Thread-modular Analyses by Abstract Interpretation

Original author: Antoine Miné

presented by Adam

Masaryk University

Brno, Czech Republic

November 12, 2017

Motivation

- We would like to analyse concurrent programs by analysing each thread in isolation.
- That is not sound, but we can use thread-modular approach.
- We can analyse the threads separately, if we know what interferences from other threads can happen.

Paper's goals

- To extend *Astrée*, a static analyser of embedded C programs, with ability to analyse concurrent avionic programs.[1]
- To lower number of false alarms by using good abstractions.
- To reduce cost of such analyses by using thread-modular approach.

[1] These do not use, for example, recursion or dynamically allocated threads.

Example program

t_a	t_b
1: **while** random **do**	1: **while** random **do**
2: **if** $X < Y$ **then**	2: **if** $Y < 100$ **then**
3: $X \leftarrow X + 1$	3: $Y \leftarrow Y + [1, 3]$
4: **end if**	4: **end if**
5: **end while**	5: **end while**

Rely-guarantee proof method

Basis of thread-modular analysis

- Extension of Hoare's logic to concurrent programs.
- $\{P\}s\{Q\}$ is extended to $R, G \vdash \{P\}s\{Q\}$
  - R – all changes from all other threads
  - G – all changes from this thread
- Modular – doesn't check other threads, but only R

Rely-guarantee proof method

Basis of thread-modular analysis

- Extension of Hoare's logic to concurrent programs.
- $\{P\}s\{Q\}$ is extended to $R, G \vdash \{P\}s\{Q\}$
  - R – all changes from all other threads
  - G – all changes from this thread
- Modular – doesn't check other threads, but only R

Our example program is tightly coupled:

$R_1 = G_2$: Y is incremented
X is unchanged
$0 \leq Y \leq 102$

$R_2 = G_1$: Y is unchanged
$0 \leq X \leq Y$

Rely-guarantee proof method

Basis of thread-modular analysis

- Extension of Hoare's logic to concurrent programs.
- $\{P\}s\{Q\}$ is extended to $R, G \vdash \{P\}s\{Q\}$
  - R – all changes from all other threads
  - G – all changes from this thread
- Modular – doesn't check other threads, but only R

Our example program is tightly coupled:

$R_1 = G_2$: Y is incremented
X is unchanged
$0 \leq Y \leq 102$

$R_2 = G_1$: Y is unchanged
$0 \leq X \leq Y$

This proves that $X \leq Y$ holds, but relies on annotations.

Goal. Infer such invariants automatically.

Transition systems

Definition

Transition system is a quadruple $(\Sigma, \mathcal{A}, I, \tau)$, where

- Σ is a set of states
- $\mathcal{A}$ is a set of actions
- $I \subseteq \Sigma$ is a set of initial states
- $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$ is a transition relation

For transitions $\langle \sigma, a, \sigma' \rangle \in \tau$, we will use notation $\sigma \xrightarrow{a}_{\tau} \sigma'$

Programs

Program is composed of

- $\mathcal{T}$ – finite set of threads
- $\mathcal{L}$ – set of program locations
- $\mathcal{V}$ – set of (global) variables with values from domain $\mathbb{V}$
- Inst – instructions (assignments and comparisons)

Each thread $t \in \mathcal{T}$ has associated

- $e_t \in \mathcal{L}$ – an entry point
- $\mathit{inst}_t \subseteq \mathcal{L} \times \mathit{Inst} \times \mathcal{L}$ – instruction set

Programs as transition systems

- $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M}$, where
  - *Control state* $L \in \mathcal{C} \stackrel{\text{def}}{=} \mathcal{T} \to \mathcal{L}$
  - *Memory state* $\rho \in \mathcal{M} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{V}$
- $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T}$
- $I \stackrel{\text{def}}{=} \langle \lambda t.e_t, \lambda V.0 \rangle$
- τ models execution steps of the program, so that only one thread changes its control state at the time.

Monolithic concrete semantics

"The old way"

Trace semantics $\mathcal{F}$ – finite partial traces of a transition system.

$$\mathcal{F} \stackrel{\text{def}}{=} \mathbf{lfp}\, F$$

$$F \stackrel{\text{def}}{=} \lambda X.I \cup \left\{\sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \stackrel{a_{i+1}}{\to} \sigma_{i+1} \mid \sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \in X \wedge \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1}\right\}$$

Monolithic concrete semantics

"The old way"

Trace semantics $\mathcal{F}$ – finite partial traces of a transition system.

$$\mathcal{F} \stackrel{\text{def}}{=} \mathbf{lfp}\, F$$

$$F \stackrel{\text{def}}{=} \lambda X.I \cup \{\sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \stackrel{a_{i+1}}{\to} \sigma_{i+1} \mid \sigma_0 \stackrel{a_1}{\to} \cdots \sigma_i \in X \wedge \sigma_i \stackrel{a_{i+1}}{\to}_\tau \sigma_{i+1}\}$$

Abstracting $\mathcal{F}$ by stripping the state history, we obtain *state semantics* $\mathcal{R}$ – states reachable in a program trace.

$$\mathcal{R} \stackrel{\text{def}}{=} \mathbf{lfp}\, R$$

$$R \stackrel{\text{def}}{=} \lambda S.I \cup \{\sigma' \mid \exists \sigma \in S, a \in \mathcal{A} : \sigma \stackrel{a}{\to}_\tau \sigma'\}$$

Thread-modular concrete semantics

Local states

$\mathcal{RL}(t)$ – reachable local states of thread t

- Subset of $\Sigma_t \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}_t$
- $\mathcal{M}_t \stackrel{\text{def}}{=} \mathcal{V} \cup \{pc_{t'} \mid t' \neq t\} \to \mathbb{V} \cup \mathcal{L}$
- Control state is reduced to t's location in program
- Other threads' locations are stored in auxiliary variables $pc_{t'}$

We define transformation function from monolithic to local state:

$$\pi_t : \langle L, \rho \rangle \longmapsto \langle L(t), \rho[\forall t' \neq t : pc_{t'} \mapsto L(t')] \rangle$$

Then, $\mathcal{RL} \stackrel{\text{def}}{=} \{\pi_t(\sigma) \mid \forall \sigma \in \mathcal{R}\}$.

Interferences

$\mathcal{I}(t)$ – interferences caused by thread t

- Subset of $\Sigma \times \Sigma$
- Transitions caused by t in an existing program trace
- Correspond to the R, G assertions in the rely-guarantee system

$$\mathcal{I}(t) \stackrel{\text{def}}{=} \{\langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \xrightarrow{a_1} \sigma_1 \cdots \xrightarrow{a_n} \sigma_n \in \mathcal{F} : a_{i+1} = t\}$$

Fixpoint characterisation

$$\mathcal{R}\ell(t) = \mathbf{lfp}\, R_t(\mathcal{I})$$
$$\mathcal{I}(t) = B(\mathcal{R}\ell)(t)$$

- $R_t(Y)(X)$ interleaves steps by t and interferences from Y.
- $B(Z)(t)$ collects state transitions by t from local states $Z(t)$.

Fixpoint characterisation

$$\mathcal{R}\ell(t) = \mathbf{lfp}\, R_t(\mathcal{I})$$
$$\mathcal{I}(t) = B(\mathcal{R}\ell)(t)$$

- $R_t(Y)(X)$ interleaves steps by t and interferences from Y.
- $B(Z)(t)$ collects state transitions by t from local states $Z(t)$.

Theorem.

$$\mathcal{R}\ell = \mathbf{lfp}\left(\lambda Z.\lambda t.\mathbf{lfp}\, R_t(B(Z))\right)$$

Flow-insensitive abstraction

Forgets control state of other threads.

$\alpha^{nf}_{\mathcal{R}} : \mathcal{P}(\mathcal{L} \times \mathcal{M}_t) \to \mathcal{P}(\mathcal{L} \times \mathcal{M})$ strips auxilliary variables.
$\alpha^{nf}_{\mathcal{I}} : \mathcal{P}(\Sigma \times \Sigma) \to \mathcal{P}(\mathcal{M} \times \mathcal{M})$ strips control parts.
Fixpoint characterisation of $\mathcal{R}\ell^{nf}$ is similar to that of $\mathcal{R}\ell$.

Flow-insensitive abstraction

Forgets control state of other threads.

$\alpha^{nf}_{\mathcal{R}} : \mathcal{P}(\mathcal{L} \times \mathcal{M}_t) \to \mathcal{P}(\mathcal{L} \times \mathcal{M})$ strips auxilliary variables.
$\alpha^{nf}_{\mathcal{I}} : \mathcal{P}(\Sigma \times \Sigma) \to \mathcal{P}(\mathcal{M} \times \mathcal{M})$ strips control parts.
Fixpoint characterisation of $\mathcal{R}\ell^{nf}$ is similar to that of $\mathcal{R}\ell$.

Intuition. When computing $\mathcal{R}\ell^{nf}$, we assume that if other threads' interferences can cause a transition at *some* point, then it can happen *anytime*.

Flow-insensitive abstraction – example

1. On our example program, we can still infer some interesting properties (like $X \leq Y \leq 102$).
2. For other programs, the abstraction is too coarse:

$$t_a: \quad {}^{(a1)}\, X \leftarrow X + 1\, {}^{(a2)} \quad \Big| \quad t_b: \quad {}^{(b1)}\, X \leftarrow X + 1\, {}^{(b2)}$$

Why?

Flow-insensitive abstraction – example

1. On our example program, we can still infer some interesting properties (like $X \leq Y \leq 102$).
2. For other programs, the abstraction is too coarse:

$$t_a\colon \quad {}^{(a1)}\, X \leftarrow X + 1\, {}^{(a2)} \quad \Big| \quad t_b\colon \quad {}^{(b1)}\, X \leftarrow X + 1\, {}^{(b2)}$$

Why?

Interferences from $\mathcal{I}(t_b)$ are of form $\langle\langle b1, x\rangle, \langle b2, x+1\rangle\rangle$. After the abstraction, these become $\langle x, x+1\rangle$. When analysing t_a, we expect these interferences to happen at any point, thus we cannot infer any upper bound of X.
Resulting invariant: $X \geq 1$

Non-relational interference abstraction

Further abstraction of interferences; we keep only mapping which variables can be changed to which values.

$$\alpha^{nr}_{\mathcal{I}} : \mathcal{P}(\mathcal{M} \times \mathcal{M}) \longrightarrow (\mathcal{V} \to \mathcal{P}(\mathbb{V}))$$
$$\alpha^{nr}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \lambda V.\{x \in \mathbb{V} \mid \exists \langle \rho, \rho' \rangle \in Y : \rho(V) \neq x \wedge \rho'(V) = x\}$$

Non-relational interference abstraction

Further abstraction of interferences; we keep only mapping which variables can be changed to which values.

$$\alpha^{nr}_{\mathcal{I}} : \mathcal{P}(\mathcal{M} \times \mathcal{M}) \longrightarrow (\mathcal{V} \to \mathcal{P}(\mathbb{V}))$$
$$\alpha^{nr}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \lambda V.\{x \in \mathbb{V} \mid \exists \langle \rho, \rho' \rangle \in Y : \rho(V) \neq x \wedge \rho'(V) = x\}$$

Example

$$\alpha^{nr}_{\mathcal{I}}(\mathcal{I}(t_a)) = [X \mapsto [1, 102], \quad Y \mapsto \emptyset \quad]$$
$$\alpha^{nr}_{\mathcal{I}}(\mathcal{I}(t_b)) = [X \mapsto \emptyset, \quad Y \mapsto [1, 102] \quad]$$

Sufficient to infer variable bounds, but not $X \leq Y$.

Invariant interferences

We join the domain and codomain of memory interferences.

$$\alpha^{inv}_{\mathcal{I}} : \mathcal{P}(\mathcal{M} \times \mathcal{M}) \longrightarrow \mathcal{P}(\mathcal{M})$$
$$\alpha^{inv}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \{\rho \mid \exists \rho' : \langle \rho, \rho' \rangle \in Y \vee \langle \rho', \rho \rangle \in Y\}$$

This keeps relations between modified and not modified variables.
To also remember, which variables have been modified, we combine $\alpha^{inv}_{\mathcal{I}}$ and $\alpha^{nr}_{\mathcal{I}}$ in a *reduced product*.

Monotonicity interference

Cheap abstraction of interferences; decides whether each variable is monotonic. Useful for programs with counters or clocks.

$\alpha_{\mathcal{I}}^{mon} : \mathcal{P}(\mathcal{M} \times \mathcal{M}) \longrightarrow (\mathcal{V} \to \mathbb{D})$ where $\mathbb{D} \stackrel{\text{def}}{=} \{\nearrow, \top\}$

$\alpha_{\mathcal{I}}^{mon}(Y) \stackrel{\text{def}}{=} \lambda V.$ **if** $\forall \langle \rho, \rho' \rangle \in Y : \rho(V) \leq \rho'(V)$ **then** $\nearrow$ **else** $\top$

Monotonicity interference

Cheap abstraction of interferences; decides whether each variable is monotonic. Useful for programs with counters or clocks.

$\alpha_{\mathcal{I}}^{mon} : \mathcal{P}(\mathcal{M} \times \mathcal{M}) \longrightarrow (\mathcal{V} \to \mathbb{D})$ where $\mathbb{D} \stackrel{\text{def}}{=} \{\nearrow, \top\}$

$\alpha_{\mathcal{I}}^{mon}(Y) \stackrel{\text{def}}{=} \lambda V.$ **if** $\forall \langle \rho, \rho' \rangle \in Y : \rho(V) \leq \rho'(V)$ **then** $\nearrow$ **else** $\top$

Example

We can infer that $X < Y$ still holds before t_a executes line 3 after considering all posible interferences, as $\alpha_{\mathcal{I}}^{mon}(\mathcal{I}(t_b))(Y) = \nearrow$, because all assignments to Y have the form $Y \leftarrow Y + \text{positive}$.

Mutual exclusion locks

We extend our transition systems with mutexes:

- $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M} \times \mathcal{S}$, where $s \in \mathcal{S} \stackrel{\text{def}}{=} \mathbb{M} \to (\mathcal{T} \cup \{\bot\})$ is a scheduler that remembers which thread owns each mutex
- $\mathbb{M}$ is a finite set of mutexes
- $I \stackrel{\text{def}}{=} \langle \lambda t.e_t, \lambda V.0, \lambda m.\bot \rangle$
- Instructions **lock**(m) and **unlock**(m) that model the usual mutex semantics (i.e. at most one thread may own a mutex)

We also extend local states of $\mathcal{R}\ell(t)$ with a set of mutexes t owns. Interferences are not changed, i.e. have type $(\mathcal{C} \times \mathcal{M}) \times (\mathcal{C} \times \mathcal{M})$.

Mutual exclusion locks

Interferences from a thread t are of two kinds:

1. $\mathcal{I}^u(t)(M)$ – t does not change the set of owned mutexes M
2. $\mathcal{I}^s(t)(m)$ – critical sections between **lock**(m) and **unlock**(m)

Mutual exclusion locks

Interferences from a thread t are of two kinds:

1. $\mathcal{I}^u(t)(M)$ – t does not change the set of owned mutexes M
2. $\mathcal{I}^s(t)(m)$ – critical sections between **lock**(m) and **unlock**(m)

No interference from $\mathcal{I}^u(t')(M')$ can happen while thread $t \neq t'$ is in local state $\langle \ell, \rho, M \rangle$ and $M \cap M' \neq \emptyset$.

Mutual exclusion locks

Interferences from a thread t are of two kinds:

1. $\mathcal{I}^u(t)(M)$ – t does not change the set of owned mutexes M
2. $\mathcal{I}^s(t)(m)$ – critical sections between **lock**(m) and **unlock**(m)

No interference from $\mathcal{I}^u(t')(M')$ can happen while thread $t \neq t'$ is in local state $\langle \ell, \rho, M \rangle$ and $M \cap M' \neq \emptyset$.

In well-synchronized programs, where every access to a variable V is guarded by a dedicated mutex, all interferences to V are in $\mathcal{I}^s(t')(m_V)$, thus we can use a coarser abstraction for $\mathcal{I}^u(t')(M')$ that will be used only when data races occur.

Bibliography

TODO. Naučit se citovat v LaTeXu.