Journal of Computer and System Sciences 78 (2012) 939–959

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Ambient Abstract State Machines with applications ☆

Egon Börger *, Antonio Cisternino, Vincenzo Gervasi

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy

ARTICLE INFO

Article history:
Received 14 April 2010
Received in revised form 12 May 2011
Accepted 5 August 2011
Available online 12 August 2011

Keywords:
Ambient concept
Abstract State Machines
Naming disciplines
Memory sharing disciplines
Object-oriented design patterns
Mobile agents

ABSTRACT

We define a flexible abstract ambient concept which turned out to support current programming practice, in fact can be instantiated to apparently any environment paradigm in use in frameworks for distributed computing with heterogeneous components. For the sake of generality and to also support rigorous high-level system design practice we give the definition in terms of Abstract State Machines. We show the definition to uniformly capture the common static and dynamic disciplines for isolating states or concurrent behavior (e.g. handling of multiple threads for Java) as well as for sharing memory, patterns of object-oriented programming (e.g. for delegation, incremental refinement, encapsulation, views) and agent mobility.

1. Introduction

In [5] the first author has used the framework of Abstract State Machines (ASMs) to analyze the behavioral features of the object-oriented programming patterns proposed in [15]. This was intended as a first step towards understanding what genuine high-level model patterns could be defined which support what in [20] is called 'normal' high-level *system* design practices, and are not limited by the low-level view of object-oriented class and similar programming structures (which belong to 'normal' *program* design). In particular the parameterization of functions was used to represent the omnipresent binding or instantiation of methods and operations to given objects, which often are notationally suppressed because implicitly known from the context (as done so successfully in physics). The parameterization scheme can be expressed by the following equation:

$$this.f(x) = f(this, x) \quad \text{or} \quad f(x) = f(this, x)$$

This parameterization equation has a simple precise explanation in terms of the abstract states (Tarski structures) on which ASMs operate. This explanation sufficed to rigorously model in [5] the behavioral features of characteristic patterns from [15].

In a recent project we started an attempt to discover the pattern underlying the large number of different client-server architectures for concurrent (distributed) web applications. The goal is to make such a structure explicit by defining precise high-level models which can be refined to the major current implementations of WEB application architectures so that as a result their differences can be precisely analyzed, stated and hopefully evaluated and classified. Common to all WEB

☆ This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA). Part of the work of the first author was done when he was on a sabbatical leave, visiting the Computer Science Department of the ETH Zürich. The material has been presented by the first author to the Amir Pnueli Memorial Symposium at Courant Institute, NYU, New York, 7–9.5.2010.

* Corresponding author.
E-mail addresses: boerger@di.unipi.it (E. Börger), cisterni@di.unipi.it (A. Cisternino), gervasi@di.unipi.it (V. Gervasi).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.08.004

application architectures is the view of an application as a set of server components which communicate with client-side WEB browsers via data sent through the HTTP protocol. The state underlying a WEB application is distributed among the interacting components in the browser, the server and/or the application together with its application framework. A browser comes with agents managing multiple browsing contexts; parts of the state of interest reside in the document buffer of the renderer, in the state of the Javascript interpreter and in the DOM (Document Object Model). A WEB server may be designed to support the execution of programs belonging to a particular programming language, like the Java-based Tomcat server which features a modular architecture built around Java classes; but it may also support the runtime execution for programs written in different programming languages (like PHP or ASP, Python, JSF or ASP.NET) and coming from different libraries. Therefore a *simple yet general and flexible ambient concept* is needed to succinctly model the interaction of distributed components acting in heterogeneous environments.

This led us to further investigate the parameterization power the ASM framework offers and to use it for a definition of the needed ambient concept which generalizes the above parameterization equation. It turned out that the definition can be based upon the semantics of traditional ASMs without need to change or add to it. In this paper we define that concept and show that it allows one to uniformly express a variety of ambient concepts known from various domains and used there for modularization purposes. We illustrate the generality of the definition, which is largely due to the generality of the two concepts of ASM and of ASM refinement, by applying it to concrete examples in the following rather different domains:

- Static naming disciplines to isolate states, i.e. methods for binding names to environments as used in programming languages (reflecting notions like scope, module, package, library, etc.) and generally where name spaces play a role to define the meaning of names in given contexts. See Sections 3.1–3.2.
- Dynamic disciplines to isolate computations, reflecting notions of processes, executing agents, threads, etc. and their instantiations. In Section 3.3 we provide two typical examples:
  - Multi-Threading, illustrated by defining two example models, namely for:
    - a MultiThreadJavaInterpreter, where the definition starts from a given component SingleThreadJavaInterpreter,
    - the task management by the ThreadPoolExecutor in the Java 2 Standard Edition Version 5.0 (J2SE 5.0) [21], starting from scratch.
  - Process instantiation.
- Memory sharing disciplines, illustrated by a model for the *Visitor* pattern [15] in Section 3.4.
- Characteristic patterns of object-oriented programming. We illustrate this in Section 3.5 for four features with behavioral impact:
  - Delegation. The ambient notion allows us to define one pattern we call *Delegation* of which the well-known patterns *Template*, *Responsibility*, *Proxy*, *Strategy*, *State* and *Bridge* are instances.
  - Incremental refinement, also called conservative extension, illustrated by the *Decorator* pattern.
  - Encapsulation, illustrated by the *Memento* pattern.
  - Views, illustrated by the *Publish-Subscribe* pattern.

  Through this analysis it becomes explicit that some of these patterns, which are treated in the literature as distinct from each other, instead share the same or a strikingly similar form of their parameterization equations and have underlying class structures which are variations of a common scheme (a sort of 'structural pattern'). This reflects that the underlying semantical meaning of the parameterization (namely the implicit instantiation of a machine) is the same; what differs is the specific intentions pursued when using these parameterizations in programming, intentions which determine the small variations of the involved class structure.

  We expect that this approach to pattern analysis will be developed further to lift programming patterns to a body of design patterns which are focused on high-level model behavior and independent of specific syntactic (in particular programming language) representations.
- Mobile agents with moving ambients. We exemplify this in Section 3.6 by a succinct formulation of Cardelli's and Gordon's calculus of mobile agents by three simple ASM rules describing the fundamental operations ambient *Entry*, *Exit* and *Opening*.

We provide the definition of ambient ASMs in Section 2 and illustrate it in Section 3 by the above listed application examples, where the accent is on the diversity of the domains and the simplicity and uniformity of the applications.[1]

Since ASMs are well-known and have been extensively described and used in the literature over the last 25 years we do not repeat their definition here, also because the definition in Section 2.2 is complete by itself and can be understood correctly interpreting the occurring constructs as pseudo-code. We refer those who want to check the technical details to the recursive definition in the textbook [7, Table 2.2].

[1] Since the goal of this paper is to develop a general, uniform, succinct and simple notation practitioners can use with advantage above all in high-level system design, the reader will find a definition and its experimental application to a variety of non-trivial examples, but no theorem. This reduction is also the reason why there is no connection at all to the sequential ASM thesis and its proof from three natural postulates one reviewer wants us to mention (referring to the textbook version in [7, Chapter 7.2]). We are quite satisfied that there was no need to extend basic ASMs; we are here concerned only with an expressivity problem.