Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling Contex-Aware Behavior by Ambient ASMs

Thread Behavior and Thread Management Case Study

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

See Ch. 4.2 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Theme: Modeling context-aware system behavior

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

- Often contexts are called environments, not to be confused with env understood as interpretation of free variables

In this lecture we use the following 2 examples:

- *Encapsulation of process runs* to separate concurrent process executions, e.g.
  - runs of threads in Java
  - computations of instances of processes which execute similar or even the same program but with different data
- *Encapsulation of state* by scoping disciplines

Idea: *use parameterization* to model context dependency.

The role of parameterization for ASMs

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: **amb** exp **in** P

where to achieve generality in the widest terms

- exp is any expression (term)
- P is any (already defined) ASM program

Intended behavior (see definition below):

- PUSH the $eval(exp, S, env, amb)$ 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.

Ambient dependent functions and their evaluation

$AmbDependent(f)$ (wrt Amb) **iff**

forsome $a, a' \in Amb$ **with** $a \neq a'$ **forsome** x $f(a,x) \neq f(a',x)$

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

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

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 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 **amb** exp **in** P is then executed with the new stack value.

$Yields($**amb** exp **in** $P, S, env, amb, U)$ **if**

$$Yields(P, S, env, \text{PUSH}(eval(exp, S, env, amb), amb), U)$$

- NB. Often the execution of P (read: the interpretation $f_{S,amb}$) depends only on the top of the stack, i.e. on $eval(exp, S, env, amb)$ (*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 *exp*ression in the ambient declaration

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

to a logical variable, say $curamb$:

$$(\textbf{amb}_{\textbf{flat}}\ exp\ \textbf{in}\ P)^* = (\textbf{let}\ curamb = exp^*\ \textbf{in}\ P^*)$$

- 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 $\mathbf{amb_{flat}}$):

    $$(\mathbf{amb_{flat}}\ exp\ \mathbf{in}\ P)^* = \ (\mathbf{let}\ curamb = exp^*\ \mathbf{in}\ P^*)$$

  - for the other rules use induction on ASM programs, e.g.

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

Encapsulating computations in distributed runs

Goal: *isolate executions of various tasks by different agents*

- Technical issue: separate concurrent (e.g. multi-core) executions and scheduling from task execution by single agents
- In this lecture we illustrate how to use ambient ASMs for the purpose by two concrete examples (though the principles can be applied also to other distributed systems and concurrent request management schemes):
  - using a component which encapsulates single-thread executions of Java code to model concurrent runs of multiple Java threads
  - modeling Java thread pool management (for J2SE 5.0), using a component which encapsulates to $\textsc{Run}$ a $thread$ to $\textsc{Execute}$ a $task$

From single-thread to multi-thread Java interpreter

Goal: Define a concurrent MultiThreadJavaInterpreter model which separates the thread instances of the underlying runs of a SingleThreadJavaInterpreter.

A SingleThreadJavaInterpreter ASM has been defined in:

- R. Stärk and J. Schmid and E. Börger, *Java and the JVM*. Definition, Verification, Validation. Springer 2001. (Called JBook)

It has been used there (Ch. 7) to define a multi-thread Java interpreter model to execute multiple tasks with shared main and local working memory. It adopts an abstract scheduling mechanism which

- selects each time one *Runnable* thread, out of the current *Thread* class instances in the *heap*, to Run it
- makes a *Synchronizing* or *Notified* thread *Active* before Running it

Using ambient ASMs saves the explicit restoring/saving of the current state of a thread when the thread is scheduled to Run *or descheduled.*

Encapsulating single-thread Java code executions

$\textsc{MultiThreadJavaInterpreter} =$

if $Runnable(t)$ **then**

$\textsc{AcquireLocks}(t)$ **seq** $\textsc{Run}(t)$

where

$\textsc{Run}(t) =$ **amb$_{\textbf{flat}}$** t **in** $\textsc{SingleThreadJavaInterpreter}$

Restoring/saving current thread state can be skipped by declaring:

- $Amb = Thread$ (i.e. implicit parameterization by threads)
- as $AmbDependent$ the thread state functions used in the ASM $\textsc{SingleThreadJavaInterpreter}$, namely:
  - $meth, restbody, pos, locals$ constituting the current frame
  - $frames$ denoting the frame stack
  - $thread$ denoting the executing thread

Mono-core vs multi-core model

- The JBook *execJavaThread* ASM chooses one $t \in Runnable$ and to $\textsc{Run}(t)$ assigns it as the currently executing $thread := t$
  – In JBook *thread* is the agent which executes the $\textsc{SingleThreadJavaInterpreter}$ (single-core view)
- The $\textsc{MultiThreadJavaInterpreter}$ ASM as defined here triggers a *concurrent ASM run of all threads which are* $\textsc{Run}$*ning.*
  – NB. $\textsc{Run}(t)$ can be refined by assigning this execution to a specific computer or core.
  – NB. *Thread* and *Runnable* are dynamic sets.

NB. *Ambient separation supports modular verifications*:

- see ASM-based analysis of C# thread model (LNCS 3052, TCS 343)
- see proofs for conservative theory extensions corresponding to incremental model extensions in D. Batory/E. Börger: Modularizing Theorems for Software Product Lines: The Jbook Case Study. J.UCS 2008

Submachines of $\textsc{MultiThreadJavaInterpreter}$

Pro memoria definitions from the JBook:

$Runnable(t)$ **iff**

$\quad mode(t) = active$

$\qquad$ **or** $(mode(t) = synchronizing$ **and** $locks(syncObj(t)) = 0)$

$\qquad$ **or** $(mode(t) = waiting$ **and** $locks(waitObj(t)) = 0)$

$\textsc{AcquireLocks}(t) =$

$\quad$ **if** $mode(t) = synchronizing(t)$ **then** $\textsc{Synchronize}(t)$

$\quad$ **if** $mode(t) = notified(t)$ **then** $\textsc{WakeUp}(t)$

$\quad Active(q) := true$

$\textsc{Synchronize}(t) =$ Refresh $sync(t)$ by $syncObj(t)$

$\textsc{WakeUp}(t) =$ Reaquire all sync claims on $waitObj(t)$

Isolating application-logic task exec from task management

Goal: separate thread management

- creation, deletion and scheduling of threads t to concurrently run tasks
  - assign t to task, decouple t from task, suspend t

and its specification from the execution and application-logic-level specification of tasks

Exl: J2SE 5.0 (see S. Oaks and H. Wong: *Java Threads*, O'Reilly 2004)

NB. Principles can be applied also to other concurrent request management systems, e.g. web servers

Thread management actions

- assignment of threads to tasks upon TaskEntry
- decoupling of threads from tasks upon TaskCompletion
- creation of threads
- suspension of threads
  - making them idle to possibly RunTaskFromQueue
- deletion of threads
  - if one cannot any more RunTaskFromQueue so that the thread has to Exit

J2SE 5.0 thread pool requirements (1)

corePoolSizeReq. The thread pool should be kept as much as possible within $corePoolSize$. When a new task is submitted and fewer than $corePoolSize$ threads are running, a new thread is created to handle the request, even if there are idle threads. If when the task is submitted the pool has reached or exceeds the $corePoolSize$ but not the $maxPoolSize$ and there are idle threads in the pool, one of them is assigned to run the task.

maxPoolSizeReq. If a task is submitted for execution when the pool is full and all threads in the pool are running, the task is inserted into a $queue$. If the $queue$ is full the task is rejected.

QueuePriorityReq. If a task is submitted for execution when $corePoolSize$ or more threads are running but the pool is not yet full and no idle thread is available, then a new thread is created and assigned to run the task only if the task cannot be placed to the $queue$ without blocking it.

J2SE 5.0 thread pool requirements (2)

RunCompletionReq. If when a thread has completed its current run there are tasks in the queue, the thread is assigned to run one of them. Otherwise it becomes idle.

IdleThreadReuseReq. When there is a task in the queue, an idle thread, if there is one, is assigned to run the task.

IdleThreadExitReq If there are more than $corePoolSize$ threads in the pool, excess threads will be terminated if they have been idle for more than the $keepAliveTime$.

ThreadPoolMngr

- first three requirements ask to HandleNewTasks
- last three concern idle threads: how to HandleQueuedTasks

In both an idle thread may be assigned to run the task in question, but *only one task per thread*:

- abstract from scheduling by letting the ThreadPoolMngr choose in each step one of the two components for execution

ThreadPoolMngr =

  one of ({HandleNewTask, HandleQueuedTask})

where

  one of $(Rules) =$

    choose $R \in Rules$ **do** R

$\textsc{HandleNewTask}(task)$ **component**

$\textsc{HandleNewTask}(task) =$ **if** $Submitted(task)$ **then**

if $\mid Pool \mid < corePoolSize$ -- first create $corePoolSize$ threads

then let $t =$ **new** $(Pool)$ **in** $\textsc{Run}(t, task)$

else if $\mid Pool \mid < maxPoolSize$ **then** -- first try $Idle$ threads

choose $t \in Pool$ **with** $Idle(t)$ $\textsc{Run}(t, task)$

if none if $BlockingFreePlaceable(task, queue)$

then $\textsc{Insert}(task, queue)$ -- first fill $queue$

else let $t =$ **new** $(Pool)$ **in** $\textsc{Run}(t, task)$

else if forall $t \in Pool$ $Running(t)$ **then**

if $\mid queue \mid < maxQueuesize$ **then** $\textsc{Insert}(task, queue)$

else $\textsc{Reject}(task)$

where $\textsc{Run}(thread, task) = (mode(thread) := running$ **par**

amb$_{\textbf{flat}}$ $task$ **in** $\textsc{Execute}(pgm(task)))$

HandleQueuedTask component

The *RunCompletionReq*uirement seems to establish a reuse priority of *Terminated* threads over idle ones. Thus we try:

- first to FindTaskForTerminated threads satisfying the RunCompletionReq
- then to FindTaskForIdle threads satisfying the two idle thread requirements

HandleQueuedTask =
choose $thread \in Pool$ **with** $Terminated(thread)$
FindTaskForTerminated($thread$)
if none
choose $thread \in Pool$ **with** $Idle(thread)$
FindTaskForIdle($thread$)

$Terminated(t)$ **iff** $mode(t) = idle$ expresses that a $thread$ has completed the run for the assigned task.

$\textsc{FindTaskForTerminated}(thread)$ submachines

NB. If there is no task in the queue, a terminated $thread$ becomes idle.

$\textsc{FindTaskForTerminated}(thread) =$

choose $task \in queue$

$\textsc{Run}(thread, task)$

$\textsc{Delete}(task, queue)$

if none

$\textsc{MakeIdle}(thread)$

where

$\textsc{MakeIdle}(thread) =$

$mode(thread) := idle$

$terminationTime(thread) := now$

-- set timer for $keepAliveTime(thread)$

$\textsc{FindTaskForIdle}(thread)$ submachine

NB. An idle $thread$ may be killed if the pool grew over the $corePoolSize$ and the $keepAliveTime(thread)$ has expired

$\textsc{FindTaskForIdle}(thread) =$

choose $task \in queue$

$\textsc{Run}(thread, task)$

$\textsc{Delete}(task, queue)$

if none $\textsc{TryToKill}(thread)$

where

$\textsc{TryToKill}(t) =$

if $\mid Pool \mid > corePoolSize$ **then**

if $Expired(aliveTime(t))$ **then** $\textsc{Delete}(t, Pool)$

$Expired(aliveTime(t))$ iff

$now - terminationTime(t) > keepAliveTime(t)$

Encapsulating state: scoping disciplined evaluation scheme

Value of id in a $state$ depends on

- the position in the program where id occurs
- an env among those which have a binding for id (i.e. a defined associated value) and whose $scope$ (program part where its bindings are valid) includes pos

$\textsc{EnvSensitiveEval}(id, pos, state) =$

  if $Occurs(id, pos, state)$ **then**

    let $E = Env \cap \{e \mid inScope(pos, e, state) \wedge HasBinding(e, id)\}$

      let $env = select(E)$ -- stands for **choose** $env \in E$

        amb env **in** $\textsc{GetValue}(id)$

$\textsc{GetValue}$ retrieves the value of an id given an env.

Variations: refining $select$ function and $inScope$ predicate

- lexical scoping (Pascal,C), dynamic scoping (Logo), combination of lexical and dynamic scoping (Java)

Encapsulating state: scoping discipline instances

Using $select$ to shadow bindings in case id is bound in multiple envs all of which are valid at pos

Example: stack-based last-in/first-out (LIFO) scope selection

$\textsc{EnvSensitiveLifoEval}(id, pos, state) =$

$\quad \textsc{EnvSensitiveEval}(id, pos, state)$

where $select(E)$ is constrained to yield an env satisfying

$\quad$ **forall** $e \in E \; env \sqsubseteq e$ -- language defined $\sqsubseteq$

Encapsulating state: TCL scoping discipline

TCL: allows to select any binding of names established in any enclosing scope (possibly hidden by a nearer scope)

$\textsc{TclEval}(id, pos, state) =$ **if** $Occurs(id, pos, state)$ **then**
  let $e = selectenv(pos, state)$ **amb** e **in** $\textsc{GetValue}(id)$
where let $n = length(Env)$

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

`upvar` k id v binds v to id as bound in the k-th scope "up" from the current scope

dto `upvar` #k id v with k-th scope "down" from the global one

References

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 contex-aware system models and further references
- E. Börger and and A. Cisternino and V. Gervasi: Ambient Abstract State Machines with Applications.
  J. Computer and System Sciences 78.3 (2012) 939-959.

On multi-threaded Java and Java thread management:

- R. Stärk and J. Schmid and E. Börger, Java and the JVM. Definition, Verification, Validation. Springer 2000
- S. Oaks and H. Wong: Java Threads. O'Reilly 2004

References on modular property verifications

- R. F. Stärk and E. Börger: An ASM Specification of C# Threads and the .NET Memory Model. LNCS 3052 (2004) 38-60
- R. F. Stärk: Formal specification and verification of the C# thread model.Theoretical Computer Science 343 (2005) 482–508

- D. Batory and E. Börger: Modularizing Theorems for Software Product Lines: The Jbook Case Study. J.UCS 2008

Copyright Notice