Egon Börger (Pisa) & Alexander Raschke (Ulm)

Refining Synchronous to Asynchronous ASMs

Extrema Finding Example

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

See Ch. 3.1 of Modeling Companion

Goal

Illustrate how multi-agent ASMs allow one to pass by an ASM refinement step

- from a synchronous (easy to grasp) understanding
- to an asynchronous (harder to check) view and from there
- to a CoreASM executable version of algorithms/systems

Example: Franklin's leader election algorithm ExtremaFinding

- finitely many processes arranged in a bidirectional ring find out who is the largest among them
  - efficiently, without any central control and passing information only to the respective two neighbors

Randolph Franklin: On an improved algorithm for decentralized extrema finding in circular configurations of processors.
Commun. ACM 1982, 25(5) 336-337

Multi-Agent ASMs

A multi-agent ASM $\mathcal{M}$ is a family

$$(ag(p), pgm(p))_{p \in Process}$$

of single-agent ASMs consisting of a set of $Process$es p viewed as

- agents $ag(p)$ which execute step by step ('sequentially')
- each its program $pgm(p)$, a finite set of ASM rules
- interacting with each other via reading/writing in designated (shared or input/output) locations

NB. ag and pgm may be dynamic functions.

A concurrent run of a multi-agent ASM $\mathcal{M}$ is a sequence $(S_0, P_0), (S_1, P_1), \ldots$ of states S_n and subsets $P_n \subseteq Process$ such that each state S_{n+1} is obtained from S_n by applying to it the updates computed by the processes $p \in P_n$

$\textsc{ConcurrencyPattern}(Process) =$

forall $p \in Process$ $\textsc{ConcurStep}(p)$

$\textsc{ConcurStep}(p) =$ -- may be publicly visible or local

if $mode = public$ **then choose** $step \in \{publicStep, publicRead\}$

if $step = publicStep$

then $\textsc{PublicExec}(p)$ -- read&write interaction (synchronized)

else

$\textsc{ReadPublicData}(p)$ -- read interaction (synchronized)

$mode := local$ -- switch to local (not publicly visible) step

if $mode = local$ **then choose** $step \in \{localStep, publicWrite\}$

if $step = localStep$ **then** $\textsc{LocalExec}(p)$ -- asynchronous step

else

$\textsc{WriteBack}(p)$ -- write interaction (synchronized)

$mode := public$ -- switch to next read/write interaction

Extrema Finding Requirements (1)

Plant&FunctionalReq. Finding without a central controller the largest element in a bidirectional ring of processes whose size N is not known in advance.

BeingActiveReq. We will define an inactive process as one that knows that it is not the largest; the other processes are active.

ActiveNeighbReq. The two neighbors of an active process are those active processes closest to it in each direction along the ring. In the degenerate case of a ring with only two active processes, each becomes the two neighbors of the other; similarly, if there is only one active process, it becomes both of its neighbors.

ExchangeInfoReq. Each active process sends a message with its value to each of its neighbors and receives such messages from its two active neighbors. If either of the messages it receives is larger than its value, then it makes itself inactive.

Extrema Finding Requirements (2)

ForwardInfoReq. The process of sending a message to an active neighbor is apparently complicated by the fact that a given process does not know the exact locations of its active neighbors. This is, in fact, no problem if we pass messages by the convention that inactive processes simply pass on received messages from either direction in the same direction, while active processes do not.

StepReq. Thus, during each step every inactive process receives and forwards two messages, while each active process transmits and receives two messages.

TerminationReq. The repetition of steps terminates when in some step a process receives a message from itself; this implies that it is the only active process left and that its value is the largest of the set. As a final action, that process announces that fact to all the other processes in N message passes.

Static background structure

Plant&FunctionalReq Finding ... the largest element in a bidirectional ring of processes whose size N is not known in advance.

$Process = \{p_o, \ldots, p_{N-1}\}$ with

- linear order $>$ ('being larger than')
- ring structure:

  $l, r : Process \to Process$ satisfying for $0 \leq i < N-1$

  $l(p_0) = p_{N-1} \qquad l(p_{i+1}) = p_i$

  $r(p_i) = p_{i+1} \qquad r(p_{N-1} = p_0)$

- $mode_p \in \{active, inactive, terminated\}$. Initially $mode_p = active$
  - values $(in)active$ by *BeingActiveReq*
  - value $terminated$ to reflect the terminiation described by *TerminationReq*

Information exchange assumptions

- all processes perform their 'steps' synchronized in rounds
- in each round every process
  - receives two messages (by StepReq)
    - $fromRightMsg_p$ (initially $\mathbf{undef}$), sent (in the previous round) by the right neighbor (ExchangeInfoReq and ForwardInfoReq)
    - $fromLeftMsg_p$ (initially $\mathbf{undef}$) sent (in the previous round) by the left neighbor (ExchangeInfoReq and ForwardInfoReq)
  - sends (by StepReq transmits or forwards) two messages, one message to each neighbor (by ExchangeInfoReq and ForwardInfoReq)
- message delivery is
  - reliable: no message gets lost or corrupted
  - immediate: a message sent in a round by a process, to its left or right neighbor, is received and read by the addressee in the next round

Modeling the ExchangeInfoRequirement

$\textsc{ExchangeInfo} =$

if $mode = active$ **then**

$\textsc{TransmitInfo}(\textbf{self})$ -- see below for a restriction

$\textsc{CheckForLargerMsg}(\textbf{self})$

where

$\textsc{TransmitInfo}(p) =$ -- p sends 'its value'

fromRightMsg$_{l(p)} := p$ -- to its left neighbor

fromLeftMsg$_{r(p)} := p$ -- to its right neighbor

$\textsc{CheckForLargerMsg}(p) =$ -- p reads & checks received msgs

$\textsc{CheckWhetherLarger}(p, fromLeftMsg_p)$

$\textsc{CheckWhetherLarger}(p, fromRightMsg_p)$

$\textsc{CheckWhetherLarger}(p, msg) =$

if $msg > p$ **then** $mode_p := inactive$

Modeling the ForwardInfoReq for inactive processes

... inactive processes simply pass on received messages from either direction in the same direction, while active processes do not.

$\textsc{ForwardInfo} =$

  if $mode = inactive$ **then** $\textsc{PassMsgs}(\mathbf{self})$

where

  $\textsc{PassMsgs}(p) =$ -- 'in the same direction'

    $fromRightMsg_{l(p)} := fromRightMsg_p$ -- from right to left

    $fromLeftMsg_{r(p)} := fromLeftMsg_p$ -- from left to right

Modeling TerminationReq for active processes

- *... terminates when ... a process receives a message from itself*

  p 'receives a msg from itself' means $msg = p$.
- *this implies that ... its value is the largest of the set*

  We distinguish 'the largest of the set' by setting an attribute $recognizedAsLargest_p$ to true.
- *... final action ... announces that fact to all ... processes*

  We describe the termination 'announcement' 'to all the other processes' using a ForwardNotify component.
  - for simplicity, instead of msg passing we use a location $notified$ which is shared between p and its right neighbor

  'final action' is translated by $mode_p := terminated$

The CheckNotifyLeaderDetected(p) component expresses the above, together with a rule to forward the notification.

Leader Detection and Notification

$\textsc{CheckNotifyLeaderDetected}(p) =$

$\textsc{NotifyLeaderDetected}(fromLeftMsg_p)$

$\textsc{NotifyLeaderDetected}(fromRightMsg_p)$

$\textsc{NotifyLeaderDetected}(msg) =$

if $msg =$ **self then** -- process receives a msg from itself

$recognizedAsLargest_{\mathbf{self}} := true$ -- is the largest of the set

$\textsc{FinalAction}(\mathbf{self})$

$\textsc{FinalAction}(p) =$

$notified_{r(p)} := true$ -- right neighbor is *notified*

$mode_p := terminated$ -- terminates with 'final' action

The rule to forward the notification is as follows:

if $mode_p = inactive$ **and** $notified_p = true$ **then** $\textsc{FinalAction}(p)$

Result: Synchronous $\textsc{ExtremaFinding}$ ground model

TerminationReq requests to stop the 'repetition of steps' once a process has been $recognizedAsLargest$:

- for inactive processes disable $\textsc{ForwardInfo}$ rule when $notified$
- for active p guard $\textsc{TransmitInfo}$ by **not** $recognizedAsLargest(p)$

$\textsc{ExtremaFinding} =$ **forall** $p \in Process$

  if $mode_p = active$ **then**

    if not $recognizedAsLargest_p$ **then** $\textsc{TransmitInfo}(p)$

    $\textsc{CheckForLargerMsg}(p)$

    $\textsc{CheckNotifyLeaderDetected}(p)$

  if $mode_p = inactive$ **then**

    if $notified_p = true$ **then** $\textsc{FinalAction}(p)$

      else $\textsc{PassMsgs}(p)$

Data refinement leads to CoreASM executable model (Soldani 2014)

Why refine ExtremaFinding to an asynchronous ASM

- The *StepReq* leading to the **forall** synchronization easens the complexity analysis of the ExtremaFinding algorithm.
- But for a truly distributed algorithm
  – *Plant&FunctionalReq*: the algorithm should work 'without a central controller', excluding a common clock

  an asynchronous model appears to be more appropriate.
- In an asynchronous model
  – sending a message (via TransmitInfo or PassMsgs)
  – receiving a message and CheckForLargerMsg or check the leader detection in NotifyLeaderDetected

  happen without synchronization.

How to turn ExtremaFinding into an asynchronous ASM

Idea: refine $fromRightMsg$ and $fromLeftMsg$ locations to mailboxes $FromRightMsgs$ resp. $FromLeftMsgs$.

Assumptions on Concurrent Extrema Finding Runs (implicitly made already in the ExtremaFindingReq) to preserve correctness:

- Every process which is enabled will eventually perform a step.
- No msgs are lost or corrupted and msgs arrive in sending order.

Consequence:

- mailboxes $FromRightMsgs, FromLeftMsgs$ are queues
- $fromRightMsg := q$ is refined to $\textsc{Enqueue}(q, FromRightMsgs)$ (same with $Left$)
- readings of $fromRightMsg$ are refined to readings of $head(FromRightMsgs)$ (the same with $Left$)
- implicit $fromRight/LeftMsg$ overwriting refined to $\textsc{Dequeue}$

CONCURREXTREMAFINDING **program**

CONCUREXTREMAFINDING =

-- same structure as EXTREMAFINDING

```
if mode = active then
  if not recognizedAsLargest then  AsyncTransmitInfo
  forall q ∈ {FromLeftMsgs, FromRightMsgs}      -- for both queues
    if q ≠ [] then
      CheckWhetherLarger(self, head(q))
      NotifyLeaderDetected(self, head(q))
      Dequeue(q)                                  -- remove msg head(q)
if mode = inactive then
  if notified = true
    then FinalAction
    else  AsyncPassMsgs
```

Submachines of ConcurExtremaFinding

$\text{AsyncTransmitInfo}(p) =$ -- p sends 'its value'

$\quad \text{Enqueue}(p, FromRightMsgs_{l(p)})$ -- to the left neighbor

$\quad \text{Enqueue}(p, FromLeftMsgs_{r(p)})$ -- to the right neighbor

$\text{AsyncPassMsgs}(p) =$

$\quad$ **if** $FromRightMsgs_p \neq [\,]$ **then**

$\quad\quad \text{Enqueue}(head(FromRightMsgs_p), FromRightMsgs_{l(p)})$

$\quad\quad \text{Dequeue}(FromRightMsgs_p)$

$\quad$ **if** $FromLeftMsgs_p \neq [\,]$ **then**

$\quad\quad \text{Enqueue}(head(FromLeftMsgs_p), FromLeftMsgs_{r(p)})$

$\quad\quad \text{Dequeue}(FromLeftMsgs_p)$

For the CoreASM executable refinement see op.cit. Soldani 2014

Refinement correctness for ConcurExtremaFinding (1)

Let any pair of ExtremaFinding/ConcurExtremaFinding runs be given, started in equivalent initial states.

Due to the simple msg producer/consumer protocol among neighbors, the corresponding actions of interest are the following ones (disregard trivial notification subprocess):

- for active processes: corresponding send actions in TransmitInfo resp. AsyncTransmitInfo and corresponding checks in
  – CheckWhetherLarger(**self**, msg)
  – NotifyLeaderDetected(msg)

  concerning corresponding locations of interest with related updates
- for inactive processes pairs of an abstract send action (with implicit overwrite) and the corresponding refined Enqueue (together with the explicit Dequeue).

Correctness proof for ConcurExtremaFinding (2)

Since messages are not lost, arrive in order and with their original content, the update effects of corresponding actions on the corresponding locations of interest are equivalent.

- Note that one abstract double-check action CheckForLargerMsg may split in the refined model into two separate check-actions CheckWhetherLarger(**self**, $head(q)$), for each neighbor's mailbox q.
  - But this does not affect the combined result of the two actions.

The same holds for NotifyLeaderDetected.

The refinement correctness preserves the correctness of the abstract machine.

References

- Randolph Franklin: On an improved algorithm for decentralized extrema finding in circular configurations of processors.
  Commun. ACM 1982, 25(5) 336-337
- Jacopo Soldani: Modeling Franklins Improved Algorithm For Decentralized Extrema Finding In Circular Configurations Of Processors.
  Computer Science Department of University of Pisa, Internal Report January 2014. The CoreASM code for the algorithm can be accessed via https://github.com/szenzaro/WebASM/blob/master/src/main/ide/ExtremaFinding.casm or the website http://modelingbook.informatik.uni-ulm.de
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018.
  http://modelingbook.informatik.uni-ulm.de

Copyright Notice