Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières

JFLA
9 janvier 2014

David Declerck
Université Paris-Sud

Travail réalisé conjointement avec :

► Sylvain Conchon, Alain Mebsout (Université Paris-Sud)
► Luc Maranget (INRIA)
Synchronization Barriers

Data structure used to synchronize the execution of a group of threads at a program point.

POSIX libraries implement barriers defined as

- a data type `barrier_t`
- an initialization function `barrier_init`
- a synchronization function `barrier_wait`
Sense-Reversing Barriers

Demo.
Safe barrier

Good synchronization:

- There does not exist a thread before the barrier and another thread after the barrier

Annotations in C source of these program points with

```c
///exclusive
wait_barrier(...);
///exclusive
```
Contributions of this work

Proving safety of synchronization barriers

- written in C
- for any number of threads
- automatically by model checking

We assume sequential consistency:

- Interleaving semantics
- Preservation of operations order
A limited fragment of C (basically, just what we need for the implementation of our benchmarks)

- `int` and `void` data types
- restricted usage of structures
- `pointers` limited to passing by reference
- loops, assignments, conditionals,
- arithmetic and relational operations
Target language

- Transition systems with states described by global variables (of type int, bool and enumerations) and infinite arrays indexed by thread identifiers.

- Transitions are encoded by logical formulas and they can be parameterized by thread identifiers (existential quantification):

\[
\exists i. \ T[i] = \text{true} \land X \leq 100 \land \forall k. k \neq i \implies T[k] = \text{false} \\
\land X' = X + 1 \land T'[i] = \text{false}
\]

(here $T'$ and $X'$ denote respectively the value of array $T$ and variable $X$ after the execution of the transition)

We can only check safety properties characterized by bad states.
(very simple) Memory Model

A set of **global variables** shared between threads, and for each thread $i$

- a **program counter** $PC[i]$ of type $t$, where
  
  $type \ t = Idle \ | \ End \ | \ L1 \ | \ L2 \ | \ ...$

- a **stack** represented by a set of $k$ global variables $STACK_j[i]$
Compilation schema: Example

\[ x = x + 1 \quad || \quad \ldots \]

corresponds to the following instructions

\[
\begin{align*}
L_0 : & \text{STACK}_0[i] \leftarrow x \\
L_1 : & \text{STACK}_0[i] \leftarrow \text{STACK}_0[i] + 1 \\
L_2 : & x \leftarrow \text{STACK}_0[i]
\end{align*}
\]

which are compiled as three transitions

\[
\begin{align*}
\exists i. & \text{PC}[i] = L_0 \land \text{STACK}_0'[i] = x \land \text{PC}'[i] = L_1 \\
\exists i. & \text{PC}[i] = L_1 \land \text{STACK}_0'[i] = \text{STACK}_0[i] + 1 \land \text{PC}'[i] = L_2 \\
\exists i. & \text{PC}[i] = L_2 \land x' = \text{STACK}_0[i]
\end{align*}
\]
Compiling Thread Counters

How to encode the arbitrary number of threads $N$?

```c
#define N ...  
int cpt = N;
```

$\begin{array}{c|c|c}
\text{cpt--;} & \ldots \ 1 \ 1 \ 1 \ldots & \forall i. \ cpt'[i] = 1 \\
\downarrow & \ldots \ 1 \ 0 \ 1 \ldots & \exists i. \ cpt[i] = 1 \land cpt'[i] = 0 \\
& \ldots \ 1 \ \ 0 \ \ 0 \ \ldots & \forall i. \ cpt[i] = 0 
\end{array}$
Proving the Sense-Reversing Barrier

Demo.
# Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Nodes</th>
<th>Inv.</th>
<th>Restarts</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb_alt.c</td>
<td>598</td>
<td>180</td>
<td>53</td>
<td>11m27s</td>
</tr>
<tr>
<td>sb.c</td>
<td>414</td>
<td>156</td>
<td>34</td>
<td>5m21s</td>
</tr>
<tr>
<td>sb_nice.c</td>
<td>303</td>
<td>139</td>
<td>49</td>
<td>28m8s</td>
</tr>
<tr>
<td>sb_single.c</td>
<td>174</td>
<td>99</td>
<td>54</td>
<td>17m44s</td>
</tr>
<tr>
<td>sb_loop.c</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TO</td>
</tr>
</tbody>
</table>
1. We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean
   - SMT solver more efficient on booleans
   - Invariant generation of model checker is not good with integers

2. Elimination of spurious traces arising from crash failure model present to handle universal quantifiers of thread counters’ encoding
   - Reduce number of backtracking (restarts) in model checker
### Benchmarks: typing optimization

<table>
<thead>
<tr>
<th></th>
<th>with typing</th>
<th></th>
<th>without typing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inv.</td>
<td>Restarts</td>
<td>Time</td>
<td>Inv.</td>
</tr>
<tr>
<td>sb_alt.c</td>
<td>152</td>
<td>7</td>
<td>7.64s</td>
<td>180</td>
</tr>
<tr>
<td>sb.c</td>
<td>226</td>
<td>10</td>
<td>20.7s</td>
<td>156</td>
</tr>
<tr>
<td>sb_nice.c</td>
<td>106</td>
<td>9</td>
<td>11.6s</td>
<td>139</td>
</tr>
<tr>
<td>sb_single.c</td>
<td>115</td>
<td>5</td>
<td>3.11s</td>
<td>99</td>
</tr>
<tr>
<td>sb_loop.c</td>
<td>1577</td>
<td>33</td>
<td>14m49</td>
<td>-</td>
</tr>
</tbody>
</table>
## Benchmarks: crash failure model optimization

<table>
<thead>
<tr>
<th></th>
<th>Refinement</th>
<th>No refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inv.</td>
<td>Restarts</td>
</tr>
<tr>
<td>sb_alt</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>sb.c</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>sb_nice.c</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>sb_single.c</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>sb_single_us.c</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>sb_loop.c</td>
<td>275</td>
<td>1</td>
</tr>
</tbody>
</table>
Future work

- Experiment with other types of synchronization barriers
- Larger subset of C
- C11 standard (semantic for concurrent programs)
- Improve Cubicle’s invariants generation mechanism for numerical candidates
Merci.
Optimization 1: Integer as Booleans

We designed a (simple) typing analysis to determine when a variable of type `int` is used as a `Boolean`
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean

```c
int x, y, z;
x = 0;
y = x;
z = y;
if (z) { x = x + 1; }
```
Optimization 1: Integer as Booleans

We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean.

```c
int x, y, z;
x = 0;               x is int or bool
y = x;
z = y;
if (z) { x = x + 1; }
```
We designed a (simple) typing analysis to determine when a variable of type \texttt{int} is used as a \texttt{Boolean}

```c
int x, y, z;
x = 0;  // x is \texttt{int} or \texttt{bool}
y = x;
z = y;  // x, y and z have the same type
if (z) { x = x + 1; }
```
We designed a (simple) typing analysis to determine when a variable of type \texttt{int} is used as a \texttt{Boolean}

```c
int x, y, z;
x = 0;  // \texttt{x is int or bool}
y = x;
z = y;  // \texttt{x, y and z have the same type}
if (z) { x = x + 1; }  // \texttt{z is bool, and x is int}
```
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean.

```c
int x, y, z;
x = 0;        x is `int` or `bool`
y = x;        y is `int` or `bool`
z = y;        z is `bool`, and x is `int`
if (z) { x = x + 1; }
```

The program is rejected
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a **Boolean**

```c
int x, y;
y = 0;
if (y == 0) { x = 0; }
```
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean

```c
int x, y;
y = 0;                        // y is int or bool
if (y == 0) { x = 0; }
```

The program is well typed:
- `x`: int (for safety reasons)
- `y`: int
We designed a (simple) typing analysis to determine when a variable of type int is used as a Boolean.

```cpp
int x, y;
y = 0;
if (y == 0) {
    x = 0;
}
```

- y is int or bool
- y is int and x is int or bool
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean.

```c
int x, y;
y = 0;                      // y is int or bool
if (y == 0) { x = 0; }     // y is int and x is int or bool
```

The program is well typed:

- `x: int` (for safety reasons)
- `y: int`
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a **Boolean**

```c
int x, y, z;
x = 0 && 1;
if (x != y && y != z && x != z) ...
```
Optimization 1: Integer as Booleans

We designed a (simple) typing analysis to determine when a variable of type \texttt{int} is used as a \texttt{Boolean}

```c
int x, y, z;
if (x != y && y != z && x != z) ...
```

\texttt{x} is \texttt{bool}
We designed a (simple) typing analysis to determine when a variable of type `int` is used as a **Boolean**

```c
int x, y, z;
x = 0 && 1;  // x is bool
if (x != y && y != z && x != z) ...
    // x, y and z are bool, but only x is initialized
```
Optimization 1: Integer as Booleans

We designed a (simple) typing analysis to determine when a variable of type `int` is used as a Boolean

```c
int x, y, z;
x = 0 && 1;
```

```
x, y and z are bool, but only x is initialized
```

The program is rejected
Optimization 2: Elimination Spurious Traces

Crash Failure Model

\[
\begin{align*}
\text{type } & t = A \mid B \mid C \\
\forall i. & X[i] = A \quad \text{(initial states)} \\
\text{t}_1 : & \exists i, j. i \neq j \land X[i] = A \land X[j] = A \land X'[i] = B \\
\text{t}_2 : & \exists i. X[i] = B \land \forall j. j \neq i \implies X[j] \neq A \land X'[i] = C
\end{align*}
\]
Optimization 2: Elimination Spurious Traces

Crash Failure Model

\[
\begin{align*}
t & = A \mid B \mid C \\
\forall i. \ X[i] & = A \quad \text{(initial states)} \\
\end{align*}
\]

\[
\begin{align*}
t_1 : \exists i, j. \ i \neq j \wedge X[i] & = A \wedge X[j] = A \wedge X'[i] = B \\
t_2 : \exists i. \ X[i] & = B \wedge \forall j. \ j \neq i \implies X[j] \neq A \wedge X'[i] = C
\end{align*}
\]
Optimization 2: Elimination Spurious Traces

Crash Failure Model

\[
\begin{align*}
\text{type } t &= A \mid B \mid C \\
\forall i. \ X[i] &= A \quad \text{(initial states)} \\
\text{t}_1 &: \exists i, j. \ i \neq j \land X[i] = A \land X[j] = A \land X'[i] = B \\
\text{t}_2 &: \exists i. \ X[i] = B \land \forall j. \ j \neq i \implies X[j] \neq A \land X'[i] = C
\end{align*}
\]

the program could contain only one thread
Optimization 2: Elimination Spurious Traces

Crash Failure Model

\[
\begin{align*}
\text{type } & t = A \mid B \mid C \\
\forall i. & X[i] = A \quad \text{(initial states)} \\
t_1 : & \exists i, j. i \neq j \land X[i] = A \land X[j] = A \land X'[i] = B \\
t_2 : & \exists i. X[i] = B \land \forall j. j \neq i \implies X[j] \neq A \land X'[i] = C
\end{align*}
\]

the program contains \textbf{at least two} threads

the program could contain only \textbf{one} thread
Optimization 2: Elimination Spurious Traces

Crash Failure Model

\[
\begin{aligned}
t & = A \mid B \mid C \\
\forall i. \ X[i] = A & \quad \text{(initial states)} \\
t_1 & : \exists i, j. \ i \neq j \land X[i] = A \land X[j] = A \land X'[i] = B \\
t_2 & : \exists i. \ X[i] = B \land \forall j. \ j \neq i \implies X[j] \neq A \land X'[i] = C
\end{aligned}
\]

the program contains \textbf{at least two} threads

the program \textbf{could} contain only \textbf{one} thread

thus, \( t_2 \) is not possible from that state
How to Refine the Crash Failure Model?

\[ \exists i. \quad \exists ij. \quad \exists i. \quad \forall j. \quad j \neq i \Rightarrow G(j) \]
How to Refine the Crash Failure Model?

\( \exists ij. \mathcal{R}(i,j) \)

\( \exists i. \ldots \forall j. j \neq i \Rightarrow \mathcal{G}(j) \)
How to Refine the Crash Failure Model?

$\exists_{ij} \mathcal{P}(i,j)$

$\exists_{ij} \ldots$

$\exists_{i} \ldots \forall_{j} \ j \neq i \Rightarrow \mathcal{G}(j)$
How to Refine the Crash Failure Model?
How to Refine the Crash Failure Model?

\[ \exists i. \mathcal{A}(i,j) \]

\[ \exists i. \mathcal{H}(i) \]

\[ \exists i. \mathcal{F}(i) \]

\[ \exists i. \quad \forall j. \ j \neq i \Rightarrow \mathcal{G}(j) \]
How to Refine the Crash Failure Model?

∃i. (i)
∃ij. (i,j)
∃i. H(i)
∃i. F(i)
∃i. ... ∀j. j≠i ⇒ C(j)