

Modelling System Behaviour by Abstract State Machines

Artur Andrzejak

ZIB Berlin

Takustrasse 7, D-14195 Berlin, Germany

andrzejak@zib.de

ABSTRACT

Using a language derived from the theory of Abstract State Machines, we are able to concisely describe both the actual and desired behaviour of systems and their components. The resulting specifications may be utilized in a variety of ways: for modelling and monitoring actual system state; as an executable description of intended behaviour; for static verification and testing of this behaviour; to evaluate a running system against a model; and as input for an automatic decision system that plans and adapts system actions according to states in the model. In addition to describing complete programs, the language (AsmL) can also indicate actions or workflows by specifying only the pre and post conditions, leaving room for adaptive decision making. By integrating specification, executable code and system state monitoring into a single model, we provide a framework for self-managing behaviours in devices and components at different scales.

1. INTRODUCTION

Modelling and monitoring system attributes, system state space, possible behaviours and desired/undesired conditions is essential to automatic system management. While a subset of these capabilities may suffice for most systems, the lack of a framework and accompanying tools for modelling, implementing, and integrating system behaviours may severely hamper the development of the system, due to reduced flexibility in modelling and restricted support for verification and testing of the implementation. Because developing such a framework is immensely challenging in itself, most solutions to automatic system management are proprietary, with unnecessarily high implementation costs and low reliability.

One of the essential features of such modelling frameworks is the ability to specify relevant system features. We must be able to describe:

- A. hierarchies of system components/resources,
- B. attributes of components/resources,
- C. constraints defining legal system states,

- D. conditional actions and their compositions,
- E. effects of executing actions on the system state,
- F. desired system states and optimization goals.

The model specification should be complemented by a set of tools that facilitate tasks such as static verification of system behaviour against imposed constraints, monitoring of running systems and mapping of system state into the model, checking the validity of the current state and executing the behavioural specification as a program (the last feature is particularly interesting for workflows, which are usually “programs” of limited complexity). Additional tools from the kernel of Autonomic Computing such Constraint Satisfaction Problem Solvers and heuristic decision engines (e.g. using Bayesian Networks or rule systems) should also be included. Such tools are designed to automatically find suitable sequences of actions in order to bring a system into a desired state, move it out of a forbidden state, and optimize the system for specific targets.

In this paper we evaluate the applicability of AsmL [1], a specification language developed at Microsoft Research, as the basis for a framework of the type described above. AsmL is derived from the theory of Abstract State Machines, in which each step in the machine computes a set of updates to the machine's variables. AsmL has a rich set of data structures, the ability to execute, integration into the .NET framework (including .NET reflection) and existing test and conformance tools, yet it is also relatively simple. The language is fully object-oriented as well, and it provides complex data types such as sets, finite mappings, sequences and structures. AsmL can be translated into C# and executed on the .NET platform (letting programs interoperate with external software). Reflection in .NET allows an AsmL specification to be translated into input for external tools (such as Constraint Satisfaction Problem Solvers) at runtime, depending on the system state. Microsoft also provides test generator and conformance checker tools, which let developers to test specific actions as well as verify system state against the AsmL model at runtime.

This paper proposes an Object-Oriented extension to AsmL. The extension includes specification conventions for Object-Oriented systems as well as additional interfaces to external OO tools for supporting adaptive (system state-dependent) decision making.

2. RELATED WORK

Scientific and business workflows have received recently a lot of attention in the web services and the Grid computing communities. Among the emerging standards for business process execution languages BPEL4WS [2] is the most popular one. Triana [7] is an example for an environment which employs an own XML-based language as well as BPEL4WS for managing serial and distributed scientific workflows. These languages are more specialized for business and/or scientific processes than the presented approach. Further, they do not allow pre- and postconditions, focusing on description of actions.

Policies have been widely used in network management [3]. They govern functions as firewalling, encryption or proxy-caching. A policy-based approach for automatic configuration of resources in data centers is presented in [5].

Specification languages for software design such as VDM or Z have been widely studied but are rarely used in practice. More popularity has gained the design by contract approach [4]. It applies the pre- and post-conditions to enforce correctness of interfaces and component behaviour.

3. MODELLING APPROACH

We take advantage of the built-in OO capabilities of AsmL and express resources (hardware, software) and actions or workflows as classes. Modelling the system is thus conceptually similar to object-oriented design. This similarity is illustrated by the following example of a system that consists of a single host and two dependent actions: installation of Java software and installation of the Eclipse IDE.

3.1 MODELLING THE RESOURCES

We model hierarchies of resource types as classes derived from the abstract class Resource, and actual resources as instances of these types. The attributes of a resource are represented by the member variables of the respective object. The state of a resource is the value of the variable. This approach satisfies requirements A and B from the introduction. Requirement C is covered by AsmL's ability to constrain the set of allowed resource states by the keyword constraint. The following figure shows the AsmL for our example:

```

abstract class Resource

class Software extends Resource
  var name as String
  var version as String
  var targetOS as Set of String
  var hdRequired as Integer // in MByte
  constraint Size(targetOS) > 0

class Host extends Resource
  var installedOS as Set of String
  var installedSoftware as Set of Software
  var RAM as Integer // in MByte
  var freeHd as Integer // in MByte
  constraint Size(installedOS) > 0

```

3.2 MODELLING BEHAVIOUR

Actions (atomic or composed) are associated with resources they act upon. Technically, each action is instantiated as an object which holds references to the objects of corresponding resources. Each action has two mandatory methods: pre() and post(). The former one specifies the preconditions necessary for an action to execute (and returns true iff they are satisfied), and the latter method describes the conditions that should hold after the execution.

We employ two kinds of action specification: *implicit actions* (discussed below) are completely described by their pre() and post() conditions, while *explicit actions* are additionally required to implement a method execute() that contains a detailed “program” for achieving a desired state (i.e. one allowed by post()). Thus pre() and execute() together satisfy requirement D above, while post() fulfills requirement E.

The following AsmL code shows an explicit action that installs Java on a host. The keyword **step** enforces sequential execution and indicates that all statements in a **step**-block can be evaluated in parallel but must be finished at the end of the block.

```

interface ImplicitAction
  pre() as Boolean
  post() as Boolean

interface ExplicitAction extends ImplicitAction
  execute() as Boolean

class InstallJava implements ExplicitAction
  var host as Host
  java as Software
  var sysHandle as RealSystemHandle

  pre() as Boolean
  return(
    (exists OS in host.installedOS
     where OS in java.targetOS)
    and (java.hdRequired < host.freeHd))

  post() as Boolean
  return (exists soft in host.installedSoftware
         where
           soft.name eq java.name and
           soft.version >= java.version)

  execute() as Boolean

```

```

step
  if post() or (not pre()) then
    skip
    WriteLine ("InstallJava not executed")
  else
    if sysHandle.reallyDo then
      step
        sysHandle.issueCommand(host,
          "javaInstall.bat")
      step
        sysHandle.updateModel(host)
    else
      add java to host.installedSoftware
  step
    return post()

```

The inclusion of `sysHandle.reallyDo` demonstrates how the specification and “implementation” can be combined in a single file. If the value of this Boolean variable is **true** (implementation mode), then an external Java installation script is executed (by simply calling the installer), and the attribute variables of the host model `host` are updated (after the script, because of **step**) according to the state of the real host. In the other mode (specification or modelling), a reference to `java` is simply added to the variable `host.installedSoftware`, which records the set of software installed on the host. The class `RealSystemHandle` is responsible for the communication with the “real world” and is simplified in this example:

```

class RealSystemHandle
  var reallyDo as Boolean
  issueCommand(res as Resource, scriptName as
    String)
  updateModel(res as Resource)

```

The second explicit action in our example attempts to install the Eclipse IDE on the host. The precondition of this action is that Java is already installed. Here we employ the convention that each “atomic” constraint in `pre()` and `post()` is formulated as a separate method named `pre_*` or `post_*`. This is helpful in debugging a workflow execution, and it also facilitates the extraction of the pre and postconditions of an action as input for external tools. For this purpose we use .NET reflection to discover methods with the `pre_` and `post_` prefixes, then parse our specification source code to extract the discovered constraints.

```

class InstallEclipse implements ExplicitAction
  var host as Host
  eclipse as Software
  var sysHandle as RealSystemHandle

  pre() as Boolean
    return (pre_targetOS() and pre_enoughHd()
      and pre_javaVersion())

  pre_targetOS() as Boolean
    return (exists OS in host.installedOS
      where OS in eclipse.targetOS)

  pre_enoughHd() as Boolean

```

```

    return (eclipse.hdRequired < host.freeHd)

  pre_javaVersion() as Boolean
    return (exists soft in host.installedSoftware
      where soft.name eq "java" and
        soft.version >= "1.4.3")

  post() as Boolean
    return (post_eclipseInstalled())

  post_eclipseInstalled() as Boolean
    return (exists soft in host.installedSoftware
      where soft.name eq eclipse.name and
        soft.version >= eclipse.version)

  execute() as Boolean
    step
      if post() or (not pre()) then
        skip
        WriteLine ("InstallEclipse not executed")
      else
        if sysHandle.reallyDo then
          step
            sysHandle.issueCommand(host,
              "eclipseInstallation.bat")
          step
            sysHandle.updateModel(host)
        else
          step
            add eclipse to host.installedSoftware
    step
      return post()

```

The defined actions and resources can be then used in the following code fragment which instantiates the resources (`host`, `Java` and `Eclipse` objects) and attempts to execute both installation steps:

```

Main()
  sysHandle = new RealSystemHandle(false)
  // create a host instance and initialize
  var myComputer = new Host({"WinXP"}, {}, 512,
    15000)
  // create software instances as constants
  java = new Software("java", "1.4.3", {"linux",
    "Solaris", "HPUX", "Win2000", "WinXP"}, 80)
  eclipseIDE = new Software("eclipse", "3.0",
    {"linux", "Solaris", "HPUX", "Win2000",
    "WinXP"}, 300)
  step
    var installJava = new InstallJava(
      myComputer, java, sysHandle)
    installJava.execute()
  step
    var installEclipse = new InstallEclipse(
      myComputer, eclipseIDE, sysHandle)
    installEclipse.execute()
  step
    if installEclipse.post() then
      WriteLine ("Eclipse installed.")
    else
      WriteLine ("Eclipse installation failed.")

```

3.3 IMPLICIT ACTIONS

The pre and postconditions are specified in terms of the attributes of involved resources, which comprise the sets of necessary and resulting system component states, respectively. From this perspective the states described by `post()` can be seen as a specification of the *goal* to be reached by an action, an implicit description of the desired action -- implicit,

because they lack the method `execute()`. In this way we partially satisfy requirement F (the optimization goal description is still missing).

Such implicit action specifications allocate space for implementing self-managing system behaviour and automatic planning of execution steps. In the first phase of a particular self-managing activity, the pre and postconditions of the available explicit actions are extracted (as described in Section 3.2) in order to construct a pool of executable “atomic” actions. In the next phase, the current system state and the desired goal state (the `post()` of an implicit action) are used as input to an external “reasoning” tool. A resource that recovers from execution failure of an action might serve as an example: here the current resource status is given by the real resource state, which is mapped to the model, and the `post()` of the failed action is the goal specification. In a successive phase the external tool computes the sequence of atomic actions, which are subsequently executed via .NET reflection. Of course, during the execution new failures can occur, which induces “recursive” application of the process described above.

3.4 TOOL SUPPORT

AsmL comes with several tools: an execution framework (compiler, debugger, Visual Studio IDE integration), a test case generator, and a conformance tester. In our context the most important component is the execution framework. Using this tool we are able to integrate the model specification and the implementation (`sysHandle.reallyDo` is **true**) within the `execute()` method. This allows us to avoid the separation of the specification from the program, eliminating a source of duplication, inconsistencies, and additional development effort.

The AsmL test generator supports test case-based model checking. After the ranges of the input values are specified and the relevant model attributes indicated by so-called “filters”, the tool generates a Finite State Machine and runs the model with random inputs from the specified ranges.

The conformance tester enables runtime verification of the implementation against the AsmL model. By binding model classes to implementation classes, execution steps of the implementation that do not adhere to the model are automatically detected. In the above example, we can run our code in the “model mode” (`sysHandle.reallyDo` is **false**) and another instance of the code in the “implementation mode” (which implies that the states of resources are derived from the real system). This lets us determine whether the real system behaves as expected. (However, this scenario can be currently used only for testing purposes, as the set of input values is restricted, and

an inconsistency causes a termination). In general, model classes can be bound to classes in any .NET assembly, which may be derived from a variety of implementation languages.

We are currently working on additional tools for supporting self-managing behaviour, including programs for extracting the constraints from `pre()` and `post()` and selecting them according to the model state, a framework for mapping the state of a real system to the model attributes, and a heuristic decision support engine.

4. CONCLUSION

Modelling system behaviour using AsmL allows us to take advantage of built-in data types, existing tools, and the interoperability of .NET. We have proposed a simple object-oriented extension for AsmL modelling that fulfills the specification language requirements enumerated in the introduction and offers enhanced tool support.

Our future work will focus on the self-management tools mentioned in Section 3.4, a graphical editor for designing AsmL-specified workflows, and real-world workflow examples such as server farm configuration [5] and protein threading in bioinformatics [6].

5. REFERENCES

1. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes: *Model-Based Testing with AsmL.NET*, 1st European Conference on Model-Driven Software Engineering, Dec. 11-12, 2003.
2. Business Process Execution Language for Web Services BPEL4WS, Version 1.1, <http://www.siebel.com/bpel>, 2003.
3. R. Haas, P. Droz, B. Stiller: *Autonomic service deployment in networks*, IBM Systems Journal, Vol. 42, No. 1, 2003.
4. Bertrand Meyer: *Eiffel: The Language. Object-Oriented Series*, Prentice Hall, New York, NY, 1992.
5. A. Sahai, S. Singhal, R. Joshi, and V. Machiraju: *Resource Management in Utility Computing Environments*, HP Technical Report, HPL-2003-176, Aug. 2003.
6. M. Shah, S. Passovets, D. Kim, K. Ellrott, L. Wang, I. Vokler, P. LoCascio, D. Xu, and Y. Xu: *A computational Pipeline for Protein Structure Prediction and Analysis at Genome Scale*, Third IEEE Symposium on Bioinformatics and BioEngineering (BIBE'03), March 10-12, 2003.
7. M. Shields and I. Taylor: *Programming Scientific and Distributed Workflow with Triana Services*, Global Grid Forum 10, Berlin 2004.