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)

```
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)

```
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)

```
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.

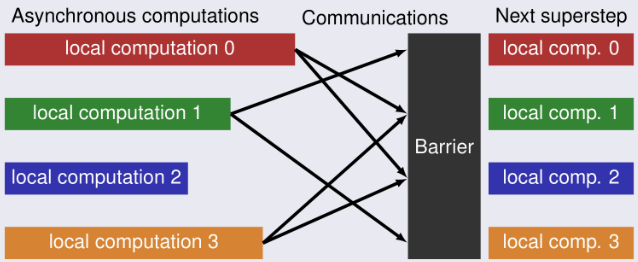

Figure: A BSP superstep

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...

BSPlite

- Toy-language "BSPlite": WHILE-language with parallel primitives (nprocs, pid and `sync`).
- Grammar of BSPlite:

$$
\begin{array}{llll}
expr & \ni & e & ::= & nprocs \mid pid \mid x \mid n \mid e + e \mid e - e \mid e \times e \\
bexpr & \ni & b & ::= & \texttt{true} \mid \texttt{false} \mid e < e \mid e = e \mid b \textbf{ or } b \mid b \textbf{ and } b \mid !b \\
cmd & \ni & c & ::= & x := e \mid \texttt{skip} \mid \texttt{sync} \mid c ; c \mid \texttt{if}\, b\, \texttt{then}\, c\, \texttt{else}\, c\, \texttt{end} \\
 & & & & \mid \texttt{while}\, b\, \texttt{do}\, c\, \texttt{end}
\end{array}
$$

- *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:

$$\begin{aligned} \rightarrow^i &: cmd \times \Sigma \rightarrow T \times \Sigma \\ \Sigma &= \mathbb{X} \rightarrow \mathbb{N} \\ T &= \{\mathbf{Ok}\} \cup \{\mathbf{Wait}(c) \mid c \in cmd\} \end{aligned}$$

- $\langle c, \sigma \rangle \rightarrow^i \langle t, \sigma' \rangle$ denotes one step of local-computation with termination state t by process with id i.
- Local semantics are standard (big-step, operational), except `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:

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

- 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

$$
\begin{aligned}
c_{nok} = {} & [\mathtt{I} := 0]^1; \\
& [\mathtt{X} := pid]^2; \\
& \texttt{while } [\mathtt{I} < 100]^3 \texttt{ do} \\
& \quad [\texttt{sync}]^4; \\
& \quad \texttt{if } [\mathtt{X} = 0]^5 \texttt{ then} \\
& \qquad [\texttt{sync}]^6 \\
& \quad \texttt{else} \\
& \qquad [\texttt{skip}]^7 \\
& \quad [\texttt{end}]; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^8 \\
& \texttt{end}
\end{aligned}
$$

Correct program

$$
\begin{aligned}
c_{ok} = {} & [\mathtt{I} := 0]^1; \\
& \texttt{while } [\mathtt{I} < 100]^2 \texttt{ do} \\
& \quad [\texttt{sync}]^3; \\
& \quad [\mathtt{I} := \mathtt{I} + 1]^4 \\
& \texttt{end}
\end{aligned}
$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
σ	σ	
$\mathbf{Wait}(c_4), \sigma_0^0$	$\mathbf{Wait}(c_4), \sigma_1^0$	

```
[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
```

$$\langle c_{nok}, \sigma \rangle \to^0 \langle \mathbf{Wait}(c_4), \sigma_0^0 \rangle \quad \& \quad \langle c_{nok}, \sigma \rangle \to^1 \langle \mathbf{Wait}(c_4), \sigma_1^0 \rangle$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
		$[\texttt{I} := 0]^1$;
		$[\texttt{X} := pid]^2$;
		while $[\texttt{I} < 100]^3$ do
	$\textbf{Wait}(c_4), \sigma_1^1$	$[\texttt{sync}]^4$;
		if $[\texttt{X} = 0]^5$ then
$\textbf{Wait}(c_6), \sigma_0^0$		$[\texttt{sync}]^6$
		else
		$[\texttt{skip}]^7$
		$[\text{end}]$;
		$[\texttt{I} := \texttt{I} + 1]^8$
		end

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

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
$\mathbf{Wait}(c_4), \sigma_0^1$	$\mathbf{Wait}(c_4), \sigma_1^2$	see program below

```
[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
```

$$\langle c_6, \sigma_0^0 \rangle \rightarrow^0 \langle \mathbf{Wait}(c_4), \sigma_0^1 \rangle \quad \& \quad \langle c_4, \sigma_1^1 \rangle \rightarrow^1 \langle \mathbf{Wait}(c_4), \sigma_1^2 \rangle$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
		$[\mathtt{I} := 0]^1;$
		$[\mathtt{X} := pid]^2;$
		$\mathtt{while}\ [\mathtt{I} < 100]^3\ \mathtt{do}$
	$\mathbf{Wait}(c_4), \sigma_1^3$	$[\mathtt{sync}]^4;$
		$\mathtt{if}\ [\mathtt{X} = 0]^5\ \mathtt{then}$
$\mathbf{Wait}(c_6), \sigma_0^1$		$[\mathtt{sync}]^6$
		$\mathtt{else}$
		$[\mathtt{skip}]^7$
		$[\mathrm{end}];$
		$[\mathtt{I} := \mathtt{I} + 1]^8$
		end

$$\langle c_4, \sigma_0^1 \rangle \rightarrow^0 \langle \mathbf{Wait}(c_6), \sigma_0^1 \rangle \quad \& \quad \langle c_4, \sigma_1^2 \rangle \rightarrow^1 \langle \mathbf{Wait}(c_4), \sigma_1^3 \rangle$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
$\mathbf{Wait}(c_4), \sigma_0^2$	$\mathbf{Wait}(c_4), \sigma_1^4$	(see program below)

```
[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
```

$$\langle c_6, \sigma_0^1 \rangle \rightarrow^0 \langle \mathbf{Wait}(c_4), \sigma_0^2 \rangle \quad \& \quad \langle c_4, \sigma_1^3 \rangle \rightarrow^1 \langle \mathbf{Wait}(c_4), \sigma_1^4 \rangle$$

BSPlite example programs

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

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

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

...

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
$\textbf{Wait}(c_4), \sigma_0^{44}$	$\textbf{Wait}(c_4), \sigma_1^{99}$	

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

...

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
		$[\mathtt{I} := 0]^1$;
		$[\mathtt{X} := pid]^2$;
		while $[\mathtt{I} < 100]^3$ do
		$[\mathtt{sync}]^4$;
		if $[\mathtt{X} = 0]^5$ then
$\mathbf{Wait}(c_6), \sigma_0^{45}$		$[\mathtt{sync}]^6$
		else
		$[\mathtt{skip}]^7$
		$[\mathrm{end}]$;
		$[\mathtt{I} := \mathtt{I} + 1]^8$
	Ok	end

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

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
		$[\mathtt{I} := 0]^1;$
		$[\mathtt{X} := pid]^2;$
		`while` $[\mathtt{I} < 100]^3$ `do`
		$[\mathtt{sync}]^4;$
		`if` $[\mathtt{X} = 0]^5$ `then`
Wait$(c_6), \sigma_0^{45}$		$[\mathtt{sync}]^6$
		`else`
		$[\mathtt{skip}]^7$
		$[\mathrm{end}];$
		$[\mathtt{I} := \mathtt{I} + 1]^8$
	Ok	`end`

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 $p = 2$

$Proc.0$	$Proc.1$	c_{nok}
σ	σ	

```
while [I < 100]^1 do
     [sync]^2;
     [I := I + 1]^3
 end
```

$$\langle c_{ok}, \sigma \rangle \rightarrow^{0} \langle \textbf{Wait}(c_4), \sigma^0 \rangle \quad \& \quad \langle c_{ok}, \sigma \rangle \rightarrow^{1} \langle \textbf{Wait}(c_2), \sigma^0 \rangle$$

BSPlite example programs

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

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

$\textbf{Wait}(c_2), \sigma^0$	$\textbf{Wait}(c_2), \sigma^0$	`while` $[\mathtt{I} < 100]^1$ `do`
		 $[\mathtt{sync}]^2$;
		 $[\mathtt{I} := \mathtt{I} + 1]^3$
		 `end`

$$\langle c_2, \sigma^0 \rangle \rightarrow^0 \langle \textbf{Wait}(c_2), \sigma^1 \rangle \quad \& \quad \langle c_2, \sigma^0 \rangle \rightarrow^1 \langle \textbf{Wait}(c_2), \sigma^1 \rangle$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
$\mathbf{Wait}(c_2), \sigma^1$	$\mathbf{Wait}(c_2), \sigma^1$	`while` $[\mathtt{I} < 100]^1$ `do` $[\mathtt{sync}]^2$; $[\mathtt{I} := \mathtt{I} + 1]^3$ `end`

$$\langle c_2, \sigma^1 \rangle \rightarrow^0 \langle \mathbf{Wait}(c_2), \sigma^2 \rangle \quad \& \quad \langle c_2, \sigma^1 \rangle \rightarrow^1 \langle \mathbf{Wait}(c_2), \sigma^2 \rangle$$

BSPlite example programs

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

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

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

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

BSPlite example programs

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

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

$\ldots$	$\ldots$	`while` $[\mathtt{I} < 100]^1$ `do`
		 $[\mathtt{sync}]^2$;
		 $[\mathtt{I} := \mathtt{I} + 1]^3$
		 `end`

$\ldots$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
$\textbf{Wait}(c_2), \sigma^{99}$	$\textbf{Wait}(c_2), \sigma^{99}$	$\texttt{while } [\texttt{I} < 100]^1 \texttt{ do}$ $\quad [\texttt{sync}]^2;$ $\quad [\texttt{I} := \texttt{I} + 1]^3$ $\texttt{end}$

$\ldots$

BSPlite example programs

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

Proc.0	*Proc*.1	c_{nok}
		`while` $[\mathtt{I} < 100]^1$ `do` $[\mathtt{sync}]^2$; $[\mathtt{I} := \mathtt{I} + 1]^3$ `end`
Ok	**Ok**	

$$\langle c_2, \sigma^{99} \rangle \rightarrow^0 \langle \mathbf{Ok}, \sigma_0^{100} \rangle \quad \& \quad \langle c_2, \sigma^{99} \rangle \rightarrow^1 \langle \mathbf{Ok}, \sigma^{100} \rangle$$

BSPlite example programs

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

$Proc.0$	$Proc.1$	c_{nok}
		while $[\mathtt{I} < 100]^1$ do
		$[\mathtt{sync}]^2$;
		$[\mathtt{I} := \mathtt{I} + 1]^3$
Ok	**Ok**	end

Ok = **Ok**: coherent termination states. Global computation is finished.

Problem formulation

- A program c is *synchronization error free*, if

$$\nexists E, \langle [c]_{i \in \mathbb{P}}, E \rangle \rightarrow \Omega$$

- 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*.

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
            [sync]^4;
            if [X = 0]^5 then
                [sync]^6
             else
                [skip]^7
             [end];
            [I := I + 1]^8
         end
```

Correct program

```
c_ok = [I := 0]^1;
       while [I < 100]^2 do
           [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

Statical 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:

$$RS(c) = \bigwedge_{(l,b,c') \in guards(c)} [\mathtt{sync}] \notin c' \vee (FV(b) \subseteq PI(l) \wedge pid \notin b)$$

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.

Evaluating "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

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

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
  - Implementated as a Frama-C plugin, ∼1200 lines of OCaml-code
- Future work includes:
  - Use as a building block for further analyses: communication, cost-analysis . . .