Automatic Verification of Message-Passing Concurrency

Luke Ong

(Joint work with Jonathan Kochems and Emanuele D'Osualdo)

University of Oxford

Kröning Group Seminar, 6 March 2014

Outline

Outline

Erlang

– designed by Ericsson in 1980s to program real-time, distributed, fault-tolerant telecoms systems.

1. Each process (actor) is a sequential, higher-order functional program.
2. Each process has an unbounded mailbox. Processes communicate by asynchronous message passing – send is non-blocking.
3. Each process has a unique name or pid, which is datum and passable as message.
4. A process may block while waiting to receive a message that matches a given pattern: message retrieval is first-in-first-firable-out (FIFFO).
5. A process may spawn new processess (and remember their names).

Erlang

– designed by Ericsson in 1980s to program real-time, distributed, fault-tolerant telecoms systems.

1. Each process (actor) is a sequential, higher-order functional program.
2. Each process has an unbounded mailbox. Processes communicate by asynchronous message passing – send is non-blocking.
3. Each process has a unique name or pid, which is datum and passable as message.
4. A process may block while waiting to receive a message that matches a given pattern: message retrieval is first-in-first-firable-out (FIFFO).
5. A process may spawn new processess (and remember their names).

Natural fit for programming "irregular concurrency". E.g. multicore CPUs, networked servers, parallel databases, GUIs and interacting programs.

Erlang: "a gold standard in concurrency-oriented programming"

Goal: automatically verify safety properties (e.g. race freedom and mailbox boundedness).

Approach: by abstract interpretation and infinite-state model checking.

Goal: automatically verify safety properties (e.g. race freedom and mailbox boundedness).

Approach: by abstract interpretation and infinite-state model checking.

Verifying Erlang programs is inherently difficult.

Theorem (Turing Completeness)

The following (tiny) fragment of Erlang is already Turing powerful.

(1) *finite data types (in particular, finite message space)*
(2) *each process computes a first-order recursive function*
(3) *static spawning: the number of processes is 2*
(4) *bounded mailbox: mailboxes have a fixed capacity of 1*

Proof is by encoding Minsky's counter machine.

Goal: automatically verify safety properties (e.g. race freedom and mailbox boundedness).

Approach: by abstract interpretation and infinite-state model checking.

Verifying Erlang programs is inherently difficult.

Theorem (Turing Completeness)

The following (tiny) fragment of Erlang is already Turing powerful.

(1) *finite data types (in particular, finite message space)*

(2) *each process computes a first-order recursive function*

(3) *static spawning: the number of processes is 2*

(4) *bounded mailbox: mailboxes have a fixed capacity of 1*

Proof is by encoding Minsky's counter machine.

Replacing (1) and (2) by the following is also Turing powerful.

(1') constructors with arity at most 2

(2') order-0 function, equivalently, a finite-state transduceer

Take (Core) Erlang code as source.

1. Perform a k-CFA-like analysis—specialised from the generic abstract interpretation—to construct abstractions of data and control-flow.

The analysis is parametric and can be tuned for accuracy.

An Automatic Verification Pathway

Take (Core) Erlang code as source.

1. Perform a k-CFA-like analysis—specialised from the generic abstract interpretation—to construct abstractions of data and control-flow.
2. Bootstrap the analysis to yield an Actor Communicating System (ACS)—a CCS-like infinite-state model—that soundly approximates the program.

The analysis is parametric and can be tuned for accuracy.

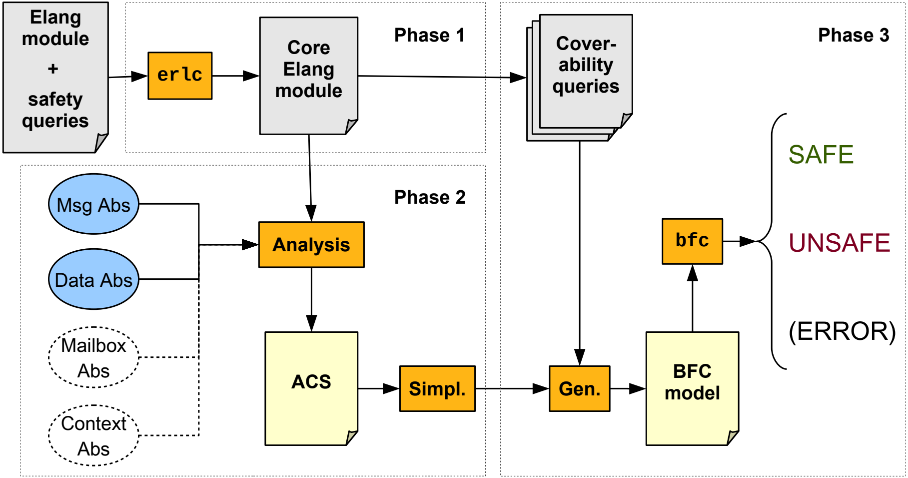
Take (Core) Erlang code as source.

1. Perform a k-CFA-like analysis—specialised from the generic abstract interpretation—to construct abstractions of data and control-flow.
2. Bootstrap the analysis to yield an Actor Communicating System (ACS)—a CCS-like infinite-state model—that soundly approximates the program.
3. Model-check the ACS using a vector addition system (or Petri nets, or multicounter automata) coverability checker (BFC)
   Counter abstraction. Three quantities: ι, q, m:
   - Counter (ι, q) counts # processes in pid-class ι currently in state q
   - Counter (ι, m) sums the occurrences of message m in the mailbox of a process p, as p ranges over pid-class ι

The analysis is parametric and can be tuned for accuracy.

Soter: Workflow in 3 Phases

http://mjolnir.cs.ox.ac.uk/soter/



Empirical Evaluation

Example	LOC	SAFE?	ABS		ACS		TIME (sec.)			
			D	M	#Pl.	Rat.	Ana.	Sim.	BFC	Total
reslockbeh	507	yes	0	2	40	4%	1.94	0.41	0.85	3.21
reslock	356	yes	0	2	40	10%	0.56	0.08	0.82	1.48
sieve	230	yes	0	2	47	19%	0.26	0.03	2.46	2.76
concdb	321	yes	0	2	67	12%	1.10	0.16	5.19	6.46
state_factory	295	yes	0	1	22	4%	0.59	0.13	0.02	0.75
pipe	173	yes	0	0	18	8%	0.15	0.03	0.00	0.18
ring	211	yes	0	2	36	9%	0.55	0.07	0.25	0.88
parikh	101	yes	0	2	42	41%	0.05	0.01	0.07	0.13
unsafe_send	49	no	0	1	10	38%	0.02	0.00	0.00	0.02
safe_send	82	no*	0	1	33	36%	0.05	0.01	0.00	0.06
safe_send	82	yes	1	2	82	34%	0.23	0.03	0.06	0.32
firewall	236	no*	0	2	35	10%	0.36	0.05	0.02	0.44
firewall	236	yes	1	3	74	10%	2.38	0.30	0.00	2.69
finite_leader	555	no*	0	2	56	20%	0.35	0.03	0.01	0.40
finite_leader	555	yes	1	3	97	23%	0.75	0.07	0.86	1.70
stutter	115	no*	0	0	15	19%	0.04	0.00	0.00	0.05
howait	187	no*	0	2	29	14%	0.19	0.02	0.00	0.22

Soter 0.1: References and Limitations

Soter tool: http://mjolnir.cs.ox.ac.uk/soter/

D'Osualdo, Kochems & O.: Soter: an Automatic Safety Verifier for Erlang. AGERE! '12.

D'Osualdo, Kochems & O.: Automatic Verification of Erlang-style Concurrency. SAS 2013.

Limitations: Two Sources of Imprecision

The rest of the talk aims to address (1) above; for (2) see Further Directions.

Soter 0.1: References and Limitations

Soter tool: http://mjolnir.cs.ox.ac.uk/soter/

D'Osualdo, Kochems & O.: Soter: an Automatic Safety Verifier for Erlang. AGERE! '12.

D'Osualdo, Kochems & O.: Automatic Verification of Erlang-style Concurrency. SAS 2013.

Limitations: Two Sources of Imprecision

(1) Each process is abstracted as a finite-state machine (even though the ACS is an infinite-state model).

- Cannot analyse non-tail-recursive functions accurately. Undesirable because Erlang processes are (higher-order) functional programs, and definition-by-recursion is standard.
- Cannot support stack-based reasoning.

The rest of the talk aims to address (1) above; for (2) see Further Directions.

Soter 0.1: References and Limitations

Soter tool: http://mjolnir.cs.ox.ac.uk/soter/

D'Osualdo, Kochems & O.: Soter: an Automatic Safety Verifier for Erlang. AGERE! '12.

D'Osualdo, Kochems & O.: Automatic Verification of Erlang-style Concurrency. SAS 2013.

Limitations: Two Sources of Imprecision

(1) Each process is abstracted as a finite-state machine (even though the ACS is an infinite-state model).

- Cannot analyse non-tail-recursive functions accurately. Undesirable because Erlang processes are (higher-order) functional programs, and definition-by-recursion is standard.
- Cannot support stack-based reasoning.

(2) Pids (**p**rocess **id**s) are abstracted as finitely many pid equiv. classes

- Cannot fully support analysis that requires precision of process identity.
- Because mailboxes are merged, certain patterns of communication cannot be analysed accurately.

The rest of the talk aims to address (1) above; for (2) see Further Directions.

Outline

Working Example: Server in Asynchronous-Programming Style

Background on Asynchronous Programming

- A ubiquitous systems programming idiom for managing concurrent interactions with the environment.
- The programmer can make conventional (synchronous) function calls, where a caller waits until the callee completes computation.
- However, for time-consuming tasks, the programmer makes (non-blocking) asynchronous procedure calls: the tasks are not immediately executed but are rather posted in a task bag.
- A despatcher picks and executes callback tasks from the task bag to completion (and these callbacks can post further callbacks to be executed later).

Working Example: Server in Asynchronous-Programming Style

```
server() →
    init_despatcher(), do_server(), post_task(),
    case (*) of
        true →server();
        false →system ? stop
    end,
    task_bag ! stop.

post_task() →task_bag ! task, task_bag ? ok.

init_despatcher() →task_bag ! init, task_bag ? ready.

despatcher() →
    task_bag ? init, task_bag ! ready,
    task_bag ? task, task_bag ! ok, do_task(),
    case (*) of
        true →despatcher();
        false →task_bag ? stop, system ! despatcher_done.

main() →spawn(server), spawn(despatcher), system ! stop.
```

Working Example: Server in Asynchronous-Programming Style

```
server() →
    init_despatcher(), do_server(), post_task(),
    case (*) of
        true →server();
        false →system ? stop
    end,
    task_bag ! stop.

post_task() →task_bag ! task, task_bag ? ok.

init_despatcher() →task_bag ! init, task_bag ? ready.

despatcher() →
    task_bag ? init, task_bag ! ready,
    task_bag ? task, task_bag ! ok, do_task(),
    case (*) of
        true →despatcher();
        false →task_bag ? stop, system ! despatcher_done.

main() →spawn(server), spawn(despatcher), system ! stop.
```

Question. Can the system reach a state s.t. `ready` $\in$ `task_bag` and `despatcher_done` $\in$ `system`?

The `server` is an instance of a widely studied concurrency model, ACPS.

Asynchronously Communicating Pushdown Systems (ACPS)

- Each process is a pushdown system.
- Processes may be spawned dynamically.
- Processes communicate asynchronously by message passing—non-blocking send, and blocking receive—via a fixed, finite number of unbounded, unordered channels (or message buffers).

The `server` is an instance of a widely studied concurrency model, ACPS.

Asynchronously Communicating Pushdown Systems (ACPS)

- Each process is a pushdown system.
- Processes may be spawned dynamically.
- Processes communicate asynchronously by message passing—non-blocking send, and blocking receive—via a fixed, finite number of unbounded, unordered channels (or message buffers).

Unfortunately reachability is undecidable in ACPS.

"Any context-sensitive and synchronisation-sensitive analysis is undecidable." (Ramalingam: TOPLAS 2000)

The server is an instance of a widely studied concurrency model, ACPS.

Asynchronously Communicating Pushdown Systems (ACPS)

- Each process is a pushdown system.
- Processes may be spawned dynamically.
- Processes communicate asynchronously by message passing—non-blocking send, and blocking receive—via a fixed, finite number of unbounded, unordered channels (or message buffers).

Unfortunately reachability is undecidable in ACPS.
"Any context-sensitive and synchronisation-sensitive analysis is undecidable." (Ramalingam: TOPLAS 2000)

A common restriction of ACPS sufficient for decidability

A process may only receive a message when its call stack is empty.

Large literature: see, e.g., (Sen & Viswanathan: CAV 2006), (Jhala & Majumdar: POPL 2007).

Questions

1. Find a model of asynchronous concurrency that relaxes the Receiveable-Only-When-Stack-is-Empty restriction (hence extending the paradigm), while preserving decidablity of reachability.
2. Is the new model realistic and useful?
3. How hard is safety verification of these models? What is the precise complexity of (ExpSpace-hard) reachability / coverability?
4. Are there "realistic algorithms"?

- Asynchronous procedure calls
  (Sen & Viswanathan: CAV06), (Jhala & Majumdar: POPL07), (Ganty et al.: POPL09)
- Hierarchical communication
  (Bouajjani & Emmi: POPL12), (Bouajjani et al.: Concur05)
- Synchronisation over locks
  (Kahlon: LICS09), etc.
- Variously bounded by: context, phase and scope
  (Lal & Reps: FMSD09), (Bouajjani & Emmi: TACAS12), (Torre et al.: Concur11)
- Pattern-based verification
  (Esparza & Ganty: POPL11)

Idea

- Because channels are unordered, the precise sequencing of non-blocking actions (i.e. **send** and **spawn**) are unobservable.
- Thus we postulate: certain actions commute with each other over sequential composition, while others (notably **receive**) don't.

Idea

- Because channels are unordered, the precise sequencing of non-blocking actions (i.e. **send** and **spawn**) are unobservable.
- Thus we postulate: certain actions commute with each other over sequential composition, while others (notably **receive**) don't.

Independence Relation and Commutative / Non-Comm. Actions

1. An independence relation $\# \subseteq \Sigma^2$ is an irreflexive and symmetric relation; it induces a congruence between terms, $\simeq_\# \subseteq (\Sigma^*)^2$. [Intuition: if $a \# b$ then "a commutes with b"]

Idea

- Because channels are unordered, the precise sequencing of non-blocking actions (i.e. **send** and **spawn**) are unobservable.
- Thus we postulate: certain actions commute with each other over sequential composition, while others (notably **receive**) don't.

Independence Relation and Commutative / Non-Comm. Actions

1. An independence relation $\# \subseteq \Sigma^2$ is an irreflexive and symmetric relation; it induces a congruence between terms, $\simeq_{\#} \subseteq (\Sigma^*)^2$. [Intuition: if $a \# b$ then "a commutes with b"]
2. $a \in \Sigma$ is $\#$-non-commutative if $\forall a' \in \Sigma : (a, a') \notin \#$
3. $a \in \Sigma$ is $\#$-commutative if $\forall a' \in \Sigma$: either a' is $\#$-non-commutative or $(a, a') \in \#$.

Idea

- Because channels are unordered, the precise sequencing of non-blocking actions (i.e. **send** and **spawn**) are unobservable.
- Thus we postulate: certain actions commute with each other over sequential composition, while others (notably **receive**) don't.

Independence Relation and Commutative / Non-Comm. Actions

1. An independence relation $\# \subseteq \Sigma^2$ is an irreflexive and symmetric relation; it induces a congruence between terms, $\simeq_{\#} \subseteq (\Sigma^*)^2$. [Intuition: if $a \# b$ then "a commutes with b"]
2. $a \in \Sigma$ is $\#$-non-commutative if $\forall a' \in \Sigma : (a, a') \notin \#$
3. $a \in \Sigma$ is $\#$-commutative if $\forall a' \in \Sigma$: either a' is $\#$-non-commutative or $(a, a') \in \#$.
4. An independence relation $\#$ is unambiguous just if it partitions Σ into $\#$-commutative (written Σ^{com}) and $\#$-non-comm. ($\Sigma^{\neg\mathrm{com}}$) parts.

A New Model of Asynchronous Concurrency: Notation

Fix finite sets: $Chan$ (channels), Msg (messages), $Labels$ and $\mathcal{N}$ (non-terminal symbols, for procedures). Define (concurrency) actions

$$\begin{array}{lcl} Spawns & := & \{\, \nu_X \mid X \in \mathcal{N} \,\} \\ Sends & := & \{\, c\,!\,m \mid c \in Chan, m \in Msg \,\} \\ Receives & := & \{\, c\,?\,m \mid c \in Chan, m \in Msg \,\} \end{array}$$

Set terminal symbols

$$\Sigma \;:=\; Labels \cup Sends \cup Receives \cup Spawns.$$

A New Model of Asynchronous Concurrency: Notation

Fix finite sets: $Chan$ (channels), Msg (messages), $Labels$ and $\mathcal{N}$ (non-terminal symbols, for procedures). Define (concurrency) actions

$$
\begin{array}{lcl}
Spawns & := & \{\, \nu_X \mid X \in \mathcal{N} \,\} \\
Sends & := & \{\, c\,!\,m \mid c \in Chan, m \in Msg \,\} \\
Receives & := & \{\, c\,?\,m \mid c \in Chan, m \in Msg \,\}
\end{array}
$$

Set terminal symbols

$$\Sigma \;:=\; Labels \cup Sends \cup Receives \cup Spawns.$$

1. Easy to define an unambiguous $\#$: partitioning Σ into commutative actions Σ^{com} and non-commutative actions $\Sigma^{\neg\mathrm{com}}$ as follows:

$$\Sigma \;:=\; \underbrace{(Labels \cup Spawns \cup Sends)}_{\text{Commutative}} \quad \cup \quad \underbrace{Receives}_{\text{Non-Comm.}}$$

A New Model of Asynchronous Concurrency: Notation

Fix finite sets: $Chan$ (channels), Msg (messages), $Labels$ and $\mathcal{N}$ (non-terminal symbols, for procedures). Define (concurrency) actions

$$\begin{array}{lcl} Spawns & := & \{\, \nu_X \mid X \in \mathcal{N} \,\} \\ Sends & := & \{\, c\,!\,m \mid c \in Chan, m \in Msg \,\} \\ Receives & := & \{\, c\,?\,m \mid c \in Chan, m \in Msg \,\} \end{array}$$

Set terminal symbols

$$\Sigma \;:=\; Labels \cup Sends \cup Receives \cup Spawns.$$

1. Easy to define an unambiguous $\#$: partitioning Σ into commutative actions Σ^{com} and non-commutative actions $\Sigma^{\neg\mathrm{com}}$ as follows:

$$\Sigma \;:=\; \underbrace{(Labels \cup Spawns \cup Sends)}_{\text{Commutative}} \quad \cup \quad \underbrace{Receives}_{\text{Non-Comm.}}$$

2. We can lift $\# \in \Sigma^2$ to an unambiguous $\widehat{\#} \subseteq (\Sigma \cup \mathcal{N})^2$, and so partition $\mathcal{N} = \mathcal{N}^{\mathrm{com}} \cup \mathcal{N}^{\neg\mathrm{com}}$

A New Model of Asynchronous Concurrency: APCPS

Given $Chan$, Msg, $Labels$ and $\mathcal{N}$, an **asynchronous partially commutative pushdown system** (APCPS) is a tuple $(\Sigma, \#, \mathcal{N}, \mathcal{R}, S)$ where

- $\Sigma := Labels \cup Sends \cup Receives \cup Spawns$ is a finite set of terminal symbols (= concurrency actions) as defined above
- $\mathcal{N}$ is a finite set of non-terminal symbols (=procedure names); $S \in \mathcal{N}$ is a start symbol
- $\# \subseteq \Sigma^2$ is an unambiguous independence relation (defined above) giving partitions: $\Sigma = \Sigma^{\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}}$ and $\mathcal{N} = \mathcal{N}^{\mathrm{com}} \cup \mathcal{N}^{\neg\mathrm{com}}$
- $\mathcal{R}$ is a set of rewrite rules of the forms $A \to a$, or $A \to B\,C$, where $a \in \Sigma \cup \{\epsilon\}$, $A, B, C \in \mathcal{N}$

The induced leftmost derivation relation, $\to$, is a binary relation over $(\Sigma \cup \mathcal{N})^* / \simeq_{\#}$.

Cf. Partially commutative context-free grammar (Czerwinski et al.:Concur 2009).

Example: APCPS

```
server() →
    init_despatcher(), do_server(), post_task(),
    case (*) of
        true →server();
        false →system ? stop
    end,
    task_bag ! stop.

post_task() →task_bag ! task, task_bag ? ok.

init_despatcher() →task_bag ! init, task_bag ? ready.
```

Define a APCPS with rules:

$$\begin{array}{lcl} S & \rightarrow & I \cdot D \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ S^{\text{case}} & \rightarrow & S \mid \texttt{system}\,?\,\texttt{stop} \\ S^{\text{stop}} & \rightarrow & \texttt{task\_bag}\,!\,\texttt{stop} \\ \ldots & & \end{array}$$

Commutative non-terminal: S^{stop}

Non-commutative non-terminals: S, I, P, S^{case}

Example: APCPS

```
despatcher() →
    task_bag ? init, task_bag ! ready,
    task_bag ? task, task_bag ! ok, do_task(),
    case (*) of
        true →despatcher();
        false →task_bag ? stop, system ! despatcher_done.
```

Further rules:

$$
\begin{array}{lcl}
D & \to & \mathtt{task\_bag}\,?\,\mathtt{init}\cdot \ell_1 \cdot \mathtt{task\_bag}\,!\,\mathtt{ready}\cdot D^{\mathrm{init}} \\
D^{\mathrm{init}} & \to & \mathtt{task\_bag}\,?\,\mathtt{task}\cdot \mathtt{task\_bag}\,!\,\mathtt{ok}\cdot T \cdot D^{\mathrm{msg}} \\
D^{\mathrm{msg}} & \to & D \mid \mathtt{task\_bag}\,!\,\mathtt{stop}\cdot \ell_2 \cdot \mathtt{system}\,!\,\mathtt{d\_done}
\end{array}
$$

Labels: ℓ_1, ℓ_2

Labels are commutative actions: reasonable because we are interested in the reachability of, not sequencing properties about, labels.

Write $\mathit{Terms} := (\Sigma \cup \mathcal{N})^* / \simeq_{\#}$. The configurations are elements of

$$\mathbb{M}[\mathit{Terms}] \times (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$$

where $\mathbb{M}[A]$ is the set of multisets of A.

For simplicity, we write a configuration

$$([\alpha, \beta, \alpha], \quad \{ c_1 \mapsto [m_1, m_1], c_2 \mapsto [\,] \})$$

as

$$\alpha \parallel \beta \parallel \alpha \blacktriangleleft c_1 \mapsto [m_1, m_1], c_2 \mapsto [\,]$$

$$\begin{array}{lll} S & \to & I \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ I & \to & \mathtt{bag}\,!\,\mathtt{init} \cdot \mathtt{bag}\,?\,\mathtt{rdy} \\ P & \to & \mathtt{bag}\,!\,\mathtt{task} \cdot \mathtt{bag}\,?\,\mathtt{ok}. \end{array} \qquad \begin{array}{lll} D & \to & \mathtt{bag}\,?\,\mathtt{init} \cdot \ell_1 \cdot \mathtt{bag}\,!\,\mathtt{rdy} \cdot D^{\text{init}} \\ D^{\text{init}} & \to & \mathtt{bag}\,?\,\mathtt{task} \cdot \mathtt{bag}\,!\,\mathtt{ok} \cdot D^{\text{msg}} \end{array}$$

A transition sequence of standard semantics

$$\begin{array}{ll}
 & S \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [] \\
\to & I \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [] \\
\to & \mathtt{bag}\,!\,\mathtt{init} \cdot \mathtt{bag}\,?\,\mathtt{rdy} \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [] \\
\to & \begin{array}{ll} & \mathtt{bag}\,!\,\mathtt{init} \cdot \mathtt{bag}\,?\,\mathtt{rdy} \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & \mathtt{bag}\,?\,\mathtt{init} \cdot \ell_1 \cdot \mathtt{bag}\,!\,\mathtt{rdy} \cdot D^{\text{init}} \end{array} \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [] \\
\to & \begin{array}{ll} & \mathtt{bag}\,?\,\mathtt{rdy} \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & \mathtt{bag}\,?\,\mathtt{init} \cdot \ell_1 \cdot \mathtt{bag}\,!\,\mathtt{rdy} \cdot D^{\text{init}} \end{array} \blacktriangleleft \mathtt{bag} \mapsto [\mathtt{init}], \mathtt{sys} \mapsto []
\end{array}$$

$$S \to I \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \qquad D \to \texttt{bag}\,?\,\texttt{init} \cdot \ell_1 \cdot \texttt{bag}\,!\,\texttt{rdy} \cdot D^{\text{init}}$$
$$I \to \texttt{bag}\,!\,\texttt{init} \cdot \texttt{bag}\,?\,\texttt{rdy} \qquad D^{\text{init}} \to \texttt{bag}\,?\,\texttt{task} \cdot \texttt{bag}\,!\,\texttt{ok} \cdot D^{\text{msg}}$$
$$P \to \texttt{bag}\,!\,\texttt{task} \cdot \texttt{bag}\,?\,\texttt{ok}.$$

A transition sequence of standard semantics (cont'd)

$$\to \begin{array}{ll} & \texttt{bag}\,?\,\texttt{rdy} \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & \ell_1 \cdot \texttt{bag}\,!\,\texttt{rdy} \cdot D^{\text{init}} \end{array} \blacktriangleleft \texttt{bag} \mapsto [\,], \texttt{sys} \mapsto [\,]$$

$$\to \begin{array}{ll} & \texttt{bag}\,?\,\texttt{rdy} \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & \texttt{bag}\,!\,\texttt{rdy} \cdot D^{\text{init}} \end{array} \blacktriangleleft \texttt{bag} \mapsto [\,], \texttt{sys} \mapsto [\,]$$

$$\to^* \begin{array}{ll} & P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & D^{\text{init}} \end{array} \blacktriangleleft \texttt{bag} \mapsto [\,], \texttt{sys} \mapsto [\,]$$

$$\to^* \begin{array}{ll} & S^{\text{case}} \cdot S^{\text{stop}} \\ \parallel & D^{\text{msg}} \end{array} \blacktriangleleft \texttt{bag} \mapsto [\,], \texttt{sys} \mapsto [\,]$$

In the sequential setting, the control-state reachability problem (of pushdown systems) is of central interest.

APCPS Program-Point Reachability Problem

Given an APCPS and $\ell_1, \cdots, \ell_n \in \mathit{Labels}$, are there $\alpha_1, \cdots, \alpha_n \in \mathit{Terms}$ and channel contents Γ s.t. $S \blacktriangleleft \varnothing \ \rightarrow^* \ \ell_1\,\alpha_1 \parallel \cdots \parallel \ell_n\,\alpha_n \parallel \cdots \blacktriangleleft \Gamma$ (possibly in parallel with some other processes)?

Safety Verification Problems

In the sequential setting, the control-state reachability problem (of pushdown systems) is of central interest.

APCPS Program-Point Reachability Problem

Given an APCPS and $\ell_1, \cdots, \ell_n \in \mathit{Labels}$, are there $\alpha_1, \cdots, \alpha_n \in \mathit{Terms}$ and channel contents Γ s.t. $S \blacktriangleleft \varnothing \ \rightarrow^* \ \ell_1\,\alpha_1 \parallel \cdots \parallel \ell_n\,\alpha_n \parallel \cdots \blacktriangleleft \Gamma$ (possibly in parallel with some other processes)?

APCPS Program-Point Coverability Problem

Given an APCPS and $\ell_1, \cdots, \ell_n \in \mathit{Labels}$, are there configuration $\Pi \blacktriangleleft \Gamma$ and $\alpha_1, \cdots, \alpha_n \in \mathit{Terms}$ such that

1. $S \blacktriangleleft \varnothing \ \rightarrow^* \ \Pi \blacktriangleleft \Gamma$, and
2. $\ell_1\,\alpha_1 \parallel \cdots \parallel \ell_n\,\alpha_n \blacktriangleleft \varnothing \ \leq \ \Pi \blacktriangleleft \Gamma$ (for a fixed well quasi-ordering $\leq$, see next slide).

Question: Is Coverability decidable?

An Approach to Deciding Coverability

A well-structured transition system (WSTS) is a triple $(S, \to, \leq)$ such that

1. $(S, \leq)$ is a well-quasi-order (WQO) i.e. a preorder such that $\forall s_0\, s_1\, s_2 \cdots \in S^\omega \,.\, \exists i < j \,.\, s_i \leq s_j$
2. transition relation $(S, \to)$ is $\leq$-monotone i.e. if $s \to t$ and $s \leq s'$ then there exists t' s.t. $s' \to t'$ and $t \leq t'$
3. for each $s \in S$, $\min(\mathrm{pred}(\uparrow s))$ is computable.

WSTS Coverability Problem

Given a WSTS $(S, \to, \leq)$, a start state and an (error) state s_{err}, is there a reachable element s that covers s_{err} i.e. $s \geq s_{\mathsf{err}}$?

WSTS Coverability is decidable.

(Abdulla et al.: LICS96), (Finkel & Schnoebelen: TCS 2001)

A well-structured transition system (WSTS) is a triple $(S, \rightarrow, \leq)$ such that

1. $(S, \leq)$ is a well-quasi-order (WQO) i.e. a preorder such that $\forall s_0\, s_1\, s_2 \cdots \in S^\omega \,.\, \exists i < j \,.\, s_i \leq s_j$
2. transition relation $(S, \rightarrow)$ is $\leq$-monotone i.e. if $s \rightarrow t$ and $s \leq s'$ then there exists t' s.t. $s' \rightarrow t'$ and $t \leq t'$
3. for each $s \in S$, $\min(\mathrm{pred}(\uparrow s))$ is computable.

WSTS Coverability Problem

Given a WSTS $(S, \rightarrow, \leq)$, a start state and an (error) state s_{err}, is there a reachable element s that covers s_{err} i.e. $s \geq s_{\mathsf{err}}$?

WSTS Coverability is decidable.

(Abdulla et al.: LICS96), (Finkel & Schnoebelen: TCS 2001)

Thus we seek conditions on APCPS that guarantee a well-quasi-ordering of the configurations, with respect to which the (APCPS) transition relation is monotone.

Idea: An APCPS process has shape:

$$\alpha \, \beta_0 \, X_1 \, \beta_1 \, X_2 \, \beta_2 \cdots X_j \, \beta_j \quad \in \quad (\Sigma \cup \mathcal{N})^* / \simeq_{\#}$$

where $\alpha \in \underbrace{\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup \Sigma \cup \{ \epsilon \}}_{\mathit{CtrlState}}$, $\beta_i \in (\mathcal{N}^{\mathrm{com}} \cup \Sigma^{\mathrm{com}})^*$ and $X_i \in (\mathcal{N}^{\neg\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}})$

Idea: An APCPS process has shape:

$$\alpha\ \beta_0\ X_1\ \beta_1\ X_2\ \beta_2 \cdots X_j\ \beta_j \quad \in \quad (\Sigma \cup \mathcal{N})^* / \simeq_{\#}$$

where $\alpha \in \underbrace{\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup \Sigma \cup \{\,\epsilon\,\}}_{\mathit{CtrlState}}$, $\beta_i \in (\mathcal{N}^{\mathrm{com}} \cup \Sigma^{\mathrm{com}})^*$ and $X_i \in (\mathcal{N}^{\neg\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}})$

1. View α as control state, $\beta_0\ X_1\ \beta_1 \cdots X_j\ \beta_j$ as (pushdown) stack

Idea: An APCPS process has shape:

$$\alpha\ \beta_0\ X_1\ \beta_1\ X_2\ \beta_2 \cdots X_j\ \beta_j \quad \in \quad (\Sigma \cup \mathcal{N})^* / \simeq_\#$$

where $\alpha \in \underbrace{\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup \Sigma \cup \{\, \epsilon \,\}}_{\mathit{CtrlState}}$, $\beta_i \in (\mathcal{N}^{\mathrm{com}} \cup \Sigma^{\mathrm{com}})^*$ and $X_i \in (\mathcal{N}^{\neg\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}})$

1. View α as control state, $\beta_0\ X_1\ \beta_1 \cdots X_j\ \beta_j$ as (pushdown) stack
2. "Summarise" the stack as $M_0\ X_1\ M_1 \cdots X_j\ M_j$ where each $M_i := \mathbb{M}[\beta_i]$, is the Parikh image[1] of β_i.

[1] The Parikh image of a word is the number of occurrences of each letter in the word. E.g. Take $\Sigma = \{\, a, b, c, d \,\}$. $\mathbb{M}_\Sigma(b\,a\,c\,a)$ is the multiset $\{\, (a,2), (b,1), (c,1), (d,0) \,\}$

An Abstract Semantics by Summarisation

Idea: An APCPS process has shape:

$$\alpha\ \beta_0\ X_1\ \beta_1\ X_2\ \beta_2 \cdots X_j\ \beta_j \quad \in \quad (\Sigma \cup \mathcal{N})^* / \simeq_{\#}$$

where $\alpha \in \underbrace{\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup \Sigma \cup \{\,\epsilon\,\}}_{\mathit{CtrlState}}$, $\beta_i \in (\mathcal{N}^{\mathrm{com}} \cup \Sigma^{\mathrm{com}})^*$ and $X_i \in (\mathcal{N}^{\neg\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}})$

1. View α as control state, $\beta_0\ X_1\ \beta_1 \cdots X_j\ \beta_j$ as (pushdown) stack
2. "Summarise" the stack as $M_0\ X_1\ M_1 \cdots X_j\ M_j$ where each $M_i := \mathbb{M}[\beta_i]$, is the Parikh image[1] of β_i.
3. The non-commutative non-terminals X_is act as separators of the caches M_js of commutative actions.

[1] The Parikh image of a word is the number of occurrences of each letter in the word. E.g. Take $\Sigma = \{\,a, b, c, d\,\}$. $\mathbb{M}_\Sigma(b\,a\,c\,a)$ is the multiset $\{\,(a, 2), (b, 1), (c, 1), (d, 0)\,\}$

An Abstract Semantics by Summarisation

Idea: An APCPS process has shape:

$$\alpha\ \beta_0\ X_1\ \beta_1\ X_2\ \beta_2 \cdots X_j\ \beta_j \quad \in \quad (\Sigma \cup \mathcal{N})^* / \simeq_\#$$

where $\alpha \in \underbrace{\mathcal{N} \cup (\Sigma \cdot \mathcal{N}) \cup \Sigma \cup \{\, \epsilon \,\}}_{\mathit{CtrlState}}$, $\beta_i \in (\mathcal{N}^{\mathrm{com}} \cup \Sigma^{\mathrm{com}})^*$ and $X_i \in (\mathcal{N}^{\neg\mathrm{com}} \cup \Sigma^{\neg\mathrm{com}})$

1. View α as control state, $\beta_0\ X_1\ \beta_1 \cdots X_j\ \beta_j$ as (pushdown) stack
2. "Summarise" the stack as $M_0\ X_1\ M_1 \cdots X_j\ M_j$ where each $M_i := \mathbb{M}[\beta_i]$, is the Parikh image[1] of β_i.
3. The non-commutative non-terminals X_is act as separators of the caches M_js of commutative actions.
4. Whenever the top separator is popped, the actions of the top cache M_0 is despatched at once.

[1] The Parikh image of a word is the number of occurrences of each letter in the word. E.g. Take $\Sigma = \{\, a, b, c, d \,\}$. $\mathbb{M}_\Sigma(b\,a\,c\,a)$ is the multiset $\{\, (a, 2), (b, 1), (c, 1), (d, 0) \,\}$

$$
\begin{array}{lll}
S & \to & I \cdot P \cdot S^{\text{case}} \cdot S^{\text{stop}} \\
S^{\text{case}} & \to & S \mid \mathtt{sys} \mathbin{?} \mathtt{stop} \\
S^{\text{stop}} & \to & \mathtt{bag} \mathbin{!} \mathtt{stop} \\
I & \to & \mathtt{bag} \mathbin{!} \mathtt{init} \cdot \mathtt{bag} \mathbin{?} \mathtt{rdy}
\end{array}
\qquad
\begin{array}{lll}
P & \to & \mathtt{bag} \mathbin{!} \mathtt{tk} \cdot \mathtt{bag} \mathbin{?} \mathtt{ok} \\
D^{\text{init}} & \to & \mathtt{bag} \mathbin{?} \mathtt{tk} \cdot \mathtt{bag} \mathbin{!} \mathtt{ok} \cdot D^{\text{msg}} \\
D^{\text{msg}} & \to & D \mid \mathtt{bag} \mathbin{!} \mathtt{stop} \cdot \ell_2 \cdot \mathtt{sys} \mathbin{!} \mathtt{d\_done} \\
D & \to & \mathtt{bag} \mathbin{?} \mathtt{init} \cdot \ell_1 \cdot \mathtt{bag} \mathbin{!} \mathtt{rdy} \cdot D^{\text{init}}
\end{array}
$$

A transition sequence

$$
\begin{array}{ll}
 & S \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}] \\
\to & I \cdot P \cdot S^{\text{case}} \cdot [\mathtt{bag} \mathbin{!} \mathtt{stop}] \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}] \\
\to^* & P \cdot S^{\text{case}} \cdot [\mathtt{bag} \mathbin{!} \mathtt{stop}] \parallel D^{\text{init}} \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}] \\
\to^* & S \cdot [\mathtt{bag} \mathbin{!} \mathtt{stop}] \parallel D^{\text{msg}} \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}] \\
\to^* & I \cdot P \cdot S^{\text{case}} \cdot [\mathtt{bag} \mathbin{!} \mathtt{stop}, \mathtt{bag} \mathbin{!} \mathtt{stop}] \parallel D \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}] \\
\to & P \cdot S^{\text{case}} \cdot [\mathtt{bag} \mathbin{!} \mathtt{stop}, \mathtt{bag} \mathbin{!} \mathtt{stop}] \parallel D^{\text{init}} \blacktriangleleft \mathtt{bag} \mapsto [], \mathtt{sys} \mapsto [\mathtt{stop}]
\end{array}
$$

Standard Coverability Reduces to Abstract Coverability

Program-Point Coverability Problem (Abstract Semantics)

Given an APCPS and labels $\ell_1, \cdots, \ell_n$, is there a configuration $\Pi \blacktriangleleft \Gamma$ such that for each $i \in \{1, \cdots, n\}$, there is a process $\lambda_i\ \beta_i \in \Pi$ such that $\lambda_i = l_i$ or ($\lambda_i = M_i$ and $l_i \in M_i$)?

Theorem (Reduction)

An instance of the Program-Point Coverability Problem is a yes-instance according to the standard semantics iff it is a yes-instance according to the abstract semantics.

A Decidable Subclass: APCPS with Shaped Stacks

An APCPS has k-shaped stacks just if (the "stack" of) every reachable process is separated by at most k non-terminals.
An APCPS has shaped stacks if it has k-shaped stacks, for some k.

Theorem

Using the abstract semantics, APCPS with shaped stacks give rise to a WSTS.

Corollary

The Program-Point Coverability Problem is decidable and EXPSPACE*-hard.*

Is the Shaped Constraint Useful in Practice?

The shaped constraint is a "semantic" condition and undecidable. But there is a sufficient syntactic condition.

Proposition (Well-foundedness)

If an APCPS satisfies

Well-foundnedness. There is a well-founded preorder $\succeq$ s.t. for all $A \in \mathcal{N}$ and $B \in \mathrm{RHS}(A) \cap \mathcal{N}$

1. $A \succeq B$, *and*
2. *if* $A \to B\,C$ *is a* $\mathcal{G}$*-rule where* $C \in \mathcal{N}^{\neg\mathrm{com}}$ *then* $A \succ B$

then it has k-shaped stacks for some k.

N.B. The k above is the length of the longest $\succ$-chain.

The condition is quite general and seems practically useful.

Example: The APCPS `server` satisfies the condition.

Outline

1. A Survey of Soter: Automatic Safety-Verification of Erlang Programs

2. A New Model of Asynchronous Message-Passing Concurrency

3. Conclusions and Further Directions

Summary

1. We introduce a new model of computation for asynchronous procedure calls—asynchronous partially commutative pushdown systems (APCPS)—that relaxes the Receivable-Only-When-Stack-is-Empty constraint.
2. Coverability of APCPS with shaped stacks is decidable and EXPSPACE-hard.
3. We give a syntactic sufficient condition for APCPS to have shaped stacks. The condition seems practically useful.

J. Kochems & O.: Safety Verification of Asynchronous Pushdown Systems with Shaped Stacks. Concur 2013.

Further Directions: Asynchronous Partially Commutative Pushdown Systems (APCPS)

1. Determine the precise complexity of deciding Coverability of APCPS with k-shaped stacks
   - We (Kochems) use a new variant Petri nets—Nets with Nested Coloured Data—and have a conjecture.
2. Extend the APCPS framework to higher-order processes.
3. Is the BFC algorithm the basis of an efficient solution for model-checking APCPS?
4. Clarify the connexions between the APCPS approach and partial order reduction. Cf. [Abdullah et al.: POPL14]
   - Is there scope to use (static / dynamic) partial order reduction to further optimise APCPS?

Further Directions, cont.

4. Use π-calculus (rather than ACS) as intermediate models of computation
   - Fragments of π-calculus that are decidable models of computation: depth-bounded / mixed-bounded / breadth-bounded fragments map ("bisimilarly") into WSTS, Petri nets and bounded Petri nets. (Roland Meyer: PhD thesis 2008)
   - Membership of these fragments are undecidable. We (D'Osualdo) aim to develop static analysis based on behavioural types and / or graph-grammatical analysis.