to assume that the mapping between concrete actions in the plan and their implementations is correct. In other words, the two following constraints are relevant:

$$\forall a \in plan.\exists i \in arch.(a \in i) \wedge \exists c \in arch.prov(c, i)$$

$$\forall c, i \in arch.req(c, i) \longrightarrow \exists c2 \in arch.prov(c2, i)$$

Where $arch$ denotes the set of components and interfaces in the current configuration, $plan$ denotes the current plan (reduced to a set of actions) and $prov(c, i)$ and $req(c, i)$ denote that component c provides (respectively requires) interface i. An interface i is regarded as containing a set of (names of) actions a. The first constraint states that for all actions, there should be a component for the corresponding interface, and the second constraint is simply that all component requirements are satisfied. We do not (at this point) employ further structural, compatibility, or performance constraints.

The mechanisms described so far account for the first configuration change required in the example of Section 3. The Fetch Controller provides the functionality needed for actions in the fetch plan (Figure 6), and the other components in the initial configuration are requirements of the Fetch Controller. The plan checks the *rover1_no_power* predicate which causes a *rover1_replenish_battery* action. When this action is encountered, a subplan is generated which enables a rescue beacon which cycles, transmitting the current location, until the battery is refuelled. As discussed in the previous section, this subplan is a refinement of the *rover1_replenish_battery* action. In this case, the Beacon component is selected because it provides that functionality, and the Location Service (1) and the Transmitter are retained as dependencies of the Beacon, giving Figure 2.

The second case requires the system to cope with entirely unexpected faults. Our approach is to allow the manager to request replanning without using the actions associated with the component that has failed (detected by some suitable mechanism). Of course, planning comes at some cost, so there is a trade-off to be made between that and allowing the manager to perform low-level changes independently, such as substituting a component which implements the same interfaces for the one which failed. It is this latter case which is most appropriate to arrive at Figure 3. The Beacon merely cares about getting location information, and if an alternative implementation is available, it should be used without replanning.

Our implementation of this approach is built upon the Backbone system [9] which allows us to construct arbitrary configurations of components, which are implemented as Java classes. A number of problem domains have been described and executed on a set of Koala robots running a JVM.

5. DISCUSSION AND FUTURE WORK

We have described an initial scheme which addresses the problem of arbitrary dynamic reconfiguration. Reconfiguration is driven by a plan which dictates what functionality the current configuration must provide. Component selection works within the limitations of the current environment which may prevent certain components from being used.

Currently, non-functional and structural constraints on the architecture are not supported. For example, one can imag-

ine a situation where the autonomous system must avoid using components which result in the hardware drawing large amount of power, or where components must be distributed in a particular manner to meet some load balancing constraint.

It remains to be seen whether such constraints can be combined with the reactive plan which at present only prescribes the system's behaviour; the architecture is a consequence of that.

Indeed, another approach would be to employ an explicit architecture plan [1], or include reconfiguration operations within the behavioural plan. One disadvantage of following this path is that the state space for planning becomes larger, with the concomitant reduction in performance.

Other issues we seek to address are those regarding the safety of the adaptation procedure. Clearly if some components are to be replaced, then their dependants must not initiate communications with them for the duration of the change. This is the notion of quiescence [7]. It is especially important for an autonomous system to be able to keep the unaffected parts of the architecture running while reconfiguration is taking place. For the same reasons, components may require special shut down procedures before they are removed from the architecture. For example, any motor control system must ensure those motors are halted before control is released.

6. 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.

7. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 2003.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[3] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.

[5] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[6] F. Giunchiglia and P. Traverso. Planning as Model Checking. *5th European Conference on Planning*, 1999.

[7] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[8] Malik Ghallib, Dana Nau, Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman, 2005.

[9] A. McVeigh, J. Kramer, and J. Magee. Using resemblance to support component reuse and evolution. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 49–56, New York, NY, USA, 2006. ACM Press.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[11] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 (20th) International Conference on Software Engineering*, pages 177–186, 1998.

[12] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso. MBP: A Model-Based Planner. *Proc. of IJCAI'01 Workshop on Planning Under Uncertainty and Incomplete Information*, 2001.

[13] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso. NuSMV 2: An Open Source Tool for Symbolic Model Checking. *Proc. of International Conference on Computer-Aided Verification*, 2002.

[14] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Using critics to analyze evolving architectures. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 90–93, New York, NY, USA, 1996. ACM Press.

[15] J. Zhang and B. Cheng. Modular model checking of dynamically adaptive programs. Technical report, Michigan State University, 2006.