Egon Börger (Pisa)

Recursive Abstract State Machines

or

Concurrent ASMs with Partial Order Runs

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it

See E. Börger and K.-D. Schewe: *A Behavioural Theory of Recursive Algorithms*. (2019)

Goal of the lecture

Explain the relation between

- partial order distributed ASM runs (Gurevich 1995)
- concurrent ASM runs (Börger/Schewe 2016)

The analysis of this relation led us to

- define recursive ASMs (intrinsically finitely composed of 'sequential' ASMs),
- show them to capture recursive algorithms,
- relate their runs to partial order runs by the following theorem:

Theorem (Börger/Schewe). *Finitely-composed concurrent ASMs $\mathcal{C}$ with sequential (possibly non-deterministic) components for which all concurrent $\mathcal{C}$-runs are definable by partial-order runs have a behavior that is equivalent to that of recursive ASMs, and vice versa*.

NB. Only knowledge of the definition of single-agent ASMs is assumed.

The history behind: what triggered the investigation

- Blass and Gurevich used what are called 'partial order runs of distributed ASMs' to describe recursive algorithms by ASMs
  – see Bulletin of the EATCS, 77 (2002), 96-119.
- Börger/Schewe defined a notion of concurrent ASM runs (2016) which is more general than that of partial order distributed ASM runs.
- A closer investigation revealed that (roughly speaking) recursive ASMs are exactly those finitely composed multi-agent ASMs (with sequential components) whose concurrent runs are definable as partial order runs.
- The investigation was carried out in more general terms extending
  – Gurevich's Sequential ASM Thesis by a Recursive ASM Thesis
  – sequential ASMs and their runs to recursive ASMs and their runs.

Recap of Gurevich's postulates for 'sequential' algorithms

To characterize *recursive* algorithms we incorporate call steps into the one-step transition for non-deterministic sequential (nd-seq) algorithms.

Branching Time. *One-step state transition* relation $\tau \subseteq \mathcal{S} \times \mathcal{S}$ with a set of initial states $\mathcal{I} \subseteq \mathcal{S}$. (NB. Gurevich used functions τ.)

- Runs = sequences of states related by τ, starting in an $\mathcal{I}$-state.

Abstract State. States are structures (interpretations of a finite signature Σ of function symbols f over a base set U, called universe).

- View a state S as set of memory locations $(f, args)$ with a value $f(args) \in U$. State changes come via sets of updates $((f, args), v)$.
- $\mathcal{S}$ and $\mathcal{I}$ are closed under isomorphisms. τ does not modify the universe U and is carried over by isomorphisms.

Bounded Exploration. A finite witness set W of ground terms (over Σ) s.t. for all states S, S', steps $\tau(S, S')$ depend only on the interpretation of W in S.

What is the intuition underlying recursive steps?

A recursive algorithm starts with a $main$ program. This program and its subprograms have the ability to

- *perform recursive call steps*, i.e.
  - to trigger for some input an instance of an algorithm
    - among a fixed maximal number of algorithms, including itself
  - and to remain waiting until the callee has computed, independently, an output for the given input
- *issue in one step possibly multiple calls*
  - a fixed maximal number of finitely many synchronous parallel calls
  - of callees which perform their subcomputations independently of each other and of the caller (modulo the input/output relation)
    - think of Fibonacci, mergesort, etc.

To capture this intuition we must characterize the input/output call relationship and encapsulate subcomputations of callees.

The call relationship for input/output algorithms

An i/o-algorithm $\mathcal{A}$ is a nd-seq algorithm with call steps satisfying the Call Step Postulat below and with signature $\Sigma = \Sigma_{in} \cup \Sigma_{loc} \cup \Sigma_{out}$ (disjoint union) satisfying the following input/output assumptions:

1. Input locations of $\mathcal{A}$ are only read by $\mathcal{A}$, but never updated by $\mathcal{A}$.
2. Output locations of $\mathcal{A}$ are never read by $\mathcal{A}$, but can be written by $\mathcal{A}$.
3. Any initial state of $\mathcal{A}$ depends only on its input locations.

A call relationship holds for two i/o-algorithms $\mathcal{A}^p$ (parent) and $\mathcal{A}^c$ (child) iff they share only in/out locations:

- $\Sigma_{in}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$. Furthermore, $\mathcal{A}^p$ may update but never reads input locations of $\mathcal{A}^c$.
- $\Sigma_{out}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$. Furthermore, $\mathcal{A}^p$ may read but never updates output locations of $\mathcal{A}^c$.
- $\Sigma_{loc}^{\mathcal{A}^c} \cap \Sigma^{\mathcal{A}^p} = \emptyset$ (no other shared locations).

The Call Step Postulate

- When a parent i/o-algorithm p calls a finite number of child i/o-algorithms $c_1, \ldots, c_n$, a call relationship holds between the caller and each callee, which are recorded in a set $CalledBy(p)$.
- The caller activates a fresh instance of each callee c_i so that they can start their computations.
- These computations are independent of each other and the caller remains waiting—i.e. performs no step—until every callee has terminated its computation (read: has reached a final state).
- For each callee, the initial state of its computation is determined only by the input passed by the caller.
- The only other interaction of the callee with the caller is to return in its final state an output to p.

A recursive algorithm $\mathcal{R}$ is a finite set of i/o-algorithms

- i.e. of components satisfying Gurevich's and the Call Step Postulate

one of which is distinguished as *main* algorithm.

What is a recursive run?

In a recursive run

- differently from runs of a nd-seq algorithm where in each state at most one step of the nd-seq algorithm is performed

a recursive algorithm $\mathcal{R}$ can perform in one recursive step simultaneously one step of each of finitely many not $Terminated$ and not $Waiting$ $Called$ instances of its i/o-algorithms, where we define:

$Active(q)$ **iff** $q \in Called$ **and not** $Terminated(q)$

$Waiting(p)$ **iff forsome** $c \in CalledBy(p)$ $Active(c)$

$Called = \{main\} \cup \bigcup_p CalledBy(p)$

Recursive Run Postulate. A recursive $\mathcal{R}$-run is a sequence of pairs (S_i, C_i) with states S_i and sets C_i of instances of $\mathcal{R}$-components s.t. the Recursive Run Constraint and the Bounded Call Tree Branching condition stated below are satisfied.

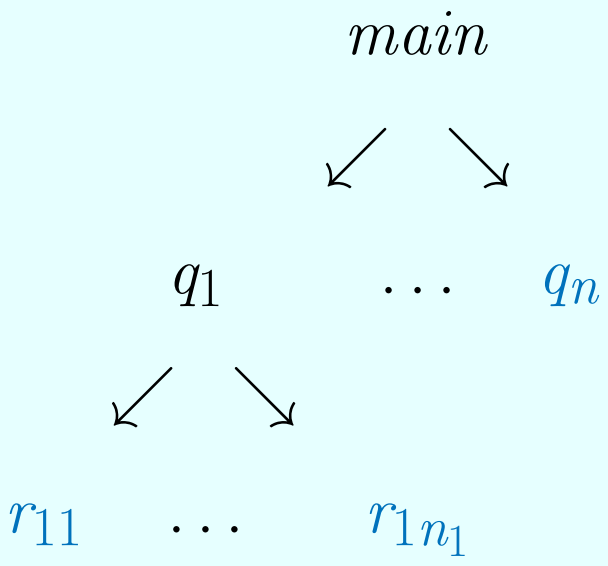
Recursive Run Postulate

Recursive run constraint.

- $C_0 = \{main\}$, i.e. every recursive run starts with $main$,
- in each recursive step a finite number of $Active$ and not $Waiting$ instances of components of $\mathcal{R}$ is $Called$ to make a step, i.e.
  - every C_i is a finite set of in S_i $Active$ and not $Waiting$ instances of components of $\mathcal{R}$,
  - every S_{i+1} is obtained by performing in S_i simultaneously one step of each i/o-algorithm in C_i.
    - Such an $\mathcal{R}$-step is also called a recursive step of $\mathcal{R}$.

Bounded call tree branching. There is a fixed natural number $m > 0$, depending only on $\mathcal{R}$, which in every $\mathcal{R}$-run bounds the number of callees which can be called by a call step.

Call trees



leaves are $Active$ and not $Waiting$

- Calls of $main$ create a finitely branched call tree.
- Further calls happen at leaves and extend the tree.
- When the algorithm at a child of a node has $Terminated$ its computation, the child is deleted from the tree.
- When $main$ has $Terminated$ its computation, the call tree is reduced again to the root.

Extending nd-seq ASMs by Call rules to recursive ASMs

The same way a recursive algorithm consists of finitely many i/o-algorithms, a recursive ASM $\mathcal{R}$ consists of finitely many recursive ASM rules, also called components (or component ASMs) of $\mathcal{R}$.

- A recursive ASM $\mathcal{R}$ consists of a finite set of recursive ASM rules, one of which is declared to be the main rule.
- Recursive ASM rules are defined as usual by induction, adding to the nd-seq rules the following Call rule:
  - Let $t_0, \ldots, t_n$ be terms where the outermost function symbol f_0 of t_0 is different from the outermost fct symbol f_i of t_i for every $i \neq 0$.
    - We declare: $f_0 \in \Sigma_{out}$ and $f_i \in \Sigma_{in}$ for $i \neq 0$.
  - Let $N \in \mathcal{N}$ be the name of a rule declared by $N(x_1, \ldots, x_n) = r$, where r is a recursive ASM rule all of whose free variables are contained in $\{x_0, \ldots, x_n\}$.
  - Then $t_0 \leftarrow N(t_1, \ldots, t_n)$ is a recursive ASM rule.
    - called a *named i/o-rule* or simply a call rule.

What are recursive ASM runs?

- Recursive ASMs are 'multi-agent' machines, consisting of a *set* of recursive rules instances of which may be called to be executed independently.
- To separate the state spaces of different agents we define *instances of a rule* r by ambient ASMs of form **amb** a **in** r with agents a.

Definition. A recursive run of a recursive ASM $\mathcal{R}$ is a sequence of pairs (S_i, A_i) of states S_i and subsets $A_i \subseteq Agent$, where each $a \in Agent$ is equipped with a $pgm(a)$ that is an instance **amb** a **in** r of a rule $r \in \mathcal{R}$, such that the following holds:

- A_0 is a singleton set $A_0 = \{a_0\}$, which in S_0 equals the set $Agent$, a_0 is equipped with $pgm(a_0) =$ (**amb** a_0 **in** $main$).
- A_i is a finite set of in S_i $Active$ and not $Waiting$ agents.
- S_{i+1} is obtained from S_i—we say in one $\mathcal{R}$-step—by performing for each agent $a \in A_i$ one step of $pgm(a)$.

The auxiliary predicates concerning recursive runs

$Active(a)$ (in state S) **iff**

$a \in Agent$ **and not** $Terminated(pgm(a))$ (in S)

$Waiting(a)$ (in state S) **iff**

forsome $a' \in CalledBy(a)$ $Active(a')$ (in S)

- Instances **amb** a **in** r of a called rule r permit to isolate the state space of agent a from that of other agents
  - by evaluating terms t in state S considering also the agent parameter a, using $val_S(t, a)$ instead of $val_S(t)$.
- To establish the call relationship we require the following:
  - when a recursive ASM rule r, executed by a parent agent p, calls a rule q to be executed by a child agent c, then the input/output functions f of q are also functions in r and are interpreted in the state space of p the same way as in the state space of c.

Defining the behavior of Call rules (by an ASM rule)

We define the update set computed by a Call rule $t_0 \leftarrow N(t_1, \ldots, t_n))$ as the set of updates computed by the following caller ASM rule:

$\textsc{Call}(t_0 \leftarrow N(t_1, \ldots, t_n)) =$

let $N(x_1, \ldots, x_n) = q$ -- declaration of N

forall $1 \leq i \leq n$ **let** $v_i = t_i$ -- evaluation of input in caller env

let $t_0 = f(t'_1, \ldots, t'_k)$

forall $1 \leq j \leq k$ **let** $v'_j = t'_j$ -- evaluation of output location

let $c = \mathbf{new}\ (Agent)$

$pgm(c) := \mathbf{amb}\ c\ \mathbf{in}\ q$ -- equip callee with its program instance

$\textsc{Insert}(c, CalledBy(\mathbf{self}))$

$\textsc{Initialize}(q_c, v_1/x_1, \ldots, v_n/x_n, f(v'_1, \ldots, v'_k)/x_o)$

$CalledBy(c) := \emptyset$

This makes the callee ready to run and the caller $Waiting$.

Recursive ASMs capture recursive algorithms

Theorem.

- Each recursive ASM $\mathcal{M}$ defines a recursive algorithm that is behaviourally equivalent to $\mathcal{M}$ with respect to recursive runs.
- For each recursive algorithm $\mathcal{R}$ there exists a recursive ASM which is behaviourally equivalent to $\mathcal{R}$ with respect to recursive runs.

The proof uses and extends Gurevich's proof (2000) that sequential ASMs capture sequential algorithms (which can be extended to hold also for the case of non-deterministic sequential algorithms).

Recursive MERGESORT example

$\textsc{Mergesort} = \{l \leftarrow \textsc{Sort}(l'), l \leftarrow \textsc{Merge}(l_1, l_2)\}$

$sorted\_list \leftarrow \textsc{Sort}(unsorted\_list) =$ -- with 3 recursive calls

if $sorted\_list =$**undef then**

let $n = length(unsorted\_list)$

if $n \leq 1$ **then** $sorted\_list := unsorted\_list$

if $n > 1$ **and** $sorted\_list_1 =$**undef**$=$ $sorted\_list_2$ **then**

let $concat(list_1, list_2) = unsorted\_list$ **with** $length(list_1) = \lfloor \frac{n}{2} \rfloor$

$sorted\_list_1 \leftarrow \textsc{Sort}(list_1)$ -- split list into sublists

$sorted\_list_2 \leftarrow \textsc{Sort}(list_2)$

if $n > 1$ **and** $sorted\_list_1 \neq$**undef**$\neq$ $sorted\_list_2$ **then**

$sorted\_list \leftarrow \textsc{Merge}(sorted\_list_1, sorted\_list_2)$

Recursive MERGE ASM example

$merged\_list \leftarrow \text{MERGE}(inlist_1, inlist_2) =$ -- with 2 recursive calls

if $merged\_list =$**undef then**

if $inlist_1 = []$ **then** $merged\_list := inlist_2$

elseif $inlist_2 = []$ **then** $merged\_list := inlist_1$

if $inlist_1 \neq [] \neq inlist_2$ **then**

for $i = 1, 2$ **let** $x_i = head(inlist_i)$ $restlist_i = tail(inlist_i)$

if $x_1 \leq x_2$ **and** $merged\_restlist =$**undef**

then $merged\_restlist \leftarrow \text{MERGE}(restlist_1, inlist_2))$

if $x_1 > x_2$ **and** $merged\_restlist =$**undef**

then $merged\_restlist \leftarrow \text{MERGE}(inlist_1, restlist_2))$

if $x_1 \leq x_2$ **and** $merged\_restlist \neq$**undef**

then $merged\_list := concat([x_1], merged\_restlist))$

else $merged\_list := concat([x_2], merged\_restlist))$

Recursive ASMs vs ASMs with unbounded parallelism

- In a recursive ASM run, in each step only finitely many instances of the components can be executed at the same time
- and these instances do not stand in an ancestor relationship.

Therefore, the relevant locations—on copies of which these instances work—can be made explicit by naming them and working on them in parallel, each instance snippet working with its own 'named' locs.

- Technically, to name individual locations one can use finite sequences I of indeces $0, 1, 2, \ldots$ as parameters, e.g. for MERGESORT
  – $unsorted\_list(I)$, $sorted\_list(I)$, $merged\_list(I)$, $restlist(I)$, etc.

The resulting parallelism on the input parameters is unbounded (**forall** $I \in \{0, 1, 2, \ldots\}^*$), although in each step only finitely many parallel branches are executed.

Conclusion: *Parallel ASMs (algorithms) are a wider class than recursive ASMs (algorithms)*.

Relation of recursive to partial order runs

- Gurevich in 1993 proposed partial order (po) runs to define distributed ASM runs and used them in 2002 with Blass to describe recursive runs.
- Börger/Schewe defined in 2016 a more comprehensive notion of concurrent ASM runs, reflecting a *Concurrency Postulate*.
- The analysis of concurrent ASM runs showed the following relation to partial order runs:

Theorem (Börger/Schewe 2019). Recursive algorithms (ASMs) are exactly those concurrent finitely-composed algorithms (ASMs) $\mathcal{C}$ with non-deterministic sequential components whose concurrent $\mathcal{C}$-runs are po-definable.

NB. The two restrictions on component ASMs are needed for 2 reasons:

- Recursive algorithms are finitely-composed.
- Unbounded parallel components lead to ASMs which are more powerful than recursive ASMs.

The Concurrency Postulate (Börger/Schewe 2016)

- A concurrent system (or process) is given by a set $\mathcal{A}$ of agents a each of which is equipped with an algorithm $alg(a)$—e.g. an ASM—that is executed autonomously by the agent.
- A concurrent $\mathcal{A}$-run is a sequence $S_0, S_1, \ldots$ of 'interaction states':
  - S_0 is an initial state
  - S_{n+1} is obtained from S_n *by combined single interacting moves* of a finite set A_n of agents a—each with its own clock—which
    - started the execution of their current $alg(a)$-move by a read interaction ('receive action') in state S_{j_a} ($j_a \leq n$ depending on a),
    - complete this (otherwise internal, purely local) move by a write interaction ('send action') in state S_n.

NB. *'Combining interacting moves of autonomous agents' in* $S_0, S_1, \ldots$ expresses the concurrent view of multi-agent system runs.

A way to formalize the concurrent ASM run step scheme

A nd-seq ASM $\textsc{ConcurStep}(pgm(a))$ permits agent a to choose

- whether to perform a global step
  - whose updates of globally visible (shared or output) locations are synchronized with globally visible updates by other processes
    - updates which are subject to the consistency constraint

  i.e. to directly perform a global atomic read&write step (including interaction locations) of $pgm(a)$ in the current run state
- or to perform a series of local steps affecting private memory, using local copies for the previously read values of globally visible locations
  - to only later interact again with other processes by
    - writing back to globally visible locations and
    - reading again possibly changed values of monitored/shared locs

  i.e. to trigger performing in the run three consecutive atomic actions: read&$\textsc{SaveGlobalData}$, $\textsc{LocalWriteStep}$, $\textsc{WriteBack}$ asynchronously, in different states (read: at clock ticks of a)

Formula defining the concurrent ASM run step scheme

Then one can define S_{n+1} formally

- as obtained from S_n by applying to S_n all the update sets U_a any agent $a \in A_n$ computes in S_n using $\textsc{ConcurStep}(pgm(a))$.

Expressed equationally:

$$S_{n+1} = S_n + \bigcup_{a \in A_n} U_a$$

NB. Here we use bootstrapping: instances $\textsc{ConcurStep}(pgm(a))$ of a single-agent ASM are used to define the behavior of concurrent ASM steps

- computing the updates in S_n either by $pgm(a)$ or by one of the simulation components
  - asynchronously at the agent's clock tick
    - due to the separation of reads/writes in concurrent steps

Gurevich's partial order ASM runs: the definition

Let $\mathcal{M} = (a, asm_a)_{a \in Agent}$ be a family of ASMs. A partial order $\mathcal{M}$-run (called also distributed ASM run or po-run of $\mathcal{M}$) is a partially ordered non-empty set $(M, \leq)$ of *moves* m of its agents $ag(m)$ coming with an initial segment function σ that satisfies the following conditions:

finite history: each move has only finitely many predecessors, i.e. $\{m' \in M \mid m' \leq m\}$ is finite for each $m \in M$,

sequentiality of agents: for each agent a the set of its moves in M is linearly ordered, i.e. $ag(m) = ag(m')$ implies $m \leq m'$ **or** $m' \leq m$,

coherence: each finite initial segment I of $(M, \leq)$ has an associated state $\sigma(I)$ —interpreted as the result of all moves in I with m executed before m' if $m < m'$ — which for every maximal element $m \in I$ is the result of applying move m in state $\sigma(I - \{m\})$.

A move m is an application of the agent's rule $pgm(ag(m)) = asm_a$.

NB. In general programs or agents are not restricted to finitely many.

Partial order ASM run example IndependentWrite

Consider IndependentWrite with two agents a, b, rule $f(\textbf{self}) := 1$ and initial state $\sigma(\emptyset)$ where $f(a) = f(b) = 0$. Let m_a resp. m_b be a move of a resp. b. The induced partial order is $\{(m_a, m_a), (m_b, m_b)\}$ with initial segments $\emptyset, \{m_a\}, \{m_b\}, \{m_a, m_b\}$ and σ illustrated by:

$$f(a) = f(b) = 0$$

$$\emptyset$$

$$\swarrow \quad \searrow$$

$$f(a) = 1 \;\; f(b) = 0 \quad \{m_a\} \quad \{m_b\} \quad f(a) = 0 \;\; f(b) = 1$$

$$\searrow \quad \swarrow$$

$$\{m_a, m_b\}$$

$$f(a) = f(b) = 1$$

Characteristics of partial order ASM runs

NB. In IndependentWrite the updates $f := 1$ by a resp. b are independent of each other because the two agents a, b come with separate states (i.e. interpretation of f as agent-dependent fcts f_a, f_b).

Let I be any finite initial segment of a partial order ASM run.

- *Linearization Property*.
  All linearizations of I yield runs with the same final state $\sigma(I)$.
- *Unique Result Property*.
  Two partial order ASM runs with same moves and same initial state yield for every finite initial segment the same associated state, i.e. $\sigma(I) = \sigma'(I)$ for each finite initial segment I.
- *Sequential Consistency Property.*
  Each linearization l of I yields a witness for the sequential consistency of the set of sequential I-subruns (one sequential run for each agent which makes a move in I).

RacyWrite has no not-single-agent partial order ASM runs

RacyWrite =

if $mode_{a_i} = start$ **then**

if $f \neq i$ **then** $f := i$

$mode_{a_i} := stop$

-- two agents a_i with $i = 1, 2$

-- NB. f is a *shared* fct

- There is no partial order ASM run of RacyWrite, started in $mode_{a_i} = start$, where each a_i makes a move m_i.
  - As for IndependentWrite, initial segments are $\emptyset$, $\{m_1\}$, $\{m_2\}$, $\{m_1, m_2\}$ and every state assignment σ satisfies $f = i$ in $\sigma(\{m_i\})$.
  - m_1 **seq** m_2 and m_2 **seq** m_1 are *two linearizations* of $\{m_1, m_2\}$ *with different final state*.

Only single-agent 1-step runs are po-runs of RacyWrite.

Characterizing the computational power of po-runs

Consider finitely composed concurrent ASMs (satisfying stipulations Gurevich made for partial order ASM runs), i.e. concurrent ASMs $\mathcal{C} = (asm_a)_{a \in Agent}$ such that

- each component asm_a is an instance **amb** a **in** M of an ASM rule M from a *finite program base $\mathcal{B}$ of*
  - (possibly non-deterministic) *sequential* ASMs
  - *programs which may create new agents* (potentially infinitely many)
    - via rules **let** $a = \mathbf{new}\ (Agent)$ **in** r
- initially there are only finitely many $Agent$s (wlog say $Agent = \{a_0\}$)

Theorem (Börger/Schewe 2019).

- For every recursive ASM $\mathcal{R}$ there is a behaviourally equivalent finitely composed concurrent ASM $\mathcal{C}_\mathcal{R}$ whose concurrent runs are po-definable.
- For each finitely composed concurrent ASM $\mathcal{C}$ whose concurrent runs are po-definable there is a behaviourally equivalent recursive ASM $\mathcal{R}_\mathcal{C}$.

Simulating recursive $\mathcal{R}$-runs by partial order runs of $\mathcal{C}_{\mathcal{R}}$

Define $\{r^* \mid r \in \mathcal{R}\}$ as program base of $\mathcal{C}_{\mathcal{R}}$ where

$r^* =$ **if** $Active(r)$ **and not** $Waiting(r)$ **then** r -- for nd-seq ASM r

$(t_0 \leftarrow N(t_1, \ldots, t_n))^* =$

if $Active(r)$ **and not** $Waiting(r)$ **then** $\textsc{Call}(t_0 \leftarrow N(t_1, \ldots, t_n))$

- This guarantees the *behavioral equivalence*.

Let $(S_0, A_0), (S_1, A_1), \ldots$ be a concurrent run of $\mathcal{C}_{\mathcal{R}}$.

- ag defined by $S_i \rightarrow_{A_i} S_{i+1}$.
- Let M_i be the set of all moves which result in state S_i of the initial run segment $[S_0, \ldots, S_i]$, i.e. finish by writing to S_j for some $j < i$.
- Let $M = \cup_i M_i$. NB: formally moves are pairs $(read_m, write_m)$.
- $m < m'$ iff move m contributes to update some state S_i (read: finishes in S_i) and move m' starts reading in a later state S_j with $i + 1 \leq j$. Thus, by definition, M_i is an initial segment of M.

From $\mathcal{R}$ to $\mathcal{C}_\mathcal{R}$: finite history, sequentiality of agents and σ

- *finite history*: given $m' \in M$ let S_j be the state in which it is started. There are only finitely many earlier states $S_0, \ldots, S_{j-1}$, and in each of them only finitely many moves m can finish.
- *sequentiality of agents* holds since the $\mathcal{R}$-components are seq ASMs.
- let $\sigma(I)$ be the result of the application of the moves in I in any total order extending the partial order $\leq$. Therefore $\sigma(\emptyset) = S_0$.
- σ *is well-defined*:
  - two incomparable moves $m \neq m'$ either both start in the same state or say m starts earlier than m'. But m' also starts earlier than m finishes. This is only possible for agents $ag(m) = a$ and $ag(m') = a'$ whose programs $pgm(a), pgm(a')$ are not in an ancestor relationship in the call tree. Therefore these programs have disjoint signatures, so that the moves m and m' could be applied in any order with the same resulting state change.
- This implies the definability $S_i =_{def}$ (result of all $m \in M_i) = \sigma(M_i)$

From recursive $\mathcal{R}$-runs to po $\mathcal{C}_{\mathcal{R}}$-runs: coherence property

- let I be a finite initial segment of finished moves of M.
  - NB. A not yet finished move is a $read_m$ whose $write_m$ did not yet follow. A pure $read_m$ does not change the state.
- let $I' = I - I_{max}$ where $I_{max} = \{m \in I \mid m \text{ maximal in } I\}$
- Then $\sigma(I)$ is the result of applying to $\sigma(I')$ simultaneously (or in any order) all moves $m \in I_{max}$.
  - NB. The maximal moves are incomparable so that they can be executed in any order without changing the result.

This implies in particular the coherence property for σ.

From partial order $\mathcal{C}$-runs to recursive $\mathcal{R}_\mathcal{C}$-runs

- Let $\mathcal{C}$ be a finitely composed concurrent ASM with program base $\{r_i \mid i \in I\}$ of nd-seq ASMs
  - whose concurrent runs are definable by partial-order runs.
- To construct: a recursive ASM $\mathcal{R}_\mathcal{C}$ whose (recursive) runs simulate the concurrent runs of $\mathcal{C}$ as follows:
  - Let $(S_0, A_0), (S_1, A_1), \ldots$ be a concurrent $\mathcal{C}$-run, defined by a partial order run $(M, \leq, ag, pgm, \sigma)$, i.e. such that
    - $S_i = \sigma(M_i)$ for the set M_i of moves which led from S_0 to S_i
    - the concurrent step $S_i \Rightarrow S_{i+1}$ is performed by parallel independent moves $m \in M_{i+1} \setminus M_i$ of $ag(m) = a$ with program $pgm(a) =$ **amb** a **in** r, an instance of a base program r
- *Idea*: simulate each move m by letting its *agent a act as caller of a named rule* $in_r \leftarrow \textsc{OneStep}_r(out_r)$.
  - The callee agent c acts as delegate for one step of a: it executes **amb** a **in** r and makes its program immediately $Terminated$.

Recursive delegation rule

Refine $\textsc{Call}$ to $\textsc{Call}^*(in_r \leftarrow \textsc{OneStep}_r(out_r))$ by

- adding to $\textsc{Initialize}$ the update $Terminated(pgm(c)) := false$
- defining $\textsc{OneStep}_r$ to perform **amb** $caller(c)$ **in** r and terminate
- **let** $in_r =$ set of exploration witnesses of $r = \{t_1, \ldots, t_n\}$
- **let** $out_r =$ set of updatable terms of $r = \{s_1, \ldots, s_m\}$

forall $t_i \in in_r$ **let** $v_i = t_i$ -- evaluation of witness terms by $caller$

forall $s_j \in out_r$ **let** $s_j = f_j(s_{j,1}, \ldots, s_{j,k_j})$

  forall $1 \leq k \leq k_j$ **let** $v_{j,k} = s_{j,k}$ -- evaluation of updatable locations

let $c = \mathbf{new}\ (Agent)$

  $pgm(c) :=$**amb** c **in** $\textsc{OneStep}_r$

  $\textsc{Insert}(c, CalledBy(\textbf{self}))$

  $\textsc{Initialize}(q_c, \ldots v_i/x_i, \ldots, f_j(v_{j,1}, \ldots, v_{j,k_j})/y_j, \ldots)$

  $CalledBy(c) := \emptyset$

Definition of $\mathcal{R}_{\mathcal{C}}$

$\mathcal{R}_{\mathcal{C}} = \{in_r \leftarrow \textsc{OneStep}_r(out_r) \mid r \in$ program base of $\mathcal{C}\}$

$\textsc{OneStep}_r =$

amb $caller(\textbf{self})$ **in** r -- the delegate executes the step of its $caller$

$Terminated(pgm(\textbf{self})) := true$ -- ... and immediates stops

NB. **amb** a **in** $in_r \leftarrow \textsc{OneStep}_r(out_r)$ with $a = caller(c)$

- is equivalent to **amb** a **in** $\textsc{Call}^*(in_r \leftarrow \textsc{OneStep}_r(out_r))$
  - by definition
- triggers the delegate to execute program **amb** c **in** $\textsc{OneStep}_r$
- which triggers **amb** c **in** (**amb** a **in** r) (by definition)
- which is equivalent to **amb** a **in** r
  - since the innermost ambient binding counts

$\mathcal{R}_\mathcal{C}$-run simulates $\mathcal{C}$-run $(S_0, A_0), (S_1, A_1), \ldots$

let $A_i = \{a_{i_1}, \ldots, a_{i_k}\} \subseteq Agent$ where i_j and k depend on i

let $a_{i_j} = ag(m_{i_j})$ **with**

$m_{i_j} \in M_{i+1} \setminus M_i$ **and** $pgm(a_{i_j}) =$ **amb** a_{i_j} **in** r_{i_j}(**forall** $1 \leq j \leq k$)

- The $\mathcal{R}_\mathcal{C}$-run has the same agents $a_{i_j} \in A_i$, but with program $in_{r_{i_j}} \leftarrow \textsc{OneStep}_{r_{i_j}}(out_{r_{i_j}})$.
- Their step in the recursive run leads to a state S_i' where all callers a_{i_j} are $Waiting$ and the delegates c_{i_j} are $Active$ and not $Waiting$.
- We choose the delegates for the next $\mathcal{R}_\mathcal{C}$ step, whereby
  - all rules r_{i_j} are performed simultaneously in the ambient of $caller(c_{i_j}) = a_{i_j}$ thus leading as desired to the state S_{i+1},
  - the delegate programs are $Terminated$, whereby their callers a_{i_j} become again not-$Waiting$ and thereby ready to take part in the next $\mathcal{C}$-step.
    - NB. Agents created in the $\mathcal{C}$-run are initialized to not-$Waiting$.

Finite concurrent ASMs with partial order definable runs

Finite concurrent ASMs, i.e. with a fixed number of agents and non-deterministic sequential component programs, whose runs are po-definable can be simulated by nd-seq ASMs.

More precisely:

Theorem. For each finite concurrent ASM $\mathcal{C} = \{(a_i, r_i) \mid 1 \leq i \leq n\}$ with nd-seq ASMs r_i such that all concurrent $\mathcal{C}$-runs are definable by partial-order runs, one can construct a nd-seq ASM $\mathcal{M}_\mathcal{C}$ such that each $\mathcal{M}_\mathcal{C}$-run is equivalent to a concurrent $\mathcal{C}$-run and vice versa.

- **Corollary**. Each Petri net can be simulated by a non-deterministic sequential ASM.

Definition of $\mathcal{M}_{\mathcal{C}}$

Idea: *linearize concurrent $\mathcal{C}$-runs* by relating their states S_i to the states $\sigma(M_i)$ associated with initial segments M_i of a corresponding partial order run $(M, \leq, ag, pgm, \sigma)$

- where each step leading from S_i to S_{i+1} consists of pairwise incomparable moves of some agents in $M_{i+1} \setminus M_i$
- these moves consist in applying a finite non-empty subset of $\mathcal{C}$-rules —executed by their agents in any order (read: in parallel)

$\mathcal{M}_{\mathcal{C}} =$

choose $R \subseteq \{r_j \mid 1 \leq j \leq n\}$ **with** $R \neq \emptyset$

forall $r \in R$ **do** r

NB. By assumption, **choose** and **forall** range over a fixed finite number of elements so that $\mathcal{M}_{\mathcal{C}}$ is a non-deterministic sequential ASM.

$\mathcal{M}_{\mathcal{C}}$-runs are equivalent to linearized $\mathcal{C}$-runs

For $\mathcal{M}_{\mathcal{C}}$-runs $S_0, S_1, \ldots$ define $(M, \leq, ag, pgm, \sigma)$ by induction:

- $S_0 = \sigma(\emptyset)$
- let S_{i+1} result from S_i by moves m_{j_k} for some $1 \leq j_k \leq n$, applying to S_i all update sets Δ_{j_k} computed by the $\mathcal{C}$-rule r_{j_k}.
- By induction $S_i = \sigma(M_i)$ for some initial segment M_i of a partial order run $(M, \leq)$.
  - Case 1. The moves m_{j_k} are pairwise independent, i.e. their application in any order produces the same state S_{i+1}. Then $(M, \leq)$ can be extended by these moves such that adding all m_{j_k} to M_i yields an initial segment M_{i+1} for which $S_{i+1} = \sigma(M_i)$ holds.
  - Case 2. If the moves m_{j_k} are not pairwise independent, the union of their update sets is inconsistent, hence the run terminates in state S_i.

Linearized $\mathcal{C}$-runs are equivalent to $\mathcal{M}_\mathcal{C}$-runs

Let a linearized $\mathcal{C}$-run with $S_i = \sigma(M_i)$ for all $i \geq 1$ be given.

- Then S_{i+1} results from S_i by applying in parallel all moves m_{j_k} in $M_{i+1} \setminus M_i = \{m_{j_1}, \ldots, m_{j_l}\}$, for some $1 \leq l \leq n$.
- Applying a move m_{j_k} means to apply some rule r_{j_k}, which by assumption is a rule $r_{j_k} \in \mathcal{C}$.
- Applying the moves m_{j_k} in parallel means to apply their rules in parallel, which yields a consistent update set (because S_{i+1} is defined).
- Then the definition of $\mathcal{M}_\mathcal{C}$ implies that $S_i \Rightarrow_{\mathcal{M}_\mathcal{C}} S_{i+1}$.

This implies that $S_0, S_1, \ldots$ is a run of $\mathcal{M}_\mathcal{C}$.

References

E. Börger and K.-D. Schewe: *A Behavioural Theory of Recursive Algorithms*. (2019, Submitted)

E. Börger and K.-D. Schewe: *Concurrent Abstract State Machines*. Acta Informatica 53 (5), 469-492 (2016)

Y.Gurevich: *Evolving algebras 1993: Lipari Guide*. in: E.Börger (Ed.): Specification and Validation Methods. Oxford Univ. Press 1995, 9–36

Y.Gurevich: *Sequential abstract-state machines capture sequential algorithms*. ACM Trans.Comp.Logic 1 (1), 77-111 (2000)

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de
- E. Börger and R. Stärk: Abstract State Machines. Springer 2003.

Copyright Notice