Modeling Architectural Change:
Architectural scripting and its applications to reconfiguration

Mads Ingstrup
Aarhus University
Department of Computer Science
Aabogade 34, 8200 Aarhus, Denmark
ingstrup@cs.au.dk

Klaus Marius Hansen
Aarhus University/University of Iceland
Dunhaga 7, 107 Reykjavík, Iceland
kmh@hi.is

Abstract

We detail the notion of architectural scripting (ASL) as a way to model the dynamic aspects of runtime and deployment-time software architecture. This is complementary to the ability of architecture description languages to model architectures statically in that we define scripting operations to modify architectures at runtime.

The scripting operations have as verification of the approach been implemented in an interpreter bundle on the OSGi platform. This implementation is used in our self-management system for generating correct reconfiguration plans in a self-managed system.

1. Introduction

Past research in software architecture description has been prolific [8], but only in so far as the modeling of static aspects of software architecture is concerned. While this work has proved fruitful, there is a growing need for the ability model dynamic aspects of runtime architectures. This is evident in particular in the trends towards autonomic and ubiquitous computing.

First, an autonomic computing system depends on the ability to automatically reason about and modify its own state to adapt to changes in, for instance, the QoS requirements to that system or to changes in its execution environment. Existing research indicates that handling self management at the architectural level is appropriate in terms of level of abstraction and generality [6]. Therefore the ability to model, analyze and realize runtime architectural change is key to building autonomic systems.

Secondly, the trend towards ubiquitous computing implies more complex deployment environments due to significant increase in the heterogeneity of devices. Thus setting up a particular configuration of a complex distributed system and running it can benefit from tool support, much like compilation of source code has come to rely on tools to manage the build-process. Such tools to assist runtime configuration, however, must incorporate some model of the actions they perform, or help perform.

In this paper we address how to model the dynamic aspects of runtime software architecture. We propose to model, analyze and implement such changes within the deployment and C&C views with the notion of an architectural scripting language (henceforth ASL). In an ASL script, change is expressed as an array of discrete steps; each step is an invocation of an operation. Modeling runtime change to an architecture with the notion of a script has several advantages:

- A script can be analyzed independently of the particular system configuration it is executed on, to assess, for instance, whether it may introduce violations to a certain style.
- A script can be reused across different contexts. For instance in autonomic computing a reconfiguration script may capture a reusable solution to a specific problem.
- A script, as we will show, is operational enough to support direct implementation while at the same time being amenable to useful analyses.

In the remainder of this paper we present our contributions. We first turn to our model of architectural change.

2. Modeling architectural change

Since the modeling of architectural change has received relatively little attention it is worthwhile to be clear about the the model we propose. We first introduce an abstract description.
2.1. Ontology and Model

Operations operate on architectural configurations which are expressed in the set of concepts shown in figure 1. A Device is physical or virtual (VM) device. A Component is unit of deployment, that is a package of executable code with explicit dependencies. Typically a binary file. A component may export modules that other components can require. Components are deployed to devices. They can provide services. Provided services can be instantiated form the components that provide them. A Module is a typed library, class, API etc. that a component may require or provide. A Service is a typed unit of runtime software with explicit dependencies in the form of required interfaces, and explicit capabilities in the form of provided interfaces. An Interface is a typed unit of association between services. In many cases implemented as an object reference (required) or object (provided).

Formally, the execution of an ASL script can be modeled as a trace. An ASL script is a list of ASL operations with specific values as parameters. A script operates on a start configuration when it is executed. After the first operation, say start_device(), this configuration is modified by having a new active device; as such we have a new configuration after each step in the script execution. The trace of a script execution is a list of such intermediate configurations, and the transition between two adjacent configurations is an invocation of an ASL operation. That is, the call of an ASL operation with specific values assigned to its parameters and to the script-properties it reads.

Definition 1  An architectural trace is a sequence of architectural configurations, $S_0, S_1, ..., S_n$, where each pair $S_i, S_{i+1}$ fulfills an architectural operation predicate.

Definition 2  An architectural script is a sequence, $O_0, O_1, ..., O_n$, of architectural operations.

A configuration should be thought of as a set of related architectural elements (in the sense of, e.g., [1]) and an architectural operation transforms one configuration into another through, e.g., deployment of a component. We have precisely modeled architectural operations using the Alloy modeling language [4]. In this way, proposed scripts can be verified for preservations of constraints such as architectural styles. The details are out of the scope of this paper, but can be found in Ingstrup and Hansen [3].

2.2. Practical Considerations

Having introduced the largely abstract model of architectural scripting, this section discusses a few practical issues that must be addressed in an implementation of the ASL operations.

Component Model Support. The realization of the ASL operations on a particular platform requires that platform to have a notion of deployment, instantiation and binding of components. In the case of OSGi¹, the notion of a bundle which an be installed/uninstalled and started/stopped supports that.

Transactional Script Execution. When running a script that modifies a system’s runtime architecture, it is important to minimize the risk that the system will be left in an inoperative or otherwise undesirable state. Supporting atomic execution of a sequence of operations can help achieve that. For instance, if a service is updated by a sequence of deployment, unbinding and rebinding operations, then it is desirable that either the update is not done at all, or done in entirety.

Naming scheme for architectural entities in ASL. The naming scheme used for variables within ASL is decoupled from whatever naming scheme may be used in the system (or model) it operates on. When a variable is initialized it is resolved by the ASL interpreter to an entity in the architecture. We do not rely on the assumption that every architectural entity has a unique identifier. They must however all have unique identities, and be practically distinguishable from each other.

3. Case: Reconfiguration planning in a self-managed system

To describe how our model can be applied to analyze a practical problem, we consider reconfiguring a runtime architecture, a challenge we are exploring in the eu-funded Hydra project². The self-management design follows Kramer and Magee’s three layer model [6], and is described in more detail in [3].

We have implemented the operations making up the architectural scripting language (ASL) on top of the OSGi platform. Doing this was relatively straightforward because

¹http://www.osgi.org
²http://www.hydramiddleware.eu
the platform is already dynamic and has a reasonably expressive component model that was easily mapped to the ontology of ASL (figure 1).

To do reconfiguration, the system must, first, select a target configuration of the system and, second, bring the system into that state. The first problem must allocate resources based to achieve a goal expressed by a high level policy. In Hydra this task is accomplished with genetic algorithms as described by Zhang and Hansen [12]. The second problem is to plan an appropriate sequence of operations that can will bring the system from its current state to the target state chosen by the optimization component.

Planning is a mature research topic within AI, and some algorithms now perform well enough to be applied to practical problems [11]. Many planning problems can be expressed in the Planning Domain Definition Language (PDDL) language [7]. A PDDL specification consists of a domain specification and of a problem specification. A domain specifies the terms in which planning problems in a particular application domain are modeled in the form of predicates. It also specifies which actions are possible in that application domain. An action is specified in terms of its pre-conditions and effects. The problem specification expresses an initial state and a goal state using the predicates defined in the domain specification. The planning problem is then to select a sequence of operations that will transform the initial state to the goal state.

A simple domain for deployment would declare predicates to designate that an object is a component, a device or a package, along with predicates for modeling their relative state, e.g., At(device, component) to say that a given component is deployed to a given device. In our example components can be deployed to and undeployed from devices. The operation deploy_component is expressed as follows:

```prolog
(action DEPLOY
  :parameters (?d ?c)
  :precondition (and (Component ?c) (Device ?d) (initiated ?d) (not (At ?d ?c))))
)
```

Listing 1. Action definition for the deploy operation.

This definition of the deploy_component operation requires, first, that the entities passed to the operation are of the correct type.

```prolog
(define (problem pb1)
  (:domain adl)
  (:objects C1 C2 D1 client server csint P1)
  (:init
    (:types
      Component C1 C2 Device D1
      Service client server
cinter
dime

Listing 2. A simple problem instance in the ASL domain.

In this planning problem instance we have as start configuration a running device with one component deployed to it. Facts that are not defined as true are false, so since only C1 is stated to be on D1, no other components or services are available at it. Both client and server services are available in the configuration being provided by components C1 and C2 respectively. The component C2 is not stated as deployed on D1 initially, but it is declared in the configuration. This means it is available for the ASL interpreter to be installed from a file in the file system of the ASL interpreters host device.

The goal in this problem instance is that the client service is bound to the server service. The preconditions and effects on the ASL operations means that the two services has to be bound with bind_interfaces. That, in turn, can only be done if the two services are running, so they have to be started with the start_services operation first. To generate a plan we have used the IPP planner [5]. When it is executed on the above problem instance it generates the following plan/asl script:

```prolog
start_service(D1 C2 server)
deploy_component(D1 C1)
start_service(D1 C1 client)
bind_interfaces(client server csint)
```

We leave a rigorous experimental performance analysis to future work, as our focus here is on modeling. However it is worth noting that in all cases we have tried the performance was satisfactory, with a correct plan being generated in less than 10 ms on a MacBook Pro with 2.33 GHz Intel Core 2 Duo processor. The performance of planning does not appear to be a problem in practise. We suspect that even for large systems most problem instances will be small, because many changes are local and only affect a few architectural entities. For instance, in pervasive computing as we are exploring in Hydra, many changes are local to a small or embedded device. Further, many wide-ranging changes, e.g., changing a security setting on all connectors in a system will be of the same kind, so that instead of a large planning
problem these planning for these changes can be characterized as many instances of a few generic planning problems.

4. Related Work

Although modeling the dynamic aspects of software architectures has not received the same amount of attention as the static aspects, there have been some sporadic but significant contributions to the area.

One branch of related work concerned with capturing changes to architecture is in the area of configuration management. Mae [10] is representative of this. The difference between this work and our own is that we model change as a series of operations rather than as a sequence of configurations. This is significant because it allows the change to be described and analyzed independently of a specific architecture, thus yielding the advantages listed in the introduction.

The other significant branch of work related to our own concerns implementation of reconfiguration. Both Oreizy et al. [9] and Hofmeister et al. [2] analyze the problem of runtime architectural change thoroughly, but do not provide a model or facility for reasoning about particular changes. Their analysis elicits several requirements a platform must conform to in order to support dynamic reconfiguration. Some of these can be realized by the choice of implementation platform, such as OSGi in our two cases, but their work remains a good complement to the work presented here, and is recommended for anyone who wishes to implement the architectural scripting operations we have described. Hofmeister et al. [2] mentions a reconfiguration language, but it is used as an implementation construct, rather than a tool for modeling and analysis as in our work.

5. Future work and Conclusion

Although we believe the set of operations chosen is expressive enough to specify most reconfigurations, some extensions may be useful, such as for component replacement which Oreizy et al.[9] describe as component removal followed by component addition, but with the additional feature of state-transfer between the removed component and its replacement.

Concerning reconfiguration planning, an interesting problem remains in how to modify the planning process in such a way that a style description can be taken into account to ensure that the generated script will not lead to style violations.

In conclusion, we have introduced the notion of architectural scripting and begun to explore both its theoretical and practical utility, in the hope that it will enable asking and answering new and interesting questions about the dynamic aspects of software architectures.

Acknowledgements

This work was partly inspired by the term “architectural scripting language” which was mentioned by Eric Dashofy in an informal chat, but we had no further occasion to discuss the meaning of the term. Thanks also to João Fernandes for giving a tutorial on the Equinox OSGi implementation, and to Weishan Zhang for commenting on an earlier draft of this paper. The work has been supported by the Hydra EU project (IST-2005-034891).

References