

Defining CONTROL in the System Model

OO vs. statemachines vs. process

Manfred Broy, Victoria Cengarle, Bernhard Rumpe

Technische Universitäten München und Braunschweig

Objects and Statemachines

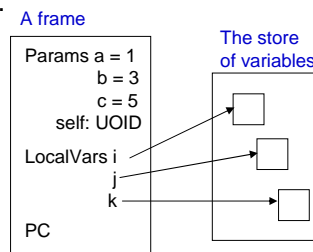
- Objects provide a successful programming paradigm
- Statemachines are a successful specification paradigm
- However there is a mismatch:
- Objects allow method calls
 - Method invocations open up new variables
 - (Parameters and local variables)
 - Method invocations can be recursive
 - The number of local variables is unbounded
- State in Statemachines has to comprise that.

State

- State of the system consists of three parts:
 - $STATE = DataStore \times ControlStore \times MessagePool$
- Control comprises:
 - Stacks
 - Threads
 - Interaction between threads
 - Relation to objects
- We have two extremes:
 - forget about the control store: Too abstract
 - include all details mentioned above: No abstraction, but detailed

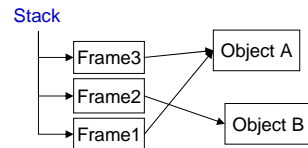
A stack frame of a method

- Let **METHOD** be the universe of methods.
- Each method m comes equipped with
 - a class where it belongs to
 - **classOf**: $METHOD \rightarrow CLASS$
 - a record of parameters
 - **parOf**: $METHOD \rightarrow UTYPE$
 - a record of local variables
 - **localsOf**: $METHOD \rightarrow UTYPE$
 - a set of possible PC-states
 - **pc**: $METHOD \rightarrow PC$
- a frame with parameters and local vars is of form
 - $Rec(self: C, par_1: TP_1, \dots, par_n: TP_n, var_1: Var TV_1, \dots, var_n: TV_n)$
 - **PC** denotes the universe of program counters
- We do not need to deal with shadowed variables or partial visibility in blocks: this is statically resolvable



Sequential case: single stacks

- All frames are from the universe
- $\text{FRAME} = \text{PC} \ \& \ \{ \text{Rec}(\text{self}: C, \dots) \}$
- Each method then has a frame structure describing which frames it may have
 - **framesOf**: $\text{METHOD} \ ! \ \varnothing(\text{FRAME})$
- A single stack is a stack of frames
 - **Stack FRAME**
- Each frame belongs to an object:



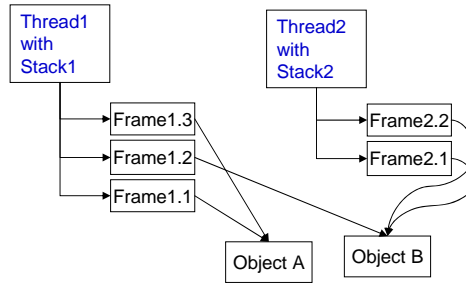
Concurrent case: threads with multiple stacks

- Let **THREAD** be the universe of threads.
- Assumptions:
 - concurrency is not necessarily connected to objects, but orthogonal
 - multiple threads per objects
 - multiple objects may belong to one process/thread
 - “active” and “passive” objects
- But: there are (simple) **actions** that are **atomic**

Concurrent case: threads with multiple stacks

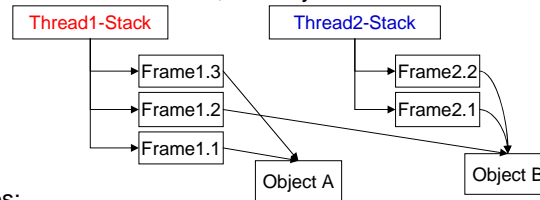
- Let **THREAD** be the universe of threads.
- A system snapshot is then given by the known
 - DataStore
- and the
 - ControlStore = (THREAD ! Stack FRAME)
- with one Stack per Thread

Remark: This is version1:
which will be extended later

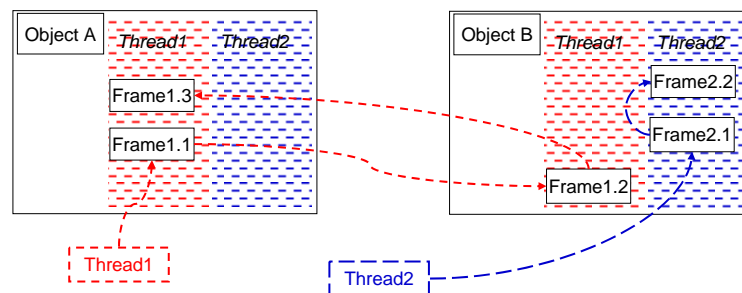


Localized view on the control structure

- Instead of centralized stacks, we may distribute stack frames over objects:

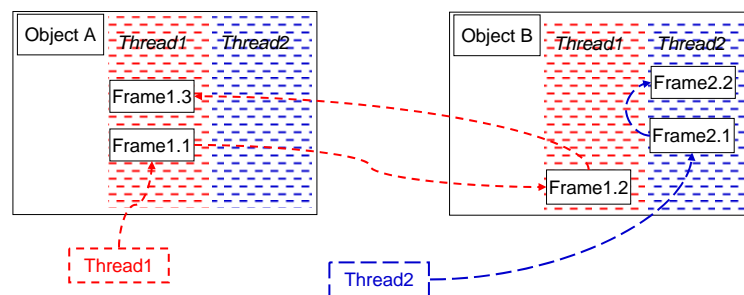


becomes: _____



Localized view on the control structure

- Instead of centralized stacks, we may distribute stack frames over objects:
- Formally
 - $\text{ControlStore} = \text{UOID} \times \text{THREAD} ! \text{Stack FRAME}$
- describing the frames that belong to an object and a thread

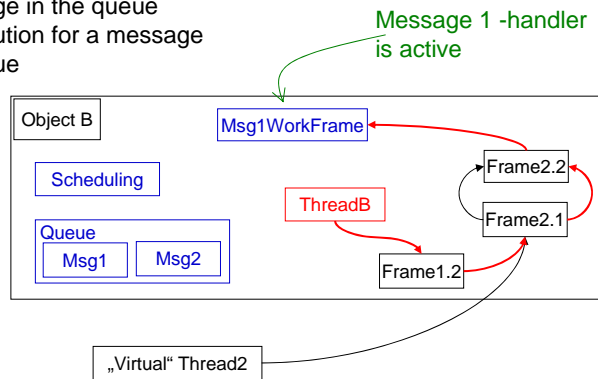


Messages and message passing

- Let **MESSAGE** be the universe of messages.
- Classes can accept messages (if not, the set is just empty)
 - $\text{messOf: CLASS} ! \emptyset (\text{MESSAGE})$
- Message consist of a
 - name, sender, receiver, arguments, ...
 - or is a timeout indicator,
- Objects deal with messages by
 - queueing and prorizing
 - starting a method to handle the message
- Consequence: some **objects are active**, by providing a **local thread** that handles messages

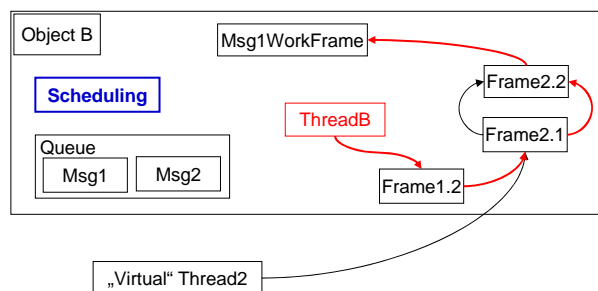
The object in total ...

- has three components:
 - DataStore = (UVAR → UVAL)
 - ControlStore = UOID × THREAD ! Stack FRAME
 - MsgQueue = Queue MESSAGE
- and there are actions like
 - adding a message in the queue
 - starting an execution for a message
 - rearranging queue



Integrating calls and messages ... (and exceptions, returns, timeouts)

- Calls are integrated by viewing them as messages with special purpose. A **call** consists of
 - a **call-message** of type
 - Rec(sender: UOID, receiver: UOID, thread: THREAD, par1: ...)
 - a **return-message** of type
 - Rec(sender: UOID, receiver: UOID, thread: THREAD, result: UVAL)
- Calls are scheduled together with ordinary messages: This gives us enormous flexibility on priorities, run to completion, concurrency, etc.



Conclusions

- 1) State machines can describe OO procedural communication
- 2) It's clumsy
- 3) There is no other way to cope with the mismatch between
 - Object recursion and
 - State Transition Systems

Fin.

- Questions, comments and suggestions welcome.