From Goals To Components: A Combined Approach To Self-Management

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 or semi-autonomous systems are deployed in environments where contact with programmers or technicians is infrequent or undesirable. To operate reliably, such systems should be able to adapt to new circumstances on their own. This paper describes our combined approach for adaptable software architecture and task synthesis from high-level goals, which is based on a three-layer model. In the uppermost layer, *reactive* plans are generated from goals expressed in a temporal logic. The middle layer is responsible for plan execution and assembling a configuration of domain-specific software components, which reside in the lowest layer. Moreover, the middle layer is responsible for selecting alternative components when the current configuration is no longer viable for the circumstances that have arisen. The implementation demonstrates that the approach enables us to handle non-determinism in the environment and unexpected failures in software components.

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. This requires that the autonomous system is able to cope with changes in the environment, changing goals, and failures in its software or hardware, all while deployed in the field. Contact with the operator or programmer may be infrequent at best.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEAMS'08, May 12–13, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-037-1/08/05 ...$5.00.

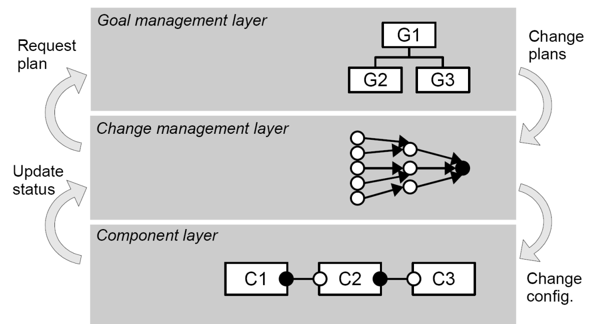

Figure 1: Three-layered conceptual model

A three-layer reference model, originally described by Gat [8], was proposed in [13] as a framework which could provide an integrated approach to the many challenges inherent in this area. In this model, the uppermost layer, which we call the goal management layer, comprises expensive deliberative planning; the middle layer, called the change management layer, is responsible for sequencing, that is, plan execution; and the lowest level, the component layer, handles reactive control concerns. Two feedback loops exist between the layers. The goal management layer pushes new plans down to the change management layer, while this layer may request a new plan. The change management layer generates new component configurations in the component layer, while the component layer may cause a new configuration to be generated by reporting a status change such as component failure. Figure 1 shows this arrangement graphically.

Our approach is aligned with this model, as reactive planning from abstract goals forms the top layer, a plan interpreter and configuration generator occupy the middle layer, and a configuration of software components resides in the lowest layer. This arrangement allows us to deal with several challenging aspects of autonomous systems. The planning layer allows us to develop complex behaviours from an abstract description of the goal; the use of reactive plans provides a mechanism for dealing with an uncertain environment; and the component control layer is able to adapt in response to either a change in the environment or a software fault. Only if these schemes fail to handle a particular situation, or the goal changes, do we resort to replanning.

In the goal management layer, so-called *reactive* plans are generated from high-level goals given in a temporal logic, and a description of the capabilities of the system. A reactive plan consists of a set of condition-action rules, which

specify the behaviour of the system. In addition to the sequence of actions intended to lead to the goal, a reactive plan also includes alternative paths which can be followed, should the environment change in an unpredictable way.

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 adaptation.

We exploit the presence of a plan in order to automatically assemble a configuration of components, as the actions in the plan implicitly give the functional requirements of the configuration. Having divorced the specification of the architecture from the programmer, we are able to permit arbitrary adaptations, provided the resulting components are sufficient to continue plan execution. While we focus on architectural change, 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.

In Section 2 we briefly mention some of the related work. Sections 3-5 describe our elaborations of the three layers. Section 6 describes our implementation, and several of the outstanding issues are discussed in Section 7.

2. RELATED WORK

The three-layer model is now widely used in the robotics community, where it developed from the early sense-plan-act (SPA) and subsumption [2] architectures. An SPA system consists of a single control loop where sensed data passes to an analysis or deliberation component, which then determines what actions to perform. The main limitation of this approach was that it could not react quickly enough for low-level behaviours. Subsumption was proposed to mitigate this by introducing several layers. The lowest layer was concerned with tight feedback loops such as obstacle avoidance, while the highest layers dealt with abstract goals. However, subsumption suffered from difficulties in specifying the interplay between the layers, and so it was not easy to modify a given program.

Previous work by the authors [20] introduced the idea of automatically deriving software component configurations from automatically generated plans. This paper builds on that work by presenting an algorithm for component selection, explicitly instantiating the reference architecture of [13] and describing a pair of case studies in which the approach is implemented in a system comprising several robots.

As we observed in [20], 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.

Garlan and Schmerl [7, 3] achieve dynamic change by describing an architectural style [5] 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. Zhang, Cheng *et al.* [22] derive the state machines of two programs and examine the guarantees of each to determine if a programmer-specified transition between them is safe. Dashofy *et al.* [4] use an architectural model and design critics [19] to determine whether a set of changes (an architectural 'delta') is safe to apply to a given configuration. Georgas and Taylor [9] describe a system where change is enacted by architectural policies which are invoked in response to certain events such as component failure. Georgiadis [10] describes a distributed approach to enforcing architectural constraints, though ultimately repairs are specified by the programmer.

The previous work that perhaps bears most resemblance to our combined approach is that by Garlan *et al.* [6] where adaptations are driven by a change of goal, and that by Arshad *et al.* [1] where adaptations are found by resorting to replanning. Unfortunately, planning for all reconfigurations comes at some cost, as it corresponds to an SPA architecture in an adaptive context.

3. GOAL MANAGEMENT LAYER

The top layer of the conceptual model concerns the generation of reactive plans from high-level goals. These plans specify the behaviour of the system in terms of actions which lead from an initial state to a goal state. The plan interpreter iterates through the rules of the plan to completion, unless a situation is detected which requires reconfiguration or replanning.

A reactive plan, unlike a sequential STRIPS-style plan [14], 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. 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 implementation, reactive plans are generated using planning-as-model-checking technology [11]. A description of the system's environment is specified in SMV [18], expressing state predicates and pre- and postcondition constraints on the actions that may be performed. Intuitively, these actions describe the capabilities of the autonomous system. This description is submitted to the Model-Based Planner tool (MBP) [17] along with a specification of the initial state and a goal, typically expressed in computational tree logic (CTL). The following gives an overview of plan generation in the goal management layer. Fuller details and examples are given in a previous paper [20].

3.1 Domain Model

In order to generate plans automatically, the environment of the system must be described formally. This description is known as the domain model. The domain model represents certain facts about the environment with logical propositions. A given state of the environment is then represented by the set of propositions that are true in that state. The propositions of the domain model are typically abstractions of properties sensed by the system, such as location for a robot. This abstraction is necessary to make planning tractable. The domain model also includes information about the capabilities of the system. These are encoded as actions that the system can perform and the effects these actions have on the environment. Formally, a domain

model is a 4-tuple

$$D = \langle Props, States, Actions, Trans \rangle$$

where $Props$ is a finite set of propositions, $States \subseteq 2^{Props}$ is a set of states, $Actions$ is a finite set of actions, and $Trans \subseteq States \times Actions \times States$ is a transition relation. Note that actions in the environment not performed by the system are not encoded in the domain model. However, the effects of such actions may be represented by particular states that the system may find itself in. Note also that domain model captures the fact that the effects of certain actions in the environment may be non-deterministic.

3.2 Plan Generation

Plans to achieve a particular goal in a given environment are automatically generated from a domain model and a goal expressed in a temporal logic (in our implementation CTL). The model-checking technology underpinning MBP identifies those states in the domain satisfying the specified goal and searches breadth-first for paths to those states from all other states in the domain model from which the goal states are reachable. If the search terminates having found a path from the current state of the system to a goal state, then the goal is achievable. As we mentioned above, the plans generated are reactive plans and thus contain paths to the goal from all states from which the goal is achievable. In a non-deterministic environment, if an execution of an action does not have its expected outcome, there is often still a path to the goal from the "unexpected" state in which the system finds itself.

We have noted that a plan is a set of condition-action rules. Formally, a plan P can be represented as a partial map from states of the domain to actions of the domain:

$$P : States \rightarrow Actions$$

As we will see in the next section, the component configuration of the system is determined by the actions required by a plan. The actions required by a given plan P are simply the range of P. For a state $s \in States$ and action $a \in Actions$, $P(s) = a$ only if $(s, a, s') \in Trans$, for some $s' \in States$:

$$Actions_P = \text{range}(P)$$

Finally, we note that in order to better manage potentially enormous state spaces, we organise domain descriptions into an abstraction hierarchy and generate plans in a corresponding hierarchy. This is discussed in [20] and will not be explored further in this paper.

4. CHANGE MANAGEMENT LAYER

The middle layer is concerned with using the plans generated by the layer above to construct component configurations and direct their operation to achieve the goal addressed by the plan. It is also concerned with making changes to the component configuration when this becomes necessary (on component failure). Crucially, we allow for arbitrary change which has not been specified by the user.

4.1 Component Model

For the purposes of deriving configurations we use a simplified component model based on Darwin [15]. In this model a component explicitly states its requirements (dependencies) and provisions. A component has a number of

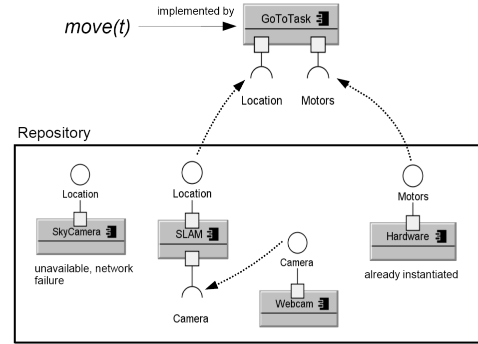

Figure 2: Components are selected according to the plan actions

(named) *ports*, and each port requires or provides a single named interface. We refer to the set of provisions of a component x as $prov(x)$, and the set of requirements as $req(x)$.

A configuration is constructed by instantiating components and connecting required ports to the provided ports of another component where the interface type matches. A complete configuration, then, contains no component with an unsatisfied requirement. The detail of how connections are implemented is not relevant for generating configurations, and is deferred until Section 5. We do not at present permit components to be defined as compositions of other components.

4.2 Deriving Component Configurations

Since reactive plans are composed of condition-action rules, the change management layer is able to use the plan to derive the functional requirements of the system's architecture. Once a configuration satisfying these requirements has been instantiated, plan execution can begin.

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. With each type of action, we associate a particular interface type, and thus the plan interpreter 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 interpreter 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 (or selected for instantiation) it should be reused. Hence, there can be only one instance of a given component type (or interface type) in a configuration. Reuse requires an explicit model of the current configuration to be maintained in this layer. Components can be marked as "unavailable", which may be a result of the component recently failing, or because a particular robot does not have the hardware necessary for that component. Clearly, components which are unavailable cannot be selected for use.

Figure 2 shows how a configuration would be generated to perform a move action. The interpreter is aware of an interface provided by the GoToTask that implements the move action. Then, to complete the configuration, it considers the requirements of the GoToTask, and their requirements in turn. In this case the Hardware controller provides an interface to the motors, and there are two alternatives for the Location requirement. The SkyCamera provides location information by connecting to the external infrastructure which can detect where the robot is located. However, SLAM (simultaneous localisation and mapping) is used in this case as the SkyCamera has previously experienced a failure. SLAM has a further requirement satisfied by the on-board camera.

4.2.1 Configuration Generator

The relationship between the actions required by a plan and the interfaces that implement those actions can be represented as a binary relation:

$$Implements \subseteq Actions \times Interfaces$$

At present we specify this relation manually, though it may be possible to derive the relation by matching the interface specification with that of the action. For a given plan p, the image of $Actions_P$ under $Implements$,

$$Implements\ [Actions_P] =$$
$$\{y \in Interfaces : \exists\ x \in Actions_P \cdot (x, y) \in Implements\}$$

is thus the set of interfaces that implement the actions in the plan. Figure 3 shows the core of the algorithm in pseudocode. An initial configuration can be generated with the call

$$\texttt{Construct}(\emptyset, Implements\ [Actions_P])$$

which will locate components which implement the interfaces in $Implements\ [Actions_P]$, and then complete the configuration by selecting further components to satisfy the requirements of those already selected. Subsequent configurations can be generated with a call such as

$$\texttt{Construct}(oldArch, Implements\ [Actions_P])$$

where the first parameter indicates the existing configuration.

```
Construct(arch, interfaces)
  ∀ i ∈ interfaces
    if ( ∃ c∈ arch : i ∈ prov(c) )
      interfaces := interfaces - {i}
    else
      providers = {x_j : x_j ∈ Components ∧
                         i ∈ prov(x_j) ∧
                         available(x_j)}
      archs = {Construct(arch ∪ {x_j}, req(x_j))
               : x_j ∈ providers}
      if ( ∃ a ∈ archs : a ≠ null )
        interfaces := interfaces - {i}
        arch := a
      else
        return null
  return arch
```

Figure 3: Configuration generator

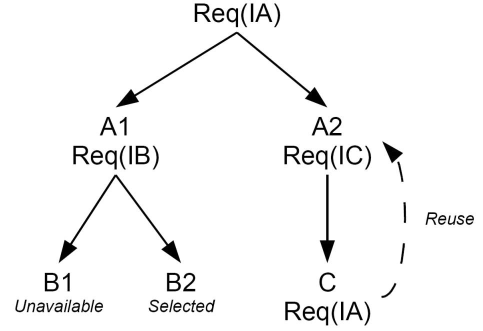

Figure 4: Component selection search tree for interface IA

The algorithm is expressed as a function which takes an existing configuration (expressed as a set of component type names, called `arch`), and a set of names of a desired interfaces. The `arch` parameter may be the empty set, to generate the initial configuration. In the first branch, the algorithm checks whether there is a component in the current configuration which already provides the desired interface. If so, the interface can be removed from the set of desired interfaces. In the second branch of the algorithm, the repository of known components (here expressed as a set called *Components*) is searched for components which provide the desired interface, and are available. If there are no providers of the desired interface, the special value `null` is returned, indicating that no configuration can be found. The function is called recursively to find the requirements of each of these components, returning a set of configuration choices in `archs`. Some of the providers may have unsatisfiable requirements, indicated by `null`. However, if there is a non-null configuration, then this is selected.

If the algorithm can find a configuration, then the configuration has the capability to execute the plan to completion, notwithstanding unexpected failures. The exception to this 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. The generated configuration, $arch$, satisfies the following constraints:

$$\forall i \in Implements\ [Actions_P] : \exists c \in arch : i \in prov(c)$$

$$\forall c \in arch : i \in req(c) \longrightarrow \exists c_2 \in arch : i \in prov(c_2)$$

The first constraint states that for all initial requirements, there should be an implementing component, and the second constraint is simply that all component requirements are satisfied.

The selection process forms a depth-first search tree for a complete configuration where all requirements are satisfied, as in Figure 4. Here, an implementation of interface IA is required. There are two alternatives, $A1$ and $A2$, both of which have further requirements. $A1$ requires interface IB, for which there are two implementations. $B1$ is unavailable (for whatever reason), so $B2$ is selected, giving a complete configuration $\{A1, B2\}$. $A2$, however, requires IC, which is provided by C. C in turn requires IA. In order to avoid a potential infinite loop where more and more instances of

$A2$ and C are added, the requirement for IA is satisfied by $A2$, which has already been selected. Consequently, the procedure will only produce configurations which have a single implementation of a given interface type. The complete configuration in this second branch is $\{A2, C\}$.

The example shows how it is possible to find multiple solutions for a given search. It may be the case that these alternatives have differing non-functional properties which can be used to make a choice. For example, $A1$ may require a large amount of CPU attention, but give accurate results, while $A2$ may run more quickly at the cost of reliability. To differentiate them, the system would have to rank CPU cost against accuracy. This will be addressed in our ongoing work, however, at present, the selection algorithm makes an arbitrary choice between the alternatives.

4.3 Adaptation

Adaptation at the component level is enacted by responding to failure events from components. It is often the case that components are best placed to detect their own failures, and monitor their dependencies. This concept closely mimics the use of exceptions in programming languages, and forms the first feedback loop between the lower two layers in Figure 1. We do not currently consider adaptation in response to widespread problems (such as performance issues) which may not be observable by a single component.

Systems such as [7, 3], which are able to observe global properties, break encapsulation to a certain extent as something must be known about the component implementation. This reduces the opportunities for reuse, particularly with commercially available components.

Failure events are handled by the change management layer which uses the event to determine which components must be replaced in order to continue the plan. For example, if an action component fails due to some internal (non-architectural) problem, it must be replaced. However, if it fails due to a problem in its dependencies, then its dependencies must be replaced. The component types that have failed are marked as unavailable, and the component selection process runs to locate alternatives. The delta computed, for some current configuration, *current*, and a failed component, *failed*, is:

$$delta = current - \texttt{Construct}(current, prov(failed))$$

If no alternatives can be found, then the second part of the feedback loop in Figure 1 from the change management layer to the goal management layer must be used in order to do dynamic update of the domain and replanning. We will address this problem in future work.

5. COMPONENT LAYER

The domain-specific components reside in the bottom layer of the conceptual model. Components are implemented as objects in Java, following the component model prescribed by the Backbone language [16] and its associated graphical design tool, jUMbLe. This model is compatible with UML2.0 which in turn resembles Darwin [15] (and is a superset of the simplified model described above).

Backbone allows us to design a configuration graphically using jUMbLe. It then performs component instantiation and connection. Provisions are indicated using the normal `implements` syntax, while required ports are implemented as fields. Hence, to create a connection, a reference to the provider is assigned to the appropriate requirement field. Interactions over this connection then consist of method invocations on the provider.

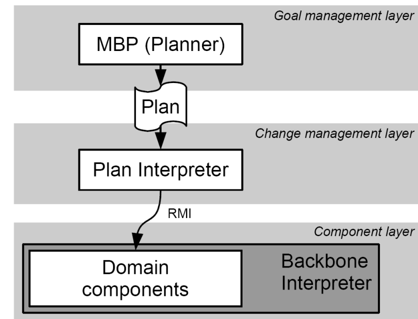

Figure 5: Implementation entities

The Backbone interpreter runs on a JVM on a number of Koala[1] robots and a Katana robot arm which we use for demontrating the work. The plan interpreter uses RMI to instruct Backbone to instantiate the desired configurations, and then makes the appropriate calls to execute actions in the plan.

6. CASE STUDIES

In order to demonstrate the generality of our approach, we have developed and implemented a number of scenarios, two of which are described below. Figure 5 shows all parts of the implementation and how these entities relate to the conceptual model.

In the top layer, the MBP planner generates reactive plans from goals and a domain description. These plans are passed to the plan interpreter in the middle layer. The plan interpreter then generates a new configuration which contains an implementation of one or more interfaces. Backbone maintains a model of the current configuration, which is referred to in order to reuse component instances. When the new configuration is ready, the plan interpreter can execute the plan by calling the methods provided by the domain-specific components which have been instantiated.

6.1 Reconnaissance Scenario

The first scenario requires a robot to move to a location and take a photograph of a target. This scenario demonstrates adaptation in response to a (simulated) failure in the components which control the primary robot's motion, and includes a second robot which is able to guide the first one towards the target.

6.1.1 Plan

The goal specified for this simple scenario is for a target t to be photographed. If we represent the property of having been photographed with the predicate *photographed*, we can express the goal as $photographed(t)$. Submitting this goal along with the domain model and current state of the robot to MBP, we generate a reactive plan prescribing actions towards this goal from each state in the domain from which $photographed(t)$ can be made true, of which one must be the current state. In other words, for all states in which the CTL formula $\mathbf{EF}\ photographed(t)$ is true (read $\mathbf{EF}\ \phi$ as

[1] http://www.k-team.com

```
(case (photographed_t))
   (done))
...
(case (and
      (not (photographed_t))
      (koala1_at_loc1)
      (t_at_loc3))
   (action koala1_move_loc3))
...
```

Figure 6: Fragment of reactive plan output by MBP

"some path eventually satisfies ϕ"), the plan contains a path from that state to the goal.

There is not space to illustrate the entire plan but a small sample of the condition-action rules generated is given in Figure 6. This shows, for example, that if t has not been photographed and the robot "koala1" is at "loc1" and the target is at "loc3" then koala1 should move to loc3. If t has been photographed then the goal has been achieved and plan execution can terminate.

If we let P denote the generated plan, then we can see that $Actions_P$ includes the actions *koala1_take_photo* and *koala1_move* ($Actions_P$ will of course also include the other actions not shown in Figure 6). Thus, the component configuration automatically constructed will include components that implement these actions, as described in Section 4.

When plan execution stops, due to a failure in the components controlling motion, an alternative path to the goal is found that requires koala1 to follow a second robot "koala2" (using short-range infrared sensors) to the target's location. The actions required by the plan along this path to the goal include $koala1_init_follow_koala2$ and $koala2_move$. The failed components in koala1 are swapped out and replaced by the components that implement the short-range following mechanism. If needed, the software on koala2 is also reconfigured.

6.1.2 Components

Figure 7 shows the component configuration instantiated on the robot in order to perform a (normal) move action. The GoToTask provides the implementation of this action, while the Koala component provides the interfaces to the motors and sensors of the hardware. The ObstacleAvoider component uses the proximity sensors to detect obstacles.

The VectorMotionController combines the outputs of the GoToTask and ObstacleAvoider to determine the direction in which the robot should move. In this case, the ObstacleAvoider serves to push the robot away from obstacles, and the GoToTask pushes the robot towards the target location. The GoToTask needs to know where the robot is located relative to the target. To provide this information we use a number of external webcams which observe the 'arena' and locate robots by detecting the distinctively-coloured patches adorning them.

Figure 8 shows the component configuration for abnormal operation, where the VisualFollowingTask encapsulates following behaviour using an on-board camera provided by the Webcam component.

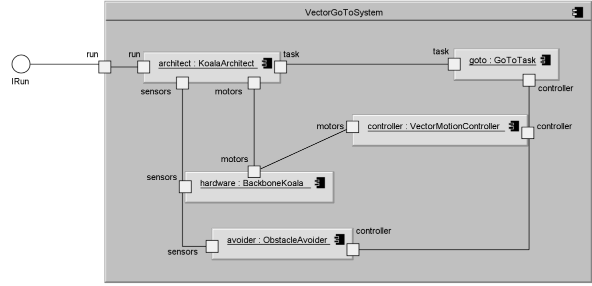

Figure 7: Configuration for performing a move action (in Backbone's graphical notation)

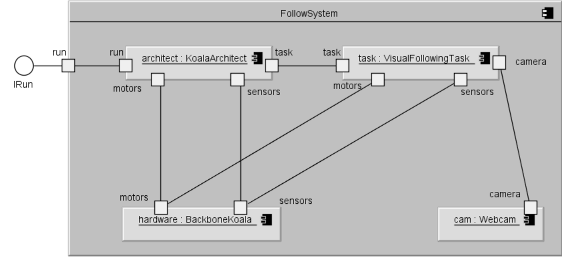

Figure 8: Abnormal configuration for moving by following

6.2 Cleanup Scenario

The second scenario we have considered requires a mobile robot to co-operate with a robot arm in order to collect a number of coloured balls. In this scenario, the goal is a continuous one, that is, balls must be picked up as long as there are balls remaining. Hence, the plan is cyclic.

6.2.1 Plan

This scenario illustrates a slightly more sophisticated form of goal. The balls in the environment can be loaded on or off the robot (one ball at a time) and whether or not the robot is carrying a ball is represented in the domain model with the predicate *loaded*. The aim is for the robot to continue indefinitely carrying balls back and forth between the area to be cleaned up and the arm that unloads it. In generating a plan, MBP searches for paths that lead to a cycle satisfying the CTL formula

$$\mathbf{AG}\ ((loaded \Rightarrow \mathbf{EF}\ \neg loaded) \wedge (\neg loaded \Rightarrow \mathbf{EF}\ loaded))$$

(read $\mathbf{AG}\ \phi$ as "all paths always satisfy ϕ"). In other words, whenever the robot is loaded it behaves in a way that will lead to being unloaded and whenever the robot is unloaded it behaves in a way that will lead to being loaded.

Plans generated for cyclic goals do not differ conceptually from those outlined above and we omit the details in this case to save space. Again, the actions required by the generated reactive plan drives the automatic configuration of components.

6.2.2 Components

The configuration used by the mobile robot in this case is identical to Figure 7, as the only action it is required to perform is the move action. The other actions in the plan refer to the robot arm which can perform a pickup

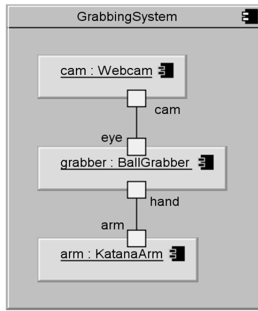

Figure 9: Configuration for picking up balls

action (shown in Figure 9). The components used to achieve this are the KatanaArm which provides the interface to the arm hardware, the Webcam component which deals with the camera hardware, providing an interface to read the current image and perform basic image processing functions, and a BallGrabber component which encapsulates searching for, gripping, and depositing the coloured ball in the box.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced work based upon Gat's three-layer model [8]. This reference architecture requires task planning from high-level goals and a domain description in the uppermost layer, task execution and derivation of architectural configurations in the middle layer, and component implementations in the lowest layer. Planning occurs before the system is started, and configurations are derived from the actions indicated in the plan.

Our approach also provides two mechanisms for dealing with unexpected events in the environment. Firstly, the use of reactive plans allows us to recover from reaching unexpected states after performing an action, so execution can continue without having to replan. Secondly, the middle layer is able to react to component failure — potentially due to a software bug or an environmental problem — by selecting alternative components where possible. This layer is capable of deriving heretofore unseen configurations which the user may not have pre-empted. Ultimately, if both of these mechanisms are insufficient for some problem, the system should be able to resort to replanning.

However, as has already been alluded to in previous sections, there remains significant research to be carried out in the various aspects of the approach. We have not yet achieved the complete feedback loop indicated in the three-layer model, as this requires dynamic replanning.

Dynamic replanning will occur in two categories of scenario. Firstly, when execution of the current plan fails, possibly due to a change in the environment or a system failure (as in the reconnaissance scenario above), a new plan must be requested from the goal management layer. In this case, it is likely that the currently held domain descriptions need to be updated with information about the failure so that a new plan can be generated which takes the environment change or system failure into account. In particular, if a component in the current configuration has failed then a new plan should of course be generated that does not require this component. Further, this updating of the domain descriptions must be carried out by the system automatically. The details of updating the domain descriptions in this way are still to be investigated.

Secondly, replanning will generally be required if the current goals are changed during execution. In this case, the domain descriptions will remain the same but the goal specification will need to be updated before generating a new plan.

There is also a need to prevent the system from entering cycles where it switches between two alternative plans or configurations, neither of which make any progress. This suggests a mechanism is required for recording what has previously been attempted under what circumstances. Indeed, this kind of logging would also remove the need to recompute plans or configurations when the system has already encountered the situation before.

Section 4.2.1 shows that there may be multiple configurations which satisfy the requirements of the plan to be executed. Since each configuration may have differing non-functional properties, it would be desirable for the selection process to take account of these.

For instance, consider a single requirement where there are two candidates. One candidate is highly reliable, while the other performs its computations in real time, at some lower reliability. If the component descriptions are annotated with these properties, the selection algorithm can produce two solutions: one which maximises reliability, and one which minimises delay. If the user specifies a preference for performing the current plan by ranking these attributes, the selection algorithm can select the best alternative.

The user may also wish to introduce extra structural constraints. For example, two components may not be compatible in a way not explicit to the selection process (they might contend for a piece of hardware, for instance). Alternatively the user may require that certain components are always selected, such as the ObstacleAvoider, and may require that the configuration generated conform to an architectural style. We are at present experimenting with ways of expressing and solving such constraints. One approach has been to use the Alloy analyser[2] to find a configuration which satisfies constraints expressed in the Alloy language. This is akin to the work in [21] where candidate configurations are checked (and possibly vetoed) by Alloy, except that we also require Alloy to propose the configuration. Unfortunately, the performance of this method is often insufficient for use in a running system, so we are exploring other possibilities.

To ensure a smooth transition from one configuration to another, special attention must be paid to the way in which the adaptation is performed, to ensure it is safe. Clearly if some components are to be replaced, then their dependants must not initiate communications with them for the duration of the change. There is much previous work, notably that on quiescence [12], which is applicable here. 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.

[2] `http://alloy.mit.edu`

8. 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.

9. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. A planning based approach to failure recovery in distributed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 8–12, New York, NY, USA, 2004. ACM Press.

[2] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.

[3] S. W. Cheng, D. Garlan, B. R. Schmerl, J. P. Sousa, B. Spitnagel, and P. Steenkiste. Using architectural style as a basis for system self-repair. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 45–59, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[4] 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.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proc. of 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York, NY, USA, 1994.

[6] D. Garlan, V. Poladian, B. Schmerl, and J. P. Sousa. Task-based self-adaptation. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 54–57, New York, NY, USA, 2004. ACM Press.

[7] D. Garlan and B. Schmerl. Model-Based Adaptation for Self-Healing Systems. In *Proc. of 1st Workshop on Self-healing Systems*, New York, NY, USA, 2002.

[8] E. Gat. Three-Layer Architectures. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, pages 195–210, 1998.

[9] J. C. Georgas and R. N. Taylor. Towards a knowledge-based approach to architectural adaptation management. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 59–63, New York, NY, USA, 2004. ACM Press.

[10] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

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

[12] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11), 1990.

[13] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. *International Conference on Software Engineering*, pages 259–268, 2007.

[14] M. Ghallib, D. Nau, P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman, 2005.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[16] A. McVeigh, J. Kramer, and J. Magee. Using Resemblance to Support Component Reuse and Evolution. In *Proc. of SIGSOFT/FSE Workshop on Specification and Verification of Component-based Systems*, New York, NY, USA, 2006. ACM Press.

[17] 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.

[18] 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.

[19] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Using Critics to Analyze Evolving Architectures. In *Joint Proc. of ISAW-2 and Viewpoints '96 on SIGSOFT '96 Workshops*, New York, NY, USA, 1996.

[20] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-directed architectural change for autonomous systems. In *Proc. of SIGSOFT/FSE Workshop on Specification and Verification of Component-based Systems*. ACM Press New York, NY, USA, 2007.

[21] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering*, Washington, DC, USA, 2006.

[22] J. Zhang and B. Cheng. Modular Model Checking of Dynamically Adaptive Programs. Technical report, Michigan State University, 2006.