Verification of Recursive Asynchronous Concurrency

Luke Ong

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

University of Oxford

IFIP WG 2.2 Meeting, Munich, 15-18 September 2014

Outline

1. A Quick Overview of SOTER: Automatic Safety Verification of Erlang Programs
2. A New Model of Recursive Asynchronous Concurrency: Asynchronously Communicating Context-Free Grammar (ACCFG) and the Shaped Constraint
3. Nested Nets with Coloured Tokens (NNCT) and TOWER-Complete Coverability
4. Conclusions and Further Directions

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.

Verifying Erlang programs is inherently difficult.

Erlang's state space has many sources of infinity	Abstraction
1. Sequential Erlang is already Turing complete.	finite
2. Message space is unbounded.	finite
3. Value domains are infinite.	finite
4. Arbitrarily many processes can be spawned dynamically.	counter
5. Mailboxes have unbounded capacity.	counter

What's decidable about Erlang?

Almost nothing interesting!

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 fixed at 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

Soter – A Safety Verification Tool

Take (Core) Erlang code as source. Two-stage abstraction (A1, A2) + model checking (MC).

A1 Perform a k-CFA-like analysis to construct abstractions of data and control-flow.

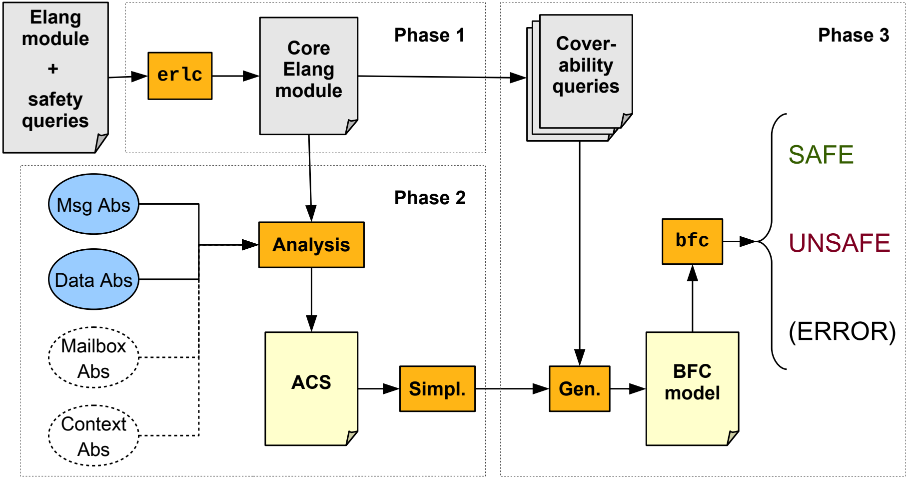
A2 Bootstrap the analysis to yield an Actor Communicating System (ACS)—a CCS-like infinite-state model—that soundly approximates the program via a counter abstraction of three quantities: ι, q, m

- 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

MC Model-check the ACS using a vector addition system coverability checker (BFC)

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: 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. *Static Analysis Symposium* (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

- Unable to 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.

Asynchronous Programming Style

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

E.g. `async` – a common construct in modern concurrency-oriented languages; e.g. Microsoft's F#, IBM's X10, Haskell, etc.

Asynchronous programming is used to build fast servers, routers, sensor networks; basis of web programming in Ajax.

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`?

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).

Asynchronous Communicating Pushdown Systems: Related Work

Various ways to achieve decidability:

- Asynchronous procedure calls – empty-stack constraint
  (Sen & Viswanathan: CAV06), (Jhala & Majumdar: POPL07), (Ganty et al.: POPL09)
- Hierarchical communication topology
  (Bouajjani & Emmi: POPL12), (Bouajjani et al.: Concur05)
- Synchronisation via 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)

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 practical algorithms?

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**) do not.

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. ($\Sigma^{\neg\text{com}}$) parts.

A New Model of Asynchronous Concurrency: Notation

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

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

Set terminal symbols (= concurrency/communication actions)

$$\Sigma := \mathit{Sends} \cup \mathit{Receives} \cup \mathit{Spawns}.$$

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

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

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: ACCFG

Given $Chan$, Msg, and $\mathcal{N}$, an **asynchronously communicating context free grammar** (ACCFG) is a tuple $(\Sigma, \#, \mathcal{N}, \mathcal{R}, S)$ where

- $\Sigma := Sends \cup Receives \cup Spawns$ is a finite set of terminal symbols (= conc./comm. 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 context-free 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 $(\Sigma \cup \mathcal{N})^* / \simeq_{\#}$.

N.B. Equivalent presentation using asynchronously communitating pushdown systems, ACPS.

Example: ACCFG

```
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 ACCFG 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}

Standard Semantics of ACCFG

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

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

Standard Semantics: Some Rules

Fix ACCFG $(\Sigma, \#, \mathcal{N}, \mathcal{R}, S)$. Set $\mathit{Terms} := (\Sigma \cup \mathcal{N})^* / \simeq_\#$.

Define binary relation $\rightarrow$ over $\mathit{Config} := \mathbb{M}[\mathit{Terms}] \times (\mathit{Chan} \rightarrow \mathbb{M}[\mathit{Msg}])$.

$$A\,\gamma \parallel \Pi \blacktriangleleft \Gamma \;\rightarrow\; B\,C\,\gamma \parallel \Pi \blacktriangleleft \Gamma \qquad (\text{`}A \rightarrow B\,C\text{'} \;\in\; \mathcal{R})$$

$$(c\,?\,m)\,\gamma \parallel \Pi \blacktriangleleft ([m] \oplus l)^c, \Gamma \;\rightarrow\; \gamma \parallel \Pi \blacktriangleleft l^c, \Gamma$$

$$(c\,!\,m)\,\gamma \parallel \Pi \blacktriangleleft l^c, \Gamma \;\rightarrow\; \gamma \parallel \Pi \blacktriangleleft ([m] \oplus l)^c, \Gamma$$

$$(\nu X)\,\gamma \parallel \Pi \blacktriangleleft \Gamma \;\rightarrow\; \gamma \parallel X \parallel \Pi \blacktriangleleft \Gamma$$

We order processes (elements of Terms) $\delta\,\pi_0 \leq_{\mathit{Procs}} \delta\,\pi_1$ just if there exist π_0' and π_1' such that $\delta\,\pi_0' \leq_{\mathrm{Hig}} \delta\,\pi_1'$ and both $\delta\,\pi_i \simeq_{\#} \delta\,\pi_i'$.

We lift $\leq_{\mathit{Procs}}$ to a preorder $\leq$ over Config, using the multiset and function extension.

ACCFG Coverability Problem

Given an ACCFG and configuration $\Pi_0 \blacktriangleleft \Gamma_0$, is there a configuration $\Pi \blacktriangleleft \Gamma$ such that

1. $S \blacktriangleleft \varnothing \;\rightarrow^* \; \Pi \blacktriangleleft \Gamma$, and
2. $\Pi_0 \blacktriangleleft \Gamma_0 \;\leq\; \Pi \blacktriangleleft \Gamma$?

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)

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

An Abstract Semantics by Summarisation

Idea: An ACCFG process (element of Term) 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 $X_i \in \mathcal{N}^{\neg\mathrm{com}}$

1. View α as control state, and $\beta_0\ X_1\ \beta_1 \cdots X_j\ \beta_j$ 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.

Standard Coverability Reduces to Abstract Coverability

Theorem (Reduction)

An instance of the 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: ACCFG with Shaped Constraint

An ACCFG is k-shaped just if every reachable process has at most k occurrences of non-commutative non-terminals.

An ACCFG satisfies the shaped constraint if it is k-shaped, for some k.

Theorem

Using the abstract semantics, shaped ACCFG gives rise to a WSTS.

Corollary

The Coverability Problem for shaped ACCFG is decidable and EXPSPACE*-hard.*

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

Is the Shaped Constraint Useful in Practice?

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

Proposition (Well-foundedness)

If an ACCFG $\mathcal{G}$ 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 is k-shaped, 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 ACCFG `server` satisfies the condition.

Nested Nets with Coloured Tokens (NNCT)

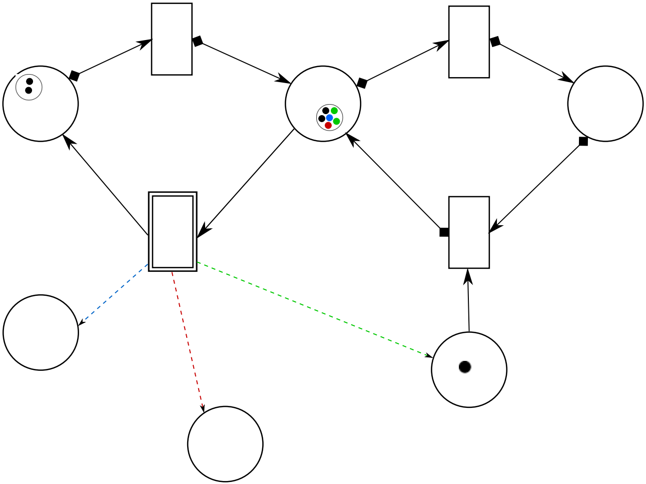
Question

What is the complexity of coverability for k-shaped ACCFG?

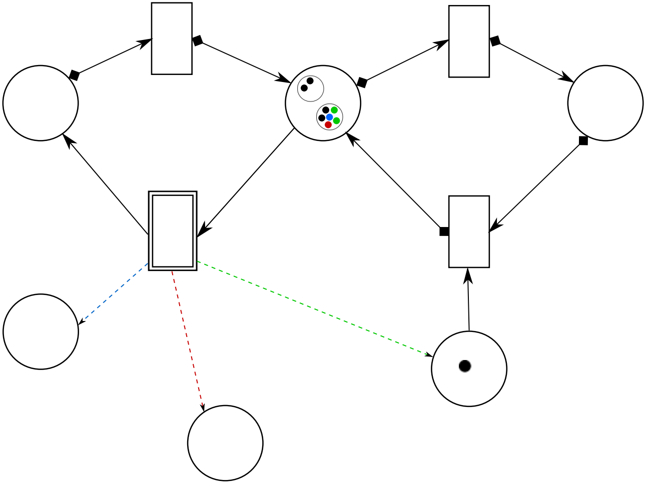
We introduce a new extension of Petri nets as a model of ACCFG.

- Simple places may contain (ordinary) tokens.
- Complex places may contain complex tokens. A complex token is a bag of coloured tokens.
- Three kinds of transition:
  1. simple transition: new complex tokens may be created, and empty complex token removed
  2. complex transition: complex tokens may be removed from a complex place and added (with possibly an injection of tokens) to another complex place
  3. transfer transition: coloured tokens of a complex token are flushed out to simple places

Example: Nested Nets with Coloured Tokens (NNCT)

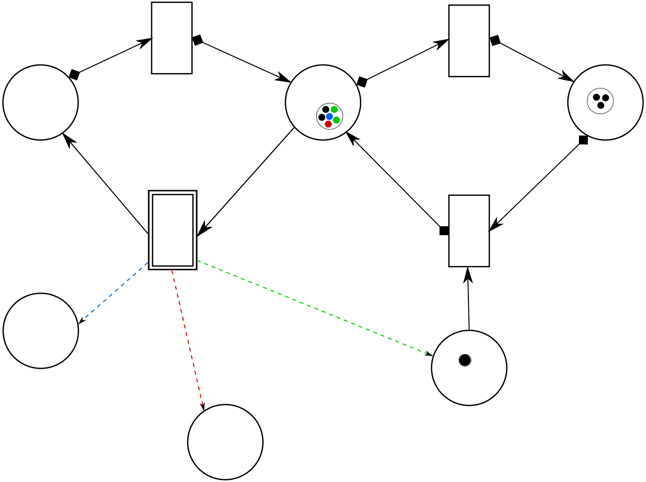


Example: NNCT

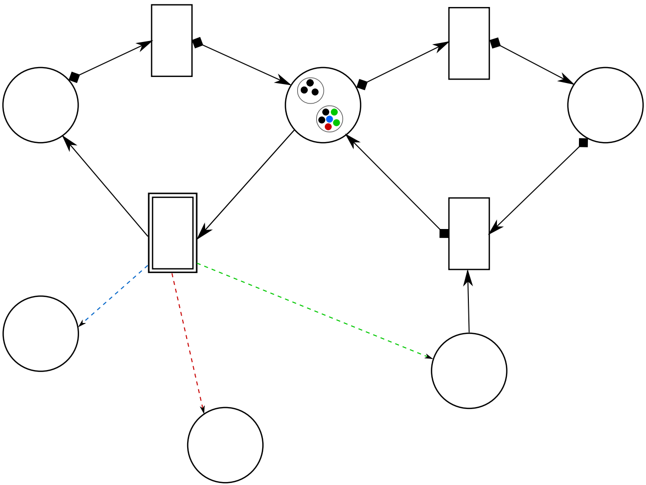


Complex transition

Example: NNCT



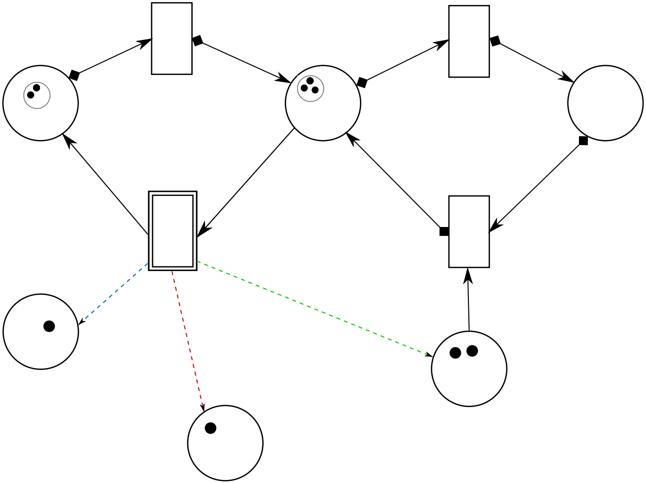
Complex transition



Complex transition

Consider the complex token with 2 black, 2 green, 1 blue, and 1 red tokens.

Example: NNCT



Transfer transition: contents of the complex tokens (2 black, 2 green, 1 blue, 1 red) are "flushed out".

Extensions of Petri nets and Complexity of Coverability

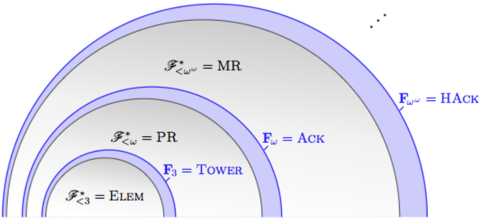
- Petri nets with transfer arcs: non-PR (Schnoebelen MFCS 2010)
- Petri nets with reset arcs: non-PR (Dufourd, Finkel and Schnoebelen ICALP 1998)
- Nested Petri nets: Ackermanian (Lamazova & Schnoebelen Ershov 1999)
- Data nets (various versions): undecidable, Ackermanian and TOWER-hard but no upper bound (Lazic, Newcomb, Ouaknine, Worrell and Roscoe ICATPN07)

In all these extensions, coverability is not primitive recursive, if decidable.

In contrast, NNCT coverability is in TOWER.

To our knowledge, NNCT is the first extension of Petri nets (with infinite token types) that has primitive recursive coverability. (Cf. Branching VAS, Lazic & Schmitz LICS14)

TOWER (Schmitz 2013): A new complexity class between ELEM and PR
Intuitively a TOWER-complete problem "spans" infinitely-many finite towers-of-exponentials.



Sylvain Schmitz: Complexity Hierarchies Beyond Elementary. arXiv 20 Dec 2013.

Examples of TOWER-complete problems:

1. SFEq (Stockmeyer & Meyer STOC 1973)
2. WS1S Satisfiability
3. Higher-Order Model Checking (Ong LICS 2006)
4. NNCT Coverability

Complexity of ACCFG Coverability via NNCT

Theorem (Inter-reducibility)

Shaped-ACCFG coverability and NNCT coverability are elementarily inter-reducible.

Idea: Given an ACCFG, we define a simulating NNCT:

- Use simple places to monitor channel contents & (pending) spawns
- Encode processes as complex tokens: for each $a \in \Sigma$ and $i < K$, allocate a colour for (a, i) in order to encode summaries as coloured tokens.

Theorem

Coverability of NNCT is $\textsc{Tower}$*-complete.*

- Upper bound: a novel "nested" Rackoff argument
- Lower bound: a modified Stockmeyer's ruler-construction

Summary

1. We introduce a new model of computation for asynchronous procedure calls—*shaped* asynchronously communicating context free grammar (ACCFG)—that relaxes the standard Receivable-Only-When-Stack-is-Empty constraint.
2. Coverability of shaped ACCFG is decidable and TOWER-complete.
3. We give a syntactic sufficient condition for ACCFG to have shaped stacks. The condition seems practically useful.
4. We introduce the first extension of Petri nets (with infinite token types)—Nested Nets with Coloured Tokens (NNCT)—with PR coverability.

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

J. Kochems & O.: Decidable Models of Recursive Asynchronous Concurrency. Preprint, 2014.

Further Directions

1. Extend the ACCFG framework to higher-order processes.
2. Is the BFC algorithm the basis of an efficient solution for model-checking ACCFG?
3. 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 and Ong) aim to develop a type-based static analysis for a fragment of depth-bounded π-terms.

FAQ

Surely TOWER complexity is too high for any practical purposes!

Answer

Verification problems of Tower-complete worst-case complexity is not doomed to fail.
Recent advances in algorithm design for problems of comparable or higher complexity:

1. Higher-order Model Checking (Ramsay et al., POPL 2014; Tower-complete) `http://mjolnir.cs.ox.ac.uk/web/preface`
2. Safety verification of concurrent C programs with broadcast by BFC, equivalent to Petri nets with transfer arcs (Kaiser et al., CONCUR 2012; non-primitive recursive)

FAQ

Is "shaped ACCFG" relevant to real-life verification?

Answer

Yes. Here are some examples:

1. When modelling recursion and values returned asynchronously by procedures e.g. via promises, delay, or C++11's std::future (see Sec. 6 para. 2).
2. Erlang behaviour, a functional interface that abstracts away a variety of concurrent interactions. This promotes a programming style that does not fit the empty-stack restriction.
3. Replicated worker pattern: Tasks are recursively decomposed and possibly returned to the distributor. Workers also interact with a shared resource. Such programs are modelled by shaped ACCFG and naturally arise from abstracting Erlang programs produced by SOTER.

FAQ

What are the differences between empty-stack constraint and 1-shaped constraint?

Answer

Empty-stack restriction limits a process to remember only a bounded amount of information along a receive transition (thus finite state).

A 1-shaped ACCFG can remember a commutative stack of arbitrary height along a receive transition (and so, infinite state). We exploit the latter in the lower-bound proof.