A SELF-ADAPTIVE ARCHITECTURE FOR AUTONOMIC SYSTEMS DEVELOPED WITH ASSL

Emil Vassev[1], Mike Hinchey[2] and Aaron Quigley[1]
[1]*Lero – The Irish Software Engineering Research Centre, University College Dublin, Ireland*
[2]*Lero – The Irish Software Engineering Research Centre, University of Limerick, Ireland*

Keywords: Autonomic Systems, Software Architecture, Adaptive Architecture, ASSL, Runtime.

Abstract: We address the need to realize a runtime self-modifiable architecture for autonomic systems, specified and generated with the ASSL (Autonomic System Specification Language) framework. This framework generates such systems with a special hierarchical multi-granular architecture composed of singleton classes. Base ASSL is designed to support runtime evolving systems, whereas in this approach, we extend the generated architecture to allow for both *code generation* and *code management* at runtime. We provide a tailored algorithm to demonstrate how this approach can be applied to customized ASSL models specified to trigger runtime changes in the structure of the generated autonomic systems.

1 INTRODUCTION

Ongoing developments demonstrate that many information systems are moving towards service-oriented architectures running on grids that may include thousands of geographically distributed systems. In addition, such systems are dependent on multi-site collaboration and fast, reliable access to shared resources and data. The success of such systems depends on their ability to follow changes in the business environment and requirements by reacting and adapting so as to better handle their service-level objectives. In practice, such reaction and adaptation is typically a manual process of identification, redesign and change. By contrast, autonomic computing (AC) (Murch, 2004) is widely regarded as a suitable approach to the development of software systems capable of *self-management*. In general, AC strives to fulfill two main objectives – self-regulation and complexity hiding.

The Autonomic System Specification Language (ASSL) (Vassev, 2008) is a tool dedicated to AC development that addresses the problem of formal specification and code generation of autonomic systems (ASs) within a framework. The work described here is motivated by the need to complement the ASSL framework with suitable constructs and in-built mechanisms for specifying and generating runtime evolving ASs. With such extents, ASSL can generate ASs capable of architecture self-modification at runtime. ASSL currently defines statements allowing runtime changes in the AS structure but those are not implemented at the level of code generation, and thus, generated systems do not have the ability to change their structure at runtime.

The rest of this paper is organized as follows. In Section 2, we review related work on adaptive architectures for software systems. As a background to the remaining sections, Section 3 provides a brief description of the ASSL framework. Section 4 presents both the current and the future adaptive ASSL architecture for ASs. In addition, this section presents our algorithm for AS architecture evolution at runtime. Finally, Section 5 presents some concluding remarks and future work.

2 RELATED WORK

Considerable research effort has been directed at the question of self-adaptation of systems, from rewriting schemes to multi-agent systems. Kitano proposes an approach to evolving architectures of artificial neural networks using a special matrix rewriting system that manipulates adjacency matrices (Kitano, 1990).

Oreizy et al. propose an architecture-based approach to self-adaptation. In their approach, new software components are dynamically inserted into deployed, heterogeneous systems without requiring a system restart. The architecture changes rely on autonomous analysis that includes feedback of current performance. Changes are encoded in the system behavior by the application developers. In the case of major changes, the system can request and require human approval (Oreizy et al., 1999).

Rainbow (Garlan et al., 2004) is a framework that relies on adaptation mechanisms to specify adaptation strategies for multiple system concerns. These help Rainbow to integrate a reusable infrastructure into software architectures to support self-adaptation of software systems. The adaptation strategies let developers of self-adaptation capabilities choose what aspects of a system to change and how to adapt the system.

Brogi et al. target the problem of adapting heterogeneous software components (Brogi et al. 2006). Their approach is based on a special adaptation methodology where components are presented with interfaces extended with protocol information to describe their interaction behavior. In addition, a high-level notation is used to express the intended connection between component interfaces. In this approach, a special adaptor is specified as a *component-in-the-middle* to help two components interact successfully, considering certain constraints.

A special compositional adaptation approach is tackled by McKinley et al. where to improve the system's fit to its environment, both algorithms and structural components are exchanged with other systems (McKinley et al., 2004). Here, by adopting new algorithms, a system can address concerns unforeseen during development.

In general, adaptive software provides some of the functionality required for building AC systems, as it allows system behavior or structure to be changed at runtime to fulfill high-level objectives (Murch, 2004). Research in autonomic architectures consists of general architectures for individual components or complete ASs, based on the integration of advanced technologies such as grid computing, web services, and multi-agent technologies, e.g., intelligent swarm systems (Truszkowski et al., 2004).

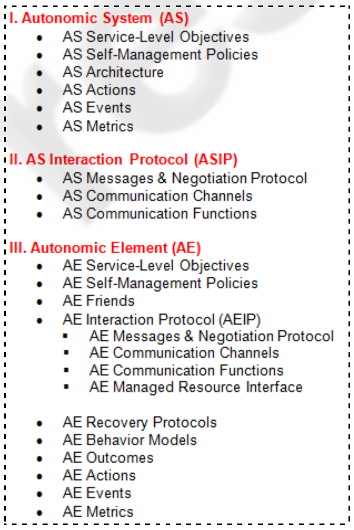
The work presented here is an AC approach where the ASSL framework is extended to provide more suitable constructs for exploiting the benefits of AC. Note that in our approach both formal notation and tools help ASs adapt at runtime by changing their structure if necessary. Moreover, we also consider runtime exchange of structures and algorithms together with code generation and hot plugging of system components.

3 ASSL

The ASSL framework is a development environment that delivers a powerful combination of ASSL notation and ASSL tools (Vassev, 2008). The tools allow specifications written in the ASSL notation to be edited and validated. ASSL can generate an operational implementation per a valid specification.

3.1 Multi-tier Specification Model

In general, ASSL considers ASs as composed of autonomic elements (AEs) interacting over interaction protocols. To specify ASs, ASSL exposes a multi-tier specification model that exposes a judicious selection of infrastructure elements and mechanisms needed by an AS. By their nature, the ASSL tiers are abstractions of different aspects of the AS in question, including *self-management policies*, *communication interfaces*, *execution semantics* and *actions*. There are three major tiers, each composed of sub-tiers (cf. Figure 1).



I. Autonomic System (AS)
- AS Service-Level Objectives
- AS Self-Management Policies
- AS Architecture
- AS Actions
- AS Events
- AS Metrics

II. AS Interaction Protocol (ASIP)
- AS Messages & Negotiation Protocol
- AS Communication Channels
- AS Communication Functions

III. Autonomic Element (AE)
- AE Service-Level Objectives
- AE Self-Management Policies
- AE Friends
- AE Interaction Protocol (AEIP)
    - AE Messages & Negotiation Protocol
    - AE Communication Channels
    - AE Communication Functions
    - AE Managed Resource Interface
- AE Recovery Protocols
- AE Behavior Models
- AE Outcomes
- AE Actions
- AE Events
- AE Metrics

Figure 1: ASSL multi-tier specification model.

Here, the AS tier forms a general and global AS perspective, where we define the general system rules, e.g., *service-level objectives (SLO)* and *self-management policies*; the AS Interaction Protocol (ASIP) tier defines the means of communication between AEs; and the AE tier forms a unit-level perspective, where we define interacting sets of individual AEs with their own behavior.

3.2 ASSL Runtime-Evolving Systems

Initially, ASSL was conceived as a software development approach that provides a means for modifying the internal structure of an AS at runtime. Moreover, the original idea is that an AS specified with ASSL should have a runtime-evolving specification; i.e., a specification that can be updated dynamically to keep up with any architectural changes. Here, an ASSL-generated AS should carry its specification and change it at runtime; i.e., both implementation and specification evolve together.

ASSL currently allows for the specification of ASs evolving over time. The evolution of such systems takes place in the *ASSL actions* (specified at the *action* tiers; cf. Figure 1) of the system. Via a finite set of special ASSL statements — CHANGE, REMOVE, ADD and CREATE, the ASSL actions can prompt changes in the tiers and sub-tiers of the AS under consideration (Vassev, 2008). ASSL currently supports these statements at the level of ASSL specification but not at the level of code generation.

4 ASSL ARCHITECTURE FOR AUTONOMOUS SYSTEMS

By using the ASSL framework, we specify an AS (autonomic system) at an abstract formal level. Next, that formal model is translated into a Java program consisting of packages and classes that inherit names and features from the ASSL specification.

4.1 Current Architecture

The current ASSL architecture (cf. Figure 2) for ASs conforms to the ASSL multi-tier specification model (cf. Section 3.1). Here, every AS is generated with:

- a global AS autonomic manager (implements the AS tier specification) that takes care of the AS-level policies and SLO;
- a communication mechanism (implements the specification of both ASIP and AEIP tiers) that allows AEs to communicate;
- a set of AEs (implement the AE tier specification) where each AE takes care of its own self-management policies and SLO.

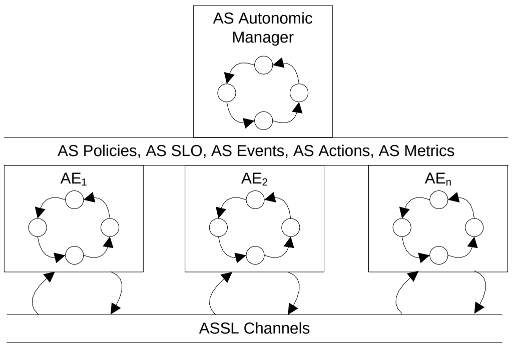


Figure 2: ASSL architecture for ASs – a design view

Both the AS manager and distinct AEs embed a special *control loop* (cf. Figure 2) generated by the framework to allow an AS manage critical situations with a sort of problem-solution mapping. In addition, the generated AS manager and the AEs orchestrate the self-management policies of the AS. Here, the AS manager coordinates the AEs via AS-level *self-management policies, SLO*, *events*, *actions* and *metrics* (cf. Figure 1).

4.1.1 AS Class Model

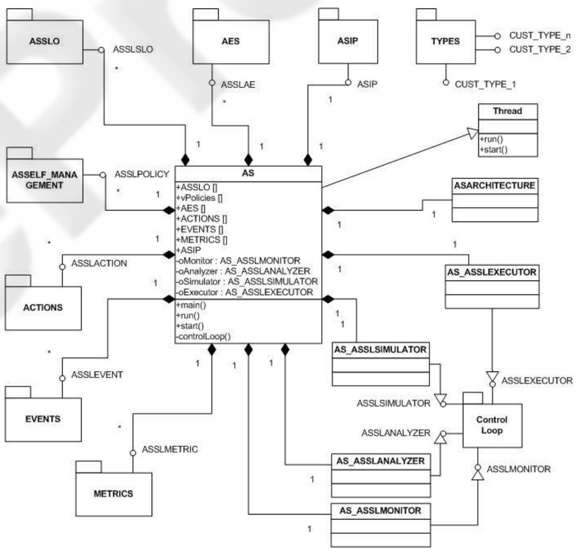


Figure 3: AS UML class diagram.

The ASSL architecture model for ASs (depicted in Figure 2) is mapped to a hierarchically organized set of Java classes. The ASSL framework generates a Java class with optional supplementary classes for each ASSL-specified tier. The generated *tier classes* are grouped into distinct *tier Java packages* derived from the ASSL specification model.

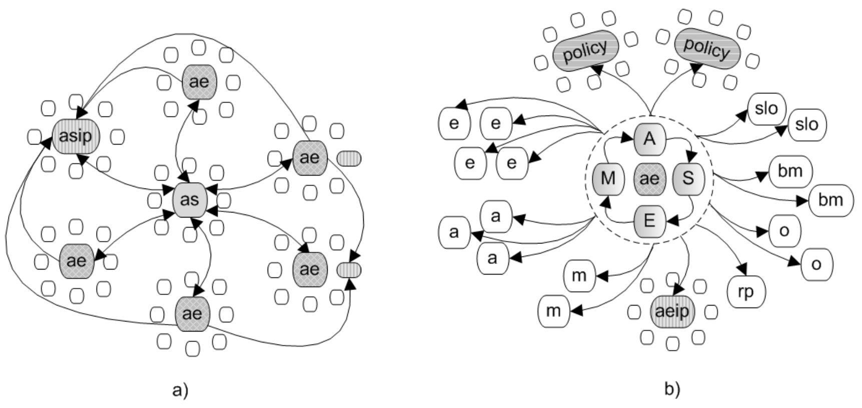


Figure 4: (a) AS runtime object model; (b) AE runtime object model.

Figure 3 presents a UML diagram depicting the class structure of ASs generated with ASSL. Here an ASSL-generated AS has, at a minimum, a main AS class (mapped to the AS tier and presenting the AS manager; cf. Figure 2) and classes implementing the AS manager's control loop (classes denoted as AS_ASSLMONITOR, AS_ASSLANALYZER, AS_ASSLSIMULATOR, and AS_ASSLEXECUTOR). In addition, the AS class maintains collections (pools) of references to *tier instances*. The latter are objects instantiated from the classes generated for the AS tiers/sub-tiers specified in the ASSL specification.

As shown in Figure 3, such tier classes are nested in proper packages. Each generated AE has a similar class structure nested in a proper AE package. In addition, all the classes generated by the ASSL framework and mapped to ASSL tiers are implemented as singletons; i.e., they define only private constructors and ensure a single instance.

4.1.2 AS Object Model

Instead of building a monolithic application for each ASSL-specified AS, the ASSL framework strives to organize generated ASs in a granular fashion. Here, at runtime, an ASSL-generated AS has a multi-granular structure composed of tier instances. All the tier instances form together the *runtime object model* of an AS (cf. Figure 4). Similar to the ASSL specification model (cf. Section 3.1), the AS runtime object model has a hierarchical composition where tier instances are grouped around instances of their *host tiers* (nesting other sub-tiers). Figure 4(a) depicts the runtime object model of an AS generated with ASSL and Figure 4(b) presents the runtime object model for AEs composing that AS. Note that, both Figure 4(a) and Figure 4(b) present *generic object models*. Thus, concrete models have an arbitrary number and types of nodes derived from their corresponding ASSL specification. As depicted by Figure 4, each node is a *tier instance* that possibly can be grouped around a *host tier instance*. For example, the AS node acts as a host tier instance for the nodes generated for the AS-level sub-tiers such as *SLO*, *policies*, *actions*, *events*, and *metrics*. Note that the AS node organizes around itself other host tier instances, such as AE nodes (generated for the AEs specified at the AE tier) and the ASIP node (cf. Section 3.1). Here both the AE nodes and the ASIP node have their own surrounding nodes, these being instances of sub-tiers specified at the AE tier and at the ASIP tier respectively. The presence of host tier nodes (AEs and ASIP) in the global AS runtime object model (cf. Figure 4(a)) makes that model multi-granular where we distinguish different levels of granularity.

Figure 4(b) presents the granular structure of the AE runtime object model. Here, at the core of the AE we can see four objects forming the AE control loop. As depicted, the latter is composed of the objects: M (monitor), A (analyzer), S (simulator), and E (executor). The AE node coordinates the tier instances of the sub-tiers specified for that AE; i.e., *metrics* (m nodes), *events* (e nodes), *actions* (a nodes), *self-management policies* (policy nodes), *service-level objectives* (slo nodes), *behavior models* (bm nodes), *outcomes* (o nodes), *recovery protocols* (rp nodes), and its *private interaction protocol* (aeip node). Here both the policy nodes and the aeip node are host tier instances themselves.

4.2 Proposed Architecture

As mentioned previously, the starting point for this work is the fact that ASSL allows the specification of ASs evolving over time, but not for the code generation of the same (cf. Section 3.2). Here our primary goal is to augment the architecture model for ASSL-generated ASs (cf. Section 4.1) with the necessary components that allows for both *code generation* and *code management* at runtime. To meet these requirements we augment both the AS object model and the AE object model with a code generator (CG) and a code manager (CM).

Figure 5 represents the new runtime object model for AEs generated with ASSL. Compared to the model presented in Section 4.1, this model has two more *host nodes* (note that these are not *host tier nodes*) — CM and CG, organizing objects needed to make the AE capable of self-modification at runtime. Here at runtime, the CG generates the needed code and compiles it by using a java compiler (javac). The CM is responsible for integrating the generated code in the currently running AS by relying on the Java VM hosting that AS. Note that both the CG and the CM access the ASSL specification, and thus, the latter should be carried with the generated AS and its AEs. Moreover, the new architecture requires a CG-CM pair be deployed with every generated AE and also with the main AS package (AS host tier node; cf. Figure 4.a). Note that the CG is the original ASSL code generator (Vassev, 2008), but deployed with every generated AE and with the AS manager. The deployed CMs are responsible for maintaining the runtime structure of the associated AE/AS object model, which includes adding new tier instances, deleting tier instances, and replacing tier instances.

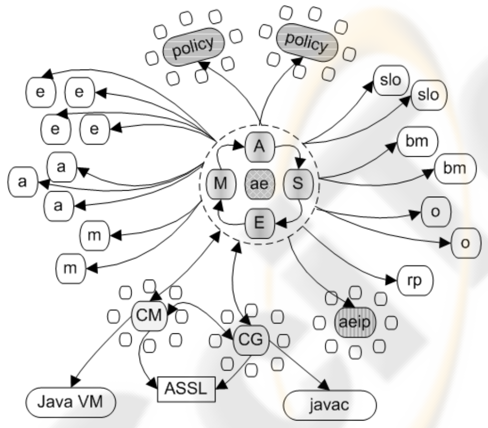


Figure 5: New AE runtime object model.

It is important to note that there are two cases where both code generation and code management are required at runtime:

1) The ASSL-generated AS calls an *action* that performs one of the ASSL statements CHANGE, REMOVE, ADD or CREATE, these triggering changes in the AS structure (cf. Section 3.2). Here while generating an AS, the ASSL framework does not generate the Java code for such statements but embeds a system call to a runtime CG with a reference to the ASSL specification block specifying structural changes in the AS.

2) An AE receives an ASSL message carrying the specification of a specific tier or sub-tier (AEs can negotiate and exchange ASSL messages in the form of ASSL-specified tiers/sub-tiers (Vassev, 2008)). Thus, if such a message has been received, a runtime CG can generate the necessary code and then compile it.

Once the code is generated, the associated CM integrates the generated code by instantiating objects and plugging those into the running AS. Plugging/unplugging objects at runtime (known as *hot-plugging*) is a difficult task. However, there a few key features of the ASSL-generated code, which facilitate this task:

1) As we have mentioned before, all the tier objects (instances) instantiate singleton classes; i.e., there is one object per tier class. This reduces the number of objects that CMs need to take care of at runtime, and also, the tier objects reference to each other via a *singleton* class access point.

2) All the host tier instances (as, ae, asip, aeip, policy, etc., cf. Figures 4 and 5) maintain pools of references to associated tier instances (singleton class instances, cf. the following example). Moreover, access to those pools is provided via predefined put(), get(), and remove() public methods, and tier instances do not keep local reference copies to other tier instances. Hence, only host tier pools keep references to tier instances.

3) An ASSL-generated system is generated as a complex multi-threaded Java application, where some of the tier instances (e.g., event tier instances) run as synchronized Java threads. Thus, an ASSL-generated AS can be easily synchronized on a "pause" system event; i.e., the entire AS can transit to a "pause" state where all the threads can be put on hold, thus allowing CMs make the needed code changes at runtime.

The following is an example of adding an event tier instance to the EVENTS pool of an AE.

```
EVENTS.put("NODEFIXED", generatedbyassl.as.
   aes.ae1.events.NODEFIXED.getInstance());
```

Adding a new tier instance to, or deleting an old one from an AS should be split into three steps – 1) put the AS on "pause"; 2) use the public methods put() and remove() to add or delete references to a tier instance to/from all the referee host tier pools; 3) if this is about deleting only (not replacing) then generate a stub (dummy instance) and replace the old tier instance with the latter, thus keeping the AS runtime object model consistent.

In order to make hot plugging possible, we should make CMs know at runtime the host tier pools that keep or must keep references to the tier instance that is about to be *added*, *replaced*, or *deleted*. A possible solution is to make existing tier instances know all the referencing host tier pools. For new tier instances this information should be derived by CGs from the ASSL specification while generating code for the same. The following is the hot plugging algorithm for *replacing* an old tier instance with a new one. Algorithms for hot plugging in *adding* or *deleting* tier instances can be deducted from this one.

1) A CG generates the code for a new tier class with all necessary supplementary classes that must replace an old tier instance.

2) That CG compiles the generated code by using a javac Java compiler.

3) The same CG notifies its paired CM that a specific tier instance is going to be replaced.

4) A CM (paired with the CG) loads the generated tier classes into the currently running Java VM running the AS and creates all the needed objects to create the replacement tier instance.

5) That CM puts the AS on hold; i.e., makes the latter transit to a "pause" state.

6) The same CM asks the tier instance to be replaced for the referee host tier pools keeping a reference to it.

7) The same CM asks each referee host tier pool to replace the old reference with the one of the new tier instance.

8) The same CM deletes the old tier instance through the Java VM garbage collector.

9) The same CM restarts the AS; i.e., makes the latter transits to a "running" state.

Both runtime code generation and hot plugging of tier instances will introduce certain overhead to the overall performance of an AS. Here, the problem is the tradeoff between the AC objectives (SLO and self-managing policies) that an AS must constantly follow and the need of restructuring to better follow these objectives. We are planning benchmark analysis to better understand the overall impact.

5 CONCLUSIONS

This paper presents our approach to the realization of a self-modifiable architecture, for ASs specified and generated with ASSL. The latter supports specification of special ASSL actions allowing for AS evolution at runtime. Via a finite set of special ASSL statements - CHANGE, REMOVE, ADD and CREATE, such ASSL actions can prompt changes in the tiers and sub-tiers of an ASSL-specified AS. However, ASSL currently does not support these statements at the level of code generation, and thus, the ASs are generated without the ability to modify their structure at runtime. Moreover, ASSL-specified AEs can exchange ASSL messages in the form of ASSL tiers, which can be used to modify the runtime structure of an AS.

To allow for runtime modifications in the structure of an AS, we propose to add a special *runtime code generation* mechanism and a special *code management* mechanism to the architecture of the ASSL-generated ASs. These mechanisms allow for runtime code generation of ASSL specifications and hot plugging of the generated code. The structure granularity of the ASs and key features of the generated code help to ensure that the difficult task of hot plugging is straightforwardly achieved.

Future work is primarily concerned with further development of the proposed mechanisms and evaluation of the degree of complexity and computational overhead these mechanisms bring to the entire system.

It is our belief that allowing ASSL-generated ASs to evolve in structure at runtime will enable broad scale development of autonomic systems.

REFERENCES

Brogi, A., Canal, C., and Pimentel, E., 2006. On the semantics of software adaptation. In *Science of Computer Programming*, vol. 61(2). Elsevier.

Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P., 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In *IEEE Computer*. IEEE Computer Society Press.

Kitano, H., 1990. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. In *Complex Systems*, vol. 4(4).

McKinley, P. K., Sadjadi, S. M., Kasten E. P., Cheng, B. H. C., 2004. Composing Adaptive Software. In *IEEE Computer*, vol. 37 (7). IEEE Computer Society Press.

Murch, R., 2004. *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall.

Oreizy P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., Wolf, A. L., 1999. An Architecture-Based Approach to Self-Adaptive Software. In *IEEE Intelligent Systems*. IEEE Computer Society Press.

Truszkowski, W., Hinchey, M., Rash, J., Rouff, C., 2004. NASA's Swarm Missions: The Challenge of Building Autonomous Software. In *IT Professional*, vol. 6(5). IEEE Computer Society Press.

Vassev, E., 2008. *Towards a Framework for Specification and Code Generation of Autonomic Systems, PhD Thesis*. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.