Automatic Software Verification of BSPlib-programs: Replicated Synchronization

Arvid Jakobsson

2017-05-31

Supervisors: G. Hains, W. Suijlen, F. Loulergue, F. Dabrowski, W. Bousdira
Context

- **Huawei**: World-leading provider of ICT-solutions
- Huawei has an increasing need for embedded parallel software
- Successful software must be **safe and efficient**
- Formal method gives **mathematical guarantees of safety and efficiency**
- **Université d’Orléans** (Laboratoire d’Informatique Fondamental): Strong research focus on **formal methods and parallel computing**
Overview of AVSBSP

- Goal of the project: develop a basis for efficient and secure, statically verified BSPlib programming
- **Bulk Synchronous Parallel (BSP)**: simple but powerful model for parallel programming,
- **BSPlib**: a library for BSP-programming in C
Overview of AVSBSP

- **Main track:** Developing *automatic* tools for verification of BSPlib programs based on *formal methods*.
  - Correct synchronization
  - Correct communication
  - Correct API usage
  ⇒ Automatic verification of *safety*

- **Side-track:** Automatic Cost Analysis
  - Automatic BSP cost formula derivation
  ⇒ Automatic verification of *performance*
Main-track: Verification

- Main track: Developing **automatic** tools for verification of BSPlib programs based on **formal methods**.
  - **Correct synchronization**
  - Correct communication
  - Correct API usage
  - Automatic verification of **safety**
Motivating example (1)

- Long scientific calculations on cluster in parallel.
- But come Monday: calculation crashed after 10 hours :( 
- What went wrong? Let’s look at the code!
Motivating example (2)

- **Single Program, Multiple data**: same program $c$ is run in parallel on $p$ processes:

  $c[pid := 0] \parallel c[pid := 1] \parallel \ldots \parallel c[pid := p-1]$
Motivating example (2)

- Single Program, Multiple data: same program $c$ is run in parallel on $p$ processes:

  $c[pid := 0] \parallel c[pid := 1] \parallel \ldots \parallel c[pid := p - 1]$

  // ...
  double $x = 0.0$;
  for (int $i = 0; i < 100; ++i) {
    $x = f(x);$  
    // ...
  }

  Figure: Parallel SPMD program: Iterative calculation
Motivating example (2)

double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
    x = f(x);

    double t1 = bsp_time();
    if (t1 - t0 > 1.0) {
        print_progress(x);
        t0 = t1;
    }
}

Figure: Buggy parallel SPMD program: Harmless printing?
Motivating example (2)

```c
void print_progress(double x) {
    int p = bsp_nprocs();
    // Print progress for process 0, 1, 2, ...
    for (int s = 0; s < p; ++p) {
        if (bsp_pid() == s) {
            printf("progress(%d): %g\n", s, x);
        }
    }
    bsp_sync();
}
```

Figure: Buggy parallel SPMD program: Harmless printing?
Motivating example (2)

```c
double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
    x = f(x);
    double t1 = bsp_time();
    if (t1 - t0 > 1.0) {
        print_progress(x); // synchronizing
        t0 = t1;
    }
}
```

**Figure:** Buggy parallel SPMD program: Harmless printing?
Motivating example (2)

```c
double t0 = bsp_time();
double x = 0.0;
for (int i = 0; i < 100; ++i) {
    x = f(x);

    double t1 = bsp_time();
    if (t1 - t0 > 1.0) { // Processes agree on this condition?
        print_progress(x); // synchronizing.
        t0 = t1;
    }
}
```

Figure: Buggy parallel SPMD program: Processes agree?
Motivating example (3): Conclusion

- **Source of bug:** Program hangs since synchronization choice depends on a value local to each process (bsp_time()).
- **Possible solution:** To synchronize or not must only depend on conditions with the same value on all processes.
- **Goal:** Enforce solution statically.
Background: Bulk synchronous parallel (1)

- Bulk synchronous parallel (BSP): model of parallel computing
- BSP computation: sequence of super-steps executed by a fixed number of $p$ processes.
- A super-step is composed of:
  1. Local computation by each process
  2. Communication between processes
  3. A synchronization barrier. Go back to Step 1 or terminate.

Figure: A BSP superstep
Background: Bulk synchronous parallel (2)

- Invented in the 80’s by Leslie Valiant. Several implementations, notably: **BSPlib**, **BSML**, most linear algebra packages...
- Domain specific languages such as Pregel and MapReduce embody BSP principles.
- Benefits of BSP compared to other models of parallel computation:
  - Deadlock and data race free
  - Simple but realistic cost model
  - Simplifies algorithm design
Background: BSPlib

- BSPlib: library and interface specification for BSP in C.
- BSPlib follows the *Single Program Multiple Data*-model (SPMD).
- Small set of primitives (20):
  - `bsp_begin`, `bsp_end`, `bsp_pid`, `bsp_nprocs`, `bsp_get`, `bsp_put`, `bsp_sync`, ...
- Several implementations exists: The Oxford BSP Toolset, Paderborn University BSP, MulticoreBSP, Epiphany BSP...
Toy-language “BSPlite”: WHILE-language with parallel primitives (nprocs, pid and sync).

Grammar of BSPlite:

\[\begin{align*}
expr & \ni e & \::= & \text{nprocs} | \text{pid} | x | n | e + e | e - e | e \times e \\
bexpr & \ni b & \::= & \text{true} | \text{false} | e < e | e = e | b \text{ or } b | b \text{ and } b | !b \\
cmd & \ni c & \::= & x := e | \text{skip} | \text{sync} | c; c | \text{if } b \text{ then } c \text{ else } c \text{ end} \\
& & & | \text{while } b \text{ do } c \text{ end}
\end{align*}\]

- \textit{pid}, returns local process id from \(\mathbb{P} = \{0 \ldots p - 1\}\): it allows processes with different id to evaluate the same program differently.
BSPlite local semantics

- Local semantics for local computation in each process:

  \[ \rightarrow^i : cmd \times \Sigma \rightarrow T \times \Sigma \]

  \[ \Sigma = \mathbb{X} \rightarrow \mathbb{N} \]

  \[ T = \{ \text{Ok} \} \cup \{ \text{Wait}(c) \mid c \in cmd \} \]

- \( \langle c, \sigma \rangle \rightarrow^i \langle t, \sigma' \rangle \) denotes one step of local-computation with termination state \( t \) by process with id \( i \).

- Local semantics are standard (big-step, operational), except \texttt{sync} which stops local computation and returns the rest of the program as a continuation.
BSPlite global semantics

- Global semantics moves the computation forward globally from one super-step to the next when all $p$ local processes has completed:

$$\rightarrow : \text{cmd}^p \times \Sigma^p \times (\Sigma^p \cup \{\Omega\})$$

- Global computation either:
  1. terminates correctly: $\langle C, E \rangle \rightarrow E'$
  2. synchronizes incorrectly: $\langle C, E \rangle \rightarrow \Omega$

- BSP meaning of program $c$ in a Single Program Multiple Data (SPMD) context: $\langle [c]_{i \in \mathbb{P}}, E \rangle \rightarrow E'$. 
BSPlite example programs

Buggy program from the introduction

\[ c_{nok} = [I := 0]^1; \]
\[ [X := pid]^2; \]
\[ \text{while } [I < 100]^3 \text{ do} \]
\[ [\text{sync}]^4; \]
\[ \text{if } [X = 0]^5 \text{ then} \]
\[ [\text{sync}]^6 \]
\[ \text{else} \]
\[ [\text{skip}]^7 \]
\[ [\text{end}]; \]
\[ [I := I + 1]^8 \]
\[ \text{end} \]

Correct program

\[ c_{ok} = [I := 0]^1; \]
\[ \text{while } [I < 100]^2 \text{ do} \]
\[ [\text{sync}]^3; \]
\[ [I := I + 1]^4 \]
\[ \text{end} \]
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

\[\begin{array}{ccc}
\text{Proc.0} & \text{Proc.1} & c_{nok} \\
\sigma & \sigma & \\
\end{array}\]

\[\begin{array}{c}
\text{Wait}(c_4), \sigma_0^0 \\
\text{Wait}(c_4), \sigma_1^0 \\
\end{array}\]

\[
[I := 0]^1; \\
[X := pid]^2; \\
\text{while } [I < 100]^3 \text{ do} \\
\quad [\text{sync}]^4; \\
\quad \text{if } [X = 0]^5 \text{ then} \\
\quad \quad [\text{sync}]^6 \\
\quad \quad \text{else} \\
\quad \quad \quad [\text{skip}]^7 \\
\quad \quad \quad [\text{end}]; \\
\quad [I := I + 1]^8 \\
\end{array}\]

\[
\langle c_{nok}, \sigma \rangle \rightarrow^0 \langle \text{Wait}(c_4), \sigma_0^0 \rangle \quad \& \quad \langle c_{nok}, \sigma \rangle \rightarrow^1 \langle \text{Wait}(c_4), \sigma_1^0 \rangle
\]
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

Proc.0  Proc.1  $c_{nok}$

$\langle c_4, \sigma_0^0 \rangle \rightarrow^0 \langle \text{Wait}(c_6), \sigma_0^0 \rangle \quad \& \quad \langle c_4, \sigma_1^0 \rangle \rightarrow^1 \langle \text{Wait}(c_4), \sigma_1^1 \rangle$

[I := 0]$^1$
[X := pid]$^2$
while [I < 100]$^3$ do
[sync]$^4$
if [X = 0]$^5$ then
[sync]$^6$
else
[skip]$^7$
[end];
[I := I + 1]$^8$
end
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

$Proc.0 \quad Proc.1 \quad c_{nok}$

\[
[I := 0]^{1}; \\
[X := pid]^{2}; \\
\text{while } [I < 100]^{3} \text{ do} \\
\quad [\text{sync}]^{4}; \\
\quad \text{if } [X = 0]^{5} \text{ then} \\
\qquad [\text{sync}]^{6} \\
\quad \text{else} \\
\qquad [\text{skip}]^{7} \\
\qquad [\text{end}]; \\
\quad [I := I + 1]^{8} \\
\text{end}
\]

\[\langle c_{6}, \sigma_{0}^{0} \rangle \rightarrow^{0} \langle \text{Wait}(c_{4}), \sigma_{0}^{1} \rangle \quad \& \quad \langle c_{4}, \sigma_{1}^{1} \rangle \rightarrow^{1} \langle \text{Wait}(c_{4}), \sigma_{1}^{2} \rangle\]
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

$Proc.0$ $Proc.1$ $c_{nok}$

$\langle c_4, \sigma_0^1 \rangle \rightarrow^0 \langle \text{Wait}(c_6), \sigma_0^1 \rangle$ \& $\langle c_4, \sigma_1^2 \rangle \rightarrow^1 \langle \text{Wait}(c_4), \sigma_1^3 \rangle$
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

$Proc.0$  $Proc.1$  $c_{nok}$

$\langle c_6, \sigma_0^1 \rangle \rightarrow^0 \langle \text{Wait}(c_4), \sigma_0^2 \rangle$  \&  $\langle c_4, \sigma_1^3 \rangle \rightarrow^1 \langle \text{Wait}(c_4), \sigma_1^4 \rangle$
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

<table>
<thead>
<tr>
<th>Proc.0</th>
<th>Proc.1</th>
<th>$c_{nok}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[I := 0]$^1$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[X := pid]$^2$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>while [I &lt; 100]$^3$ do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[sync]$^4$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if [X = 0]$^5$ then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[sync]$^6$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[skip]$^7$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[end];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[I := I + 1]$^8$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

...
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

\begin{align*}
Proc.0 & \quad Proc.1 & \quad c_{nok} \\
\text{Wait}(c_4), \sigma_0^{44} & \quad \text{Wait}(c_4), \sigma_1^{99} & \\
[I := 0] & & \\
[X := \text{pid}] & & \\
\text{while } [I < 100] & & \\
\text{do } & & \\
\text{[sync]} & & \\
\text{if } [X = 0] & & \\
\text{then } & & \\
\text{[sync]} & & \\
\text{else } & & \\
\text{[skip]} & & \\
\text{[end]}; & & \\
[I := I + 1] & & \\
\text{end} & & \\
\end{align*}
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

Proc.0  Proc.1  $c_{nok}$

$\langle c_4, \sigma_0^{44} \rangle \rightarrow^0 \langle \text{Wait}(c_6), \sigma_0^{44} \rangle \quad \& \quad \langle c_4, \sigma_1^{99} \rangle \rightarrow^1 \langle \text{Ok}, \sigma_0^{100} \rangle$
BSPlite example programs

Execution of $c_{nok}$ with $p = 2$

Proc.0  Proc.1  $c_{nok}$

$[I := 0]_1$;
$[X := pid]_2$;
while $[I < 100]_3$ do
  $[\text{sync}]_4$;
  if $[X = 0]_5$ then
    $[\text{sync}]_6$
  else
    $[\text{skip}]_7$
  $[\text{end}]$;
$[I := I + 1]_8$
$[\text{end}]$

Wait($c_6$), $\sigma_0^{45}$

Wait $\neq$ Ok: incoherent termination states of processor 0 and 1.
Computation cannot continue: a synchronization error.
BSPlite example programs

Execution of $c_{ok}$ with $p = 2$

$\begin{align*}
\langle c_{ok}, \sigma \rangle \rightarrow^0 \langle \text{Wait}(c_4), \sigma^0 \rangle & \quad \& \quad \langle c_{ok}, \sigma \rangle \rightarrow^1 \langle \text{Wait}(c_2), \sigma^0 \rangle
\end{align*}$
BSPlite example programs

Execution of \( c_{ok} \) with \( p = 2 \)

\[
\begin{align*}
\text{Proc.0} & \quad \text{Proc.1} & \quad c_{nok} \\
\text{Wait}(c_2), \sigma^0 & \quad \text{Wait}(c_2), \sigma^0 & \text{while } [I < 100]^1 \text{ do} \\
& & \quad [\text{sync}]^2; \\
& & \quad [I := I + 1]^3 \\
& & \text{end}
\end{align*}
\]

\[
\langle c_2, \sigma^0 \rangle \rightarrow^0 \langle \text{Wait}(c_2), \sigma^1 \rangle \quad \& \quad \langle c_2, \sigma^0 \rangle \rightarrow^1 \langle \text{Wait}(c_2), \sigma^1 \rangle
\]
BSPlite example programs

Execution of \( c_{ok} \) with \( p = 2 \)

\[
\begin{align*}
\text{Proc.0} \quad & \quad \text{Proc.1} \quad & \quad c_{nok} \\
\text{Wait} (c_2), \sigma^1 \quad & \quad \text{Wait} (c_2), \sigma^1 \quad & \quad \text{while } [I < 100]^1 \text{ do} \\
& \quad \quad [\text{sync}]^2; \quad & \quad [I := I + 1]^3 \\
& \quad \quad \text{end} \\
\langle c_2, \sigma^1 \rangle \rightarrow^0 \langle \text{Wait} (c_2), \sigma^2 \rangle \quad & \quad \& \quad \langle c_2, \sigma^1 \rangle \rightarrow^1 \langle \text{Wait} (c_2), \sigma^2 \rangle
\end{align*}
\]
BSPlite example programs

Execution of $c_{ok}$ with $p = 2$

<table>
<thead>
<tr>
<th>$Proc.0$</th>
<th>$Proc.1$</th>
<th>$c_{nok}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Wait}(c_2), \sigma^2$</td>
<td>$\text{Wait}(c_2), \sigma^2$</td>
<td>while $[I &lt; 100]^1$ do $[\text{sync}]^2$; $[I := I + 1]^3$ end</td>
</tr>
</tbody>
</table>

$\langle c_2, \sigma^2 \rangle \rightarrow^0 \langle \text{Wait}(c_2), \sigma^3 \rangle \quad \& \quad \langle c_2, \sigma^2 \rangle \rightarrow^1 \langle \text{Wait}(c_2), \sigma^3 \rangle$
BSPlite example programs

Execution of $c_{ok}$ with $p = 2$

<table>
<thead>
<tr>
<th>Proc.0</th>
<th>Proc.1</th>
<th>$c_{nok}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>while $[I &lt; 100]^1$ do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{sync}]^2$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[I := I + 1]^3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Execution of $c_{ok}$ with $p = 2$

Proc.0  Proc.1  $c_{nok}$

$\text{Wait}(c_2), \sigma^{99}$  $\text{Wait}(c_2), \sigma^{99}$  \\
while $[I < 100]^1$ do  \\
    $[\text{sync}]^2$;  \\
    $[I := I + 1]^3$  \\
end

...
BSPlite example programs

Execution of $c_{ok}$ with $p = 2$

\[
\begin{align*}
\text{Proc.0} & \quad \text{Proc.1} & \quad c_{nok} \\
\text{Ok} & \quad \text{Ok} & \quad \text{while } [I < 100]^{1} \text{ do} \\
& & \quad \text{sync}^{2}; \\
& & \quad [I := I + 1]^{3} \\
& & \quad \text{end}
\end{align*}
\]

\[
\langle c_{2}, \sigma^{99} \rangle \rightarrow^{0} \langle \text{Ok}, \sigma^{100}_0 \rangle \quad \& \quad \langle c_{2}, \sigma^{99} \rangle \rightarrow^{1} \langle \text{Ok}, \sigma^{100} \rangle
\]
BSPlite example programs

Execution of $c_{ok}$ with $p = 2$

<table>
<thead>
<tr>
<th>Proc.0</th>
<th>Proc.1</th>
<th>$c_{nok}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ok</td>
<td>Ok</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>while $[I &lt; 100]^1$ do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[sync]^2$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[I := I + 1]^3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

$Ok = Ok$: coherent termination states. Global computation is finished.
Problem formulation

- A program $c$ is *synchronization error free*, if
  \[
  \forall E, \langle [c]_{i \in P}, E \rangle \rightarrow \Omega
  \]
- Goal: guarantee that BSPlib programs are synchronization error free.
- $c_{ok}$ synchronization error free, $c_{nok}$ is not.
Replicated synchronization

- **Textually aligned synchronization**: in each super-step, all local processes stop at the same instance of the same sync-primitive.
- Sufficient but not necessary condition for correct synchronization.
Replicated synchronization

- *Textually aligned synchronization*: in each super-step, all local processes stop at the same instance of the same `sync`-primitive.
- Sufficient but not necessary condition for correct synchronization.
- *Replicated synchronization*: statically verified condition for having textually aligned synchronization.
- Program has replicated synchronization if all conditionals and loops with bodies which contains `sync` are *pid-independent*. 

\[ \text{Var is pid-independent when it has no data- or control-depency on pid.} \]

\[ \text{Pid-independent variables go through the same series of values on all processes at textually aligned statements.} \]
Replicated synchronization

- *Textually aligned synchronization*: in each super-step, all local processes stop at the same instance of the same `sync`-primitive.
- Sufficient but not necessary condition for correct synchronization.
- *Replicated synchronization*: statically verified condition for having textually aligned synchronization.
- Program has replicated synchronization if all conditionals and loops with bodies which contains `sync` are *pid-independent*.
- A variable is *pid-independent* when it has no data- nor control-dependency on `pid`.
- *Pid-independent* variables goes through the same series of values on all processes at textually aligned statements.
BSPlite example programs

Buggy program from the introduction

\[ c_{nok} = [I := 0]^1; \]
\[ [X := pid]^2; \]
while \([I < 100]^3 \) do
\[ [\text{sync}]^4; \]
if \([X = 0]^5 \) then
\[ [\text{sync}]^6 \]
else
\[ [\text{skip}]^7 \]
[end];
\[ [I := I + 1]^8 \]
end

Correct program

\[ c_{ok} = [I := 0]^1; \]
while \([I < 100]^2 \) do
\[ [\text{sync}]^3; \]
\[ [I := I + 1]^4 \]
end
Replicated synchronization: Good software engineering practice

- *Replicate synchronization* codifies good parallel software engineering practices
- The condition is simple to understand
- Makes parallel code easier to understand
- Majority programs we have surveyed are implicitly written in this style
- Our analysis statically verifies that BSPlib code meets this condition, and so is synchronization error free
Stational analysis for finding $pid$-independent variables

- Reformulation of type system of Barrier Inference [Aiken & Gay ’98] as a data-flow analysis
- Stronger requirements on the analyzed program: no synchronization in branches where guard-expression is not $pid$-independent.
- Idea: find variables and program locations which does not have a data- or control-dependency on $pid$
- The abstract state in the data-flow analysis for each program location contains:
  1. set of variables statically guaranteed to be $pid$-independent at that point
  2. $pid$-independence of each guard-expression in which the point is nested.
Statically verifying "Replicated synchronization"

- After data-flow analysis, simple to verify that a program has replicated synchronization: all guard-conditions for if- and while-statements which contains sync is pid-independent:

\[
RS(c) = \bigwedge_{(l,b,c') \in \text{guards}(c)} \left[ \text{sync} \notin c' \lor (\text{FV}(b) \subseteq PI(l) \land \text{pid} \notin b) \right]
\]
Implementing and evaluating “Replicated synchronization”

- Implemented as Frama-C plugin in ~1200 lines of OCaml
- Uses the data-flow functor of Frama-C.
- Implementation also handles:
  - Interprocedurality
  - Pointers, structures and arrays (using conservative assumption)
- Limitations:
  - Unstructured control flow (gotos, switch), and structures which are normalized to gotos (early return, continue, etc) are not supported.
  - Pointers, structures and arrays are never treated as pid-independent.
Evaluation “Replicated synchronization”

- Evaluation on 20 BSPlib programs: public and Huawei-developed
- Minor modifications needed:
  - Rewriting switch-statements and early returns
  - Forcing command-line arguments pid-independent.
- Synchronization of all but three is verified
- Found same bug in two programs: synchronization depending on global variables
- One program not handled: synchronization depends on the result of a global reduction
### Evaluation result

<table>
<thead>
<tr>
<th>Program</th>
<th>Result</th>
<th>Reason</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSPedupack/bspbench.c</td>
<td>Safe</td>
<td></td>
<td>198</td>
</tr>
<tr>
<td>BSPedupack/bspfft_test.c</td>
<td>Safe</td>
<td></td>
<td>165</td>
</tr>
<tr>
<td>BSPedupack/bspinprod.c</td>
<td>Safe</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>BSPedupack/bsplu_test.c</td>
<td>Safe</td>
<td></td>
<td>147</td>
</tr>
<tr>
<td>BSPedupack/bspmv_test.c</td>
<td>Safe</td>
<td></td>
<td>625</td>
</tr>
<tr>
<td>Huawei/SDN_BSP_1.c</td>
<td>Safe</td>
<td></td>
<td>1580</td>
</tr>
<tr>
<td>AlexG/as02a/assess.c</td>
<td>Safe</td>
<td></td>
<td>573</td>
</tr>
<tr>
<td>AlexG/bp03v2/brdmmain.c</td>
<td>Unsafe</td>
<td>Uninitialized variable</td>
<td>342</td>
</tr>
<tr>
<td>AlexG/bp03v2/ppfmain.c</td>
<td>Safe</td>
<td></td>
<td>336</td>
</tr>
<tr>
<td>AlexG/mult03v6/mulmain.c</td>
<td>Safe</td>
<td></td>
<td>422</td>
</tr>
<tr>
<td>AlexG/prdx14v06/prmain.c</td>
<td>Unsafe</td>
<td>Uninitialized variable</td>
<td>320</td>
</tr>
<tr>
<td>OxfBSPlib/array_get.c</td>
<td>Safe</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>OxfBSPlib/array_put.c</td>
<td>Safe</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>OxfBSPlib/helloworld.c</td>
<td>Safe</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>OxfBSPlib/helloworld_init.c</td>
<td>Safe</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>OxfBSPlib/helloworld_seq.c</td>
<td>Safe</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>OxfBSPlib/reverse.c</td>
<td>Safe</td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>OxfBSPlib/sparse.c</td>
<td>Safe</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>OxfBSPlib/sum.c</td>
<td>Safe</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>PRGPAR1/examen99/examen99.c</td>
<td>Rejected but safe</td>
<td></td>
<td>192</td>
</tr>
</tbody>
</table>
Conclusion and future work

▶ Contributions:
  ▶ Formulating the correctness criterion “Replicated synchronization”
  ▶ Formalized and proved static analysis for detecting Replicated synchronization as a data-flow analysis for BSPlite
  ▶ Implemented as a Frama-C plugin, ∼1200 lines of OCaml-code

▶ Future work includes:
  ▶ Use as a building block for further analyses: communication, cost-analysis . . .