

Ambient Abstract State Machines with Applications

Egon Börger and Antonio Cisternino and Vincenzo Gervasi

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
{boerger,cisterni,gervasi}@di.unipi.it

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

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:

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

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].

¹ 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.

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 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 Sect. 3.1-3.2.
- Dynamic disciplines to isolate computations, reflecting notions of processes, executing agents, threads, etc. and their instantiations. In Sect. 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 Sect. 3.4.
- Characteristic patterns of object-oriented programming. We illustrate this in Sect. 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 focussed on high-level model behavior and independent of specific syntactic (in particular programming language) representations.

- Mobile agents with moving ambients. We exemplify this in Sect. 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 Sect. 2 and illustrate it in Sect. 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.²

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 Sect. 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].

² 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, Ch.7.2]). We are quite satisfied that there was no need to extend basic ASMs; we are here concerned only with an expressivity problem.

2 Definition of Ambient ASMs

We start in Sect. 2.1 with some small programming examples to explain the problem and the simple idea leading to the definition in the ASM framework. Sect. 2.2 contains the details of the definition, which consists in a simple and transparent translation of machines of form **amb** *exp* **in** *M* to traditional ASMs and thus avoids to change or add anything to the semantics of basic ASMs.

2.1 Two Small Problem Examples

We start with two motivating examples.

```
Example1 =  
  amb  $a_1$  in  
     $x := 3$   
  amb  $a_2$  in  
     $y := x$ 
```

We want the execution of this example to result in $x(a_1) = 3$ and $y(a_2) = x(a_2)$, where $x(a_2)$ may be different from $x(a_1)$. That is, the innermost ambient declaration should count for determining the environment where an expression is evaluated. Therefore we transform *Example1* as follows into *Example1**, using a logical variable *curamb* which we allow to be bound again within the scope of a **let**. The definition of the semantics of the **let** construct for ASMs, which is in accordance with its usual meaning, guarantees that for each occurrence of *curamb* in the scope of a **let** *curamb* = ..., the innermost enclosing **let** *curamb* = ... determines the value of an occurrence of *curamb* (see the detailed explanations in Sect. 2 or [7]).

```
Example1* =  
  let curamb =  $a_1$  in  
     $x(\textit{curamb}) := 3$   
  let curamb =  $a_2$  in  
     $y(\textit{curamb}) := x(\textit{curamb})$ 
```

We also want to have a way to express explicitly an ambient where to evaluate an expression. We use the usual dot-notation *exp.t* to denote that *t* is evaluated in ambient *exp*.

```
Example2 =  
  amb  $a_1$  in  
     $x := 3$   
  amb  $a_2$  in  
     $y := \textit{parent}(a_2).x$ 
```

The execution of this example, where a separately defined ambient independent function *parent* is used to explicitly describe the desired ambient to evaluate *x*, should result in $x(a_1) = 3$ and $y(a_2) = x(a_1) = 3$. *parent* reflects the nesting of occurrences of **let** *curamb* = ... in the program text. We therefore transform *Example2* into *Example2** as follows, where $parent(a_2) = a_1$.

$$\begin{aligned}
 \textit{Example2}^* = & \\
 & \mathbf{let} \textit{ curamb} = a_1 \mathbf{in} \\
 & \quad x(\textit{curamb}) := 3 \\
 & \mathbf{let} \textit{ curamb} = a_2 \mathbf{in} \\
 & \quad y(\textit{curamb}) := x(\textit{parent}(a_2))
 \end{aligned}$$

Thus, the idea is to define the meaning of **amb** *exp* **in** *M* to be (roughly) **let** *curamb* = *exp* **in** *M*. For a formal definition one can follow the inductive scheme used in [7, Table 2.2] to define the semantics for basic ASMs. The details are given in the next section.

2.2 Definition

We call *ambient ASM* each ASM which can be obtained starting from basic ASMs (formally speaking the ones defined by e.g. the inductive definition in [7]) by allowing for given machines *M* also a machine of the following form:

$$\mathbf{amb} \textit{ exp} \mathbf{in} \textit{ M}$$

Thus syntactically ambient ASMs are usual ASMs where also machines of the form **amb** *exp* **in** *M* are allowed. The semantics of the new clause **amb** *exp* **in** *M* for ambient ASMs can be defined by using the **let** construct, as we do below, to bind *exp* to a logical variable *curamb*, which denotes the ambient in which *M* is executed. The reuse of *curamb* for binding nested ambient expressions will allow us to succinctly describe various scoping disciplines in a uniform way, see the discussion below.

We extend the ASM classification of functions and locations by considering an extension of static functions and locations, whose values for given arguments do not depend on any state, to *ambient independent* functions or locations, i.e. static or dynamic functions or locations whose values for given arguments do not depend on any ambient.

We want to use the dot-notation $s_1.s_2 \dots s_m.t$ where each s_i stands for an ambient expression and t for a term $f(t_1, \dots, t_n)$ describing locations $(f, (v_1, \dots, v_n))$ in the sense used in the ASM framework. A location (of an ASM *M*) is a pair $(name, args)$ of a *name* (belonging to the signature of *M*) and a sequence *args* of elements (belonging to the domain of *M*) and represents an abstract memory ‘location’ parameterized by *args* where values can be stored. To this purpose we extend the inductive definition of the set of terms (expressions) by declaring the dot symbol to not be a location symbol and by allowing dot-terms $s.t$ as terms if the following two conditions hold:

- *s* is a term

- t is a term of form $f(t_1, \dots, t_n)$ and f is a *location symbol*.

We define now the semantics of ambient ASMs by translating them into basic ASMs. The definition follows the inductive scheme used in [7, Table 2.2]; unless otherwise stated f denotes a location symbol.

Term translation ($t \rightarrow t^*$) For the transformation of terms different from dot-terms we stipulate the following, included the case of variables where $n = 0$. To guarantee that each term is evaluated in the current ambient *curamb*, we add to each location an additional argument for the ambient in which the location is evaluated. This is analogous to the object-oriented notation *this.exp* to denote the evaluation result of *exp* for the instance denoted by *this*.

- If f is a location symbol we define:

$$f(t_1, \dots, t_n)^* = f(\text{curamb}, t_1^*, \dots, t_n^*)$$

- If f is a logical variable³ or a rule name or an ambient independent function symbol, we define:

$$f(t_1, \dots, t_n)^* = f(t_1^*, \dots, t_n^*)$$

- If $t_1 \dots t_m.f(s_1, \dots, s_n)$ is a dot-term we define:

$$(t_1 \dots t_m.f(s_1, \dots, s_n))^* = f(t_1^*, \dots, t_m^*, s_1^*, \dots, s_n^*)$$

Rule translation The crucial steps for the transformation of ASMs are the following two. An assignment to a location becomes an assignment to this location in the current ambient, where all the terms involved are evaluated in this *curamb*.

$$(f(s_1, \dots, s_n) := t)^* = (f(\text{curamb}, s_1^*, \dots, s_n^*) := t^*)$$

The execution of (a step of) a machine P in a given state S with a desired ambient t is defined as execution of (a step of) the transformed machine P^* in this state with ambient value calculated as the value v of the defining expression t in state S .

$$(\mathbf{amb} \ t \ \mathbf{in} \ P)^* = (\mathbf{let} \ \text{curamb} = t^* \ \mathbf{in} \ P^*)$$

This definition implies that the ambient expression t is passed in the transformation of $\mathbf{amb} \ t \ \mathbf{in} \ P$ by value, whereas the above definition for the transformation $(t.s)^*$ of a dot-term guarantees that the explicit ambient expression t is passed in $(t.s)^*$ by name.

For an illustration of this definition consider the following machine.

³ Logical variables are bound by **let**, **choose** or **forall**.

$$\text{NESTEDAMB}(P) =$$

$$\mathbf{amb} \ t_1 \ \mathbf{in}$$

$$\mathbf{amb} \ t_2 \ \mathbf{in}$$

$$P$$

By the definition of **let** in [7, Table 2.2], a step of $\text{NESTEDAMB}(P)^*$ in state S yields the update set U under a given environment (interpretation of the free variables) env if a step of $(\mathbf{amb} \ t_2 \ \mathbf{in} \ P)^*$ yields U under the modified environment

$$env_1 = env[curamb \rightarrow val(t_1^*, S, env)]$$

which is the case if a step of P^* yields U under the modified environment

$$env_2 = env_1[curamb \rightarrow val(t_2^*, S, env_1)].$$

So when P^* computes a step, it does it with the last computed value for $curamb$, here $val(t_2^*, S, env_1)$ which typically depends on its being nested in the scope of the declaration of the ambient t_1 . For a concrete example see below the special case of the *State* pattern.

The rule name case $r(t_1, \dots, t_n)$ is covered by the definition for terms. The other steps follow the inductive scheme used in [7, Table 2.2].

$$\mathbf{skip}^* = \mathbf{skip}$$

$$(P \ \mathbf{par} \ Q)^* = (P^* \ \mathbf{par} \ Q^*)$$

$$(\mathbf{let} \ x = t \ \mathbf{in} \ P)^* = (\mathbf{let} \ x = t^* \ \mathbf{in} \ P^*)$$

$$(\mathbf{if} \ \phi \ \mathbf{then} \ P \ \mathbf{else} \ Q)^* = \mathbf{if} \ \phi^* \ \mathbf{then} \ P^* \ \mathbf{else} \ Q^*$$

$$(\mathbf{choose} \ x \ \mathbf{with} \ \phi \ \mathbf{do} \ P)^* = \mathbf{choose} \ x \ \mathbf{with} \ \phi^* \ \mathbf{do} \ P^*$$

$$(\mathbf{forall} \ x \ \mathbf{with} \ \phi \ \mathbf{do} \ P)^* = \mathbf{forall} \ x \ \mathbf{with} \ \phi^* \ \mathbf{do} \ P^*$$

$$(P \ \mathbf{seq} \ Q)^* = (P^* \ \mathbf{seq} \ Q^*)$$

Given this translation of ambient ASMs M into traditional ASMs M^* we often identify M and M^* without further mention.

Remark. Sometimes one has to deal at the same time with different types of ambients, like the declaration environment of process instances. One can support this notationally by writing in such cases

$$\mathbf{amb} \ (type) \ exp \ \mathbf{in} \ M$$

where *type* is a name denoting the kind of ambient one wants to consider. In this way one can distinguish for example the current declaration environment $\mathbf{amb} \ (env)$ of a process with a given current object $\mathbf{amb} \ (obj)$ from its current instance $\mathbf{amb} \ (inst)$ and its currently executing thread $\mathbf{amb} \ (thread)$.

3 Characteristic Applications

In this section we illustrate the use of ambient ASMs for some characteristic environment concepts as they show up in various domains.

3.1 Static Naming Disciplines

We start with a simple example from programming languages (see any book on programming, e.g. [23]). They typically come with disciplines to declare items of various types. A declaration has the effect to create an environment in which certain names are bound to specific values. One can easily describe this with the ambient construct

$$\mathbf{amb} \text{ } exp \text{ in } M$$

Here the evaluation of a declaration *expression* is assumed to yield an *environment* (imagine a hash table) which associates with each *identifier* declared in *exp* a *bindingValue*(*id*, *env*). This value is used to execute *M*, say via a *curValue* function which describes the current item values for this computation. The definition of *curValue* uses an auxiliary function *parent*, which is defined by the nesting structure of **amb** in the given program (of which **amb** *exp* **in** *M* is a part).

$$curValue(x, env) = \begin{cases} bindingValue(x, env) & \mathbf{if} \ x \text{ is declared in } exp \\ curValue(x, parent(env)) & \mathbf{else} \end{cases}$$

3.2 General Scoping

More generally, we can use the **amb** construct together with non-determinism and abstract functions to describe a high-level model of binding and scoping that can then be instantiated to the scoping disciplines of various languages.

We assume a set *Envs* including all the environments which bind a set of names to the corresponding values (including locations); environments are added to, removed from or altered in *Envs* by entering a new scope, leaving (destroying) a scope, or by declarations establishing new bindings in a scope. Notice that environments that are exited, but not destroyed, are preserved in *Envs*: this is the case, among others, of closures.

Then, to obtain the current value of an identifier *id*, referenced at lexical position *pos* and run-time state *s*⁴ the following machine can be used:

$$\begin{aligned} \text{EVALUATE}(id, pos, s) = \\ & \mathbf{choose} \ a \in \{e \in Env \mid inScope(e, pos, s) \wedge e.defines(id)\} \ \mathbf{in} \\ & \mathbf{amb} \ a \ \mathbf{in} \ \text{GETVALUE}(id) \end{aligned}$$

In this general model, the environment in which *id* is evaluated is any of those whose scope covers the current lexical and dynamic position in the program, provided they have a binding for *id*. The latter condition is expressed by

$$e.defines(id) \equiv (e.id \neq \mathbf{undef})$$

⁴ Here, *s* models the current dynamic state of the computation, e.g. the procedure, function or method call stack, the current instance if any in object-oriented languages, the executing thread in languages supporting thread-local storage, etc.

By reducing the non-determinism of **choose** and specifying *inScope*, one can refine the general model to the scoping policies of various languages.

For example, purely lexical scoping (as in Pascal, Modula-2, Ada, C) is refined from the general version by defining

$$inScope(e, pos, s) \equiv inLexicalScope(e, pos)$$

where *inLexicalScope*(*e*, *pos*) is a predicate covering all the positions in the source text of the program where environment *e* is in effect, and can be determined statically by the compiler. Similarly, purely dynamic scoping (as in Logo or certain variants of Lisp) is described by

$$inScope(e, pos, s) \equiv inDynamicScope(e, s)$$

where *inDynamicScope*(*e*, *s*) is a location of the state which is altered by an interpreter (of the language or of the executable code generated by a compiler) whenever a dynamic-state changing statement (e.g., a function invocation) is encountered. Languages that offer both scoping policies (e.g., Perl, Java) can be similarly modeled by a combination of *inLexicalScope* and *inDynamicScope*.

Most language also have *hiding/shadowing rules* to specify the behaviour when the same identifier is bound in multiple environments, all of which are in effect at the same static and dynamic position. Typical cases are when a method parameter has the same name as an instance or class (**static**) variable in Java, or when an identifier is re-declared as a local variable of an enclosed block in a given lexical scope.

For such languages, the non-determinism in the **choose** clause needs to be further specified. Most languages stipulate a last-in, first-out (LIFO) policy for scopes, where the most-proximal declaration (lexically or dynamically) hides previous bindings. This strategy could be expressed as

$$\begin{aligned} \text{EVALUATE}(id, pos, s) = \\ \text{let } E = \{e \in Env \mid inScope(e, pos, s) \wedge e.defines(id)\} \text{ in} \\ \text{choose } a \in E \text{ with } (\forall a' \in E, a \sqsubseteq a') \text{ in} \\ \text{amb } a \text{ in GETVALUE}(id) \end{aligned}$$

where \sqsubseteq provides the particular nesting order specified by the language.

A notable exception to this policy is the TCL language, where the programmer can explicitly refer to a binding established in a particular enclosing scope (and possibly hidden by a more proximal scope) by indexing. In TCL, *id* refers to the current (most recent) scope, as delimited by **procedures** or **namespaces**; the expression **global** *id* always refer to the global scope, and the declaration **upvar** *n id v* binds the local variable *v* to the identifier *id* as bound in the *n*-th scope "up" from the current scope (if the form **upvar** *#n id v* is used instead, then the statement refers to the *n*-th scope "down" from the global one).

This peculiar scoping policy allows TCL programmers to access arbitrary bindings⁵ of names. The generality of our **amb** construct allows arbitrary values

⁵ A common usage for this is to use **upvar** **1** *id v* to simulate call-by-reference, since then the local variable *v* is bound to a variable *id* in the immediate caller. The con-

to be used to represent ambients, and arbitrary structures to be built with them. Hence, we could stipulate that Env be a list of scopes, manipulated on entering and leaving a `proc` (hence, Env depends on s), and specify that for TCL,

```
EVALUATE( $id, pos, s$ ) =
  let  $a = ambref(pos, s)$  in
    amb  $a$  in GETVALUE( $id$ )
```

where

$$ambref(pos, s) = \begin{cases} Env[0] & \text{if } pos \text{ is in global } id \\ Env[n - k] & \text{if } pos \text{ is in upvar } k \text{ id } v \\ Env[k] & \text{if } pos \text{ is in upvar } \#k \text{ id } v \\ Env[n] & \text{otherwise} \end{cases}$$

with n being the size of Env . Notice that we can use pos , representing the lexical position where the id appears in the source code, to ascertain the current syntactic context, and that as expected, an unqualified id appearing in global code, a `global id` , and a `upvar $\#0$ id v` all refer to the same variable.

3.3 Disciplines to Dynamically Isolate Computations

In distributed computations strong mechanisms are needed to sufficiently isolate computations of different agents, as for example the execution of various tasks by multiple threads. We show here for two thread related examples how ambient ASMs support this goal in a clear and simple way. In Sect. 3.3 we build from a `SINGLETHREADJAVAINTERPRETER` and any given *scheduler* a `MULTITHREADJAVAINTERPRETER` controlled by that *scheduler*, separating in this way also thread scheduling from thread execution. In Sect. 3.3 we pursue this separation of handling the thread management from programming the application logic further by building a high-level model for the `THREADPOOLEXECUTOR` of J2SE 5.0.

MultiThreadJavaInterpreter Let `SINGLETHREADJAVAINTERPRETER` be a single-thread Java interpreter, for definiteness say the one defined in terms of ASMs in [29] from where we borrow the terminology. For simplicity of exposition let us assume for the moment that the underlying executing machine is a mono-core processor where at each time only one (a unique) current thread may run Java code; we explain below the little changes needed for the multi-core case. We want to separate the scheduling discipline from the thread management task, so that the construction can be used independently of the adopted particular scheduling algorithms (see [21, Ch.11] for the different scheduling methods available in J2SE 5.0 through the Scheduled Thread Pool Executor). Therefore

struct allows any index to be used — although, of course, doing so in an uncontrolled manner is considered rather poor taste.

we assume the scheduler to be given, say by a function *schedule* (to be implemented by a program computing this function) which selects one *Runnable* thread out of the current instances of the *Thread* class (in terms of the ASM model in [29] being a current instance means to be in the *heap*).

This leads us to the following definition, which generalizes and modularizes further the definition given in [29, 7.2.1] by abstracting from its specific treatment of thread context (see the explanations below). A thread is *Runnable* if it is either *Active* or *Synchronizing* or *Notified*. If it is *Synchronizing* or *Notified* and chosen for execution, it should first SYNCHRONIZE respectively WAKEUP, whereby it also becomes *Active*, before going to RUN. In a monocoore architecture only one thread can be the *lastSelectedThread* and *Active*, so that it can be associated with the SINGLETHREADJAVAINTERPRETER to continue the execution of the computation the thread is carrying around as its ambient, for an initially assigned program. This is described by the following ASM where the ambient construct is used to express what it means to RUN a thread.⁶

```

MULTITHREADJAVAINTERPRETER =
  let  $q = \text{schedule}(\{t \in \text{Thread} \mid \text{Runnable}(t)\})$ 
  if  $q = \text{lastSelectedThread}$  then
    (if Synchronizing( $q$ ) then SYNCHRONIZE( $q$ )
     if Notified( $q$ ) then WAKEUP( $q$ ))
    seq7 RUN( $q$ )
  else
    (Active( $q$ ) := true
     lastSelectedThread :=  $q$ )
    seq RUN( $q$ )
  where
    RUN( $q$ ) =
      if Active( $q$ ) and  $q = \text{lastSelectedThread}$  then
        amb  $q$  in SINGLETHREADJAVAINTERPRETER

```

This definition abstracts from the particular specification of thread contexts used in [29]. In case of rescheduling it avoids saving the context for the suspension of the *lastSelectedThread* and restoring the context for the newly scheduled thread⁸ since via the ambient construct each thread gets its context via *curamb* when called to RUN. The specification used in [29] now appears as one possible refinement of the thread ambient concept.

Under the mono-core assumption the thread suspension is achieved in the above MULTITHREADJAVAINTERPRETER by the guard of RUN, which requires the unique currently executing thread to be the *lastSelectedThread* and *Active* (not only *Synchronizing* or *Notified*, which makes threads only *Runnable*). In

⁶ All the concepts we use in this definition without further explanation are defined in [29] where the reader can check the details.

⁷ seq denotes the sequential execution of ASMs, see [7] for a definition.

⁸ In [29] the context appears as frame stack which is recorded into a *continuation* function and restored using a *switchCont* operation as part of the *run* macro.

the multi-core case the description can be simplified, since thread suspension may not be necessary any more each time a new thread is selected for execution. If upon scheduling a new thread this thread can simply be put to RUN without suspending other currently running threads (assuming a potentially unbounded number of running threads, see the THREADPOOLEXECUTOR in the next section with details for the more realistic case of a bounded number of available threads), one can define this by the following machine:

```

UNBOUNDEDTHREADJAVAINTERPRETER =
  let  $q = \text{schedule}(\{t \in \text{Thread} \mid \text{Runnable}(t)\})$ 
  if  $\text{Active}(q)$  then RUN( $q$ )
  else
    (if  $\text{Synchronizing}(q)$  then SYNCHRONIZE( $q$ )
     if  $\text{Notified}(q)$  then WAKEUP( $q$ ))
  seq RUN( $q$ )
  where
    RUN( $q$ ) = amb  $q$  in SINGLETHREADJAVAINTERPRETER

```

We deal in the next section with the case where the number of simultaneously running threads is bounded.

Thread Pool Executor The role of thread pools is to separate the formulation of thread management—details for the creation, the use and the deletion of threads to run tasks, including the control of the number of simultaneously running threads—from the description of the application logic of the to be programmed tasks, for conceptual clarity and for pragmatic reasons (e.g. possible throughput gains through time slicing, task creation overhead reduction, etc.). This separation of concerns is well supported by the ambient concept. For the sake of definiteness we illustrate this here by developing a high-level model for the thread pool executor of J2SE 5.0 following its informal description in [21, Ch.10]. Similar schemes can be described for example for web servers where arriving requests are managed by a pool of threads, etc.

The THREADPOOLEXECUTOR manages the assignment of threads to tasks which are entered for execution (a method we call TASKENTRY) and the decoupling of this association of a thread with a task upon the completion of the execution of the task (a method we call TASKCOMPLETION). If no thread can be assigned to a submitted task because the number $|\text{CreatedThread}|$ of the set of *CreatedThreads* is already the *maxPoolSize* number of threads and all of them are *Running*, then TASKENTRY inserts the task into a *queue*—if this can be done without exceeding the *maxQueueSize*; otherwise the task is REJECTED. If there are still threads to create, thread creation and task assignment takes place if no *Idle* thread is available and the task cannot be placed to the *queue* without blocking it. This privileges queue insertion with respect to the creation of a new thread via a predicate *BlockingFreePlaceable(task, queue)* which we leave abstract.

Coming naturally with its *queue*, THREADPOOLEXECUTOR also has a method (submachine) to assign under certain conditions a thread to a task from the *queue* to RUN it. In addition the J2SE 5.0 thread pool also supports a *corePoolSize* \leq *maxPoolSize* to keep the number of *CreatedThreads* as long as possible within *corePoolSize*, reassigning threads that have already been created but are currently *Idle*. In this spirit last but not least a thread exits the runtime set *CreatedThread* if it has been *Idle* for more than its *keepAliveTime* waiting to be assigned to a task in the *queue* which remained empty during this waiting period. We formulate the waiting behavior of a thread by a submachine TASKFROMQUEUEOREXIT

This description is modeled by the following definitions where we make crucial use of the ambient construct in the RUN macro definition. We omit the description of how the THREADPOOLEXECUTOR is called either for a *task* (to execute TASKENTRY) or for a (*task*, *thread*) pair (to execute TASKCOMPLETION) or for a *thread* (to execute TASKFROMQUEUEOREXIT).

```

THREADPOOLEXECUTOR =
  TASKENTRY
  TASKCOMPLETION
  TASKFROMQUEUEOREXIT

```

To describe the externally controlled submission of tasks for execution we use a monitored predicate *Enters* expressing the event that a task is submitted. We use the **new** (*S*) machine to provide for each call a fresh element and to place it into the set *S*. The submachine EXECUTE which we leave abstract is the ‘task interpreter’, similar to the SINGLETHREADJAVAINTERPRETER in Sect. 3.3. We also leave REJECT abstract.

```

TASKENTRY(task) = if Enters(task) then
  if | CreatedThread | < corePoolSize then
    let t = new (CreatedThread) in RUN(t, task)
    // first fill in corePoolSize many threads
  else
    if | CreatedThread | < maxPoolSize then // first use Idle threads
      if forsome t ∈ CreatedThread Idle(t) then
        choose t ∈ {t ∈ CreatedThread | Idle(t)} RUN(t, task)
      else
        if BlockingFreePlaceable(task, queue) then
          INSERT(task, queue) // first fill queue before creating threads
        else let t = new (CreatedThread) in RUN(t, task)
    else
      if forall t ∈ CreatedThread Running(t) then
        if | queue | < maxQueuesize then INSERT(task, queue)
        else REJECT(task)
where
  RUN(thread, task) =

```

```

program(thread) := task
amb task in EXECUTE
Running(thread) := true

```

To describe the externally controlled completion of task execution by a thread we use a monitored predicate *Completed*, which we assume without loss of generality to be preemptive. To leave the particular queue access policy open we use a not furthermore specified function *next* to determine the next to be chosen element from the *queue*.

```

TASKCOMPLETION(task, thread) =
if thread ∈ CreatedThread and Completed(task, thread)
and Running(thread) then
  if queue ≠ empty then RUNTASKFROMQUEUE(thread)
  else
    Idle(thread) := true
    completionTime(thread) := now
where
  RUNTASKFROMQUEUE(thread) =
    let task = next(queue)
    RUN(thread, task)
    DELETE(task, queue)

TASKFROMQUEUEOREXIT(thread) =
if Idle(thread) and thread ∈ CreatedThread then
  if now − completionTime(thread) ≤ keepAliveTime(thread)
  and queue ≠ empty
  then RUNTASKFROMQUEUE(thread)9
elseif | CreatedThread | > corePoolSize then
  DELETE(thread, CreatedThread)

```

Note that in the case of $keepAliveTime(thread) = 0$ and an empty *queue* with the number of created threads not exceeding the *corePoolSize*, the thread “blocks indefinitely waiting for a new task to be queued” and “runs the new task when available” [21, p.193], namely through the second clause of TASKENTRY.

Process Instances The instantiation of a process *P* by an executing agent **self** carrying its own environment, which has often been used in the literature when dealing with multi-agent ASMs, generalizes the use of Java threads in the previous section to arbitrary agents executing an instance of a given ASM in a concurrent context and corresponds to **amb self in P**. A well-known case is

⁹ The reader will notice that we do not reset *completionTime(thread)* to **undef**. This is not needed since a thread uses its *completionTime* only when attempting to perform the RUNTASKFROMQUEUE operation. But for this the thread has to be in *Idle* mode, and each time it enters the *Idle* mode its *completionTime* is set to the new value of *now*.

class and method instantiation in object-oriented programming, where $this.M(x)$ corresponds to **amb** $this$ **in** $M(x)$. Similarly the instantiation of S for execution on a given *server* can be described by **amb** *server* **in** S .

This implicit parameterization scheme provides a way to isolate executions of different M -instances, for example by defining for different host machines $host_1 \neq host_2$ separated instances **amb** $host_i$ **in** M for $i = 1, 2$. It also solves the problem (see [31]) to precisely but uniformly distinguish between different instances of a same business process model; it has been used for example in [6] to rigorously model process instantiation as proposed by the OMG standard [22].

3.4 Shared Memory

An ambient may expose memory an agent shares with another agent when executing its program M . We illustrate this by the *Visitor* pattern from [15] and by a small Request/Answer communication scheme where the receiver for ANSWERING a request can access some part of memory which is shared with the requestor. Another example is the *Publish-Subscriber* pattern described in Sect. 3.5.

Request/Answer with Shared Memory Imagine multiple senders s, s', \dots which send requests to a *mailBox* of a receiver r . The receiver for ANSWERING a request is supposed to share some part of the memory of the respective sender. The shared memory locations are assumed to be extractable by an *extractState* function applied to the sender of the request. Then one can formulate the MEMORYSHARINGRESPOND mechanism as follows. We use an abstract function *next* to determine the next message to be taken from the *mailBox* for responding.

```
MEMORYSHARINGRESPOND =
  let request = next(mailBox)
  amb extractState(sender(request)) in ANSWER
```

Visitor Pattern The idea of the *Visitor* pattern [15, pg.331] is to represent an operation on a concrete element not directly as a method of the class *ConcreteElement* it belongs to, but as a so-called ‘Visitor operation’ VISITCONCRELEM of another class *Visitor*. The execution of this operation is triggered by a concrete element ce through ‘ACCEPTING’ a *visitor* in whose class the operation VISITCONCRELEM(ce) is implemented. ce provides through ACCEPTING an appropriate access to its state for the *visitor* to execute the operation.

All classes *ConcreteElement* are subclasses of one class *Element*. *Visitor* provides for each of them an interface VISITCONCRELEM(*ConcreteElement*), each of which is refined in each subclass *ConcreteVisitor* of *Visitor* by a corresponding omonymous submachine. See Fig. 1.

The interface ACCEPT(*visitor*) is refined in *ConcreteElement* such that every concrete visited element **self** ‘supplies itself as an argument to this operation to

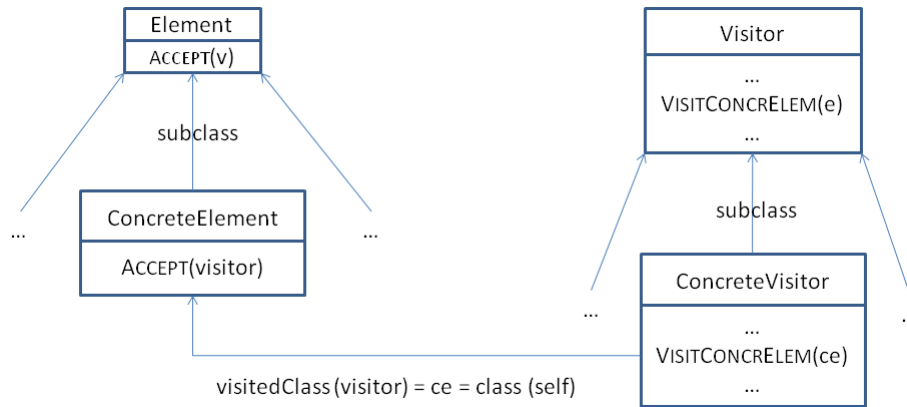


Fig. 1. Visitor Pattern Structure

let the visitor access its state, if necessary' [15, pg.335]. This can be expressed by the ambient construct as follows:¹⁰

$$\text{ACCEPT}(\text{visitor}) = \mathbf{amb} \text{ visitor in VISITCONCREELEM}(\mathbf{self})$$

The *visitor* parameter denotes an instance of a *ConcreteVisitor* class implementing VISITCONCREELEM. This is expressed by the following constraint:

$$\text{visitedClass}(\text{visitor}) = \text{class}(\mathbf{self}).$$

In this way if one wants to define a new operation on instances of *Element*, it can be done 'simply by adding a new visitor' which will trigger the new operation upon acceptance of the visitor. 'Adding a new visitor' means to define a new *Visitor* subclass *ConcreteVisitor* where each VISITCONCREELEM interface is implemented in the desired new way. As a result there is no need to change the *Element* subclass structure because every *ConcreteVisitor* instance, created by a client that uses the pattern, will have to 'traverse the object structure, visiting each element with the visitor' [15, pg.335].

3.5 Object-Oriented Patterns

In this section we illustrate applications of ambient ASMs to *uniformly* describe the behavior of some object-oriented patterns which appear in [15] as separate

¹⁰ In [15] it is supposed that 'the operation's name and signature identifies the class that sends the visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface' [15, pg.334]. To avoid having to deal with such naming conventions we use instead **self**, standing for the concrete to be visited element, to identify the relation between the classes to which the visiting and the visited object belong.

individual patterns. Where *interfaces* are mentioned, the reader may think about them as abstract machines or operations without an associated ASM rule (signature names), since the generality of the ASM refinement concept allows one to generalize the specific *implementations* referred to in [15] to ASM *refinements* [4].

Delegation Pattern In this section we illustrate the use of the ambient concept for a uniform description of what we call *Delegation* pattern behavior that is common to various object-oriented patterns in [15], which as we will see also share a common (in fact almost the same) class structure. The *Delegation* instances we show are known under the names *Template*, *Responsibility* (together with its deterministic instance *ChainOfResponsibility*), *Proxy*, *Strategy*, *State* and *Bridge*.

Delegation is used to decouple an interface OPERATION in an *AbstractClass* from its implementations such that at run-time upon a call of OPERATION an object *delegate* in an appropriate *classOf(delegate)* can be determined to carry out the call by executing the implementation provided in *classOf(delegate)*. How *delegate* is defined varies among *Delegation* instances and may also determine some variations of the underlying class structure as illustrated in Fig. 2. In the *Template*, *Responsibility*, *Strategy*, *State* and *Proxy* patterns the implementations of OPERATION are defined in subclasses *ConcreteClass* of *AbstractClass* instead of being outsourced as in *Bridge* to *Implementor* subclasses *ConcreteImplementor*.

The behavior of the *Delegation* pattern is expressed by the following ambient ASM DELEGATE which defines the delegation equation for calls of OPERATION for specific *Request* input:

$$\text{DELEGATE}(\text{OPERATION}, \text{delegate})(\text{Request}) = \\ \mathbf{amb} \text{ delegate in } \text{OPERATION}_{\text{classOf}(\text{delegate})}(\text{Request})$$

We now analyze how *delegate* is defined in DELEGATE instances. This happens either externally—this can be either statically, determined by the class structure (the *Template* pattern case) or by a data-structure related function (like the chain traversal function in the CHAINOFRESPONSIBILITY pattern), but also dynamically (e.g. via some run-time determined *selection* function as in the *Responsibility* pattern)—or internally by having *delegate* as a location in *AbstractClass* (like in the *Bridge* pattern) or in some other dedicated class (like in the *Proxy*, *Strategy* and *State* patterns).

Template Pattern For the *Template* pattern we read in [15, pg.325]:

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

This can be done by a refinement of DELEGATE(OPERATION, *delegate*), instantiating *delegate* statically to denote a subclass *ConcreteClass* (which implies

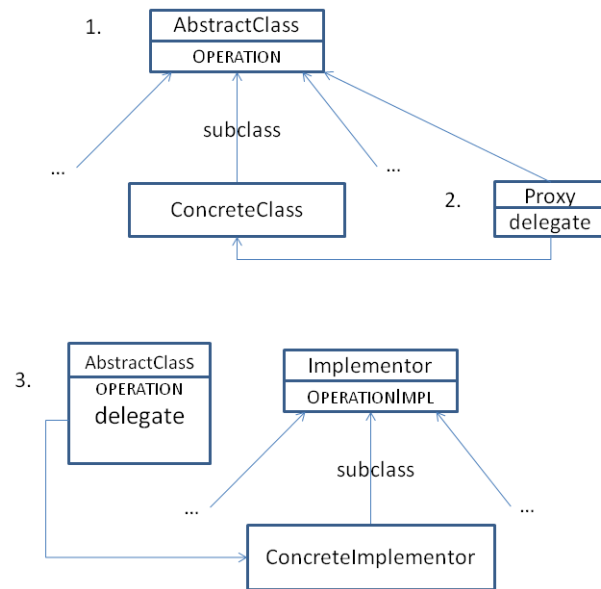


Fig. 2. Delegation Pattern Structure. 1. is for Template and Responsibility, 2. for Proxy, Strategy and State, 3. for Bridge

of course $classOf(delegate) = ConcreteClass$). The definition of `DELEGATE` has the following unfolding when applied to `TEMPLATEMETHOD`¹¹ and `ConcreteClass`:

`DELEGATE (TEMPLATEMETHOD, ConcreteClass) =`
`amb ConcreteClass in TEMPLATEMETHOD`

In this interpretation `TEMPLATEMETHOD` stands for ‘the skeleton of an algorithm’ which may call some abstract *PrimitiveOperations*, i.e. interfaces provided by the `AbstractClass`. `AbstractClass` stands for an ‘Application’ and every subclass `ConcreteClass` for an individual ‘MyApplication’ which provides its interpretation $op(ConcreteClass, x)$ of the abstract *PrimitiveOperations* $op(x)$ ‘to carry out subclass-specific steps of the algorithm’. This refinement type and various generalizations of it are frequently used with ASMs.

If one wants to restrict the subclass-specific steps to steps of abstract submachines, without modifying the interpretation of functions which are already defined in `AbstractClass`, it suffices to declare only those locations as the class ambient dependent ones which one wants to specifically implement.

¹¹ `TEMPLATEMETHOD` is just a renaming of `OPERATION` to adhere to the names used in [15, pg.325].

Responsibility Pattern The goal of the RESPONSIBILITY pattern following [15, pg.223] is to

avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

e.g. when a static or an a priori specification of the association is impossible.¹² This can be interpreted as an instantiation of DELEGATE where *delegate* is determined by a not furthermore specified external *selection*, applied to the set of *ReceivingObjects* in subclasses *ConcreteClass* which *CanHandle* the *Request* using the implementation of OPERATION in their class.

$$\begin{aligned} & \textit{delegate} = \textit{select} \\ & \{o \in \textit{ReceivingObj}(\textit{Request}) \mid \textit{CanHandle}(o, \textit{OPERATION})(\textit{Request})\} \end{aligned}$$

This determines also the *Responsibility* class structure, see Fig. 2. The *selection* mechanism is furthermore specified in the *Chain of Responsibility* pattern by the stipulation that ‘the handler should be ascertained automatically’ so that one has to ‘chain the receiving objects and pass the request along the chain until an object handles it’. This means that CHAINOFRESPONSIBILITY is a refinement of RESPONSIBILITY by specializing the *select* mechanism to choose the first element *a* which *CanHandle(a, Request)* with respect to a given order relation for the set *ReceivingObj*.¹³

Proxy Pattern The *Proxy* pattern is intended to ‘provide a surrogate or placeholder for another object to control access to it’ [15, pg.207].¹⁴ This can be interpreted as an instantiation of *Delegation* where *delegate* is ‘the real object that the proxy represents’; in fact *delegate* is renamed for this pattern to *realSubject*. More precisely *delegate* is a *ConcreteClass* instance (of one of the subclasses) which is kept in a placeholder location of a dedicated subclass of *AbstractClass* called *Proxy* such ‘that a Proxy can be used anywhere a RealSubject is expected’ [15, pg.210]. In this sense *Proxy* refines OPERATION by forwarding client calls to the *delegate* which is passed as ambient parameter to the implementation $\textit{OPERATION}_{\textit{classOf}(\textit{delegate})}$. This determines the class structure of this pattern instance as illustrated in Fig. 2.

In the same way one can formalize also various instances of proxies. For example a remote proxy is one which forwards every *Request* call to a *delegate* in a different address space. The ASM ambient concept covers this address space

¹² Therefore *AbstractClass*, *ConcreteClass* and OPERATION are renamed in [15] respectively to *AbstractHandler*, *ConcreteHandler* and HANDLEREQUEST.

¹³ To ascertain the handler automatically, it remains to program this function, e.g. by an appropriate instance of the iterator pattern. In the *Command* pattern the association order is defined by the client.

¹⁴ Therefore *AbstractClass*, *ConcreteClass* and OPERATION are renamed to respectively *Subject*, *RealSubject* and REQUEST.

aspect, so that it suffices to impose the mentioned constraint on the values of *delegate* for a remote proxy call.

Similarly one can extend the formalization to cover a virtual proxy which caches *delegate* information via some $CACHE(realSubject, Request)$ so that its access can be postponed. Analogously for protection proxies which check the caller's permission to access *realSubject*.OPERATION.

Strategy and State Patterns The *Strategy* and *State* patterns are proposed to make a variety of different implementations interchangeable, where for the common behavior the difference in the assumptions made in the two patterns about the source of the interchangeable algorithms does not matter. For *Strategy* we read:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [15, pg.315]

The intention of the *State* pattern is described as to

allow an object to alter its behavior when its internal state changes. The object will appear to change its class.[15, pg.305]

Both patterns can be interpreted as an instance of DELEGATE with the same class structure as shown for the *Proxy* pattern in Fig. 2. The class *Proxy* is just renamed to *Context*, which is required to use the OPERATION interface¹⁵ 'to call the algorithm defined by a ConcreteClass' and for this purpose is 'configured with a *ConcreteClass* object', namely *delegate*.¹⁶ No new definition is needed to satisfy the requirement that the 'context delegates state-specific requests to the current *ConcreteClass* object' and that each implementing machine hides its specific data structures and its specific algorithm by being called via a subclass instance *delegate* 'that defines the current state' of *Context*.

Remark on Strategy. The pattern description leaves it open how exactly *Strategy* and *Context* interact, except for requesting that 'clients interact with the context only' and that 'a context forwards request from its clients to its strategy'. In fact 'a context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself

¹⁵ Renaming also OPERATION to ALGORITHM respectively HANDLE, *AbstractClass* to *Strategy* respectively *State*, *ConcreteClass* to *ConcreteStrategy* respectively *ConcreteState*, the OPERATION(*Context*) in *State* to STATESPECIFICREQUEST.

¹⁶ Correspondingly in the *State* pattern, OPERATION (named HANDLE) is interpreted as 'an interface for encapsulating the behavior associated with a particular state of the Context' and each of its implementations in *ConcreteState* to 'implement a behavior associated with a state of the Context' [15, pg.306]. Furthermore, OPERATION_{Context} is called STATESPECIFICREQUEST and interpreted to represent 'the interface of interest to clients': each client request triggers a concrete HANDLE implementation that depends on the *concrState*.

as an argument to Strategy operations. That lets the strategy call back on the context as required.’ [15, pg.317]

An instance of the *Strategy* pattern is found in the Network Leader ASM, defined in [7, 6.1.5] at the abstract level out of three submachines *propose*, *proposeImprove* and *improveByProposals*. These three submachines are then refined by different algorithms to compute either only a leader, or the leader with respect to a total order, or the leader for a partial order, or the leader together with a termination (synchronization) event, or the leader together with a shortest path to it.

Bridge Pattern In the *Bridge* pattern [15, pg.151] *delegate* (which for this pattern is renamed to *imp* standing for an instance of the implementing class) is declared as a location of the *AbstractClass*¹⁷ and the implementing subclasses *ConcreteClass* are outsourced, i.e. separated from *AbstractClass* to become subclasses *ConcreteImplementor* of another class called *Implementor*. This new class provides an OPERATIONIMPL interface to be implemented in the subclasses. This comes up to the following refinement to establish the intended link between the two interfaces OPERATION and OPERATIONIMPL:

$$\begin{aligned} \text{BRIDGEDELEGATE}(\text{OPERATION}, \textit{delegate}) = \\ \text{DELEGATE}(\text{OPERATIONIMPL}, \textit{delegate}) \end{aligned}$$

In this interpretation, coming with a class structure as illustrated in Fig. 2, the pattern provides run-time choices between different refinements of abstract machines via updates of the *delegate* to determine the desired implementation of the common implementation interface OPERATIONIMPL, instead of using the static binding of an implementation to its abstraction as realized by class inheritance. In [15, pg.153] it is required that

both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.

Therefore *AbstractClass* is also refinable by some own subclasses *ConcreteClass*, independently from refinements of *Implementor*, so that different implementations of a common implementation interface become run-time configurable and run-time assignable.¹⁸

Incremental Refinement (Decorator Pattern) The declared goal of the *Decorator* pattern is to

¹⁷ In [15] this class is renamed for this pattern to *Abstraction*.

¹⁸ Following [15, pg.154] the OPERATIONIMP interface for implementation classes ‘doesn’t have to correspond exactly to *Abstraction*’s interface; in fact the two interfaces can be quite different. Typically the *Implementor* interface provides only primitive operations, and *Abstraction* defines higher-level operations based on these primitives’.

‘attach additional responsibilities to an object dynamically’ as ‘a flexible alternative to subclassing for extending functionality’[15, pg.175] .

The implementation of the interface OPERATION of the abstract class—which is called here *Component*, its implementing subclass *ConcreteComponent*—is to be considered as subject to get refined by adding behavior. To this purpose ‘a reference to a *Component* object’ is kept in a location *component*, an ‘interface for objects that can have responsibilities added to them dynamically’. The location *component* is kept in a dedicated subclass *Decorator* which comes with multiple subclasses *ConcreteDecorator*, one for each considered ADDED BEHAVIOR. This determines the class structure of the pattern illustrated in Fig. 3.

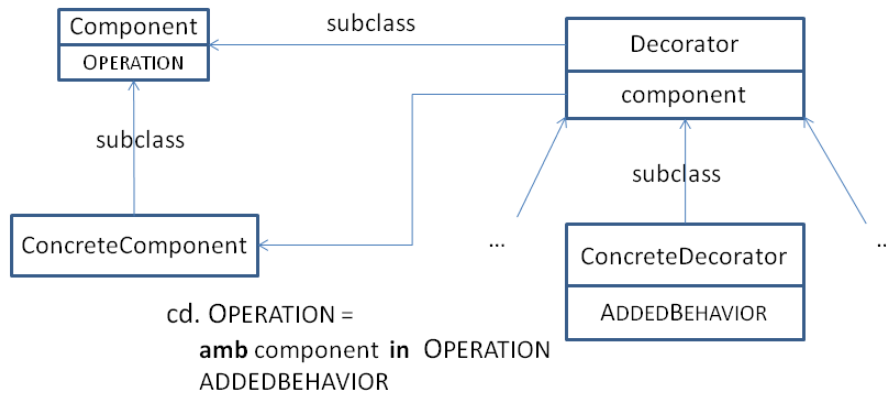


Fig. 3. Decorator Pattern Structure

The value of *component* is supposed to be an instance of *ConcreteComponent* and serves to refer to the given refined behavior $OPERATION_{Decorator}$ of the OPERATION interface.¹⁹

$$OPERATION_{Decorator} = \mathbf{amb\ component\ in\ OPERATION}$$

Each subclass *ConcreteDecorator* adds new behavior to calls of OPERATION simply by defining a new submachine ADDED BEHAVIOR.

$$OPERATION_{ConcreteDecorator} = \\
 OPERATION_{Decorator} \\
 ADDED BEHAVIOR$$

In case the ADDED BEHAVIOR is required to be executed within the *component* ambient, the equation reads as follows:

¹⁹ In the wording of [15] *Decorator* ‘defines an interface that conforms to *Component*’s interface’.

OPERATION_{ConcreteDecorator} =
amb component in
 OPERATION
 ADDEDBEHAVIOR

This kind of purely incremental refinement occurs frequently in ASM developments and is related to conservative extensions of the underlying theories to prove properties which relate the behavior of the given and of the extended machine. It has been heavily exploited in the Jbook [29] for coupling design and verification and appears also in the development of software product lines [3]. In Event-B [2] too it plays a special role and is called there superposition refinement.

Encapsulation (Memento Pattern) The *Memento pattern* illustrates how for reasons of encapsulation abstract operations which belong to one say *Originator* class can be refined in another dedicated class.

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.[15, pg.283]

To achieve this goal two interfaces CREATEMEMENTO and SETMEMENTO, provided by the *Originator* class and intended to encapsulate recording and retrieving the current *Originator* state *curState*, are implemented via SETSTATE, GETSTATE operations of a separate *Memento* class, resulting in the class structure illustrated in Fig. 4.

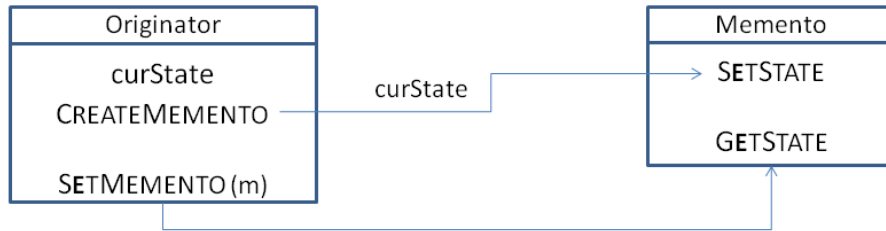


Fig. 4. Memento Pattern Structure

SETSTATE, GETSTATE ‘may store as much or as little of the originator’s internal state as necessary at its originator’s discretion’ creating or restoring snapshots *mementoState(m)* of the internal *curState* recorded in a memento instance *m* created for the purpose. In the following ambient ASM description of the encapsulation, for which *curState* is declared to be ambient indepen-

dent, the reader may think of `SETSTATE(s)` as `mementoState(m) := s` and of `GETSTATE(m)` as `RETURNING mementoState(m)`.²⁰

```

CREATEMEMENTO =
  let m = new (Memento) in
    amb m in SETSTATE(curState) // read mementoState(m) := curState
    RETURN m
SETMEMENTO(m) = RETURN amb m in GETSTATE
  // read: returning mementoState(m) //

```

Views (Publish-Subscribe Pattern) The *Publish-Subscribe* (also called *Observer*) pattern exploits the refinement mechanism to reflect different views between multiple *observers* and one *subject*. The goal is to

define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [15, pg.293].

The structure of the pattern participants is defined by two groups, each consisting of some abstract machines in a class *Subject* respectively *Observer* together with their refinements in subclasses *ConcreteSubject* and *ConcreteObserver*. See Fig. 5.

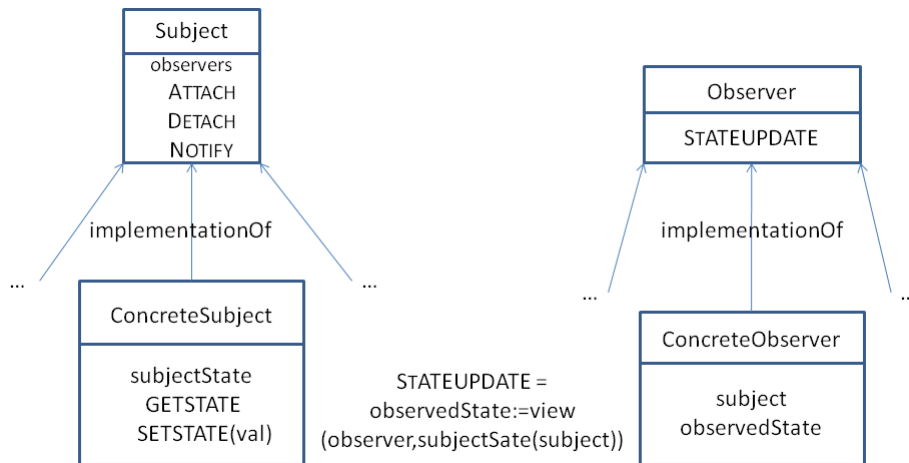


Fig. 5. Publish-Subscribe Pattern Structure

²⁰ We skip the formulation of the two *Caretaker* interfaces whose purpose is to guarantee that *Memento* ‘protects against access by objects other than the originator’, see [15, pg.285] for the details.

Subject has a location for the *observers*—the set of instances of *Observer* which are currently known to the *Subject*—and three interfaces to manipulate or notify²¹ the set of *observers*:

```
ATTACH(o) = INSERT(o, observers)
DETACH(o) = DELETE(o, observers)
NOTIFY = forall o ∈ observers amb o in STATEUPDATE
```

In addition to these operations imported from *Subject*, each *ConcreteSubject* has a *subjectState* location—intended to represent that part of its state which is of interest to concrete observers—together with two submachines GETSTATE and SETSTATE(*val*) to manipulate *subjectState*:

```
GETSTATE = RETURN subjectState
SETSTATE(val) =
  subjectState := val
  // more generally one could write MODIFY(subjectState, val) for val //
```

Each *ConcreteObserver* refines the STATEUPDATE interface imported from the *Observer* and comes with two locations:

- *subject*, denoting an instance of a class *ConcreteSubject*,
- *observedState*, denoting the concrete observer’s view of the state of its *subject*.

The imported STATEUPDATE interface is refined as follows:

```
STATEUPDATE =
  observedState := view(amb subject in GETSTATE)
```

Unfolding the two equations defining NOTIFY shows the intended memory sharing and the fact that the *ConcreteObserver* machines may differ by their *view* function which is used to refine the STATEUPDATE interface for notifying a concrete *observer* about changes of the *subjectState* in the *subject* it observes:

```
amb o in STATEUPDATE // evaluate for curamb = o
= observedState(o) := view(o,
  let curamb = subject(o) in (GETSTATE)*)
= observedState(o) := view(o, let curamb = subject(o) in
  RETURN subjectState(curamb))
= observedState(o) := view(o, subjectState(subject(o)))
NB. o, subjectState are ambient independent
```

²¹ The pattern definition in [15] does not include any scheme to explain how state changes in the subject lead to a notification of the observers.

3.6 Moving Ambients

This is not the place to discuss the huge literature on mobility for which we refer to characteristic surveys [32,11,14]. We use for our illustrative purposes in this section just one outstanding example of a calculus of mobile agents, namely the one which was defined by Cardelli and Gordon in [8,9]. The three operations studied there for changing the hierarchical structure of ambient processes are ambient ENTRY, EXIT and OPENING. There is a natural formulation for each of these operations in terms of an ambient ASM rule. These three simple rules fully capture the calculus of mobile agents which is defined in [8] in terms of roughly two dozens reduction and congruence rules.²²

MOBILEAGENTSMANAGER =
choose $R \in \{\text{ENTRY, EXIT, OPEN}\}$ **in**
 R

This machine runs, transforming the initially given current ambient process $curAmbProc$, as long as there are ambients to enter, to exit or to open in the current value of $curAmbProc$. Choosing for each step one of the above three rules reflects the deductive nature of the calculus, where in each step one reduction rule is applied.²³

Ambient processes (sometimes called also simply ambients) are written in [8] as terms $n[P]$ and interpreted as denoting process P located to run at n . In the context of ambient ASMs one can define $n[P]$ as follows:

$$n[P] = \mathbf{amb} \ n \ \mathbf{in} \ P$$

Ambient processes have a tree structure, which is induced by the nesting of ambients as resulting from the inclusion of brackets $[]$ or program texts **amb** n **in** \dots . In each ambient process $n[P]$ the following three items are distinguished, which can be accessed by appropriate functions:

- An *ambName* (here n , element of a domain *AmbName* of ambient names), which is considered as root of the tree induced by **amb** n **in** P and therefore is sometimes also used to uniquely denote the tree itself.

²² It is a different question we do not discuss here which logic might be appropriate to describe and prove properties of mobile agents, see for example [10]. From a long experience with proving properties for ASMs, either by traditional mathematical proofs (like in [29,3]) or in some dedicated logic calculus (like in [26,28]) or in a machine-assisted manner (e.g. using KIV [17,25,24,19] or PVS [18,33,30,12,13,16] or AsmTP [27]), it wouldn't surprise us if the application of Cardelli's and Gordon's ambient logic to ambient ASMs as a result of the definitional simplification also leads to a simplification of the needed logical combinatorics.

²³ None of the 17 structural congruence rules and no other reduction rules than ENTRY, EXIT and OPEN—called Red In, Red Out, RedOpen in [8]—are needed because of the **choose** construct in the three ASM rules and because of considering *subtrees*(n) as a (possibly multi-) set (without order).

- A (possibly empty) dynamic set $locAg(n)$ of (non-ambient) processes, say P_1, \dots, P_p , called local agents of the ambient process and viewed as located at n and running there.
- A (possibly empty) dynamic set $subAmb(n)$ of subambients, say **amb** m_1 **in** $Q_1, \dots, \mathbf{amb} m_q$ **in** Q_q .

The local agents and subambients of n form a (possibly multi-) set $subtrees(n)$, each element t of which has its own (possibly multi-) set $subtrees(t)$, etc. Process P of ambient **amb** n **in** P , which we denote by $ambBody(n)$, is interpreted in [8] as the parallel composition of the elements of $subtrees(n)$, written as follows, using the process algebra notation $|$ for the parallel composition operator:²⁴

$$P = P_1 | \dots | P_p | \mathbf{amb} m_1 \mathbf{in} Q_1 \dots | \mathbf{amb} m_q \mathbf{in} Q_q$$

The Entry, Exit and Opening actions change the ambient process they are applied to. Therefore we use a variable²⁵ $curAmbProc$ to keep track of the current value of the executed overall ambient process. We identify $curAmbProc$ with the tree it induces, so that the Entry, Exit and Opening actions can be formulated as tree operations applied to any nodes of $curAmbProc$, changing the current value of the dynamic function $subtrees$ at these nodes.²⁶ We freely use other (derived) tree functions, like *sibling* and *parent*, which can all be defined from the dynamic function $subtrees$, as well as tree manipulation operations to INSERT elements into and to DELETE elements from $subtrees(n)$.

Entry into an Ambient The Entry of an ambient $n[. . .]$ into a sibling ambient $m[. . .]$ (if there is one), to become one of its subambients, is triggered by a so-called *entry action* child of n , which is denoted *in* $m.P$. The inverse Exit of a subambient $n[. . .]$ out of its parent ambient $m[. . .]$ (if it exists), to become one of its sibling ambients, is triggered by a so-called *exit action* child of n denoted *out* $m.P$. Opening of a sibling ambient $m[. . .]$ (if it exists) is triggered by a process denoted *open* $m.P$. We now define the meaning of these three operations in detail.

ENTRY is triggered if $curAmbProc$ has an ambient node n with an entry action child, say *in* $m.P$. Such an m serves as target ambient (name) of the entry action. ENTRY chooses such a node n in $curAmbProc$ (if there is any) and then checks whether there is a sibling ambient of n whose $ambName$ matches the target ambient name m . If there is some, ENTRY makes **amb** n **in** (*in* $m.P$ | \dots) move away from this sibling position to let (the modified process) **amb** n **in** (P | \dots) become a subambient of ambient m . See Fig.6.

ENTRY =
if $curAmbProc$ contains an entry action **then**

²⁴ The parallelism is interpreted via interleaving.

²⁵ In the ASM framework variables are treated as dynamic 0-ary functions.

²⁶ In terms of the classification of ASM locations this means that $curAmbProc$ is a derived dynamic (0-ary) function one can define in terms of $subtrees$.

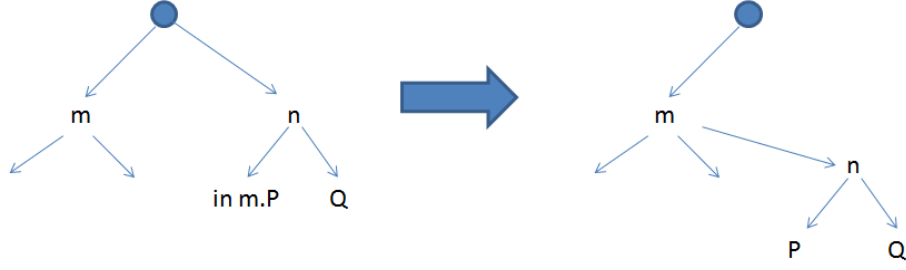


Fig. 6. Entry operation

```

choose  $S = \mathbf{amb} \ n \ \mathbf{in} \ ((in \ m.P) \mid Q) \in EntryAction(curAmbProc)$ 
if  $sibling(S)$  contains a process with ambient name  $m$  then
  choose  $\mathbf{amb} \ m \ \mathbf{in} \ R \in sibling(S)$ 
    DELETE( $S, subtrees(parent(m))$ )27
    //  $n$  disappears as sibling of target ambient  $m$ 
    INSERT( $\mathbf{amb} \ n \ \mathbf{in} \ (P \mid Q), subtrees(m)$ )
    // modified  $n$  becomes subambient of  $m$ 
where
   $curAmbProc$  contains an entry action =
     $EntryAction(curAmbProc) \neq \emptyset$ 
   $EntryAction(curAmbProc) =$ 
     $\{n \in curAmbProc \mid \mathbf{forsome} \ m, P, Q \ \mathbf{ambBody}(n) = (in \ m.P) \mid Q\}$ 
   $X$  contains a process with ambient name  $m =$ 
     $\mathbf{forsome} \ R \ (\mathbf{amb} \ m \ \mathbf{in} \ R) \in X$ 

```

Exit from an Ambient The EXIT operation is triggered by an exit action child $out \ m.P$ (if there is some) of an ambient n and transforms its ambient process, in case this process is a subambient $n[...]$ of a parent ambient $m[...]$ in $curAmbProc$, into a $sibling(m)$. See Fig.7.

```

EXIT =
if  $curAmbProc$  contains an exit action then
  choose  $S = \mathbf{amb} \ n \ \mathbf{in} \ ((out \ m.P) \mid Q) \in ExitAction(curAmbProc)$ 
  if  $parent(n) = m$  then
    DELETE( $S, subtrees(m)$ ) //  $n$  disappears as subambient of  $m$ 
    INSERT( $\mathbf{amb} \ n \ \mathbf{in} \ (P \mid Q), subtrees(parent(m))$ )
    // modified  $n$  becomes sibling ambient of  $m$ 
where
   $curAmbProc$  contains an exit action =
     $ExitAction(curAmbProc) \neq \emptyset$ 

```

²⁷ To guarantee that $parent(m)$ is always defined, we assume without loss of generality that the tree representation of $curAmbProc$ has a *root* that is not a process.



Fig. 7. Exit operation

$$\text{ExitAction}(\text{curAmbProc}) = \{n \in \text{curAmbProc} \mid \text{forsome } m, P, Q \text{ } \text{ambBody}(n) = (\text{out } m.P) \mid Q\}$$

Opening an Ambient OPEN is triggered by an ambient dissolving action $\text{open } m.P$, which “provides a way of dissolving the boundary of an ambient named m located at the same level as open ” [8, Sect.2.2]. In other words OPEN replaces a subtree pair $(\text{open } m.P, \text{amb } m \text{ in } Q)$ of siblings in curAmbProc by the new siblings pair (P, Q) . See Fig.8.



Fig. 8. Open operation

OPEN =
if curAmbProc contains an ambient dissolving action **then**
 choose $S_1 = \text{open } m.P_1 \in \text{AmbDissolvAction}(\text{curAmbProc})$
 if $\text{sibling}(S_1)$ contains a process with ambient m **then**
 choose $S_2 = \text{amb } m \text{ in } P_2 \in \text{sibling}(S_1)$
 let $p = \text{parent}(S_1)$
 forall $i \in \{1, 2\}$
 DELETE($S_i, \text{subtrees}(p)$)
 INSERT($P_i, \text{subtrees}(p)$)

where

$curAmbProc$ contains an ambient dissolving action =
 $AmbDissolvAction(curAmbProc) \neq \emptyset$
 $AmbDissolvAction(curAmbProc) =$
 $\{n \in curAmbProc \mid \mathbf{forsome} \ m, P \ \mathit{ambBody}(n) = \mathit{open} \ m.P\}$
 X contains a process with ambient $m =$
 $\mathbf{forsome} \ Q \ (\mathbf{amb} \ m \ \mathbf{in} \ Q) \in X$

Each time there is no element to choose, the tree manipulation operation cannot be performed (in the ASM framework the rule is then equivalent to **skip**, which does not change the tree $curAmbProc$).

The restriction operator $(\nu n)P$ can be expressed in the ASM framework by $P(n/new(AmbName))$, due to the new function which each time it is applied to a set provides a new, completely fresh element for this set.

4 Related Work and Conclusion

We tried in this paper to achieve a qualitative goal by a) providing a simple definition of a general ambient concept and b) illustrating its wide applicability by a series of characteristic non-trivial examples from different domains. In essence it was the arguably most general notion of ASM state which allowed us to fully exploit the power of parameterization for defining a most general abstract notion **amb env in M** of machines working in a defined environment. Numerous other ambient concepts have been proposed in the literature. Since this is not the place to list this literature, in accordance with common scientific practice we have cited only what we have used or referred to directly.

What can be said in general is that the definitions in the literature typically provide specific solutions for particular contexts like mobility of devices or of code or of the context structure for the execution of a (e.g. Java) program, whereas the construct we have defined within the ASM framework is of abstract nature, covering in a uniform way various forms of context (syntactical, computing, user or even physical context). We are not aware of any other ambient definition which covers in a simple and uniform way the challenging examples we use in this paper to illustrate the wide applicability of our definition.

Another distinction is that our definition treats ambients as first-order objects, exploiting the generality of the notion of state underlying ASMs. This simplifies enormously to define and work with the concept in different application areas.

The main purpose of going public with this definition is to trigger further uses to acquire a still wider range of experimental experience before embarking on an implementation, e.g. by programming a plug-in for the CoreASM engine [1].

Acknowledgement. We thank Richard Banach, Iain Craig, Peter Müller, Helmuth Partsch, Jens Kohlmeyer, Alexander Raschke for useful criticism on an earlier version of the paper. We thank G. Schellhorn and the KIV group in Augsburg for discussing how the ASM ambient concept could be made available in the KIV verifier system by appropriately extending the rich body of ASM concepts

already implemented in KIV. We thank M. Vardi for recognizing the potential of an ambient ASM based design pattern analysis. We thank two referees for having forced us to better phrase our goals.

To appear in: J. Computer System Science (Special Issue in honor of Amir Pnueli)

References

1. The coreasm project. <http://www.coreasm.org>.
2. J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
3. D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, 14(12):2059–2082, 2008.
4. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
5. E. Börger. Design pattern abstractions and Abstract State Machines. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*, pages 91–100. Université de Paris 12, 2005. <http://lacl.u-pec.fr/~dima/asm05/asm05-contents.html>.
6. E. Börger and O. Sörensen. *Handbook of Conceptual Modelling. Theory, Practice, and Research Challenges*, chapter BPMN Core Modeling Concepts: Inheritance-Based Execution Semantics, pages 287–332. Springer, 2011. ISBN: 978-3-642-15864-3.
7. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
8. L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.
9. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
10. L. Cardelli and A. D. Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2006.
11. G. Chen and Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dartmouth, 2000.
12. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
13. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
14. C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, and D. Rmy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer Berlin / Heidelberg, 1996.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
16. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322. Springer-Verlag, 2000.

17. M. Giese, D. Kempe, and A. Schönege. KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur. Interner Bericht 16/97, Universität Karlsruhe, Germany, 1997.
18. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzsos, editor, *Int. Conf. on Compiler Construction, Proc. Poster Session of CC'96*, Linköping, Sweden, 1996. IDA Technical Report LiTH-IDA-R-96-12.
19. D. Hagenberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Aspects of Computing*, 20(1), 2007.
20. M. Jackson. Program verification and system dependability. In P. P. Boca, J. P. Bowen, and J. I. Siddiqi, editors, *Formal Methods: State of the Art and New Directions*, pages 43–78. Springer, London, 2010.
21. S. Oaks and H. Wong. *Java Threads*. O'Reilly, third edition, 2004.
22. OmgBpmn. Business Process Modeling Notation (BPMN). Version 2.0. <http://www.omg.org/spec/BPMN/20100501>, 2009. formal/2011-01-03.
23. P. Pepper and P. Hofstedt. *Funktionale Programmierung*. Springer, 2006.
24. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Germany, 1999.
25. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
26. A. Schönege. Extending dynamic logic for reasoning about evolving algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, Germany, 1995.
27. R. F. Stärk. Formal specification and verification of the c# thread model. *Theor. Comput. Sci.*, 343:482–508, October 2005.
28. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.
29. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
30. M. M. Stegmüller. Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen. Diplom thesis, University of Ulm, Germany, 1998.
31. M. Weske. *Business Process Management*. Springer, 2007.
32. M. Wooldridge and N. Jennings. Agent theories, architectures, and languages: A survey. Springer LNCS, pages 1–39, 1995.
33. W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *J. Universal Computer Science*, 3(5):504–567, 1997.