Design Pattern Abstractions and Abstract State Machines

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it
on sabbatical leave at SAP Research, Karlsruhe, Germany egon.boerger@sap.com

Abstract. As a first step towards defining high-level modeling patterns, which support the general ASM refinement technique, we model some standard object-oriented design patterns from [12] by ASMs.

1 Introduction

In this paper we model as Abstract State Machines (ASMs) some object-oriented design patterns, which have been introduced in [12] in the context of object-oriented programmming principles. We hope to trigger by this work some investigation of genuine modeling patterns, which are placed at a higher level of abstraction than object-oriented class structures and related programming structures.

We suppose the reader to have a rough understanding of ASMs or to believe in the accurate definability in terms of ASMs of his intuitive understanding of pseudo code, i.e. rules of the form

if *Condition* **then** $f(t_1, \ldots, t_n) := t$

All the rules of one machine are simultaneously executed. Besides standard abbreviations (like **let**) we use also the following notation to explicitly name forms of non-determinism:

choose x **with** ϕ
  rule

In addition we adopt the following notation, which goes together with the synchronous parallelism for the execution of ASM rules:

forall x **with** ϕ **do**
  rule

Details can be found in the AsmBook [10].

Since their introduction in [12], the object-oriented design patterns have been widely adopted for reusable object-oriented software development. They are traditionally classified by their purpose and by their scope, distinguishing *structural* patterns, which are concerned with the composition of classes and objects, from *creational* patterns, which are concerned with the creation of objects at class instantiation, and from *behavioral* patterns which describe the interaction of classes and objects, characterizing the flow of control, abstract forms of actions of objects and the communication and cooperation between classes and objects. Concerning the scope, *class patterns* are characterized as dealing with the static (compile-time relevant) composition of (sub)classes, expressible in terms of signature declarations via inheritance and accessibility or use relations, whereas *object patterns* have a dynamic character, dealing with the composition of run-time objects. We illustrate in this paper how the notion of ASM (their abstract states and updates) together with the notion of ASM refinements can be used to classify and precisely capture the abstractions which are incorporated into design pattern.

On the basis of their abstraction mechanism, patterns can be classified into the following three groups (where we skip most patterns which are of syntactical nature and well-understandable in terms of mere signatures and associations of classes):

- *Refinement of abstract (sub)machines or operations* by a more detailed machine, which includes the case of an implementation. In the object-oriented framework in which design patterns are presented in [12], abstract classes or operations are those defined by the mere signature without any implementation. Furthermore as refinement only implementations are considered, whereas the ASM refinement notion permits the use of more general successive refinements. See its introduction into ASMs in [2,4,3,5], where standard refinement concepts in the literature are generalized exploiting the abstraction facilities offered by ASMs. See the analysis in [13,14] and the survey in [5]. As examples we discuss below the patterns *Template Method*, *Memento*, *Observer* and *Iterator*.
- *Parameterization of (sub)machines or operations by implicit arguments* for the current class instance, in the ASM framework more generally for the current machine executing agent. As examples we discuss below some typical structural patterns, namely *Bridge, Proxy, ObjectAdapter* and *Decorator*, and some behavioral patterns, namely *Strategy* and *Visitor*. Another related example is the so-called *State* pattern, where the parameterization is done by a state.
- *Run-time association of agents to actions.* This association can be non-deterministic or deterministic. The deterministic case is usually based upon some ordering, which may be predefined or defined by clients. As examples we discuss the *Set of Responsibility* pattern, defined in analogy to the *Chain of Responsibility* pattern which exhibits a predefined order-traversal scheme, whereas the order underlying the *Command* pattern can be defined by clients.

Whether a pattern falls into one or the other group is to a certain extent a pragmatic matter and certainly does not divide patterns into mutually exclusive schemes. For example the *State* pattern and the *Bridge* pattern can also be seen as an example of run-time association of actions to certain parameters.

Before explaining these three groups of pattern abstraction in the following sections, we briefly review the involved abstraction concepts and how they are expressed in the ASM framework. We transfer the abstract type and subclassing concept from [12] to the notion of ASM submachines defined in [9]. These definitions are compatible with specialized type and inheritance concepts. A type is understood as an interface. In the object-oriented reading this means a set of operations for objects of the type. In the more general ASM framework it simply means an ASM signature, consisting of function and submachine names. A type s is called a subtype of type t if and only if the interface of s contains the interface of t. We understand a class here as an implementation (more generally: as a refined ASM) of a given interface, where the refinement notion is the one I have introduced into ASMs, generalizing more restricted refinement concepts in the literature (see the above mentioned survey [5]). N is called a subclass of M if and only if N imports every submachine and function definition (read: implementation) of M.

The effect of parameterization is easily explained within the ASM framework, which contains functions in the mathematical sense, but also functions in the sense of programming (submachines). Making explicit an implicit parameter **self** of expressions $f(x_1, \ldots, x_n)$ yields a global view of an $n+1$-ary function or location. Its individual branches

$$\mathbf{self}\ .f = f(\mathbf{self}, x_1, \ldots, x_n)$$

represent independent n-ary functions (depending on arguments $x_1, \ldots, x_n$), one function for each value of **self**: an instance of f that is private to **self**. In most cases **self** is used to denote an agent, typically the one currently executing a given machine, similar to the object-oriented current class instance *this* with respect to which methods of that class are called (executed). In an object-oriented spirit the parameter **self** is often left implicit.

2 Using the refinement of abstract machines

In this section we illustrate how patterns use the refinement of abstract (sub)machines or operations by a more detailed machine or by an implementation. As examples we discuss the patterns *Template Method*, *Memento, Observer* and *Iterator*, formalizing the descriptions given in [12].
For the **Template Method** we read in [12, 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.

The structure for this pattern is characterized by two classes:

- An *AbstractClass*, standing for an 'Application' and providing a *TemplateMethod* ('the skeleton of an algorithm'). This method may call operations defined in *AbstractClass* (or in classes of other objects) and also some *PrimitiveOperations* that are abstract (read: empty or not implemented) for *AbstractClass*.
- A subclass *ConcreteClass* (of *AbstractClass*), which stands for an individual 'MyApplication' and implements the *PrimitiveOperations* 'to carry out subclass-specific steps of the algorithm'.

The ASM Method uses instead of this object-oriented subclassing and inheritance paradigm a more general and mathematically accurate refinement concept (see [5]). One can define an ASM using abstract operations (functions) or submachines (macros) which are then refined. The place where such refinements are defined can be 'elsewhere', i.e. it is not necessarily in a subclass of the machine where the abstract operation occurs. One may also introduce more than one refinement of the same abstraction; this generalizes the case that abstractions are implemented 'in various ways' by multiple subclasses differing by variations of the implementations.

We now show examples to illustrate refinements done 'elsewhere' (like in the Memento pattern) or 'in various ways' (like in the Iterator and Observer pattern).

The **Memento pattern** illustrates how an abstract operation for reasons of encapsulation is refined in a separate class. We read as intent in [12, pg.283]:

> Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

In other words the goal of this patterns is to offer abstract operations to record or retrieve the current state *currState* of one machine (the *Originator* class), which are refined by (read: encapsulated via operations of) another machine (the *Memento* class). More technically speaking the two classes are related as follows:

- The class *Memento* has state recording submachines (operations) *SetState, GetState*, which 'may store as much or as little of the originator's internal state as necessary at its originator's discretion'.
- The class *Originator* comes with a location *currState* and abstract machines CreateMemento and SetMemento, which are implemented as follows via *Memento* operations to create or restore snapshots of its internal current state via a memento instance:

$$
\begin{aligned}
&\textsc{CreateMemento} = \textit{return}\ \text{New}\ \textit{Memento}(\textit{currState})\\
&\quad = \textbf{let}\ m = \ \text{New}\ (\textit{Memento})\ \textbf{in}\\
&\qquad m.\textit{SetState}(\textit{currState})\\
&\qquad \text{Return}\ m\\
&\textsc{SetMemento}(\textit{memento}) = (\textit{originatorState} := \textit{memento.GetState})
\end{aligned}
$$

To achieve that *Memento* 'protects against access by objects other than the originator', two interfaces are introduced by a third class, called *Caretaker*, which 'is reponsible for the memento's safekeeping' and 'never operates on or examines the contents of a memento'. It 'sees a *narrow* interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state.' [12, pg.285] These features belong to the pragmatic use conditions of the pattern and are therefore not formalized here.

For the sake of generality we have distinguished two locations to denote *Originator* states, namely *currState* (from where a state is recorded) and *originatorState* (where a previously recorded state is retrieved); an implementation may identify them. The operation *m.SetState(state)* is usually implemented using an additional state location in *Memento*, say by *m.mementoState:=state*.

We rely here without further explanation upon an intuitive understanding of the value returning mechanism. A mathematical definition which supports this understanding within the synchronous parallelism of ASMs has been given in [8].

The **Iterator** pattern illustrates how an abstract operation can be refined in various ways, in particular to support polymorphic concepts. The goal here is to provide a uniform interface for multiple traversal of and access to components of different structures without refering to the internal encoding: 'provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation' [12, pg.257]. We see here an *Iterator* machine which 'defines an interface for accessing and traversing elements'. It comes with a location *currItem* and abstract submachines *First*, *Next*, *IsDone*, *Previous*, *SkipTo(Condition)*. All of these abstract machines are to be refined in a variety of subclasses (submachines) *ConcreteIterator*, which 'implement the *Iterator* interface' and 'keep track of the current position in the traversal of the aggregate'. Each *ConcreteIterator* is associated to a subclass *ConcreteAggregate* which implements the *Iterator* creation interface; this interface is an abstract method *CreateIterator* of its abstract superclass *Aggregate*, 'to return an instance of the proper *ConcreteIterator*'.

No more constraints are imposed on these implementations. We therefore see that also in this pattern, the crucial role of the underlying concept of implementation is mathematically underpinned and generalized by the ASM refinement notion. A simplification derives from the fact that the ASM framework, which first of all supports the modeling (specification) activity in system design, does not impose any *a priori* defined type system.

The **Observer** pattern, sometimes also called *Publish-Subscribe* pattern, exploits the refinement mechanism to reflect different views between some *observers* (typically many) 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' [12, pg.293]. The structure of the pattern participants is defined by two groups, each consisting of an abstract machine *Subject* respectively *Observer* together with implementing submachines.

The concrete machines *ConcreteSubject* are supposed to refine (implement) *Subject* from where they import each a location for the *observers*—a set of instances of *Observer*—which are currently known to the *Subject*. Each *ConcreteSubject* also imports the following three interfaces provided by *Subject*, with the indicated meaning to manipulate or notify[1] the set of *observers* (where we write $o \in \mathit{observers}$ for the characteristic predicate of the set *observers*):

$$\textsc{Attach}(o) = (o \in \mathit{observers}) := \mathit{true}$$
$$\textsc{Detach}(o) = (o \in \mathit{observers}) := \mathit{false}$$
$$\textsc{Notify} = \textbf{forall } o \in \mathit{observers} \;\; o.\textsc{Update}$$

In addition to importing *Subject*, each machine *ConcreteSubject* has a *subjectState* location—intended to represent the state of interest to concrete observers—together with two submachines GetState and SetState(*change*) to manipulate *subjectState* as follows:

$$\textsc{GetState} = \text{ Return } \mathit{subjectState}$$
$$\textsc{SetState}(\mathit{change}) = \mathit{subjectState} := \text{Modify}(\mathit{subjectState}, \mathit{change})$$

Each of the concrete machines *ConcreteObserver* refines the Update interface imported from the abstract machine *Observer*. Each *ConcreteObserver* comes with two locations: *subject*, denoting an instance of a class *ConcreteSubject*, and *observerState*, denoting the concrete observer's view of the state of its *subject*. Each of these machines refines the imported Update interface to guarantee the following meaning:

$$\textsc{Update} = \mathit{observerState} := \textbf{self}\ .\text{View } (\mathit{subject}.\textsc{GetState})$$

The *ConcreteObserver* machines may differ by their View function which is used to refine the Update interface for notifying an observer about changes in subject.

[1] The pattern definition in [12] does not include any scheme to explain how state changes in the subject lead to a notification of the observers.

3 Parameterization of machines

In this section we formalize patterns which use the parameterization of (sub)machines or operations by implicit arguments, in particular for the current agent or the current class instance. The examples we discuss are the structural patterns *Bridge, Proxy, ObjectAdapter, Decorator* and the behavioral patterns *Strategy* and *Visitor*, which all use the parameterization by agents. In the *State* pattern one finds a parameterization by states.

The **Bridge** pattern illustrates the parameterization of machines by agents (class instances) to 'decouple an abstraction from its implementation so that the two can vary independently' [12, pg.151]. It also serves the goal to provide at run-time the possibility to choose between different refinements of abstract machines, instead of using the fixed binding of an implementation to its abstraction as realized by class inheritance only. The structure consists of two abstract machines *Abstraction* and *Implementor*, the second one serving for the implementation of the first one. The key property is that *Abstraction* is refinable by subclasses *RefinedAbstraction* independently from refinements of the *Implementor* (see the remark below).

More specifically, *Abstraction* provides a high-level abstract interface *Operation* and a location *imp*, which serves as reference to an object (an instance) of type *Implementor* (read: the currently chosen delegator).

The machine *Abstraction* is intended to be refinable by subclasses *RefinedAbstraction* to 'extend the interface defined by *Abstraction*', independently from refinements of its *Implementor*, the abstract machine which provides an implementation interface *OperationImp* to be refined in machines *ConcreteImplementor* as follows:

$$Operation = imp.OperationImp$$

This realizes the intended forwarding of client requests of *Operation* to the current implementor *imp*. Since *OperationImp* can be implemented by different subclasses *ConcreteImplementor*, using this pattern one obtains that via updates of the implicitly given delegator *imp*, different implementations of a fixed interface become run-time configurable and run-time assignable. In particular, the interface *OperationImp* 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' [12, pg.154]. The combination of multiple *imp*-addressable *ConcreteImplementor*s with independent refinements *RefinedAbstraction* of *Abstraction* has purely pragmatic reasons and has no formal counterpart. The official explanation reads: '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.' [12, pg.153]

The **Proxy** pattern is intended to 'provide a surrogate or placeholder for another object to control access to it' [12, pg.207]. The structure is defined by the following three groups of machines:

- A machine *Subject* providing an interface (read: an abstract submachine or operation) *Request*,
- subclasses *RealSubject* of *Subject*,
- a *Subject*-subclass *Proxy*, which provides a placeholder location *realSubject*, intended to define an instance of *RealSubject*, 'the real object that the proxy represents'.

The *Subject* interface *Request* is imported by and common to the subclasses *RealSubject* and *Proxy*, 'so that a Proxy can be used anywhere a RealSubject is expected' [12, pg.210]. *Proxy* refines *Request* by forwarding client calls for its execution to the instance *realSubject*, which is given as implicit parameter. This is formalized as follows:

$$\mathbf{self}.Request = \mathbf{self}.realSubject.Request$$

One can formalize also the various instances of proxies. For example a remote proxy is one which sends *Request* to *realSubject* in a different address space. Therefore one has to add to the above formalization a notion of different address spaces and sending messages between them and to impose the corresponding constraint on *realSubject*. Similarly one can extend the formalization to cover a virtual proxy which caches *realSubject* information so that its access can be postponed. Analogously for protection proxies which check the caller's permission to access *Request*.

The **Object Adapter** pattern aims at supporting that 'clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request' [12, pg.141]. The structure to realize this pattern is given by three machines:

- a main machine *Target*, which 'defines the domain-specific interface that Client uses' in the form of an abstract submachine (read: operation) *Request* to be used by the clients,
- a machine *Adaptee*, which provides (an implementation of) a machine interface *SpecificRequest* 'that needs adapting' to serve as implementation of *Request*,
- a subclass *Adapter* of *Target*; it adapts the *SpecificRequest* interface of *Adaptee* to the *Target* interface *Request* expected by clients, using an *Adapter* location *adaptee* for an instance of *Adaptee*.

Thus the *Adapter* refinement of the *Target Request* consists in parameterizing the *Adaptee* interface *SpecificRequest* by the *adaptee* instance:

$$Request = adaptee.SpecificRequest$$

The **Decorator** pattern serves to 'attach additional responsibilities to an object dynamically', as 'a flexible alternative to subclassing for extending functionality' [12, pg.175]. The structure is defined as follows by a main machine *Component* with an implementing subclass *ConcreteComponent* and an additional subclass *Decorator*:

- The main machine *Component* provides a high-level submachine *Operation* representing an interface that is subject to get extended by added behavior, 'the interface for objects that can have responsibilites added to them dynamically'.
- A *Component* subclass *ConcreteComponent* 'defines an object to which additional responsibilities can be attached'.
- The subclass *Decorator* of *Component* provides a location *component*, to address the given behavior of *Operation* as described below, and comes with possible subclasses *ConcreteDecorator* to add new behavior to calls of *Operation*, as described below.

Decorator 'maintains a reference to a *Component* object' using a location *component*, defined as instance of *Component*. It serves to refer to the given subclass-defined behavior of *Operation* by implicit parameterization as follows (i.e. *Decorator* 'defines an interface that conforms to *Component*'s interface'):

$$Operation = component.Operation$$

The added behavior is represented by any of the subclasses *ConcreteDecorator* which refine *Operation* by adding to it new submachines *AddedBehavior* as follows:

$$\begin{aligned}&\mathbf{self}\ .Operation = \\ &\quad component.Operation \\ &\quad \mathbf{self}\ .AddedBehavior\end{aligned}$$

The **Strategy** pattern serves to make a variety of different implementations interchangeable at run-time: 'Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy les the algorithm vary independently from clients that use it.' [12, pg.315] The structure of this pattern is given by two machines *Strategy* and *Context*, where *Strategy* comes with intendently multiple subclasses *ConcreteStrategy*.

- Machine *Strategy* 'declares an interface common to all supported algorithms', say in the form of an abstract (sub)machine (read: method name) *AlgorithmInterface*.
- Each member of a family of specific machines (read: algorithms), formally subclasses *ConcreteStrategy* of *Strategy*, defines an implementation for the common interface *AlgorithmInterface* declared in *Strategy*.

- Machine *Context* uses the interface *AlgorithmInterface* 'to call the algorithm defined by a ConcreteStrategy'. It is 'configured with a *ConcreteStrategy* object' via a location *strategy* it provides, as described below.

To state it in other words: *ConcreteStrategy* is one among possibly many subclasses (i.e. machines importing *Strategy*) which refine *Strategy* by implementing its *AlgorithmInterface*, denoted as follows using the implicit parameter **self**:

$$AlgorithmInterface = strategy.AlgorithmInterface$$

In this way each implementing machine hides its specific data structures and its specific algorithm. Each machine is callable via an executing agent (subclass instance), typically instantiated by *Context* to an object in a class *ConcreteStrategy* out of a finite set of (implementations of) such concrete strategies. 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 as an argument to Strategy operations. That lets the strategy call back on the context as required.' [12, pg.317]

As an instance of the *Strategy* pattern consider the Network Leader ASM, defined in [10, 6.1.5] at the abstract level out of three submachines *propose*, *proposalsImprove* and *improveByProposals*. These three submachines are then refined in different ways (read: 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.

The intent of the **Visitor** pattern is to 'represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates' [12, pg.331]. The idea is to represent an operation on a concrete element not directly as belonging to its class *ConcreteElement*, but as a 'Visitor operation' *VisitConcreteElement* that corresponds to its class, in such a way that the operation gets executed on the element by its 'accepting' a corresponding visitor to *VisitConcreteElement(element)*. This explains that the structure of the pattern is defined by two main machines *Element* (belonging to the given *ObjectStructure*) and *Visitor* with the following subclasses:

- *Element* comes with subclasses *ConcreteElement* whose elements are to be operated on by some operations defined in the subclass and triggered by 'accepting' a corresponding 'visitor'. For this purpose the machine *Element* provides an abstract machine (interface) *Accept(visitor)* which is refined in *ConcreteElement* to visit (read: trigger the operation on) concrete elements.
- The *Visitor* class comes with subclasses *ConcreteVisitor*. *Visitor* provides for each *ConcreteElement* a parameterized abstract machine (interface) *VisitConcreteElement(concrElem)* (see the explanation below for the parameter *concrElem*). Each of these is refined (implemented) by every submachine *ConcreteVisitor* of *Visitor*. 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' [12, pg.334]. The abstract machine (interface) *Accept(visitor)* is therefore refined to visit (read: operate on) every *ConcreteElement* as follows:

$$Accept(v) = v.VisitConcreteElement(\mathbf{self})$$

The implicit parameter **self**, for a *ConcreteElement* instance upon which an operation has to be triggered, represents that 'the element supplies itself as an argument to this operation to let the visitor access its state, if necessary' [12, pg.335]. The visitor parameter v denotes an instance of a *ConcreteVisitor* machine implementing *VisitConcreteElement*. This can be formalized by the following constraint:

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

In this way in case a new operation has to be defined on *Element*, one can do this 'simply by adding a new visitor' which will trigger the new operation upon acceptance of the visitor. Technically speaking this means to define a new *Visitor* submachine *ConcreteVisitor*, which implements each *VisitConcreteElement* interface (one for each *Element* subclass *ConcreteElement*) in the desired new manner. As a result there is no need to change the given structure of subclasses *ConcreteElement* of *Element* 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' [12, pg.335].

The intention of the **State** pattern is to 'allow an object to alter its behavior when its internal state changes. The object will appear to change its class' [12, pg.305]. The structure of the pattern is defined by two machines *State* and *Context* with the following locations and submachines:

- *State* provides an abstract *Handle* machine (read: method name) that is implicitly parameterized by a state of the given *Context* (see below *currstate*). This abstract *Handle*r represents 'an interface for encapsulating the behavior associated with a particular state of the Context' [12, pg.306] and is to be refined (implemented) by submachines *ConcreteState* of *State*, each of which 'implements a behavior associated with a state of the Context'.
- The machine *Context* provides an abstract *Request* machine, which represents 'the interface of interest to clients', together with a location *currstate* that represents an instance of a subclass *ConcreteState* of *State*, the one 'that defines the current state' of *Context*. Each client *Request* triggers a concrete *Handle*r implementation that depends on the *currstate*. This is expressed by the following stipulation:

$$Request = currstate.Handle$$

This equation realizes the pattern requirement that the 'context delegates state-specific requests to the current *ConcreteState* object'. The further requirement that 'a context may pass itself as an argument to the *State* object handling the request. This lets the *State* object access the context if necessary' [12, pg.306] comes up to an corresponding parameterization of *currstate*.
Remark. The equations for the parameterization have the same form in different patterns, for example in *Adapter*, *Bridge*, *Decorator*, *Strategy*, *State*. In fact the underlying semantical meaning of the parameterization (namely the implicit instantiation of a machine) is the same; what differs is the use which is made of this refinement, as determined by the different class (read: submachine) structures reflecting the different goals of the involved patterns.

4 Run-time association of agents to actions

In this section we illustrate how patterns in [12] use the run-time association of agents to actions. We distinguish the non-deterministic case, where one gets a *Set of Responsibility* pattern, from the deterministic case for which we formalize *Chain of Responsibility* (with predefined automatic set traversal) and *Command* (with association order defined by the client).

The declared goal of the *Chain of Responsibility* pattern is to

> avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. [12, pg.223]

e.g. when a static or an a priori specification of the association is impossible. The structure of the pattern reflects this purpose: it consists of a machine *AbstractHandler* with submachines *ConcreteHandler* defined as follows:

- *AbstractHandler* provides an abstract *HandleRequest* machine (read: method name as interface to clients), which is to be implemented by every given subclass *ConcreteHandler*.
- Every *ConcreteHandler* refines *HandleRequest* by associating any request m to an agent which is chosen out of a set of given implementations that can handle m.

The question now is how this agent is chosen. A non-deterministic choice leads to what one could call **Set of Responsibility** pattern, formalized by the following refinement of *HandleRequest*:

$\textsc{HandleRequest}(m) = \textbf{choose } a \in \mathit{CanHandle}(m)$
$\quad a.m$

where the set *CanHandle(m)* ranges over the appropriate instances of the given set of subclasses that implement *HandleRequest* and are available for the given m. The set of these instances is typically dynamic.

In the **Chain of Responsibility** pattern the idea is that there should be no choice of the responsible agents, but 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'. For this purpose every *ConcreteHandler* (or alternatively the common superclass *AbstractHandler*) is required to also provide a *successor* location which is used to implement *HandleRequest* as follows:

$\textsc{HandleRequest}(\textbf{self}, m) = \textbf{if } \mathit{CanHandle}(m)$
$\quad \textbf{then self }.m$
$\quad \textbf{else } \textsc{HandleRequest}(\textbf{self }.\mathit{successor}, m)$

However, to ascertain the handler automatically, it remains nevertheless to specify the initial agent **self** and a termination criterion for passing *HandleRequest* along the *successor* chain. It also remains to specify whether the function *successor* is static or dynamic.

The declared goal of the **Command** pattern is to "encapsulate a request as an object" and to "support undoable operations". To achieve this the class *Command* provides an abstract *Execute* machine which is to be implemented by a subclass *ConcreteCommand*, one for each implementation of a concrete command. *ConcreteCommand* refines *Execute* by associating a (*Receiver* instance) *receiver* to an interface *Action* in *Receiver* where the command is implemented. The intention is that clients create a *ConcreteCommand* instance, thereby specifying an instance of *Receiver* where the *Action* is defined. Additionally *ConcreteCommand* provides a machine to *Store* a *state* for undoing of operations. This is formalized by the following refinement:

$\textsc{Execute}(\textbf{self}) = \textbf{if } \mathit{Doable}\ (\textbf{self }.\mathit{receiver}.\mathit{Action}(\textbf{self }.\mathit{receiver}))\ \textbf{then}$
$\quad \textbf{self }.\mathit{receiver}.\mathit{Action}(\textbf{self }.\mathit{receiver})$
$\quad \mathit{Store}\ \ \textbf{self }.\mathit{state}$

As a result, using this pattern one can parameterize clients with multiple requests.

5 Conclusion

We have illustrated how the design pattern abstractions, which have been introduced in the context of object-oriented programming, can be naturally captured by ASMs. This can be seen as a first step towards turning these patterns into class libraries, as advocated by B. Meyer (personal communication, Zürich, December 2004). However, this exercise leaves many interesting questions open. The descriptions given here are oo-programming-centric, direct formalizations of the semantical (mostly behavioral) aspects of the patterns defined in [12]. It remains to investigate whether interesting high-level modeling patterns can be defined (read: genuine ASM refinement patterns which support the general ASM refinement concept described in [5]), furthermore in such a way that they yield abstract information for the designer that the traditional class-oriented design patterns do not provide. It also remains to be analyzed whether the ASM pattern description method presented here can be applied also to more complex composite patterns, like Enterprise Computing Patterns [11] or J2EE Core Patterns [1], and whether the structure of such patterns can be defined in terms of simple basic patterns. This is related to the still widely open question what are appropriate ASM composition techniques to build complex machines out of components by verification-supported construction schemes, linking architectural and component level system views. Some general techniques are formulated in [6]. It would also be interesting to know whether such ASM pattern descriptions can help to distinguish useful pattern combinations from less useful ones, etc.

Acknowledgement. This is to thank W. Zimmermann and A. Cavarra (for explanations of some design pattern features) and M. Altenhofen (for a critical discussion of two draft versions of this paper, during which he also asked the questions stated in the Conclusion). Thanks also to B. Meyer for having triggered this paper by attracting my interest in his course at the ETH Zürich on *Trusted Components: Reuse, Contracts and Patterns* (winter term 2004/05, see `http://se.inf.ethz.ch/teaching/ws2004/0239/slides/AsmMethZh04.PDF`). A final version of this paper will appear as part of the *D.Rödding Gedenkschrift* paper [7].

References

1. D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2003.
2. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1990.
3. E. Börger. *Berechenbarkeit, Komplexität, Logik*, chapter Dynamische Algebren und Semantik von Prolog, pages 476–499. Vieweg, 3rd edition, 1992.
4. E. Börger. A natural formalization of full Prolog. *Newsletter of the Association for Logic Programming*, 5(1):8–9, 1992.
5. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
6. E. Börger. Linking architectural and component level system views by abstract state machines. In C. Grimm, editor, *Languages for System Specification and Verification*, CHDL, pages 247–269. Kluwer, 2004.
7. E. Börger. From Finite State Machines to Virtual Machines (Illustrating design patterns and event-B models). In E. Cohors-Fresenborg and I. Schwank, editors, *Präzisionswerkzeug Logik–Gedenkschrift zu Ehren von Dieter Rödding*. Forschungsinstitut für Mathematikdidaktik Osnabrück, 2005. ISBN 3-925386-56-4.
8. E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer-Verlag, 2003.
9. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
10. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
11. M. F. et al. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2004. Web pages at `http://www.martinfowler.com/eaaCatalog/`.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
13. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
14. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 2004.