First-Order Logic for the Analysis of Programs on Weak Memory Models

Alexei Lisitsa

Department of Computer Science
University of Liverpool,
Liverpool, UK

AI4FM Worskhop, September 1, 2015

Outline

- Verification by finite countermodel finding
- First-order encoding of concurrent programs on weak memory models
- Case study: Burns mutual exclusion protocol:
  - erroneus traces from from first-order proofs
  - finite countermodes to verify corrected version
- Concluding remarks

Reachability as deducibility

- Many problems in verification can be naturally formulated in terms of *reachability* within transition systems;
- We propose to use deducibility (or derivability) in first-order predicate logic to model reachability in transition systems of interest;
- Then verification can be treated as theorem (dis)proving in classical predicate logic;
- Many automated tools (provers and model finders) are readily available.

Reachability as deducibility

- The states of a system to be verified are encoded by (tuples of) terms;
- The transitions of the system are encoded within a first-order theory T such that $T \vdash R((t_1, t_2)$ holds iff the state encoded by t_2 is reachable from the state encoded by t_1
- A variant: $T \vdash R(t)$ iff the state encoded by t is reachable from some of the initial states
- Under such assumptions:
  - Establishing reachability $\equiv$ theorem proving
  - Establishing non-reachability $\equiv$ theorem disproving

Verification of safety

- Safety $\equiv$ non-reachability of "bad" states
- Verification of safety properties $\equiv$ theorem disproving
- To disprove $\varphi \models \psi$ it is sufficient to a find a countermodel for $\varphi \to \psi$, or which is the same a model for $\varphi \wedge \neg\psi$
- In general, such a model can be inevitably infinite and the set of satisfiable first-order formulae is not r.e.
- One can not hope for full automation here
- Our proposal: use automated finite model finders/builders

Remarks

- For the verification of safety the weaker assumption on the encoding is sufficient:
  - $s \rightarrow^* s' \Rightarrow \varphi_s \vdash \varphi_{s'}$
- For the verification of parameterized systems general idea of reachability as deducibility should be suitably adjusted
  - depends on particular classes of systems
  - unary or binary predicates modeling reachabiity can be used

Origins

- The idea of using finite model finders for verification was proposed and developed in the area of verification of security protocols in the following papers (at least):
  - C. Weidenbach Towards an Automatic Analysis of Security Protocols in First-Order Logic, in H. Ganzinger (Ed.): CADE-16, LNAI 1632, pp. 314–328, 1999.
  - Selinger, P.: Models for an adversary-centric protocol logic. Electr. Notes Theor. Comput. Sci. 55(1) (2001);
  - Goubault-Larrecq, J.: Towards producing formally checkable security proofs, automatically. In: Computer Security Foundations (CSF), pp. 224238 (2008)
  - Jan Jurjens and Tjark Weber, Finite Models in FOL-Based Crypto-Protocol Verification. Foundations and Applications of Security Analysis, LNCS 5511, 2009.

Further developments

- Lossy Channel Systems AL ATVA 2010
- Cache Coherence Protocols AL ATVA 2010
- Linear Systems of Automata and Monotone Abstraction AL CoRR abs/1011.0447: (2010), JAR 2013
- Regular Model Checking AL CoRR abs/1011.0447: (2010)
- Regular Tree Model Checking AL CoRR abs/1107.5142:(2011), TTATT 2012 Workshop, Nagoya, RTA'12
- Safety for general TRS and Tree Automata Completion RTA 2012

Verification of Programs on Weak Memory Models

- **Here:** we explore the applicability of FCM approach for Verification of Programs on Weak Memory Models

Verification of Programs on Weak Memory Models

- **Here:** we explore the applicability of FCM approach for Verification of Programs on Weak Memory Models
- **Model:** we adopt a formal model of *concurrent programs* largely from
  - A.Linden, P. Wolper, An Automata-Based Symbolic Approach for Verifying Programs on Relaxed memory Models, SPIN 2010

Verification of Programs on Weak Memory Models

- **Here:** we explore the applicability of FCM approach for Verification of Programs on Weak Memory Models
- **Model:** we adopt a formal model of *concurrent programs* largely from
  - A.Linden, P. Wolper, An Automata-Based Symbolic Approach for Verifying Programs on Relaxed memory Models, SPIN 2010
- **Semantics:** we adopt a *store-buffer semantics* for PSO memory model largely from
  - M. Faouzi Atig, A, Bouahhani, S. Burckhard, M. Musuvathi on the Verification Problem for Weak Memory Models, POPL 2010

Concurrent Programs

A *concurrent program* is a tuple $P = (\mathcal{P}, \mathcal{M}, \mathcal{D}, I)$ where

- $\mathcal{D}$ ia a finite data domain;
- $\mathcal{P} = \{p_1, \ldots, p_n\}$ is a finite set of processes;
- $\mathcal{M} = \{m_1, \ldots, m_k\}$ is a finite set of memory locations;
- A function $\mathcal{I} : \mathcal{M} \to \mathcal{D}$ represents an initial content of the memory.

Each process p_i is defined by

- a finite set $\mathcal{L}(p_i)$ of control locations;
- an initial location $l_0(p_i)$;
- and a set of transitions $T(p_i) \subseteq \mathcal{L}(p_i) \times \mathcal{O} \times \mathcal{L}(p_i)$.

Here O is a set of operations which contains the following memory operations:

- $store(p_i, m_j, d)$, a process p_i stores value $d \in \mathcal{D}$ to memory location m_j
- $load(p_i, m_j, d)$, a process p_i loads the value stored in m_i and checks that its value is d. If the stored value is different from d, the transition is not possible.

PSO semantics

In short:

- the order between writing (store) operations to *different memory* locations is not preserved and during the execution one operation may overtake another
- At the same time the order of operations issued by the same process for the same memory location is preserved.

Store-buffer semantics for PSO

- Assign a buffer (a queue) b_{ij} for every pair (p_i, m_j), where $p_i \in \mathcal{P}, m_j \in \mathcal{M}$
- The buffer b_{ij} keeps the *store* operations issued by process i concerning the memory locaton m_j which have not taken yet effect on the memory. Formally $b_{ij} \in D^*$.
- For the purpose of logical modelling we represent the buffers as terms build from contstants from $D \cup \{e\}$ and the associative binary functional symbol $*$ (concatenation). The constant e denotes the empty buffer.

Store-buffer semantics for PSO (cont.)

- The program state is defined as a triple (L, V, B), where $L \in \mathcal{L}(p_1) \times \ldots \times \mathcal{L}(p_i) \times \ldots \times \mathcal{L}(p_n)$, $V : \mathcal{M} \to \mathcal{D}$ and $B = \{b_{ij} \mid p_i \in \mathcal{P}, m_j \in \mathcal{M}\}$.
- The PSO operational semantics of the program P is given by a transition relation $\Rightarrow_P^{pso}$ defined on the program states as follows:
  $(L, V, B) \Rightarrow_P^{pso} (L', V', B')$ iff (L, V, B) coincides with (L', V', B') everywhere with one of the following exceptions:
  - there is a transition $(l, store(p_i, m_j, d), l')$ in $T(p_i)$ such that $l = L(p_i)$, $l' = L'(p_i)$, $\forall j\ V(m_j) = V'(m_j)$, $b'_{ij} = d * b_{ij}$, or
  - there is a transition $(l, load(p_i, m_j, d)l')$ in $T(p_i)$ such that $l = L(p_i)$, $l' = L'(p_i)$, and either $b_{ij} = d * x$ or $b_{ij} = e$ and $V(m_j) = d$, or
  - $L = L'$ and there exists i and j such that $b_{ij} = b'_{ij} * d$, $V'(m_j) = d$.

Store-buffer semantics: initial state

- Given a concurrent program $P = (\mathcal{P}, \mathcal{D}, \mathcal{M}, \mathcal{I})$ its *intial program state* in the PSO model is defined as $(l_0, \mathcal{I}, B_\emptyset)$ where $l_0 = l_0(p_1), l_0(p_2), \ldots, l_0(p_n)$ and $B_\emptyset = \{b_{ij} = e | p_i \in \mathcal{P}, m_j \in \mathcal{M}\}$.

Synchornization Operations

- The effect of $commit(p_i, m_j)$ operation is to commit to the memory location m_j the full content of a buffer b_{ij};
- The effect of *sync* operation is to commit to the memory the full content of all buffers;
- The model of concurrent programs is extended by allowing a transition of the form $(l, commit(p_i, m_j), l)$ and $(l, sync, l')$ to be among the transitions of a process in a program.

Safety Verification Problem

The verification problem we address in this talk is formulated as follows

Given: A concurrent program $P = (\mathcal{P}, \mathcal{M}, \mathcal{D}, \mathcal{I})$ and a finite set $U \subseteq \Pi_{i=i}^{n} \mathcal{L}(p_i)$ of unsafe vectors of local states;

Question: Is it true that none of states (L, V, B) with $L \in U$ is reachable by execution of P under PSO semantics?

We show now how to translate the safety verification problem into a (dis)proving a first-order formula.

Towards FO modeling of PSO

Program states as tuples of terms:

- $\tilde{\tau}((L, V, B)) =$
  $(s_1, \ldots, s_n, d_1, \ldots, d_m, \tau(b_{11}), \ldots, \tau(b_{1m}), \ldots, \tau(b_{n1}), \ldots, \tau(b_{nm}))$

Towards FO modeling of PSO

Program states as tuples of terms:

- $\tilde{\tau}((L, V, B)) =$
  $(s_1, \ldots, s_n, d_1, \ldots, d_m, \tau(b_{11}), \ldots, \tau(b_{1m}), \ldots, \tau(b_{n1}), \ldots, \tau(b_{nm}))$
- Optimized translation: $\tau((L, V, B)) = (s_1, \ldots, s_n; \tau(b_{11}) *$
  $d_1, \ldots, \tau(b_{1m}) * d_m, \ldots, \tau(b_{n1}) * d_1, \ldots, \tau(b_{nm}) * d_m)$

FO modeling of PSO

Claim Given a concurrent read-only sharing program P one may effectively construct a first-order formula Φ_P such that for any progam state S of P:

- $(l_0, \mathcal{I}, B_\emptyset) \Rightarrow_P^* S$, iff $\Phi_\mathcal{P} \vdash R(\tau(S))$.

R here is an unary predicate denoting reachability.

Burns mutual exclusion protocol

```
//    process[0]
1 while true
2    store flag[0]=1;
3    load _flag=flag[1];
4    if _flag==1 goto 4;
5  //Critical Section
6  store flag[0]=0;
```

```
//    process[1];
1 while true
2   store flag[1]=0;
3   load _flag=flag[0];
4   if _flag==1 goto 2;
5   store flag[1]=1;
6   load _flag=flag[0];
7   if _flag==1 goto 2;
8  //Critical Section
9   store flag[1] = 0;
```

Concurrent program for Burns mutex

Let $\mathcal{D} = \{0, 1\}$, $\mathcal{P} = \{p_0, p_1\}$ and $\mathcal{M} = \{m_o, m_1\}$. We use m_0 and m_1 to represent `flag[0]` and `flag[1]` above.
$\mathcal{L}(p_0) = \{a_0, a_1, a_2\}$ and a set $T(p_0)$ is listed below

- $(a_0, store(p_0, m_0, 1), a_1)$
- $(a_1, load(p_0, m_1, 1), a_1)$
- $(a_1, load(p_0, m_1, 0), a_2)$
- $(a_2, store(p_0, m_0, 0), a_0)$

$\mathcal{L}(p_1) = \{a_0, a_1, a_2, a_3, a_4\}$ and a set $T(p_1)$is listed below

- $(a_0, store(p_1, m_1, 0), a_1)$
- $(a_1, load(p_1, m_0, 1), a_0)$
- $(a_1, load(p_1, m_0, 0), a_2)$
- $(a_2, store(p_1, m_1, 1), a_3)$
- $(a_3, load(p_1, m_0, 1), a_0)$
- $(a_3, load(p_1, m_0, 0), a_4)$
- $(a_4, store(p_1, m_1, 0), a_0)$

Initial program state is $(a0, a0, 0, 0)$ *Unsafe vector of local states* is (a_2, a_4).

FO encoding Φ of Burns protocol

```
(x * y) * z = x * (y * z).

N(0).
N(1).
N(x) -> N(x * 1).
N(x) -> N(x * 0).

R(a0,a0,0,0).
...
R(a0,v1,x,v2) -> R(a1,v1,1*x,v2).

R(a1,xx,x,yy*1) -> R(a1,xx,x,yy*1).
R(a1,xx,x,1) -> R(a1,xx,x,1).
R(a1,xx,x,yy*0) -> R(a2,xx,x,yy*0).
R(a2,v1,x,v2) -> R(a0,v1,0*x,v2).
R(x,w1,z*0,w2) & N(z) -> R(x,w1,z,w2).
R(x,w1,z*1,w2) & N(z) -> R(x,w1,z,w2).
```

(cont. at the next page)

FO encoding Φ of Burns protocol

```
R(x1,a0,x2,y) -> R(x1,a1,x2,0*y).
R(x,a1,y*1,yy) -> R(x,a0,y*1,yy).
R(x,a1,1,yy) -> R(x,a0,1,yy).
R(x,a1,y*0,yy) -> R(x,a2,y*0,yy).
R(x,a1,0,yy) -> R(x,a2,0,yy).
R(x,a2,y,yy) -> R(x,a3,y,yy*1).
```

```
R(x,a3,y*1,yy) -> R(x,a0,y*1,yy).
R(x,a3,1,yy) -> R(x,a0,1,yy).
R(x,a3,y*0,yy) -> R(x,a4,y*0,yy).
R(x,a3,0,yy) -> R(x,a4,0,yy).
R(x1,a4,x2,yy) -> R(x1,a0,x2,0*yy).
```

```
R(w1,x,w2,z*0) & N(z)-> R(w1,x,w2,z).
R(w1,x,w2,z*1) & N(z) -> R(w1,x,w2,z).
```

Adequacy of encoding and safety condition

Proposition

A program state S of a concurrent program Burns is reachable, i.e. $(l_0, \mathcal{I}, B_\emptyset) \Rightarrow_P S$, iff $\Phi \vdash R(\tau(S))$.

Corollary

Burns protocol is safe iff $\Phi \nvdash \exists x_1 \exists x_2 R(a_2, a_4, x_1, x_2)$

Verification attempt

- Apply automated theorem prover (Prover9 in our case) to $\Phi \rightarrow \exists x_1 \exists x_2 R(a_2, a_4, x_1, x_2)$;
- it returns in 0.09s with a proof $\Rightarrow$ Burns is incorrect under PSO semantics

Verification attempt

- Apply automated theorem prover (Prover9 in our case) to $\Phi \rightarrow \exists x_1 \exists x_2 R(a_2, a_4, x_1, x_2)$;
- it returns in 0.09s with a proof $\Rightarrow$ Burns is incorrect under PSO semantics
- Execution trace violating the property can be extracted from the proof

Form proof to a trace

Prover9 is a resolution based theorem prover and the proof consists of the derivation of the contradiction from $\Phi \wedge \neg\Psi$. Here is the final part of the proof, consisting of the clauses with single atomic formulae:

```
...
31 R(a0,a0,0,0).  [assumption].
...
49 -R(a2,a4,x,y).  [deny(24)].
58 R(a0,a1,0,0 * 0).  [hyper(36,a,30,a)].
66 -R(a2,a3,x * 0,y).  [ur(44,b,49,a)].
94 -R(a2,a2,x * 0,y).  [ur(41,b,66,a)].
166 R(a0,a2,0,0 * 0).  [hyper(40,a,58,a)].
191 -R(a1,a2,x * 0,y * 0).  [ur(32,b,94,a)].
194 R(a1,a2,1 * 0,0 * 0).  [hyper(31,a,166,a)].
195 $F.  [resolve(194,a,191,a)].
```

From the proof to a trace

Unsafe trace:

$(a0, a0, 0, 0)$, $(a0, a1, 0, 0 * 0)$,
$(a0, a2, 0, 0 * 0)$,$(a1, a2, 1 * 0, 0 * 0)$,
$(a2, a2, 1 * 0, 0)$, $(a2, a3, 1 * 0, 0)$,
$(a2, a4, 1 * 0, 0)$

Safe Burns Protocol and its verification by disproving

- Burns protocol can be made safe by inserting appropriate fences (synchtonization operations);
- Its corrected version can be automatically verified by demonstrating $\Phi' \nvdash \exists x_1 \exists x_2 R(a_3, a_4, x_1, x_2)$ using a finite model finder (Mace4);
- Mace4 returns a countermodel of size 4 in 0.3s;
- The protocol is safe

Concluding Remarks

- We have demonstrated how the verification of weak memory models can be done by first-order logic (dis)proving;

Concluding Remarks

- We have demonstrated how the verification of weak memory models can be done by first-order logic (dis)proving;
- Some other classical examples (Paterson protocol, Dekker protocol) have been verified too;
- **Advantages of the approach:**
  - It is simple;
  - It re-uses the existsing tools in a modular way;

Concluding Remarks

- We have demonstrated how the verification of weak memory models can be done by first-order logic (dis)proving;
- Some other classical examples (Paterson protocol, Dekker protocol) have been verified too;
- **Advantages of the approach:**
  - It is simple;
  - It re-uses the existsing tools in a modular way;
- **Further work is required:**
  - Automation of unsafe traces extraction from proofs
  - Comparison with other approaches
  - Scalability