A Non-Intrusive Component-Based Approach for Deploying Unanticipated Self-Management Behaviour

Sandro Santos Andrade Raimundo José de Araújo Macêdo
Department of Computer Science - Distributed System Laboratory (LaSiD)
Federal University of Bahia
Campus de Ondina, 40170-110. Salvador-Ba. Brazil
{sandros, macedo}@ufba.br

Abstract

Supporting self-management behaviour by means of component-based architectural solutions has been the focus of many current research projects. In such a context, handling unanticipated changes with no impacts in application-specific software artefacts is a challenging problem. In this paper, we present a component-based solution for the specification and execution of self-management behaviour that decouples application-specific from autonomic-specific software artefacts (non-intrusive approach) and supports unanticipated changes in distributed heterogeneous environments. We describe the designed components for environment monitoring, adaptation policies specification, and changes redeployment and present a reference implementation built on top of CIAO middleware - a Lightweight CORBA Component Model implementation that supports dynamic reconfiguration of components and connectors. We also present two evaluation experiments, which provide self-optimization and self-healing behaviour in a distributed industrial supervision and control system.

1 Introduction

As the use of distributed components, software architectures, architectural styles, and ADLs (Architectural Description Languages) becomes an effective and well-established discipline, the adoption of such technologies when developing distributed autonomic systems draws the attention of some research groups [1, 4, 6, 12, 28]. By anticipating the rationale of self-management (and other non-functional) requirements to the architectural design stage of development process one can more easily meet such features using a scalable, reusable, and high-level approach.

Runtime architectural adaptation can be achieved in different degrees of autonomicity [10, 23]: changes can be enacted in a range from fully automatic to human-in-the-loop, constrained to adaptation plans previously defined at design-time (anticipated/closed-adaptive) or open to the insertion of new behaviour defined at runtime (unanticipated/open-adaptive), driven by high-level goals or by fine-grained software artefacts modifications, etc. In some particular application domains, like highly dependable and mission-critical distributed systems, continuous availability and unacceptable delays during maintenance/evolution strongly demand the support of unanticipated runtime changes. Moreover, separating application-specific concerns from autonomic-specific concerns is a design choice that facilitates the management of complexity in reusable, modular, and flexible software platforms.

In this paper we propose a component-based application framework that implements the autonomic control loop (environment monitoring - adaptation rules evaluation - changes redeployment), which defines hooks (hot-spots) for the composition of application-specific adaptation plans, architectural changes to be enacted (components and connectors creation/removal/reconfiguration), and required environment monitoring devices (CPU, network, RAM etc).

Self-management is achieved by two independent but related component configurations: the application-specific system and the autonomic observer system, in a non-intrusive approach. By deploying adaptation plans, architectural changes, and required monitoring devices as a component configuration we make the self-management model available at runtime, thus providing means for unanticipated changes support. We believe that our solution can tackle highly complex unanticipated self-management requirements by increasing the number of running configurations, in a multi-observer autonomic architecture.

We describe the framework architecture, the designed components in each module, and explain how a) adaptation...
plans and architectural changes can be created by composing the provided building blocks, b) environment monitoring devices can be plugged into the reusable autonomic core, and c) multi-observer self-management requirements can be specified (section 2).

The proposed platform comprises three main modules: Adaptation, Environment, and Redeployment. The Adaptation module defines a gray-box framework that specifies a component-based representation of adaptation policies, creating a tree-based composition of environment monitoring rules and a set of changes to be enacted whenever the evaluation of such rules returns true. The Environment module provides a white-box framework that specifies interfaces to be implemented in order to create the environment monitoring devices, such as CPU, network, and application-specific performance sensors. The Redeployment module insulates the reusable framework core from middleware-specific aspects for runtime redeployment of component configurations.

We also present a reference implementation built on top of the CIAO (Component-Integrated ACE ORB) [19] middleware - a Lightweight CORBA Component Model [21] implementation devoted to the construction of high-performance and real-time distributed systems (section 3). Our choice to develop a CIAO-based solution is justified by the wide heterogeneity support provided by CORBA in conjunction with the adaptation capabilities and high-performance and real-time features of CIAO.

Two evaluation experiments (section 4) illustrate how our framework can be used to implement self-optimization and self-healing behaviour in an industrial supervision and control system built on top of the ARCOS platform [2] - a previous component-based solution developed by the authors. We describe the environment monitoring components we have developed and the configuration that represents the expected autonomic behaviour. Finally, we discuss similar research efforts in section 5 and draw some concluding remarks and future work in section 6.

2 The Proposed Framework

Architecture-based approaches to self-managed systems have been the focus of many current research efforts, covering aspects such as architecture description languages (ADLs) [15, 17, 18], architectural styles [5, 26], planning [28], and autonomic architecture prototypes [12, 23, 29]. In order to deliver a fully self-managed architecture-based software platform some important aspects [10, 17, 22] should be considered:

- Underlying component model: components, connectors, and architectural configurations (a set of components and connectors that describes architectural structure) are the main entities defined by a component model. These entities are treated in distinct ways in existing research work, for example modeling connectors as first-class entities (like C2 [16] and Weaves [9] models) or implicit entities (like Darwin [14, 15] and RAPIDE [13] models), defining interfaces by provided/required services (like Darwin) or by request/notification (like C2), specifying component configurations by using graphs of components and connectors (like C2) or by using hierarchical composite components (like Darwin), etc. The expected degree of autonomicity and self-management capabilities provided by the platform depends heavily on the expressiveness of the underlying component model.

- Degree of autonomicity: previous work [8, 10, 23] has characterized autonomic computing in some degrees of autonomicity, ranging from simple conditional expressions to evolutionary programming and proactive goal-directed behaviour. Deciding what comprises a change, when and how changes take place, how to ensure architectural constraints, and how to describe self-management goals are important aspects when designing autonomic platforms.

- Planning and high-level reasoning: dealing with fine-grained software artefacts has become a hard task in current complex systems. Providing mechanisms for specifying high-level business-driven goals and mapping such goals onto low-level software artefacts are important topics in current research projects. Model checking based approaches, utility functions, and the use of ECA (Event-Condition-Action) rules have been evaluated as solutions for autonomic planning [11].

- Runtime support environment: in order to perform dynamic architectural changes in an autonomic system a runtime support environment with adaptive behaviour must be used. Middleware solutions that deliver services for dynamic component/connector creation and removal as well as for querying the current architectural configuration, in addition to common facilities for distributed communication and composability, have widely been used in adaptive and autonomic systems research. The mechanism provided by the middleware for applying changes in a running configuration is usually a redeployment service guided by (re)configuration descriptors in XML format or an architecture reconfiguration language that provides commands (such as create_component, remove_connector, etc) to be executed in some shell (interpreter) application.

- Consistency and system integrity maintenance: runtime changes must be applied in a manner that component states and system integrity are kept safe. This
includes checking for pending requests, blocking new connections during a component modification, and ensuring that the system reaches a valid state after changes take place. The runtime support environment can be responsible for some low-level consistency maintenance whereas application-specific system integrity should be implemented by the developer often by using architectural constraints [5] or formal models [28] for the representation of valid system states.

Taking the aspects listed above, the proposed platform for self-management - the IRIDIUM framework - provides a reusable core of component-based services that implements common features (such as autonomic control loops and building blocks for adaptation policies and architectural changes definition) while provides hooks for the connection of developer-supplied components that specializes IRIDIUM to a particular scenario of environment monitoring. For example, one can implement a component for network monitoring that gathers the average rate of packet loss. By connecting this new component in the IRIDIUM’s core such network metric would be available for use in adaptation policies that trigger changes in response to the current network status. Built-in components enable the description of adaptation rules, like "PacketLoss > 0.8 or (CpuUsage > 0.9 and MemoryUsage > 0.5)" as a tree-based component configuration that can be evaluated and even modified at runtime. Another group of built-in components is used to describe the changes that should be enacted in response to adaptation rules evaluated as true.

Figure 1 briefly depicts our non-intrusive approach. Two component-based systems constitute the whole autonomic scenario: the application-specific system, which implements the functional requirements and is unaware of autonomic issues; and the autonomic observer system (IRIDIUM instance), which maintains the adaptation rules, corresponding changes, and environment monitoring devices currently in use. The components in the autonomic observer system should be configured by the system administrator in such a way that they reflect the expected self-management in the application-specific system. The autonomic observer manages a number of adaptation policies, each of them comprising the adaptation rules, the adaptation actions (changes to be enacted when rules are evaluated as true), and the rules evaluation rate. Periodically, at the specified rate for each policy, IRIDIUM evaluates the tree-based representation of adaptation rules and triggers the corresponding adaptation actions whenever the evaluation results true.

Our solution requires the use of a middleware platform that provides some architecture reconfiguration service as part of the runtime support environment, in addition to the common services for distributed communication, resource management, naming and localization, etc. Such middleware platform should provide the basic mechanisms for consistency maintenance during changes, like awaiting pending requests and blocking new connections.

Three main modules implement core functionality for adaptation policies specification, gathering of environment monitoring data, and deployment of changes, as shown in figure 2. The most important module, Adaptation, is responsible for holding the required information about adaptation policies and triggering periodic evaluations of the supplied adaptation rules. As a gray-box sub-framework, this module provides built-in components for adaptation rules composition as well as for the most commonly used adaptation actions (component and connector creation/removal and component’s attribute reconfiguration). New adaptation actions, e.g. component migration, can be implemented and plugged into the Adaptation module.

The Environment module provides mechanisms for creating specific environment monitoring components, such as CPU or RAM usage, network-related metrics, and application-specific measurement components. Once created, monitoring components are then registered in the Environment module and become available for use in adaptation rules.
When configuring anticipated self-management behaviour, which adaptation rules should be evaluated. Architectural changes specification: the autonomic control loop, adaptation rules evaluation, and component (a hook), and contain configurable attributes (small white circle), require a service from another component. A component can provide a service (a connection point for the tree-based representation of rules) and groups the architectural changes - adaptation actions - to be enacted.

- AdaptationPolicy: represents a specific self-management behaviour, composed by adaptation rules and adaptation actions. It defines a provided interface (for connection with AdaptationManager), an attribute for evaluation rate definition, and two required interfaces: IAdaptationRule (that defines the entry point for the rule-based representation of rules) and IAdaptationActions (that groups the architectural changes - adaptation actions - to be enacted).

- AdaptationRule: basic building block for adaptation rules specification. It can play two different roles when specifying adaptation rules: a single expression involving an environment monitoring device (e.g. CpuUsage > 0.9) or a logical operator executed on underlying adaptation rules (which in turn can be a single expression or a logical operator). The AdaptationRule component defines four attributes: DeviceName, RelationalOperator, and Value (used for single expression representation); and LogicalOperators (used for logical operations on underlying adaptation rules). Figure 3 illustrates the hierarchical representation for the adaptation rule "PacketLoss > 0.8 or (CpuUsage > 0.9 and MemoryUsage > 0.5)". In this tree-based composition of adaptation rules leaves represent single expressions involving an environment monitoring device and internal nodes represent logical operators executed on their children. This component-based approach enables the flexible specification of adaptation rules since it provides the means for precedence definition and relies on an extensible set of environment monitoring devices. Furthermore, as we deploy adaptation rules as component configurations, runtime changes can be applied with the purpose of supporting unanticipated architectural changes by including new rules or modifying already deployed ones.

- Adaptation actions: represent the architectural changes to be enacted whenever rules evaluation returns true. The DeploymentPlanUUID attribute uniquely identifies the component configuration that represents the system to be self-managed. In order to implement the autonomic control loop AdaptationManager communicates with the Environment module to collect data from environment monitoring devices and asks the Redeployment module to perform architectural changes when necessary.

The Redeployment module insulates IRIDIUM’s core from middleware-specific code for deploying architectural changes and provides a clean interface to the architecture reconfiguration service. In the next sections we present a detailed view of IRIDIUM’s modules.

### 2.1 The Adaptation Module

The Adaptation module implements a simplified version of the ECA (Event-Condition-Action) [11] approach. When configuring anticipated self-management behaviour, the system administrator defines what adaptation policies should be deployed. To do this, built-in components should be composed, in a black-box framework style. An adaptation policy comprises a) the adaptation rules (expressions involving environment monitoring devices), b) adaptation actions (architectural changes to be enacted), and c) rate at which adaptation rules should be evaluated.

Figure 3 shows the designed components for the Adaptation module. A component can provide a service (a small white circle), require a service from another component (a hook), and contain configurable attributes (small white rectangles). The following components implement the autonomic control loop, adaptation rules evaluation, and architectural changes specification:

- AdaptationManager: the main component in the Adaptation module. It defines a required interface (IAdaptationPolicy) in which all the adaptation policies must be connected. For each policy AdaptationManager executes the evaluation of policy’s adaptation rules and selects the proper adaptation actions whenever rules evaluation returns true. The DeploymentPlanUUID attribute uniquely identifies the component configuration that represents the system to be self-managed. In order to implement the autonomic control loop AdaptationManager communicates with the Environment module to collect data from environment monitoring devices and asks the Redeployment module to perform architectural changes when necessary.

- AdaptationPolicy: represents a specific self-management behaviour, composed by adaptation rules and adaptation actions. It defines a provided interface (for connection with AdaptationManager), an attribute for evaluation rate definition, and two required interfaces: IAdaptationRule (that defines the entry point for the rule-based representation of rules) and IAdaptationActions (that groups the architectural changes - adaptation actions - to be enacted).

- AdaptationRule: basic building block for adaptation rules specification. It can play two different roles when specifying adaptation rules: a single expression involving an environment monitoring device (e.g. CpuUsage > 0.9) or a logical operator executed on underlying adaptation rules (which in turn can be a single expression or a logical operator). The AdaptationRule component defines four attributes: DeviceName, RelationalOperator, and Value (used for single expression representation); and LogicalOperators (used for logical operations on underlying adaptation rules). Figure 3 illustrates the hierarchical representation for the adaptation rule "PacketLoss > 0.8 or (CpuUsage > 0.9 and MemoryUsage > 0.5)". In this tree-based composition of adaptation rules leaves represent single expressions involving an environment monitoring device and internal nodes represent logical operators executed on their children. This component-based approach enables the flexible specification of adaptation rules since it provides the means for precedence definition and relies on an extensible set of environment monitoring devices. Furthermore, as we deploy adaptation rules as component configurations, runtime changes can be applied with the purpose of supporting unanticipated architectural changes by including new rules or modifying already deployed ones.

- Adaptation actions: represent the architectural changes to be enacted whenever rules evaluation returns true. IRIDIUM provides five different implementations of the IAdaptationAction interface: RemoveComponent, RemoveConnector, CreateComponent, CreateConnector, and ChangeComponentAttribute. New adaptation actions,
Figure 3. Designed components for the Adaptation module. AdaptationManager keeps track of all registered adaptation policies. Each of them presents two branches of components: the adaptation rules (on the left) and the adaptation actions (on the right).

e.g. ComponentMigration, can be implemented as long as there is a proper support in the Redeployment module. Required information to perform the adaptation action are defined in attributes for each action in particular, e.g. unique ConnectorName/ComponentName (for RemoveConnector and RemoveComponent actions); Implementation (component type), ComponentName, and Node to be deployed (for CreateComponent action); and unique ConnectorName and information about the two end-points (for CreateConnection action).

2.2 The Environment Module

The Environment module defines an extensible framework for the creation of environment monitoring devices, such as CpuUsage, MemoryUsage, PacketLoss, MethodInvocationPerformance, etc. As depicted in figure 4, the EnvironmentManager component defines a multiple hook which is expected to be connected to developer-supplied components (monitoring devices) that implement the IDevice interface. Such components should acquire information about a specific environment entity (e.g. CPU/memory usage, packet loss) and make it available.
through the IDevice interface. Each monitoring component has two attributes: a unique Name for monitoring device identification and a Type, which informs the returned data type (int, float, double, boolean, char, or string).

When defining the desired self-management behaviour the system administrator should implement (or reuse) the required monitoring components and plug them into the EnvironmentManager component. All adaptation rules defined in the Adaptation module should refer to monitoring devices connected to the EnvironmentManager by using its unique name in the DeviceName attribute of adaptation rules.

Figure 5 shows the designed IEnvironmentManager and IDevice interfaces and how to create a monitoring device (CpuUsage). Line 5 defines the get_device_by_name method, used by AdaptationManager component to get the devices to be used when recursively evaluating adaptation rules.

Lines 8-10 define the interface all monitoring component should implement. The data method returns updated information about the specific environment entity being monitored and is invoked by AdaptationManager whenever a rule that considers this device is evaluated.

Lines 11-15 show how a monitoring component is created. In this example, a CpuUsage monitoring component provides an IDevice implementation, which must be able to acquire CPU usage information, and the Name and Type attributes configured for use in Adaptation module.

### 2.3 The Redeployment Module

The goal of the Redeployment module is twofold: it insulates IRIDIUM's core from middleware-specific code for deploying architectural changes and it provides a cleaner interface to Adaptation module. Figure 6 presents the designed components: RedeploymentManager defines the IRedeploymentService interface, which is implemented by some MiddlewareSpecificImplementation component.

The MiddlewareSpecificImplementation component should implement the two method depicted in figure 7. The handle_adaptation_action method (lines 5-7) should execute any pre-redeployment operations for the provided adaptation action and include it in a set of architectural changes to be afterwards deployed. Then, the perform_redeployment method actually enacts the architectural changes previously handled.

Based on the mentioned IRIDIUM services, in order to provide self-management behaviour one should select components provided by the framework and implemented by the developer and tie them up in a component config-
Component and connectors assembling depends heavily on middleware-specific mechanisms, e.g. the use of XML deployment descriptors, visual assembly tools, or configuration-specific programming languages.

Unanticipated changes can be supported in two ways: or by manually deploying changes in the autonomic observer system either by adopting a multi-observer approach. The choice of representing the self-management model as a component configuration makes it eligible to be self-managed by another running instance of IRIDIUM, in a three (or multi) observer configurations approach. In doing so, adaptation rules and actions (and the whole IRIDIUM infrastructure) can be autonomically created, removed, and reconfigured.

3 Implementation Issues

The proposed framework was fully implemented in a Debian/GNU-Linux platform using the CIAO [19] middleware as runtime support environment. The CIAO middleware is a high-performance and real-time implementation of the Lightweight CORBA Component Model (CCM) [21] that supports runtime architectural changes by means of the ReDaC (Redeployment and Reconfiguration) service [25].

CIAO implements the OMG standard [20] for configuration and deployment of CCM-based systems in a service called DAnCE (Deployment and Configuration Engine) [7]. In such a service, component configurations are described in terms of XML deployment descriptors, which contain the component instances and connectors to be created as well as the initial values for component attributes. DAnCE provides a set of interesting features for autonomic systems implementation: distributed deployment of component instances in distinct network nodes; remote connection of components running in different network nodes; a component repository service for browsing and automatic downloading component binaries; and the ReDaC service, which provides features for browsing the current architectural topology and redeploying a new changed architecture.

For each deployed CCM system DAnCE maintains an in-memory representation of the respective XML deployment descriptor uniquely identified by a deployment plan UUID. ReDaC extends the DAnCE service by providing objects that enable browsing that in-memory representation and modifying specific XML fragments. As a consequence, components, connectors, and attributes can be changed to reflect the expected self-management behaviour and this new in-memory XML deployment descriptor can be redeployed, actually enacting the architectural changes.

In our reference implementation, the Middleware-SpecificImplementation component was implemented in the ReDaCUtils component, a helper class that wraps the ReDaC service and adapts its interface to the IRedepliymentService interface designed in IRIDIUM. In ReDaCUtils, the handle_adaptation_action method invokes the ReDaC service in order to actually apply the changes in the running system.

The AdaptationManager implementation relies on features provided by ACE (ADAPTIVE Communication Environment) [27]. ACE is an application framework devoted to the construction of high-performance distributed systems, which defines an useful object-oriented API for distributed and concurrent computing. ACE enables the multi-platform development for a number of operating systems and hardware environments and is extensively used in CIAO development.

As depicted in figure 8, after all IRIDIUM components have already been deployed and tied up the ciao_postactive method on AdaptationManager component scans all connected policies (lines 3, 4, and 7-9) and, for each of them, uses the ACE_Thread_Manager singleton (line 5) to spawn a new thread, which begins to execute the run method. The policy thread uses the ACE_Reactor design pattern [27] implementation, provided by ACE, to schedule the timer (line 21) that drives the adaptation rules in accordance with the rate specified in the AdaptationPolicy component provided as a parameter to run method (lines 19 and 20). In line 23 the thread blocks for handling ACE_Reactor events. Finally, whenever the timer expires the handle_timeout method acquire the AdaptationRule directly connected to the AdaptationPolicy (the root node of the tree-based representation of adaptation rules) (lines 28 and 29) and executes the recursive algorithm for rule evaluation (line 31). Should the rule evaluation return true the adaptation actions connected to the AdaptationPolicy are gathered and
void AdaptationManager_exec_i::ciao_postactivate() ACE_THROW_SPEC((SystemException, CCMEException))
{
    i_adaptationPolicyConnections_var policies_facet = context_−>get_connections(i_adaptationPolicy());
    CORBA::ULong len = policies_facet−>length();
    ACE_Thread_Manager *thr_mgr = ACE_Thread_Manager::instance();
    for(int index=0; index<len; index++) {
        CORBA::Object_var obj = policies_facet[index].objref−>get_component();
        AdaptationPolicy_var adaptation_policy = IRIDIUM::Adaptation::AdaptationPolicy::narrow(obj.in());
        int thread_id1 = thr_mgr−>spawn(&run, reinterpret_cast <void *>(&adaptation_policy),
         THR_NEW_LWP | THR_JOINABLE | THR_INHERIT_SCHED, 0, 0);
    }
    thr_mgr−>wait();
}

void *AdaptationManager_exec_i::run(void *arg)
{
    ACE_Reactor reactor;
    AdaptationPolicy_var adaptation_policy = reinterpret_cast <AdaptationPolicy_var *>(arg);
    ACE_Time_Value delay( adaptation_policy−>rate(), adaptation_policy−>rate() );
    reactor.schedule_timer( this, arg, delay, delay );
    for (;;) { reactor.handle_events(); }
}

int AdaptationManager_exec_i::handle_timeout(const ACE_Time_Value &tv, const void *arg)
{
    AdaptationPolicy_var adaptation_policy = reinterpret_cast <AdaptationPolicy_var *>(arg);
    IAdaptationRule_var rule_facet = adaptation_policy−>get_connection_i_adaptation_rule();
    if (evaluate_rule(rule_facet)) {
        enact_actions(adaptation_policy);
    } else { ACE_DEBUG((LM_DEBUG, "Rule\evaluation\return\false\n\n")); }
}

Figure 8. ACE-based implementation of the AdaptationManager

passed to the Redeployment module (line 32).

4 Evaluation Experiments

In order to validate the proposed platform we have implemented two evaluation experiments based on the ARCS framework [2]. ARCS is a flexible, reusable, and interoperable platform for the construction of real-time supervision and control systems that provides services for industrial data acquisition, closed control loops, and supervisory activities. In such application domain, self-optimization (to satisfy temporal constraints) and self-healing (dependability) are common requirements since safety-critical operations are usually involved.

Self-optimization experiment. Figure 9.a presents the implemented self-management behaviour using the services provided by IRIDIUM. The data acquisition implementation provided in ARCS specifies two main components: the DAISServer component - responsible for managing data acquisition sessions and periodic delivery of industrial data; and the DAISProvider component, which communicates with a specific data acquisition device (such as a parallel port or a programmable logic controller - PLC) and makes ARCS core reusable across a range of acquisition devices. In this experiment we provide self-optimization behaviour in ARCS data acquisition by automatically adding the DAISProviderDataCache component whenever the current number of data acquisition sessions reaches a specific threshold. For that purpose a SessionCounterDAISClient was implemented as a IRIDIUM environment monitoring device, which connects to the DAIS-Server, invokes its services to acquire the number of current data acquisition sessions, and makes this information available for use in adaptation policies. The DAISProviderDataCache component manages a cache table whose size and replacement policy are defined by the corresponding component attributes. Two adaptation policies were defined for adding and removing the cache component (and associated connectors) whenever the number of current data sessions becomes greater or smaller than a specific threshold, respectively.

Self-healing experiment. As indicated in figure 9.b a passive redundancy mechanism was implemented in or-
Figure 9. Evaluation experiments: a) self-optimization behaviour for industrial data caching; b) self-healing behaviour for replicated PID controllers

Architecture-based approaches to self-management are currently being considered a promising mechanism for designing flexible, scalable, and maintainable autonomic systems. We have presented a component-based platform for providing self-management behaviour in a non-intrusive approach that supports unanticipated architectural changes. We have shown the designed components for adaptation policies specification, environment monitoring devices implementation, and redeployment of architectural changes. Two evaluation experiments have demonstrated how our solution can be used to deploy self-optimization and self-healing behaviour in an industrial supervision and control system.

Future work include the definition of a planning mechanism for mapping high-level autonomic goals onto the IRIDIUM defined adaptation actions, resolution sub-system for conflicting adaptation policies, IRIDIUM extensions for coping with self-management in the presence of real-time constraints, component replacement and migration support, and reports on empirical experiments concerning aspects such as productivity, code defects, and self-management coverage when using the proposed framework.

References


