Modeling Context-Aware Behavior by Ambient ASMs

Behavioral Programming Patterns Case Study

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 4.3 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de
Theme: Modeling context-aware system behavior

General question: How to gently model contexts and programs whose run behavior depends on the context in which the program is executed?

- NB. Often contexts are called environments, not to be confused with \( env \) understood as interpretation of free variables

In this lecture we use behavioral programming patterns

- source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns (Addison-Wesley 1994)

as examples to model context-aware behavior by ambient ASMs

- we recapitulate motivation and definition of ambient ASMs

General idea: \textit{use parameterization} to model context dependency.
Parameterizing ASM functions and rules permits to directly model:

- partitioning/isolating of states and distributed computations
  - of agents which concurrently execute in heterogeneous contexts
- various forms of information hiding (encapsulation of memory)
  - statically: scope, module, package, library, etc.
  - dynamically: executing agents, threads, process instances, etc.
- patterns of programming, of communication and of control flow
- mobility (environments where agents can move)
- modularity (of specs and property verifications)

Exploit *simplicity of semantical foundation* of parameterization:

\[ f(x) = f_{params}(x) \]

in particular when used with implicit (hidden) parameters, supporting conventional implicit oo parameterization \( this.f(x) = f(x) \)
Ambient ASMs to explicitly support env-sensitive actions

Idea: enrich ASMs by an abstract ambient parameter
- with respect to which the terms involved in a step are evaluated
- for ambients (environments) at whatever level of abstraction
- which can be created, modified, deleted, also at run time

Syntactical construct: \( \text{amb } \text{exp} \text{ in } P \)

where to achieve generality in the widest terms
- \( \text{exp} \) is any expression (term)
- \( P \) is any (already defined) ASM program

Intended behavior (see definition below):
- **Push** the \( \text{eval}(\text{exp}, S, \text{env}, \text{amb}) \) of the newly declared ambient expression to the current ambient
- execute \( P \) in the new ambient

Function classification is extended by ambient (in)dependent fcts.
AmbDependent(f) (wrt Amb) iff

forsome $a, a' \in \text{Amb}$ with $a \neq a'$ forsome $x$ $f(a, x) \neq f(a', x)$

Otherwise $f$ is called $\text{AmbIndependent}$. We also say $\text{environment (in)dependent}$, hoping that $\text{Ambients}$ will not be confused with $\text{Environments}$ (interpretation of free variables in a state).

- **Case AmbDependent(f):**
  \[
  \text{eval}(f(t_1, \ldots, t_n), S, \text{env}, \text{amb}) =
  f_S(\text{amb}, \text{eval}(t_1, S, \text{env}, \text{amb}), \ldots, \text{eval}(t_n, S, \text{env}, \text{amb}))
  \]
  Here $f_S$ is turned into a family of possibly different functions $f_{S, \text{amb}}$.

- **Case AmbIndependent(f) (unchanged interpretation of $f$):**
  \[
  \text{eval}(f(t_1, \ldots, t_n), S, \text{env}, \text{amb}) =
  f_S(\text{eval}(t_1, S, \text{env}, \text{amb}), \ldots, \text{eval}(t_n, S, \text{env}, \text{amb}))
  \]
Semantics of ambient ASMs

To avoid a signature blow up by dynamic ambient nesting, we adopt Simone Zenzaro’s idea (PhD Thesis, Pisa 2016) to treat \( \text{amb} \) as a stack where new ambient expressions are pushed (passed by value) so that for each \( f^{(n)} \) one extension \( f^{(n+1)} \) suffices which offers an additional argument position for the interpretation of \( f^{(n)} \) in a given ambient.

The body of \( \text{amb\ exp\ in\ } P \) is then executed with the new stack value.

\[
\text{Yields}(\text{amb\ exp\ in\ } P, S, \text{env}, \text{amb}, U) \text{ if } \text{Yields}(P, S, \text{env}, \text{PUSH}(\text{eval}(\text{exp}, S, \text{env}, \text{amb}), \text{amb}), U)
\]

- NB. Often the execution of \( P \) (read: the interpretation \( f_{S,\text{amb}} \)) depends only on the top of the stack, i.e. on \( \text{eval}(\text{exp}, S, \text{env}, \text{amb}) \) (\textit{flat ambient ASMs}, see JCSS 2012).
Special case: flat ambient ASMs

Only the last declared (most recent) ambient is kept

- instead of an ambient stack (of nested ambients) to which each newly declared environment is **Pushed**
  - This was the original definition of ambient ASMs in JCSS 2012, which is generalized by the stack interpretation of ambients.

For flat ambients it suffices to bind the value of the *expression* in the ambient declaration

- which is computed in the current state (in the current ambient)

to a logical variable, say `curamb`:

\[
(\text{amb}_{\text{flat}} \ exp \ \text{in} \ P)^* = (\text{let} \ curamb = \ exp^* \ \text{in} \ P^*)
\]

- where `exp^*` is obtained from `exp` by replacing the names `f` of ambient dependent functions in `exp` by `f_{curamb}`
Inductive transformation of flat ambient ASMs

**transformation of terms:**

- for AmbIndependent($f$) define:
  \[(f(t_1, \ldots, t_n))^* = f(t_1^*, \ldots, t_n^*)\]

  logical variables and names of parameterized rules are classified as ambient independent

- for AmbDependent($f$) define:
  \[(f(t_1, \ldots, t_n))^* = f(curamb, t_1^*, \ldots, t_n^*)\]

**transformation of ambient ASM rules** (to standard ASM programs):

- for update rules define:
  \[(f(t_1, \ldots, t_n) = t)^* = (f(t_1, \ldots, t_n)^*: = t^*)\]

- for ambient rules define (eliminating \texttt{ambflat}):
  \[\texttt{(ambflat } exp \texttt{ in } P)^* = (\texttt{let } curamb = exp^* \texttt{ in } P^*)\]

- for the other rules use induction on ASM programs, e.g.
  \[\texttt{(let } x = t \texttt{ in } P)^* = (\texttt{let } x = t^* \texttt{ in } P^*)\]
Traditional classification of oo design patterns

by purpose:
- structural patterns concerning composition of classes and objects
- creational patterns concerning creation of objects upon class instantiation
- behavioral patterns concerning interaction of classes and objects (flow of control, actions, communication and cooperation)

by scope:
- class patterns dealing with static (compile-time relevant) composition of (sub)classes (inheritance, accessibility, use relations)
- object patterns dealing with dynamic composition of run-time objects

Using the ASM Method we lift the object-oriented subclassing and inheritance view to the more general (mathematically precise) ASM refinement concept, using ambient ASMs.
We illustrate the parameterization of (sub)machines or operations by implicit arguments by the Delegation pattern and some of its refinements:

- Delegation with various refinement examples
  - Structural examples: Template, Proxy
    - Proxy with remote, virtual and protection version
  - Behavioral examples: (Chain of) Responsibility, Bridge
  - Incremental refinement: Decorator
Ambient ASM definition of Delegation Pattern behavior

Delegation ‘separates’ instantiations of an OPERATION

- via an abstract Class interface OPERATION from implementations of OPERATION in a concrete DelegateClass

s.t. at run-time for a call of OPERATION\((x)\) one can determine a delegate in DelegateClass to execute its OPERATION instance

- i.e. the implementation provided in DelegateClass

\[
\text{Delegate(OPERATION, delegate)}(x) = \text{amb delegate in OPERATION}(x)
\]

NB. The pattern permits multiple DelegateClasses.

Pattern variations result from different ways to define delegate.

\footnote{All pattern figures below are © 2018 Springer-Verlag, reused with permission}
Instances of **Delegate** pattern

Alternatives to define *delegate*:

- **internally**: define *delegate* as a location
  - in the abstract *Class* (**Bridge** pattern)
  - in some dedicated subclass to ‘provide a placeholder for another object’ so that *delegate* is ‘the real object that the proxy represents’ (**Proxy** pattern and its refinements)

- **externally**: define *delegate*
  - statically:
    - determined by the class structure (**Template** pattern)
    - determined by a data-structure related function (like the chain traversal function in **ChainOfResponsibility** pattern)
  - dynamically (**Responsibility** pattern (using run-time determined selection function), **Bridge** and emph Decorator patterns)
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.

Idea: use *delegate* as denoting a subclass *ConcreteClass* of *AbstractClass*, determined by the static subclass structure.
NB. In an ASM, arbitrary ambient expressions are permitted, e.g. `ConcreteClass`.

\[ \text{TEMPLATE} = \text{DELEGATE \ where \ delegate} = \text{ConcreteClass} \]

**OPERATION**: ‘the skeleton of an algorithm’ (an ‘Application’), may call some abstract `PrimitiveOperations`

- interfaces in `AbstractClass` are implemented (as individual ‘MyApplication’) in a `ConcreteClass`

  – which provides its interpretation \( op(\text{ConcreteClass}, x) \) of the abstract `PrimitiveOperations` \( op(x) \) ‘to carry out subclass-specific steps of the algorithm’: an ASM refinement of type \((1,1)\)
Goal:

avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request in particular when static association of caller with delegate is impossible.

It suffices to select a delegate among Receivers in subclasses Handler 1, ... , Handler n of the AbstractClass which CanHandle the input request
Responsibility pattern: a data-refinement of Delegate

Responsibility Pattern Class Structure:

Idea: select a delegate among Receivers in subclasses Handler i (1 ≤ i ≤ n) of the AbstractClass which CanHandle the input request

Responsibility = Delegate

where delegate = select(PossibleHandler(x))

PossibleHandler(request) = Receiver(request) ∩ Handler(request)

Handler(request) = {o | CanHandle(o, OPERATION)(request)}
**ChainOfResponsibility**: data-refined Delegate

Requirements:

‘chain the receiving objects and pass the request along the chain until an object handles it ... the handler should be ascertained automatically’

This can be achieved by a data refinement:

- of the non-deterministic choice fct `select in Responsibility`
- to a deterministic function which with respect to an order `< (‘chain’) of Receiver(request) provides the first element that CanHandle the input request

$$\text{ChainOfResponsibility} = \text{Delegate}$$

where `delegate = first(PossibleHandler(x))`

NB. To ‘ascertain’ the handler ‘automatically’ means to program the `delegate` function. function
Proxy intended to ‘provide a surrogate or placeholder for another object to control access to it’ such ‘that a Proxy can be used anywhere a RealSubject is expected’.

Class structure:

delegate becomes ‘the real object that the proxy represents’ and thus is renamed to RealSubject

- value of delegate is a ConcreteClass instance
  - kept in a placeholder location of the dedicated Proxy subclass
Via *Proxy*, client calls of *Operation* are forwarded to *delegate*—which is passed as ambient parameter to the refinement of *Operation*

- i.e. to its implementation in the `classOf(delegate)`

**Proxy** = **Delegate** where *delegate* denotes a location of *Proxy*

with values in any subclass *ConcreteClass* of *AbstractClass*
Further data refinements of Proxy pattern

Refinements by constraints on where the values of delegate are stored or on the access to delegate:

- in a RemoteProxy the delegate location is required to be in a different address space
- in a VirtualProxy the delegate value is cached via some operation Cache(delegate, request) so that its access can be postponed
- in a ProtectionProxy it is checked whether the caller has the permission to access the Operation in classOf(delegate)

Example: LeadElection ASM built out of abstract components Propose, ReceiveMsg, UpdateKnowledge.

These components can be refined to compute

- the leader
- the leader plus notification of the termination
- the leader together with a shortest path to it
**Bridge pattern: an instance of Delegate**

*delegate* declared as *AbstractClass* location, its values are instances of outsourced implementing subclasses of an *Implementor* class

- with a link relating the interface *OPERATION* in *AbstractClass* to the interface *OPERATIONIMPL* in *Implementor* class

\[
\text{Bridge}(\text{OPERATION}, \text{delegate}) = \\
\text{Delegate}(\text{OPERATIONIMPL}, \text{delegate})
\]

Run-time choices bw *OPERATION* refinements via updates of *delegate* replace static binding of implementations via class inheritance.
Requirements:

‘attach additional responsibilities to an object dynamically’ as ‘a flexible alternative to subclassing for extending functionality’

- leaving the \textit{Operation} behavior of other class instances unchanged

\begin{itemize}
  \item \textbf{Idea:} keep ‘a reference to a \textit{Component} object’ in a \textit{delegate} location of a subclass \textit{Decorator}, an ‘interface for objects that can have responsibilities added to them dynamically’
  \item one \textit{ConcreteDecorator} subclass for each \textit{AddedBehavior}
\end{itemize}
As ASM program, `Decorator` is identical to `Delegate`:  

```asm
Decorator(Operation, delegate) =
    amb delegate in Operation
where delegate denotes a location of Decorator
    with values in any subclass ConcreteDecorator of Decorator
```

The difference is in how the parameterization of `Operation` by the environment is defined:

- In `Decorator`, the implementation of `Operation` extends the `Component` `Operation` by `AddedBehavior`.
- In `Delegate`, there is no extension of the implementation of `Operation`.

NB. If the pattern only extends functionality without overriding the behavior of `Operation`, it represents an instance of incremental (‘conservative’) ASM refinements.
Work to be done

- model more complex not oo-programming-centric patterns, e.g.
  - J2EE Core Patterns (Alur et al., Prentice Hall 2003)
  - Patterns of Enterprise Application Architecture (Fowler et al., Addison-Wesley 2004)
  - Enterprise Integration Patterns (Hohpe & Woolf, Addison-Wesley 2004)
  - WS-CDL Service Interaction Patterns (Barros et al. 2005)
- define genuine modeling-patterns and their refinement
On ambient ASMs:
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de
  – Ch.4 contains other applications of ambient ASMs for context-aware system models and further references
- E. Börger and A. Cisternino and V. Gervasi: Ambient Abstract State Machines with Applications.

On oo programming patterns:
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns (Addison-Wesley 1994)
It is permitted to (re-) use these slides under the CC-BY-NC-SA licence

https://creativecommons.org/licenses/by-nc-sa/4.0/

i.e. in particular under the condition that

- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial