MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation*

Victor Braberman[1], Nicolas D'Ippolito[1], Jeff Kramer[2], Daniel Sykes[2], Sebastian Uchitel[1,2]
[1] Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
[2] Department of Computing, Imperial College London, UK

ABSTRACT

An architectural approach to self-adaptive systems involves runtime change of system configuration (i.e., the system's components, their bindings and operational parameters) and behaviour update (i.e., component orchestration). Thus, dynamic reconfiguration and discrete event control theory are at the heart of architectural adaptation. Although controlling configuration and behaviour at runtime has been discussed and applied to architectural adaptation, architectures for self-adaptive systems often compound these two aspects reducing the potential for adaptability. In this paper we propose a reference architecture that allows for coordinated yet transparent and independent adaptation of system configuration and behaviour.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design

Keywords

Self-adaptive Systems, Software Architecture

1. INTRODUCTION

Self-adaptive systems are capable of altering at runtime their behaviour in response to changes in their environment, capabilities and goals. Research and practice in the field has addressed challenges of designing these systems from multiple perspectives and levels of abstraction.

It is widely recognised that an architectural approach to achieve self-adaptability promises a general coarse grained framework that can be applied across many application domains, providing an abstract mechanism in which to define runtime adaptation that can scale to large and complex systems [21].

Architecture-based adaptation involves runtime change of system configuration (e.g., the system's components, their bindings, and operational parameters) and behaviour update (e.g., component orchestration).

*This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2012-0724, UBACYT W0813, ANPCYT PICT 2011-1774, UBACYT F075, CONICET PIP 11220110100596CO, MEALS %295261.

Existing approaches to architectural adaptation (e.g. [16, 10] incorporate elements from two key areas to enable runtime adaptation: Dynamic reconfiguration [9, 21] and discrete-event control theory [12, 22, 8]. The first, key for adapting the system configuration, studies how to change component structure and operational parameters ensuring that on-going operation is not disrupted and/or non-functional aspects of the system are improved. The second, key for adapting behaviour, studies how to direct the behaviour of a system in order to ensure high-level (i.e., business, mission) goals.

Although the notions of configuration and behaviour control are discussed and applied by many authors, they are typically compounded when architectures for adaptation are presented, reducing overall architectural adaptability. Automated change of configuration and behaviour address different kinds of adaptation scenarios each of which should be managed as independently as possible from the other. Nonetheless, configuration and behaviour are related and it is not always possible to change one without changing the other. The need for both capabilities of independent yet coordinated adaptation of behaviour and configuration requires an extensible architectural framework that makes explicit how different kinds of adaptation occur.

Consider a UAV on a mission to search for and analyse samples. A failure of its GPS component may trigger a reconfiguration aiming at providing a location triangulating over alternative sensor data. The strategy may involve passivating the navigation system, unloading the GPS component and loading components for other sensors in addition to the component that resolves the triangulation. A behaviour strategy that is keeping track of the mission status (e.g. tracking areas remaining to be traversed, samples collected, etc.) should be oblivious to this change.

A reconfiguration adaptation strategy that can cope with the GPS failure can be computed automatically using approaches based on, for example, SMT solvers or planners [22] that consider the structural constraints provided in the system specification (e.g., the need for a location service), requirements and capabilities of component types (e.g., the requirements of a triangulation service) and runtime information of available component instances (e.g., the availability of other sensors).

The arrival of the UAV at an unexpected location due to, say, unanticipated weather conditions may make the current search and collection strategy inadequate. For instance, the new location may be further away from the base than expected and the remaining battery charge may be insufficient to allow visiting the remaining unsearched locations before returning

to base. In this situation the behaviour strategy would have to be revised to relinquish the goal of searching the complete area before returning to base in favour of the safety requirement that battery levels never go below a given threshold. The new behaviour strategy may reprioritise remaining areas to be searched (in terms of importance and convenience), visiting only a subset of the remaining locations as it moves towards the base station for recharging. Once recharged, the strategy may attempt to revisit the entire area under surveillance but prioritising locations previously discarded. Such behavioural adaptation should be independent to the infrastructure supporting reconfiguration control.

A behaviour strategy that can deal with unexpected deviations in the UAV's navigation plan can be computed automatically using approaches based on, for instance, controller synthesis [8] that consider a behaviour model describing the capabilities of the UAV (e.g. autonomy), environment (e.g., map with locations of interest and obstacles) and system goals (e.g. UAV safety requirements and search and analyse – liveness – requirements). Indeed, our proposal of an explicit separation of reconfiguration and behaviour strategy computation and enactment is in line with the design principles of a separation of concerns and information hiding. The behaviour strategy is oblivious to the implementation that provides the services it calls and the reconfiguration strategy supports the injection of the dependencies that are required by the behaviour strategy oblivious to the particular ordering of calls that the behaviour strategy will make. In a sense, the design principle which is known to support changeability supports runtime changeability, which ultimately is what adaptation is about.

Configuration and behaviour adaptation may however need to be executed in concert. *Consider the scenario in which the gripper of the UAV's arm that is to be used to pick up samples becomes unresponsive. With a broken gripper the original search and analyse mission is unachievable. This should trigger an adaptation to a degraded goal that aims to analyse samples via on-board sensors and remote processing. This goal requires a different behaviour strategy (e.g. circling samples once found to perform a 360 degree analysis) but also a different set of services provided by different components (e.g. infra-red camera). Not only are both behaviour and configuration adaptation required, but also their enactment requires a non-trivial degree of provisioning: To set up the infra-red camera, the UAV requires folding the arm to avoid obstructing the camera's view; performing such an operation while in the air is risky. Hence, coordination between configuration and behaviour adaptation is needed: First, a safe landing location must be found, then arm folding must be completed, and only then can the reconfiguration start. New components are loaded and activated, and finally, a strategy for in-situ analysis, rather than analysis at the base, can start.*

It is in the combined configuration and behaviour adaptation where the need for both separation of concerns and explicit architectural representation of coordination becomes most evident. Approaches to automated computation of configuration and behaviour adaptation strategies require different input information and utilise different reasoning techniques. Both automated reasoning forms are of significant computational complexity and require careful abstraction of information. Keeping resolution of configuration and behaviour adaptation separately allows reusing existing and future developments in the fields of dynamic reconfiguration and control theory and also helps keep computational complexity down.

The broken UAV gripper scenario requires a coordinated behaviour and reconfiguration adaptation strategy. The adaptation required can be decomposed into a behaviour control problem that assumes that a reconfiguration service is available and a reconfiguration problem. The resulting behaviour strategy will be computed on the assumption that the UAV's capabilities will conform to the current configuration (e.g. grip command fails) until a reconfigure command is executed, and that from then on different capabilities will be available (e.g. infra-red camera getPicture command available). The behaviour strategy computation will also consider restrictions on when the reconfigure command is allowed (e.g. when arm is folded) and new goals (360 degree picture analysis rather than collect). The computation of the reconfiguration strategy does not entail additional complexity and is oblivious to the fact that a behaviour strategy that involves a reconfiguration halfway through is being computed.

In the above scenario, what needs to be resolved at the architectural level of the self-adaptation infrastructure is which architectural element is responsible for the decomposition of the adaptation strategy into a behaviour strategy and a reconfiguration strategy, and also how strategy enactment is performed to allow the behaviour strategy to command reconfiguration at an appropriate time (and possibly even account for reconfiguration failure). Indeed, an appropriate architectural solution to this would enable guaranteeing that given a correct decomposition of the overall composite adaptation problem into configuration and behaviour adaptation problems, and given correct-by-construction configuration and behavioural strategies for these problems, the overall adaptation problem is correct.

In this paper we present MORPH, a reference architecture for behaviour and configuration self-adaptation. MORPH makes the distinction between dynamic reconfiguration and behaviour adaptation explicit by putting them as first class entities. Thus, MORPH allows both independent reconfiguration and behaviour adaptation building on the extensive work developed but also allowing coordinated configuration and behavioural adaptation to accommodate for complex self-adaptation scenarios.

2. MORPH

The MORPH reference architecture builds upon the large body of work related to engineering self-adaptive emphasising the need to make behaviour and reconfiguration control first-class architectural entities. In particular, it draws inspiration from [16, 10], which are discussed below.

2.1 Background

The MAPE-K model shows how to structure a control loop in adaptive systems. The four key activities (Monitor, Analysis, Plan and Execute) are performed over a shared data structure that captures the knowledge required for adaptation. The MAPE-K model does not prescribe what knowledge is to be captured nor what aspect of the system is to be controlled. Thus, there is no explicit treatment or distinction between configuration and behaviour adaptation let alone prescribed mechanisms for dealing with coordinated and independent configuration and behaviour adaptation.

The need to deal with hierarchies of control loops in au-

tonomous systems is widely recognised (e.g. [11]). Lower levels are typically low latency loops that focus on more tactic and stateless objectives that involve less monitored and controlled elements while higher levels tend to focus on more stateful and strategic objectives involving multiple controlled and monitored aspects that require higher latency loops. The need for hierarchy in architectural self-adaptation is discussed in [16]. A three-tier architecture is proposed to provide a separation of concerns and to address a key architectural concern related to dealing with the complexity of run-time construction of adaptation strategies. The architecture structures hierarchically the MAPE-K loops introducing a separation of concerns in which complex, strategic, resource consuming analysis is performed in top layers while simple, more tactical adaptation is performed in lower layers.

Unlike the MAPE-K model, the architecture in [16] prescribes the kind of control that is effected on the adaptive system by establishing a clear interface between the adaptation infrastructure and the component based system to be adapted. The architecture assumes an interface on which it can take action on the current system configuration by creating and deleting components, binding and unbinding components through their required and provided ports and setting component modes (i.e., configuration parameters).

Although the three-tier architecture provides a clear separation of the concerns of behavioural planning from component reconfiguration, this is purely hierarchical, with the behaviour plan dictating the required structural (re) configuration at lower layers. Although this allows for independent structural configuration alone if the behaviour plan is still satisfied, it is less clear how behaviour control and configuration control should work together, hindering the possibility of more elaborate behaviour and configuration control and the potential for reasoning about adaptation guarantees.

The Rainbow [10] framework instantiates and refines the MAPE-K architecture providing an extensible framework for sensors and actuators at the interface between the control infrastructure and the target system. The architecture recognises the complexity of the interface between the MAPE-K infrastructure (referred to by the authors as the architecture layer) and the component system to be adapted (referred to as the system layer). The Rainbow framework introduces additional infrastructure into the architecture and system layers in addition to accounting for an extra layer between the two: the translation layer. Monitoring is split amongst the three layers: probes are introduced as system layer infrastructure to support observation and measurement of low-level system states. Gauges are part of the architectural infrastructure layer and aggregate information from the probes to update appropriate properties of the knowledge base used for the MAPE activities. The translation layer resolves the abstraction gap between the system layer and the architectural layer, for instance relating abstract component identifiers in the later concrete process identifiers and machine identifiers in the former.

Rainbow focuses on achieving self-adaptation through configuration adaptation. Thus, as in [16] focus is on changing component instances and bindings and also effecting behaviour by changing operational parameters (thread pool size, number of servers, etc.). Indeed, the framework does not account explicitly for automated construction of behaviour strategies that control the functional behaviour of the system layer components. As in [16], the distinction between configuration and behaviour control is not elaborated explicitly in the architecture.

In the following, we propose an architecture that takes inspiration from the architectures discussed above but that includes the design concern related to supporting independent yet coordinated behaviour and configuration adaptation. The architecture combines the three tier structure from [16] to address varying latency of architectural self-adaptation control. We design each layer as a MAPE-K control loop, resulting in a hierarchical control loop structure as in [17]. As in Rainbow [10] we address the problem of bridging the gap between the managed component architecture and the adaptation infrastructure encapsulating the former and also providing a decoupled mechanism for aggregation and inference over logged system data.

2.2 Architectural Overview

We start with a very brief introduction of the main architectural elements to give a general picture of how the architecture works before we go into detail of the workings and rationale of each element. A graphical representation of the architecture can be found in Figure 1. In the remaining text, when we want to emphasise traceability to the figure we will use an alternative font.

The architecture is structured in three main layers that sit above the target system: Goal Management, Strategy Management and Strategy Enactment. Orthogonal to the three layers is the Common Knowledge Repository. Each layer can be thought of as a implementing a MAPE-K loop. The top layer's MAPE-K loop is responsible for reacting to changes in the goal model that require complex computation of strategic, possibly configuration and behavioural, adaptation. Its knowledge base is the Common Knowledge Repository. The Strategy Management layer's MAPE-K loop is responsible for adapting to changes that can be addressed using pre-processed strategies. It selects pre-computed strategies based on the Common Repository Knowledge and a set of internally managed pre-computed strategies. The Strategy Enactment layer's MAPE-K loop is responsible for executing strategies; its knowledge base is primarily the strategy under enactment.

The Target System abstracts the Component Architecture that provides system functionality. The Component Architecture is harnessed by effectors and probes which allow the Strategy Enactment Layer to interface with system components. The Knowledge Repository stores in a Log the execution data produced by Target System and also stores in the Goal Model) the result of Inference procedures that produce knowledge regarding the system state, goals and environment assumptions. We expect users, administrators and other stakeholders to also produce modifications to the Knowledge Repository, and in particular the goals and environment assumptions.

The three layers that provide the architectural adaptation infrastructure are each split into reconfiguration and a behaviour aspects. The Goal Management layer has a Goal Model Manager whose main responsibility is to decompose adaptation problems into reconfiguration and behaviour problems, each of which is given to a specific Solver to produce a strategy that can achieve the required adaptation. The top layer sends reconfiguration and behaviour strategies downwards. The bottom two layers have architectural elements to handle reconfiguration and behaviour strategies separately but interact with each other when and if needed to maintain

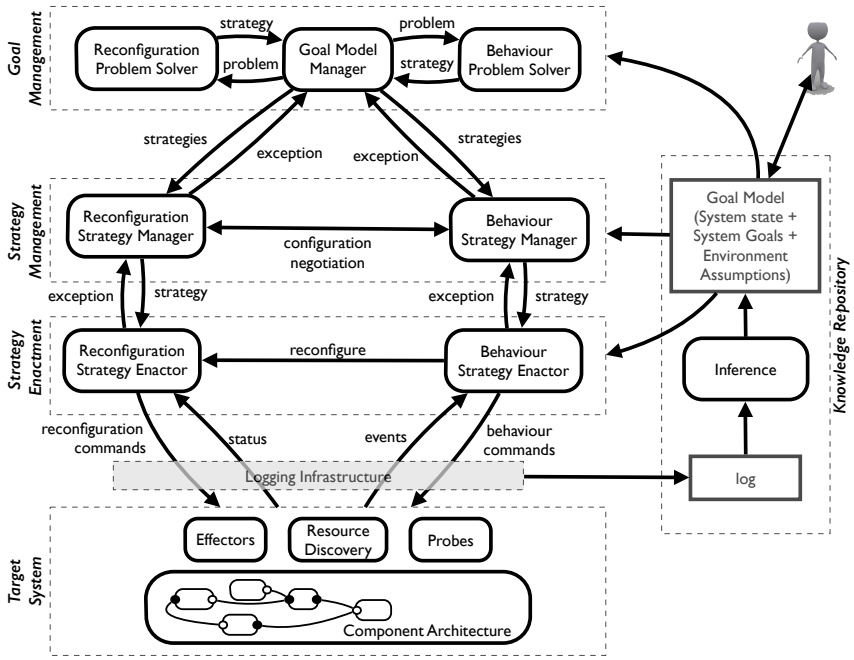

Figure 1: The MORPH Reference Architecture.

overall consistency. The Strategy Management layer entities interact to ensure that they select consistent strategies to be executed by the Strategy Enactment layer (c.f., configuration negotiation). The Strategy Enactment layer entities interact to ensure that the execution of their respective strategies is done consistently over time (c.f., reconfigure command).

2.3 Target System

Responsibility: The main responsibility is to achieve the system goals, encapsulate implementation details and provide abstract monitoring and control mechanisms over which behaviour and structure of the system can be adapted.

Rationale: This is to encapsulate the instrumentation of the system-to-be-adapted to support a flexible and reusable framework for monitoring, analysing, planning and executing adaptation strategies in the layers above.

Structure and Behaviour: The Target System, strongly inspired by [10], contains the component architecture that provides the managed system's functionality *(*e.g., GPS, video, telemetry and navigation components). It also contains instrumentation to monitoring and control of the component architecture. Two types of effectors are provided. The first provides an API to add, remove and bind components, in addition to setting operational parameters of these components. We refer to the invocation of operations on these effectors as reconfiguration commands. These effectors are application domain independent, and they provide the adaptation infrastructure an abstraction over the concrete deployment infrastructure on which the component architecture runs *(e.g., the UAVs operating system)*. The second effector type, behaviour actions, is domain dependent and provides an API that invokes functional services provided by components of the component architecture. *The UAV's navigation component may exhibit a complex API which is abstracted into simple commands (e.g. goto(Location)) that are to be used as the basis for behaviour strategies.*

The mechanism for monitoring of the component architecture can be provided by probes that reveal state information. As with effectors, monitoring information can be classified into two kinds. We have on one hand information regarding the status of components. This kind of information is application independent. Status of a component may indicate if it is active, inactive, connected or killed. On the other hand we refer to as events the application domain relevant information that flows from the Target System to the Strategy Enactment Layer. *UAV events may include notifications regarding battery depletion, or acknowledgements of having reached a requested location.*

Between the target system and the adaptation infrastructure a translation layer is required to provide translation services that aim to bridge the abstraction gap between the knowledge representation required to perform adaptation at the architectural level and the concrete information of the actual implementation. *In the UAV, this may include resolving event handlers, process ids, in addition to domain specific translations such the conversion of continuous variable for battery level to a discrete battery depleted event.*

2.4 Common Knowledge Repository

Responsibility: The key responsibility of the repository is to keep an up to date goal model at runtime based on inferences made over continuous monitoring of the environment to detect changes in goals, behaviour assumptions and available infrastructure.

Rationale: The design rationale for the repository is to decouple the accumulation of runtime information of the target system from the complex computational processes involved in abstracting and inferring high-level knowledge that can be incorporated, for subsequent adaptation, into a structured body of knowledge regarding stakeholder goals, environmental assumptions and target system capabilities.

Structure and Behaviour: The common knowledge repository stores information about the target system, the goals and environment assumptions. It consists of two data structures (a log and a goal model) and an Inference procedure.

The Goal Model: This is the key data architectural element of the repository. We use the term "goal model" in the sense of goal oriented requirements engineering [18].

The point of keeping a structured view of the world that includes requirements and assumptions, with multiple ways of achieving high-level goals and preference criteria over these alternatives is that at runtime it is possible to change the way a goal is achieved by selecting a different OR-refinement. The combinatorial explosion of possible OR-refinement resolutions can be a rich source for adaptation which is exploited in the Goal Management Layer. In addition, this representation of rationale, is amenable to being updated and changed as new information is acquired.

2.5 Goal Management Layer

Responsibility: The main responsibility of the Goal Management Layer is to deal with and anticipate changes in the stakeholder goals, environment assumptions and system capabilities by pre-computing adaptation strategies consisting of separate behaviour and reconfiguration strategies.

Rationale: The rationale for this layer is based on two core concepts: The first is that the adaptive system must be capable of performing strategic, computationally expensive, planning independently of and concurrently with the execution of pre-computed strategies (occurring in lower layers). The second is to decompose adaptation into a behaviour strategy that controls the system to an interface and a reconfiguration strategy that injects the dependencies on concrete implementations that the behaviour strategy will use. Decomposing adaptation along the modular design improves support for adaptability allowing behaviour and configuration changes independently.

Structure and Behaviour: The layer has three main entities, the Goal Model Manager, the Behaviour Problem Solver and the Reconfiguration Problem Solver.

The Goal Model Manager: This is the key element of the layer responsible for three core tasks: The first is to decide when a new adaptation strategy must be computed, the second is to resolve all OR-refinements in the goal model and select the requirements for to be achieved by the system, and third to decompose the requirements into achievable reconfiguration and behaviour problems. Concrete strategies for reconfiguration and behaviour are computed by the solvers.

Production of adaptation strategies can be triggered by requests for plans from layers below or internally due to the identification of significant changes in the goal model. The former may correspond to a scenario in which a failure is propagated rapidly upwards from the target system: *For instance, the UAV's gripper component fails. The Strategy Enactment layer, which is executing a strategy that requires the gripper, immediately informs that its current strategy is unviable and requests a new strategy to the Strategy Management Layer.* If all pre-computed plans require the arm to pick up objects, a new strategy for achieving system goals is requested to the Goal Model Manager.

The alternative, internal, triggering mechanism corresponds to scenarios in which the goal model is changed because of new information inferred from the log or input manually by some stakeholder. *For instance, weather conditions may lead to inferring higher energy consumption rates from logged information. What would follow is a revision of the assumptions on UAV autonomy stored in the Goal Model. Such alteration may trigger the re-computation and downstream propagation of search strategies to make more frequent recharging stops.*

Adaptation strategies are decomposed into a strategy for achieving the component configuration that can provide the functional services required achieve selected requirements and a behaviour strategy that can call these services in an appropriate temporal order to satisfy the requirements. *The adaptation strategy that deals with the broken gripper must reconfigure the system to use a different set of components (e.g. the infra-red camera) and coordinate its use upon reaching a position where there is a sample to be inspected.*

As discussed in the Introduction, decomposition allows adaptation of the system configuration transparently to the behaviour strategy being executed (*e.g., changing the location mechanism*) or the behaviour strategy transparently to the configuration in use (*e.g., changing the route planning strategy*). In addition, decomposition allows for the computation of multiple behaviour strategies for a given configuration (*e.g. different search and collect strategies that assume different UAV autonomy can be run on a configuration that has a gripper component*) and different configurations can be used for a given behaviour strategy (*e.g. different configurations for providing a positioning service can be used for the same search strategy*).

One of the design rationales for this layer is the precomputation of expensive adaptation strategies that are then ready to use when needed. This means that multiple reconfiguration and behaviour strategies may be constructed. Indeed, the Goal Model Manager can pre-compute, and propagate downwards, many reconfiguration and behaviour strategies for one resolution of the OR-refinements of the goal model. *This may be useful, for example, if it is known that information regarding UAV autonomy is imprecise, multiple (behaviour) search strategies for searching the area may be developed so that the infrastructure can adapt quickly as soon as the predicted UAV autonomy differs significantly from what can be inferred from the monitored energy consumption. Similarly, should the GPS-based location service be known to fail (perhaps do to environmental conditions), then various reconfiguration plans may be pre-computed to allow adaptation to alternative positioning systems when needed.*

Configuration Problem Solver: The layer has two entities capable of automatically constructing strategies for given adaptation problems. The Configuration Problem Solver focuses on how to control the target system to achieve a specified configuration given the current system configuration, configuration invariants that must be preserved and component availability. Configuration invariants may include structural restrictions forcing the architecture to conform to some architectural style or other considerations based, for instance, on non-functional. In the UAV example such restrictions may include that the attitude control components never be disabled or the total number of active components never be beyond a given threshold to avoid battery overconsumption.

Reconfiguration problem solvers build strategies that call actions that add and remove components, activate and passivate them, and establish or destroy bindings between them. These reconfiguration actions are part of the API exposed by the target system. The strategy may sequence these actions

or have an elaborate scheme that decides which actions to call depending on feedback obtained through the information on the status of components exhibited by the target system API.

To automatically construct strategies the solvers can build upon a large body of work developed in the Artificial Intelligence and Verification communities, including automatic planners (e.g. [3]), controller synthesis (e.g. [8]), and model checking. Such techniques have been applied to construction of reconfiguration strategies in [21].

Behaviour Problem Solver: This entity focuses on how to control the target system to satisfy a behaviour goal. In contrast to reconfiguration problems, the behaviour goal may not be restricted to safety and reachability (i.e. reach a specific global state while preserving some invariant). Behaviour goals may include complex liveness goals such as to have the UAV monitor indefinitely an area for samples to inspect. Behaviour problem solvers produce strategies, which can be encoded as automata that monitor target system events and invoke target system actions.

In addition to the expressiveness of goals that behaviour strategies must resolve, there is an asymmetry between reconfiguration and behaviour problems. To resolve the coordination problem between strategies (*as with folding the UAV arm before a reconfiguration to deal with a gripper failure can be executed, see Introduction*), the behaviour strategies produced by the solver can invoke a reconfigure command, which triggers the execution of a reconfiguration strategy. We explain how this triggering works in Section 2.6.

2.6 Strategy Management Layer

Responsibility: This layer selects and propagates precomputed behaviour and reconfiguration strategies to be enacted in the layer below. For this, the layer must store and manage pre-computed behaviour and reconfiguration plans, and request new strategies to the layer above when needed. It must also ensure that the behaviour and reconfiguration strategies sent to the lower layer are consistent, indicating their relationships.

Rationale: They main concept for the layer is to allow rapid adaptation to failed strategy executions (or capitalizing rapidly on opportunities offered by new environmental conditions) by having a restricted universe of pre-computed alternative behaviour and reconfiguration strategies that can be deployed independently or in a coordinated fashion.

Structure and Behaviour The layer has two entities that work in similar fashion mimicking much of the layer's responsibilities but only on either behaviour or reconfiguration strategies. However, the Behaviour Strategy Manager and the Reconfiguration Strategy Manager are not strictly peers. In some adaptation scenarios the former will take a Master role in a Master-Slave decision pattern.

Behaviour Strategy Manager: This entity stores and manages multiple behaviour strategies. From these strategies it picks a behaviour strategy to be enacted in the layer below. The selection of strategy may be triggered by an exception raised by the layer below or internally due to a change identified in the common knowledge repository. The former may occur when the behaviour strategy being executed finds itself in a unexpected situation it cannot handle. *For instance, the UAV executing a particular search strategy expects to be at a specific location with at least 50% of its battery remaining but finds that it is below that threshold, invalidating the rest of the strategy for covering the area to be searched.* At this point the Strategy Enactment Layer signals that the assumptions for its current strategy are invalid and requests a new strategy to this layer.

The other scenario that can trigger the selection of a new strategy is a change in the common knowledge repository. *Consider again the problem of unexpected energy overconsumption. An inference process in the knowledge repository may update the average energy consumption rate periodically based on Target System information being logged. This average may be well above the assumed consumption average for the behaviour strategy being executed. The Behaviour Strategy Manager may decide that it is plausible that the current behaviour strategy will fail and may decide to deploy a more conservative search strategy.*

Note that the two channels that may trigger the selection of a new strategy differ significantly in terms of latency and urgency. The exception mechanism provides a fast propagation of failures upwards, indicating that the strategy being currently enacted is relying on assumptions that have just been violated. This means that any guarantees on the success of the current strategy in satisfying its requirements are void and a new strategy is urgently required. The monitoring of changes in the knowledge repository is a process that incurs in comparatively significant delays as the inference of goal model updates based on logged information may be performed sporadically and consume a significant amount of time. The upside of this second channel is that it may predict problems sufficiently ahead of their occurrence, providing time to select pre-computed strategies to avoid them.

The selection of a behaviour strategy is constrained by the current configuration of the target system (which determines the events and actions that can be used by the strategy) and the alternative configurations that may be reached by enacting one of the pre-computed re-configuration strategies. Furthermore, the selection is informed by preferences defined in the goal model on which OR-refinement resolution is preferred. *Thus, a new strategy that can be supported by the current UAV configuration may be selected to alter the search strategy. Alternatively or a strategy that no longer picks samples up to avoid the extra consumption produced by load carrying may be chosen. In the later case, in-situ analysis is required and hence a reconfigured UAV with an infra-red camera in place is required. Selecting such a precomputed behaviour strategy is subject to the availability of a pre-computed reconfiguration strategy that can reach a configuration with an active infra-red camera module.*

The Behaviour Strategy Manager deploys the selected strategy by performing two operations. Firstly, should the selected strategy require a configuration with characteristics that are currently not provided, it commands the Reconfiguration Strategy Manager to deploy an appropriate reconfiguration strategy (c.f. Master-Slave relationship). Secondly, the manager hot-swaps the current behaviour strategy being executed in the layer below with the newly selected strategy, setting the initial state of the new strategy consistently with the current state of the Target System. Note that should the new strategy be replacing a strategy that is still valid (i.e. no exception has been raised) then the hot-swap procedure may also exploit information extracted by the current state of the strategy to be swapped out.

Should the Behaviour Strategy Manager fail to select a pre-computed behaviour strategy, the manager requests new

strategies from the layer above. This may happen, for example, because none of pre-computed strategies it manages have assumptions that are compatible with the actual observed behaviour of the system (e.g., *energy consumption is far worse than what is assumed by any pre-computed strategy*) or that they all rely on unachievable configurations (e.g. *the joint failure of the gripper component and infra-red camera was a operational scenario not considered in any of the pre-computed strategies*).

Reconfiguration Strategy Manager: This entity works similarly to its behaviour counterpart. It stores and manages multiple reconfiguration strategies and selects them for deployment constrained by the availability of instantiatable components in the Target System while maintaining consistency with the configuration requirements of the current behaviour strategy. Selection is also informed by preferences specified in the goal model. *Consequently, a precision preference may lead to selecting a reconfiguration strategy that attempts to use a GPS rather than hybrid positioning.*

When negotiating with the Behaviour Strategy Manager on a pair of strategies to be deployed, the Reconfiguration Strategy Manager takes the slave rol, informing the configurations requirements that are achievable and then selecting an appropriate reconfiguration strategy based on the selection made by the Behaviour Strategy Manger.

There are three channels that can trigger the selection of a new reconfiguration strategy. Two are similar to those that trigger the Behaviour Strategy Manager: An exception from the Reconfiguration Strategy Enactor and a change in the goal model. *Examples of these are the failure of the GPS component triggering a rapid response by the manager which selects an alternative configuration (using the hybrid positioning component) and deploys an appropriate reconfiguration strategy, or an increased response time of the GPS component leading to the decision of changing the positioning system before it (most likely) fails.* The third channel is the request of a new configuration by the Behaviour Strategy Manager (which in turn may have been triggered via de exception mechanism or a change in the goal model).

Note that deployment of new strategies at the Strategy Management layer may respond not only to problems (or forseen problems) while enacting the current strategies, but also deploy new strategies to capitalise on opportunities afforded by a change in the environment. For instance, should a new component become available, or statistics on its performance improve, (e.g. *the GPS component*) this would be reflected in the knowledge repository and an alternative preferred pre-computed strategy may be deployed.

2.7 Strategy Enactor

Responsibility: This layer's main responsibility is to execute behaviour and reconfiguration strategies provided by the layer above. Strategy execution involves monitoring the target system and invoking operations on it at appropriate times as defined by the strategy. The layer must also ensure that if the target system should reach a state unexpected by the strategy, and that consequently cannot be dealt with by the strategy, is reported to the layer above. The other key responsibility of the layer is to support both independent and transparent update of behaviour and reconfiguration strategies in addition to supporting a master-slave relation between behaviour and reconfiguration strategy execution in which the former can initiate the execution of the later.

Rationale: The aim is to provide a MAPE loop with low latency analysis to allow rapid response to changes in the state of the target system based on pre-computed strategies. In other words to achieve fast adaptation to anticipated behaviour of the target system. Allow independent handling of failed assumptions made by either the behaviour or reconfiguration strategies, thereby adapting one strategy in a way that is transparent to the other.

Structure and Behaviour: The layer has two strategy enactors, one for behaviour strategies and the other for reconfiguration strategies. Both enactors work very similarly. They monitor the target system and react to changes in the system by invoking commands on the target system. The decision of which command to execute is entirely prescribed by the strategy being enacted and requires no significant computation. The two enactors do, however, differ in the instrumentation infrastructure they use to monitor and effect the target.

Reconfiguration Strategy Enactor: This entity invokes reconfiguration commands and accesses individual software component status information through an API provided by the Target System layer. The aspects monitored and effected by this enactor are application domain independent; commands and status data are related to the component deployment infrastructure and allow operations such as adding, removing and binding components, setting operational parameters of these components and checking if they are idle, active, and so on.

In addition to sequencing reconfiguration commands, the enactor has to resolve the challenge of ensuring that state information is not lost when the configuration is modified. This can involve ensuring stable conditions such as quiescence [15] passive or quiescent before change.

Behaviour Strategy Enactor: The entity monitors and effects the target system through application domain services provided by the components of the target system via behaviour commands and event abstractions exhibited by the Target System layer. The enactor starts executing the behaviour strategy assuming that there is a configuration in place that can provide the events and commands it requires. *Thus, a new search and analyse behaviour strategy using the gripper is assuming the gripper component configured.*

Should the behaviour strategy require a different configuration at any point, it must request the configuration change explicitly. In this case a reconfigure command will be part of the behaviour strategy and the behaviour enactor will command the execution of the reconfiguration strategy stored by the reconfiguration strategy enactor *(e.g., the behaviour strategy folds the arm holding the broken gripper and then requests reconfiguration to incorporate the infra-red camera to only then proceed with in situ analysis).* Note that in this case the behaviour enactor assumes that the reconfiguration strategy is attempting to reach a target configuration that is consistent with the behaviour strategy.

Assumptions regarding the current configuration and the target configuration of the strategy loaded on the reconfiguration strategy enactor are assured by the layer above that feeds consistent behaviour and reconfiguration strategies to this layer.

3. RELATED WORK

The last decade has seen a significant build up on the body of work related to engineering self-adaptive systems.

This work builds on this knowledge, emphasising the need to make behaviour and reconfiguration control first-class architectural entities. As discussed in Section 2, the architecture proposed builds on those of [16] and others. However, existing work does not provide support for both independent and also coordinated structural and behavioural adaptation at the architectural level.

The MORPH reference architecture is geared towards the use of strategies derived from the field of control engineering referred to as discrete event dynamic system (DEDS) control [4] which naturally fits over the system abstractions used at the architecture level, which is the level we envisage self-adaptation supported by our architecture to operate. DEDS are discrete-state, event-driven system of which the state evolution depends entirely on the occurrence of discrete events over time. The field builds on, amongst others, supervisory control theory [19] and reactive planning [5].

Automated construction of DEDS control strategies have been applied for self-adaptation in many different forms. For instance, in [1] temporal planning is used to produce reconfiguration strategies that do not consider structural constraints and the status of components when applying reconfiguration actions. In [22], an architecture description language (ADL) and a planning-as-model-checking are used to compute and enact reconfiguration strategies. In [7, 2, 14] automatic generation of event-based coordination strategies is applied for runtime adaptation of deadlock-free mediators. In [13], a learning technique (the L* algorithm [6]) is applied for automatically generating component's behaviour. Note that strategies do not have to be necessarily temporal sequencing of actions or commands. For instance, in [20] reconfiguration strategies used are one-step component parameter changes.

an executable modelling language for runtime execution of models (EUREMA) facilitates seamless adaptation.

4. CONCLUSIONS

An architectural approach to self-adaptive systems involves runtime change of system configuration (e.g., the system's components, their bindings and operational parameters that act as knobs) and behaviour update (e.g., components orchestration, reactive behaviour, etc). In this paper we present MORPH, a reference architecture for behaviour and configuration self-adaptation. MORPH allows both independent reconfiguration and behaviour adaptation building on the extensive work developed but also allows coordinated configuration and behavioural adaptation to accommodate for complex self-adaptation scenarios.

5. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 2007.

[2] A. Bennaceur, P. Inverardi, V. Issarny, and R. Spalazzese. Automated synthesis of connectors to support software evolution. *ERCIM News*, 2012.

[3] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *IJCAI*, 2001.

[4] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2010.

[5] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.*, 2003.

[6] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*, 2003.

[7] A. Di Marco, P. Inverardi, and R. Spalazzese. Synthesizing self-adaptive connectors meeting functional and performance concerns. In *SEAMS*, 2013.

[8] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *TOSEM*, 2013.

[9] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[10] D. Garlan, S. Cheng, A. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 2004.

[11] E. Gat, R. P. Bonnasso, R. Murphy, and A. Press. On three-layer architectures. In *AIMR*, 1997.

[12] C. Ghezzi, J. Greenyer, and V. P. La Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *SEAMS*, 2012.

[13] D. Giannakopoulou and C. S. Pasareanu. Context synthesis. In *SFM*, 2011.

[14] V. Issarny, A. Bennaceur, and Y. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM*, 2011.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *TSE*, 1990.

[16] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

[17] P. D. L. and M. A. K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge Uni. Press, 2010.

[18] A. V. Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE*, 2001.

[19] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. IEEE*, 1989.

[20] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *FSE*, 2014.

[21] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: A combined approach to self-management. In *SEAMS*, 2008.

[22] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. Plasma: A plan-based layered architecture for software model-driven adaptation. In *ASE*, 2010.