Int J Softw Tools Technol Transfer (2014) 16:481–491
DOI 10.1007/s10009-014-0324-3

RERS

Thoughtful brute-force attack of the RERS 2012 and 2013 Challenges

Jaco van de Pol · Theo C. Ruys · Steven te Brinke

Published online: 6 August 2014

Abstract The Rigorous Examination of Reactive Systems' (RERS) Challenges provide a forum for experimental evaluation based on specifically synthesized benchmark suites. In this paper, we report on our 'brute-force attack' of the RERS 2012 and 2013 Challenges. We connected the RERS problems to two state-of-the-art explicit state model checkers: LTSMIN and SPIN. Apart from an effective compression of the state vector, we did not analyze the source code of the problems. Our brute-force approach was successful: it won both editions of the RERS Challenge.

Keywords Software verification · Multi-core model checking · Randomized depth-first search

1 Introduction

"Reactive systems appear everywhere, e.g., as web services, decision support systems, or logical controllers. Their validation techniques are as diverse as their appearance and structure. They comprise various forms of static analysis, model checking, symbolic execution and (model-based) testing, often tailored to quite extreme frame conditions. Thus, it is almost impossible to compare these techniques, let alone to establish clear application profiles as a means for recommendation.

The Rigorous Examination of Reactive Systems' (RERS) Challenges [7] aim at overcoming this situation by providing a forum for experimental profile evaluation based on specifically designed benchmark suites. These benchmarks are automatically synthesized to exhibit chosen properties, and then enhanced to include dedicated dimensions of difficulty, ranging from conceptual complexity of the properties (e.g., reachability, full safety, liveness), over size of the reactive systems (a few hundred lines to tens of thousands of them), to exploited language features (arrays and arithmetic expressions)".[1]

The 2012 [6] and 2013 [13] Challenges are the first editions of RERS. Their problems are provided as C and Java programs. The programs basically consist of a main loop, where in each iteration an input event is read and processed by the system. Depending on the internal state, the system changes its internal variables and possibly writes an output to the standard output. If an unexpected input event is provided, an error message is printed and the system terminates.

Our main motivation to compete in the RERS Challenges was to find out whether our model checking tool-set LTSMIN can deal with large, industrial-size reactive systems provided as source code. LTSMIN [1,10,15] is a tool-set for the generation, manipulation and model checking of transition systems with state and edge labels. Its strengths are language-independence and high-performance parallel model checking, with multi-core and distributed implementations. It offers symbolic model checking engines as well as explicit state model checkers with efficient state compression techniques. LTSMIN is under development within the Formal

J. van de Pol (✉) · S. te Brinke
University of Twente, Enschede, The Netherlands
e-mail: j.c.vandepol@utwente.nl

S. te Brinke
e-mail: s.tebrinke@utwente.nl

T. C. Ruys
RUwise, Deventer, The Netherlands
e-mail: theo.ruys@gmail.com

[1] Rationale of the RERS Challenge from [16].

Methods and Tools group of the University of Twente. The tool-set is freely available from our website [15].

Although the specification of the problems of both Challenges was more-or-less the same (i.e., specified in C and Java), we used two completely different approaches for the 2012 and 2013 Challenges. For the 2012 Challenge, we reverse engineered the RERS problems to mCRL2 and PROMELA specifications. LTSMIN supports both modeling languages. We used LTSMIN to model check the mCRL2 and PROMELA models. For the 2013 Challenge, however, we kept the C versions of the RERS problems more-or-less intact and constructed a small driver in C to connect the C program directly to LTSMIN. Each iteration of the system was regarded as a single, atomic transition: we simply executed the C code of the RERS problem. For the 2013 Challenge, we also used the the model checker SPIN [4,17]. Given SPIN's advanced C-interface, it was easy to implement the same *patch-and-glue* approach for SPIN.

For both approaches, we did not really analyze the source code of the problems: in 2012, we simply translated the code to different modeling formalisms and in 2013, we just executed the code. Still, our *brute-force* approaches were successful: both editions of the RERS Challenges were won!

This paper mainly focuses on the RERS 2013 Challenge. We mention some differences with the RERS 2012 Challenge where necessary. Section 2 explains our patch-and-glue approach for the RERS 2013 White-Box Challenge in detail. It describes how we link LTSMIN and SPIN to the RERS problems, respectively. It also introduces additional techniques to handle the Gray-Box Challenges and to achieve state compression. Section 3 summarizes the algorithms and heuristics used for checking errors and LTL properties, respectively. We also mention the results that we achieved using these techniques. Finally, Sect. 4 concludes the paper and discusses the limitations and some future improvements of our approach.

2 Connecting RERS problems to model checkers

For the RERS 2013 Challenge, there were three categories of reactive systems to be considered:

- *White-Box* systems: for these systems, the complete source code was provided (Java and C).
- *Black-Box* systems: for these systems only an executable binary was provided. Furthermore, the output of several test runs was given.
- *Gray-Box* systems: for these systems, source files were also provided, but the source code contained calls to `grey`-functions, for which there was no source, only object code. To test a Gray-Box system, an executable binary was also provided.

Table 1 Characteristics of some White-Box problems

C program	Features	Loc (k)	Size	Vars	Comp
Problem28.c	Plain	2.4	67 KB	139	3
Problem43.c	Plain	7,098	161 MB	13,209	683
Problem45.c	Array	4,842	119 MB	54,702	876
Problem51.c	Array	9,436	269 MB	7,349	495
Problem54.c	Array	6,606	194 MB	21,486	1,615

We started the RERS 2013 Challenge by trying to solve the *White-Box* systems. After submitting our results of the *White-Box* systems, we were fortunate enough that the deadline for the *Gray-Box* systems was extended, due to an error in several of the source files. We were able to convert the *Gray-Box* systems into *White-Box* systems, on which we could use our proven techniques. We did not look at the *Black-Box* problems.

For each White-Box (and Gray-Box) system, three semantically equivalent programs were provided: (1) a Java program which uses characters for its input and output, (2) a Java program which uses integers for its I/O (i.e., the characters A…Z are mapped upon the integers 1…26), and (3) a similar C program which also uses integers for its I/O. Due to the nature of our verification tools, we only considered the C versions of the problems. Each system has a unique number and is identified as `ProblemN`, where `N` is an integer. The C version of the `N`-th problem is called `ProblemN.c`.

In each category (White, Gray or Black-Box), there were 27 reactive systems to be considered which ranged from plain, structurally simple and small systems to structurally complex and huge systems comprising arithmetic and arrays. All these systems had to be checked for 160 properties, which fell into two categories:

- Sixty *implicit* properties, specified as assertion errors; these properties had to be checked for reachability, and
- Hundred *explicit* behavioral properties, described verbally and formalized in LTL over the input and output values.

To give an idea of the size of the problems, Table 1 shows statistics on some problems of the White-Box category. Column *loc* lists the number of lines of code of the C program. Column *size* gives the program size in kilo or megabytes. Column *vars* lists the number of global variables of the program. The *comp* column shows the number of integer variables needed when our compression techniques are enabled (we will explain these compression techniques in Sect. 2.4). `Problem28.c` is the smallest White-Box problem. It should be noted that the bigger problems can barely be compiled with a standard C compiler such as `gcc`.

```
1  int inputs[] = {1,2,3,4,5,6};
2  int a164 = 1;
3  int a148 = 32;
4  ...
5  int a95 = 34;
6  int cf = 1;
7
8  void calculate_output132(int input) { ... }
9  void calculate_output82(int input) { ... }
10 ...
11
12 void calculate_output(int input) {
13   if ((a47==32) && (a9!=1) && (a112==1)) {
14     cf = 0;
15     error_0: assert(0);
16   }
17   if ((a52==2) && (a10!=1) && (a97==2)) {
18     cf = 0;
19     error_1: assert(0);
20   }
21   ...
22
23   if ((a13==32) || (a7==2) && (cf==1)) {
24     cf = 0;
25     a28 = 33;
26     a160 = 0;
27     ...
28   }
29   if ((a82==11) || (a23==0) && (cf==1)) {
30     cf = 0;
31     a31 = 1;
32     a91 = 1;
33     ...
34     printf("%d\n", 21);
35   }
36
37   if ((a17==1) && (a3!=1) || (a27==31)) {
38     if ((a2==1) && (a3==27) && (a45=39)) {
39       calculate_output132(input);
40     }
41     if ((a7==89) && (a11==3) && (a12!=1)) {
42       calculate_output82(input);
43     }
44   }
45   ...
46
47   if (cf==1)
48     fprintf(stderr, "Invalid␣input:␣%d\n", input);
49 }
50
51 int main()
52 {
53   while(1)
54   {
55     int input;
56     scanf("%d", &input);
57     calculate_output(input);
58   }
59 }
```

Fig. 1 Skeleton of a White-Box `ProblemN.c` file

Figure 1 shows a simplified skeleton of a `ProblemN.c` RERS' program. Within the (infinite) `while`-loop of `main`, the program reads integers from the input (line 56) and for each `input`, the function `calculate_output` (line 57) is called. Within `calculate_output`, several things might happen:

- an output is emitted (line 34), or
- an error has been detected (line 19), or
- the input is considered invalid (line 48).

Within `calculate_output`, global variables might be modified. Further, note that within `calculate_output` auxiliary `calculate_outputN` functions might be called. These functions may also modify the global variables or emit an output.

2.1 White-Box

As seen in Fig. 1, a RERS system is a potentially large but finite system, which keeps reading inputs and emitting outputs indefinitely, or aborts with an error. In our brute-force approach, we connect an explicit state model checker (LTSMIN and SPIN) to a `ProblemN.c` file to systematically feed all possible sequences of inputs to the program and observe its output.

We thus regard each call to `calculate_output` as a single, atomic transition. Apart from keeping track of the internal state of the program (i.e., the global variables), we must make sure that all observable behavior (i.e., normal output, invalid input, assertions) is dealt with.

This brute-force approach has several advantages. We simply execute the code: the function `calculate_output`. Therefore, a lot of analysis issues are not required. We do not need to worry about complicated control flow (nested `if`s, multiple functions). We do not need to worry about integer arithmetic.

The `ProblemN.c` source files need some patching to connect them to a model checker. We have developed Python scripts which patch the C programs line-by-line; we do not parse the C programs. In the following sections, we go into details of the patching of the programs for LTSMIN and SPIN, respectively.

2.2 Connecting to LTSmin

LTSMIN [1,15] consists of a collection of high-performance model checking engines, providing multi-core, distributed, and symbolic model checking algorithms. These algorithms obtain on-the-fly access to models in various specification languages through a uniform interface, called partitioned interface for the next-state function (PINS). The main functionality of PINS is to provide an *initial state* and a *next-state* function. LTSMIN comes with several language modules, each implementing PINS for a specific modeling language. Language modules are available for PROMELA, DVE, MCRL2 and UPPAAL, among others. This provides two different approaches to solve the RERS Challenge: modeling a RERS problem in an already supported modeling language, or building a dedicated language module for RERS.

RERS 2012 approach In 2012, we chose the former solution. Each RERS problem was translated into an MCRL2 and a PROMELA specification. Both translations were based on a line-by-line transformation of the well-structured C code

by means of a dedicated Python script. We applied symbolic reachability, multi-core reachability, and multi-core NDFS to hunt for error-assertions and check the LTL properties. This approach was successful, since it won us the RERS 2012 Challenge.

We observed two drawbacks of this translation method. The first is related to correctness: the transformation scripts make several assumptions on the shape of a well-structured RERS problem. However, this shape is not guaranteed and the 2013 RERS Challenge problems were considerably less well-structured and provided several new features, not supported by the translation scripts. The second drawback is an efficiency detour: first we translate RERS problems from C to mCRL2/PROMELA. Then, both mCRL2 and PROMELA compile (the main part of) their models to C to speedup model checking. However, the resulting C code is much less efficient than the original C code.

For the RERS 2013 Challenge, we decided to link LTSMIN directly to the RERS C code, in particular we directly call the `calculate_output` function. This effectively provides a dedicated RERS language module.

Patch-and-glue for LTSMIN. First, all effects of the function `calculate_output` should be visible in the vector of global state variables. To this end, we still had to patch the code. Printing a value to `stdout` was replaced by assigning that (positive) value to a new global variable `output`. Instead of aborting when an error is detected, a negative error code is returned in the global variable `output`, so

```
error_N: assert(0);
```

is replaced by

```
output = -N; return.
```

Several problems also scanned input from `stdin` within the function `calculate_output`. We handled this by saving the control location in a global variable, aborting the calculation, leaving the scanning of input to the main while loop, and resuming the calculation from the last control location.

While patching the code, we also collect the number of global variables (`state_size`), the number of valid inputs (`max_input`) and the identifiers of all global variables. This information is used to generate code to copy all global RERS variables to a state vector array (`rers2pins`) and copy them back (`pins2rers`).

Given the side-effect free version of `calculate_output` and the information above, the next-state function can be readily implemented. A schematic view is provided in Fig. 2. LTSMIN calls `next-state` on a state vector in array `src` and expects a call-back to function `cb` for each successor. We generate a successor state for each possible input, by copying the source state vector to the RERS variables (line 5), calling `calculate_output` and copying the global variables to the destination state vector (line 13). In between, we also set the edge labels for input and the output or error value (line 8–12).

```
1  int next-state(...,int src[],TransitionCB cb,...) {
2    int dst[state_size];
3    char* edge_label[2];
4    for (int input=0; i<max_input; input++) {
5      pins2rers(src);
6      output = NO_OUTPUT;
7      calculate_output(input);
8      sprintf(edge_label[0],"Input(%d)",input);
9      if (output>0)
10       sprintf(edge_label[1],"Output(%d)",output);
11     else
12       sprintf(edge_label[1],"Error(%d)",-output);
13     rers2pins(dst);
14     cb(...,edge_label,dst);
15   }
16   return max_input;
17 }
```

Fig. 2 Template of the PINS' next-state in the RERS language module

Thus, to connect a RERS Problem to LTSMIN, we *patched* the `ProblemN.c` file and *glued* the code via the RERS language module to LTSMIN. We used a 100-line Python script (`rers2ltsmin.py`) to patch the `ProblemN.c` file. The newly developed RERS module consists of 150 lines of C code.

Note that by excluding the input and output variables in the state vector, we avoid an unnecessary blow-up of the state space. Instead, we treat them as edge labels. The edge labels are used in LTSMIN to search for errors, and to report readable traces to detected errors.

When dealing with LTL properties, we generated a slightly different state space. First, since LTSMIN only supports state-based LTL, for this case we added input and output variables to the state vector. This could increase the state space. Moreover, since the LTL properties are interpreted over infinite traces only, we suppress all transitions that lead to an error. This prunes the state space.

2.3 Connecting to SPIN

Although we already had a working setup with LTSMIN, we also connected the RERS problems to SPIN [4,17]. We used the same patch-and-glue approach as for LTSMIN.

The motivation for this alternative implementation was twofold: (1) one of the authors was much more comfortable and experienced with SPIN than with LTSmin and (2) we wanted an alternative implementation to check the results we obtained with LTSMIN. It turned out that some things were easier to accomplish with SPIN than with LTSMIN, most notably the 'jumpstart technique', which we will report on in Sect. 3.2.

Similar to `rers2ltsmin.py`, we have developed a Python script `rers2spin.py` to connect a RERS' `ProblemN.c` file to a verification model in PROMELA. Figure 3 shows how the script is being used. Given a `ProblemN.c` file, the script generates three files:

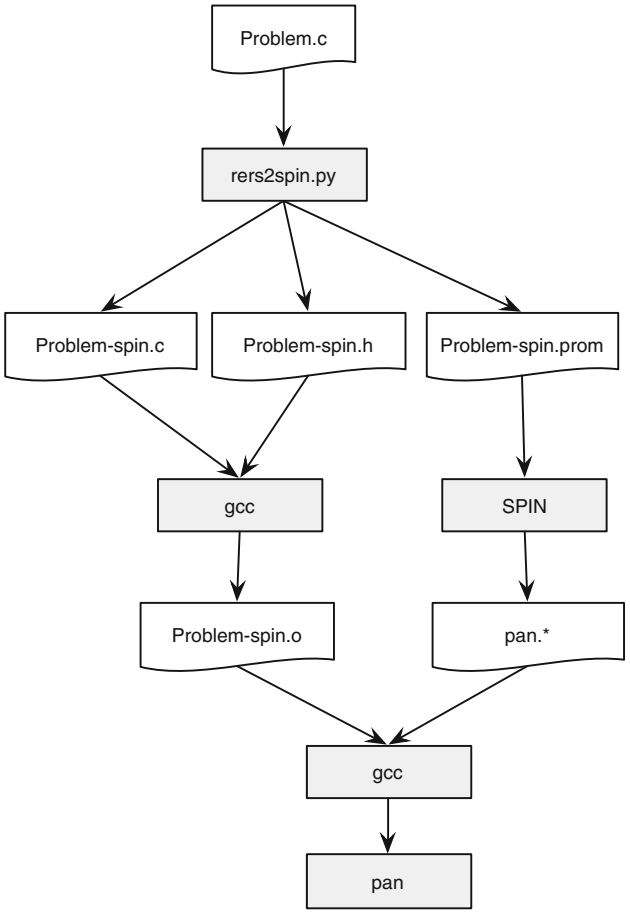

Fig. 3 Using SPIN to verify White-Box problems

```
1  int input;
2
3  c_decl {
4    int output;
5    int rersstate[N];
6  }
7  c_track "rersstate" "sizeof(int)*N"
8
9  init {
10   c_code { rers2spin(rersstate); };
11   do
12   ::  if
13       ::  input=1;
14       ::  <<input=2,3,4,5>>
15       ::  input=6;
16       fi;
17       c_code {
18           spin2rers(rersstate);
19           calculate_output(now.input);
20           rers2spin(rersstate);
21       };
22       if
23       ::  c_expr { output == INVALID } ->
24             break;
25       ::  else ->
26             if
27             ::  c_expr { output < 1 } ->
28                   assert(false)
29             ::  else /* normal output */
30             fi
31       fi;
32   od
33 }
```

Fig. 4 Promela model as generated by `rers2spin.py`

- `ProblemN-spin.c`, a modified version of the original `Problem.c` file,
- `ProblemN-spin.h`, which contains definitions of some auxiliary variables and functions, including `spin2rers` and `rers2spin`, and
- `ProblemN-spin.prom`, the PROMELA model that drives the verification.

The `rers2spin.py` script consists of 500 lines of Python code.

As seen in Fig. 1, the original `ProblemN.c` contains input and output statements to communicate with the outside world. Furthermore, the program might exit with an assertion violation in case of an error. To close the PROMELA model, the effect of the output statements and the assertions has been modeled by a global integer variable `output`:

- In case of a normal output statement, the variable `output` gets assigned the value of the (positive) number that would have been printed. So, line 34 of Fig. 1 gets replaced by `output = 21;`
- In case of invalid input, the variable `output` gets assigned the constant value `INVALID`. So, line 48 of Fig. 1 gets replaced by `output = INVALID;`
- In case of an assertion, the variable `output` gets assigned a negative value of the error label of the assertion. So, line 19 of Fig. 1 gets replaced by `output = -1;`

Promela model Our modeling approach is a straightforward adaption of the abstraction techniques of Holzmann and Joshi [5] and Ruys and Kars [12]. Figure 4 shows the PROMELA model as generated by the `rers2spin.py` script. The array `rersstate` (line 5) holds a copy of all global variables of the `ProblemN.c` program. The constant `N` is computed by the `rers2spin.py` script. Due to the `c_track` declaration of line 7, SPIN will save the contents of this array in the state vector. The C function `spin2rers` copies the contents of `rersstate` to the variables of `ProblemN.c`. The dual C function `rers2spin` fills the variable `rersstate` with the variables of `ProblemN.c`. Both functions are generated by `rers2spin.py`. The `do`-loop of the `init`-process resembles the `main` of `ProblemN.c`. The `if`-statement of line 12 non-deterministically sets the variable `input` to a value between 1 and 6. Next, the function `calculate_output` (line 19) is called, but not before SPIN's saved state is copied back to

```
a1 = 8;
a2 = a236[2];
a3 = ((((a3 * a1) % 14999) + -14863) + -73) + -35;
a4 = &a195;
```

Fig. 5 Assignments in RERS problems

the variables of `ProblemN.c`. Thereafter, the variables are copied to the array `rersstate`.

After executing the call to `calculate_output`, the variable `output` is inspected in the `if`-statement of the lines 22–31. If the `output` is invalid, the current verification trail is simply abandoned. If the `output` is less than 1, we have hit an assertion violation in `ProblemN.c` and we make sure that the violation is seen by SPIN. If there is normal output, we just continue.

2.4 Compression of state vectors

We used the C version of all problems, in which all variables are of the type integer. However, the data types in the Java source code show that many variables are actually booleans or enums. This information can be utilized for reducing the state vector, i.e., the representation of a single state. In this section, we explain how we compress state vectors, which reduces the memory consumption of model checking, but does not reduce the state space (i.e., the number of reachable states).

Our state compression is simple, we only use the number of bits needed to store every possible value of a variable instead of storing each variable as 32 bit integer. To apply the compression, we must first extract (an over-approximation of) the possible values of all variables. Extracting these values is easy because the RERS source code is generated and all assignments are on their own line and always have one of the forms shown in Fig. 5, which are explained in the next paragraphs.

When the right hand side is a constant value, as for `a1`, we assume the variable is an enum and add the constant to the possible values of the variable `a1`. Note that constants and booleans are also enums, with one or two possible values, respectively. Thus, these types are not treated differently.

When the right hand side is a variable, as for `a2`, we add all possible values of the right hand side to the variable `a2`. Array indices are always constants, so we can treat each element as its own variable without special handling of arrays.

When the right hand side is an expression, as for `a3`, we do not analyze it, but just assume that the variable `a3` can have any value, so we do not compress this variable.

Aliasing, as for `a4`, does not introduce any difficulties, because in all RERS problems, aliases are read only. Since no assignments are performed through aliases, we can treat `a4`

```
state[0] = a3;
state[1] = 0;
switch (a1) {
case 8:
    break;
case 10:
    state[1] |= 0x1 << 0;
    break;
case 11:
    state[1] |= 0x2 << 0;
    break;
default:
    assert(0);
}
if (a4 == &a195) {
} else  if (a4 == &a83) {
    state[1] |= 0x1 << 2;
} else  {
    assert(0);
}
```

Fig. 6 Storing the state vector

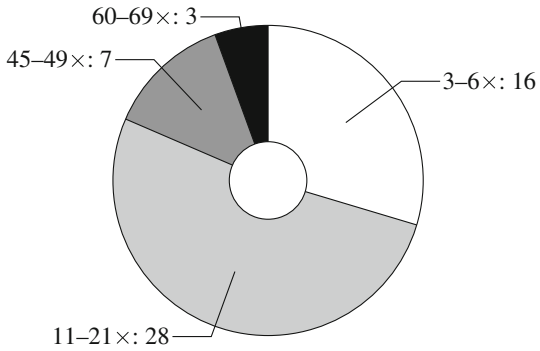

Fig. 7 Compression factor of state vectors

as an enum of which the possible values are a set of references instead of a set of concrete values.

The assignments shown in Fig. 5 could, for example, lead to the state vector shown in Fig. 6: `a1` is compressed to 2 bits, `a2` is left out, `a3` is uncompressed, and `a4` is compressed to 1 bit. We see that in the original code, at least six integers were present, whereas the state vector contains only two integers. The code for storing the state vector is generated for correctness, not speed: the assert statements are present to guarantee that our compression is correct.

When we apply the above analysis, we will have a slight over-approximation of all possible values, because we do neither analyze complex expressions nor exploit the unreachability of assignments. However, in practice we reduce the state vector significantly for most RERS problems. Figure 7 shows the various compression factors: the compressed state vectors are 3–69 times smaller than the original vectors, and for 70 % of the problems, the state size becomes more than eleven times smaller.

The RERS problems are divided into three categories based on the used language features: plain, arithmetic, and array.

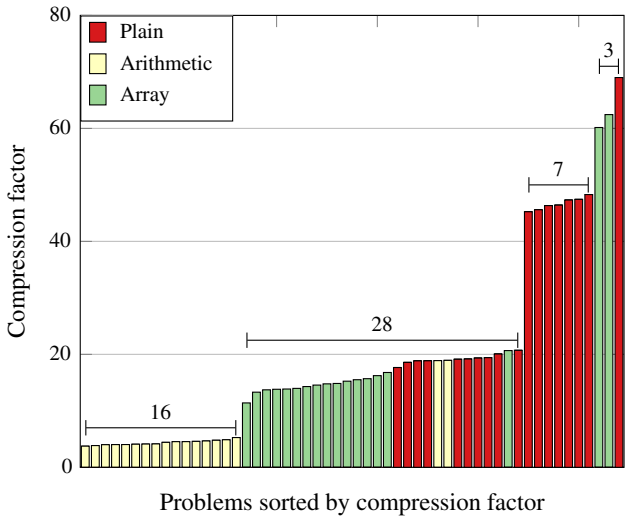

Fig. 8 Compression factor of state vectors

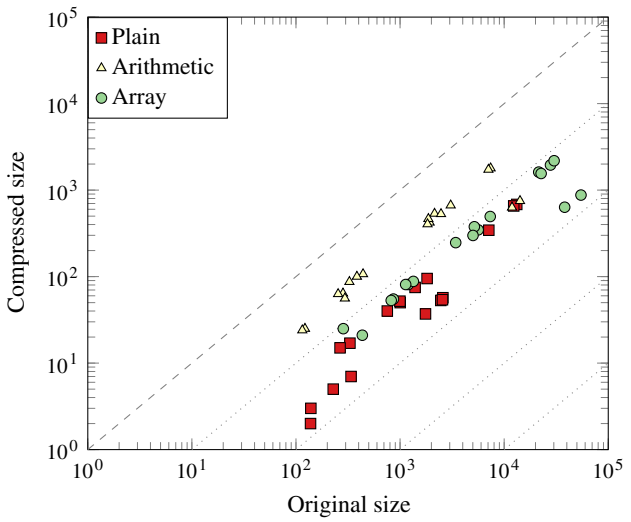

Fig. 9 Original and compressed size of state vectors

Figure 8 shows the compression factor of all problems, together with their category. Plain problems are the simplest problems and, therefore, can be compressed most. Arithmetic problems contain many expressions, for which the ranges are unknown, so these problems do not compress well. Array problems contain both expressions and arrays of which the content is constant. This results in a quite good compression.

Figure 9 shows the compressed state size compared to its original size. We see that the compressed vectors are an order of magnitude smaller, which is essential for SPIN: The number of states that can be processed increases by an order of magnitude and also the time needed for copying states reduces. The latter is also helpful in the LTSMIN-setting.

2.5 Gray-Box

A Gray-Box `ProblemN.c` source file has the same structure as a White-Box problem. The only difference is that the source code contains calls to `gray` functions, which are only available in binary form.
An example of a `gray` function call is:

```
gray3(a190,input,a32,a131);
```

A call to a `gray` function does not have any side effect: there is no hidden state, there is no I/O and the global variables are not modified. However, a gray function will dump its *intended* side-effect to the `stdout` as a list of C assignments. For example:

```
a46 = 33
a51 = 32
cf = 0
```

Recall that in our White-Box approach, we simply executed the `calculate_output` function as a single, atomic transition: we did not care what is happening within this function. If we can ensure that the *intended* side-effect of the `gray` functions is being executed in the `ProblemN.c` file, the Gray-Box problem could be transformed into a White-Box problem, essentially.

We have thus developed a Python script `gray2rers.py` which converts a Gray-Box `ProblemN.c` to a White-Box `ProblemNw.c` file on which we can use our White-Box scripts. The `gray2rers.py` script encloses each `gray` function between a `pre_gray` and a `post_gray` function. The example call above would then be translated to:

```
pre_gray();
gray3(a190,input,a32,a131);
post_gray();
```

The `pre_gray` function redirects the standard output to a temporary file. The `post_gray` function reads this temporary file, parses the emitted C assignments and executes these assignments. Finally, it redirects the standard output back to `stdout`.

The script `gray2rers.py` consists of 250 lines of Pyton code. The C code for `pre_gray` and `post_gray` encompasses 100 lines of C code. Since we have now obtained White-Box problems, we can simply reuse the LTSMIN and SPINsetups and still apply state compression.

3 Checking for errors and behavioral properties

3.1 Finding errors with brute-force reachability

Finding assertion violations boil down to plain reachability, which is supported by the symbolic, distributed as well as the multi-core tools of LTSMIN. Command-line option

Table 2 Size of state space and error traces

Problem	Levels	States	Transitions	Traces
Completed White-Box				
28 (plain)	17	155	930	5–10
49 (plain)	66	405,299	8,105,980	44–45
52 (plain)	85	1,563,872	31,277,440	77–78
Completed Gray-Box				
55 (plain)	13	244	1464	4–9
71 (arith.)	52	32,909	329,090	42–44
73 (plain)	45	4,657,893	93,157,860	9–10
Incomplete White-Box				
31 (plain)	23	3,819,763,502	22,918,581,012	5
37 (plain)	25	5,443,560,592	54,435,605,920	4
Incomplete Gray-Box				
60 (array)	20	1,295,062,475	1,912,487,616	16–20
62 (arith.)	10	1,102,921,328	1,616,818,650	26–30

`–action="error(19)"–trace` initiates a reachability procedure, looking for this specific edge label on-the-fly. On detecting the action, LTSMIN aborts and emits a trace from the initial state to this transition. To check for all 60 potential assertion errors, we would have to traverse the state space 60 times. One could also generate the full state space and check for all `"error(N)"` labels in the alphabet, but this would not provide traces. We implemented an alternative: option`–action="error.*"–trace` searches for any error, and produces a trace to the first occurrence of each error.

Table 2 gives an indication of the size of the state space generated and searched by LTSMIN. Only three White-Box and three Gray-Box problems could be investigated exhaustively. We also indicate the size of the largest state spaces that we have partially traversed. For each problem, we show the number of BFS levels, the numbers of unique states and transitions (over five billion states!). These figures were computed by multi-core BFS. We also indicate the length of the traces towards an assertion violation (i.e., the number of inputs required). Some of these were computed using the techniques of the following section.

Clearly, we cannot generate the full state space for many RERS problems (cf. Table 2). Although breadth-first search produces the shortest traces, for large Challenges all errors were beyond the feasible BFS horizon. We now focus on randomized parallel depth-first search.

Multi-core DFS Multi-core LTSMIN [10] is based on a concurrent shared hash table, with one worker per core. All workers perform their own DFS, based on a local work stack. Duplicate work (which could even lead to running on a cycle) is avoided by storing all visited states in a globally shared hash table, which has been designed to scale ideally for model checking applications [9]. A load balancer ensures that when a worker runs out of work, a portion of another worker's stack gets stolen. By selecting successors in a random order, the workers tend to swarm over different portions of the state space.

To store as many states as possible in limited main memory, we applied multi-core variants of tree compression [11] and Cleary's compact hashing [14]. These techniques maintain the ideal speedups, at least on our compute server, which consists of 48-cores and 132 GB RAM.

With this maximally brute-force strategy we got results for all but one White-Box problems, and for many Gray-Box problems. But for the largest problems, no errors could still be found. From here on, clever manual intervention was required to steer the search.

3.2 DFS with restart and jumpstart

While experimenting with the Gray-Box problems, we observed two characteristics. These observations inspired techniques to find the first error relatively quickly with LTSMIN and, given the first error, find (presumably) all other errors efficiently with SPIN.

Restart The first observation was that in randomized parallel DFS, either a couple of errors were found in the first few seconds, or no errors were found at all even after searching for hours. Apparently, the errors occur relatively high in the search tree. When all 48 workers missed these shallow errors, they would get lost deep in the state space due to their depth-first strategy. On the other hand, these errors were often beyond the horizon that we could reach using breadth-first-search: In most cases, BFS exhausted all memory even before the first error was found.

We exploited this characteristic of the RERS problems by putting a maximum to the DFS stack and setting a very short timeout for the parallel search. Typically, the DFS stack was limited to depth 30 and increased up to 80, until some error was found, corresponding to a trace of 30–80 inputs.

The timeout was typically set to 10 s. On timeout, the parallel DFS procedure was automatically aborted and relaunched. Effectively, this resets all workers to the initial state, raising the chance of finding shallow errors. We established the right values experimentally for each problem, until hitting the first error.

Applying these heuristics in LTSMIN, we found errors even in the largest Gray-Box problems within 1–2 h of experimentation. We find this remarkable, since we can have traversed only a tiny fraction of the complete state space within the given time and memory.

Jumpstart The second observation was that all traces to assertion violations would often share a common prefix. In other words, the assertion needles lay together within the haystack.

Table 3 Reported assertion errors in RERS 2013 Challenge

	White-Box ($3 \times 9 \times 60$)			Gray-Box ($3 \times 9 \times 60$)		
	Plain	Arith.	Array	Plain	Arith.	Array
Sure yes	249	222	227	239	248	242
Sure no	101	0	0	68	32	0
Guessed no	190	258	313	233	260	298
No report	0	60	0	0	0	0

To exploit this apparent characteristic of the RERS problems, we used a so-called *jumpstart* technique to find assertion violations. First, we would use LTSMIN's randomized parallel search to find a first assertion violation. When found, we would use a prefix of the error trail—typically, the error trail minus the last five inputs—as the start of a new BFS in SPIN. In most cases, the BFS would find several additional assertion errors.

We could only apply the restart and jumpstart technique to the Gray-Box problems, since their deadline was extended. As a result, for one White-Box problem, we did not find any assertion errors. For the Gray-Box problems, though, we always found assertion violations. After the correct answers were revealed by the organizers, it appeared that we had indeed found *all* assertion errors with these techniques.

Reachability results In total, the White-Box and Gray-Box Challenges consisted of 27 problems each, with 60 assertions, summing up to 3,240 yes/no answers. We summarize our reported results in Table 3, splitting the reports over the categories plain/arithmetic/array. It can be noticed that we solved more plain problems than arithmetic and array problems.

We checked the computed error trails to assertion violations on the original RERS programs, so we can report these "yes" answers with great confidence. However, only for three White-Box and three Gray-Box problems, we could compute the complete state space. Only in these cases, we can guarantee the "no" answers (together, these 6 problems had 201 no answers). For all other cases, in principle the assertion violations could happen in an uncovered part of the state space. We judged that the probability of missing such errors was very low. Hence, we reported all these cases as "no", thereby increasing our expected score. An exception was the largest arithmetic White-Box problem: Here, we found no errors at all, so we had no clue which errors are actually reachable. In that case, we didn't report any "yes/no" answers.

After the challenge, the correct answers were published by the organizers. It appeared that all sure yes/no answers were correct indeed. In the 761 guessed "no" answers in the White-Box Challenge, we actually missed 12 "yes" answers, 6 in Problem 36 and 6 in Problem 51. For Problem 53, we had not reported any answers. These problems were all categorized as either "hard" or "large" (or both). All reported results in the Gray-Box category were actually correct, so we have solved 100 % of the Gray-Box reachability Challenge.

3.3 Checking LTL properties

We have addressed the LTL properties in the LTSMIN setup only. We also spent much less time on the LTL properties than on the reachability properties. In total, each of the 27×2 White-Box and Gray-Box problems came with 100 LTL properties, yielding 5,400 subproblems. Unlike the reachability problems, we are not aware of a technique to handle multiple LTL properties simultaneously. Consequently, we could use only a fully mechanized strategy.

First, as explained at the end of Sect. 2.2, we modified the state vector by adding an input and output variable. As a result, the standard automata-based approach to state-based LTL model checking can be used [8]. Next, we translated all given LTL properties into a format that can be understood by LTSMIN. As an example, the property "input C precedes output Y and Z" was formalized by the original formula:

```
(!(oY | oZ) WU iC)
```

We translated this for LTSMIN into:

```
(!((output==25)||(output==26)) W (input
    ==3))
```

Multi-core NDFS LTSMIN uses `ltl2ba` [3] to generate a Büchi automaton that accepts all traces violating the LTL property. The product of the state space of the RERS problem and this Büchi automaton is computed on-the-fly and searched for accepting cycles, coding for counter examples to the property. We used LTSmin's CNDFS algorithm [2] to find accepting cycles. This is a multi-core algorithm, where all workers apply Nested DFS. Every worker uses its own stack, but visited states are stored in a globally shared hash table. This table also stores the status bits from the outer DFS procedure and the inner search. Conflict resolution repairs situations in which the strict DFS order is violated due to parallelism. It has been proved that all accepting cycles will be detected by at least one worker. Also, it has been demonstrated that parallel CNDFS shows linear speedups for many benchmarks.

Since there were 5,400 independent jobs, we decided to apply a cluster of smaller machines, rather than the big compute server used for reachability. A job scheduler distributed the jobs over 16 nodes, each consisting of an 8-core CPU with 32 GB RAM. So, each individual job ran CNDFS on 8 cores. In a first run, we limited the time per job to 30 min. Every experiment has four possible outcomes: "yes", property holds; "no", property does not hold and a counter example is produced; "memout" or "timeout". Later experiments showed that raising the timeout to 1 h only found very

Table 4 Reported LTL properties in RERS 2013 Challenge

	White-Box ($3 \times 9 \times 100$)			Gray-Box ($3 \times 9 \times 100$)		
	Plain	Arith.	Array	Plain	Arith.	Array
False	646	607	660	696	683	654
True	53	20	9	110	41	7
Unknown	201	273	231	94	176	239

few additional answers. For the "no" answers, we obtain a trace (actually a lasso) that can be tested on the original RERS problem. For the "yes" answers, we rely on the completeness of CNDFS to guarantee that the property holds. In all other cases, we basically had no clue of the truth value, so we did not report these cases.

LTL *results* This standard model checking procedure was relatively successful. First, in many cases, the product of the RERS problem and the Büchi automaton remained small. Apparently, the properties pruned away large parts of the state space. Second, since CNDFS is a truly on-the-fly algorithm, the computation can be aborted as soon as a counter example is found. Luckily, for most problems, the number of false properties was much higher than the true properties; we found cases where 90 out of 100 LTL properties did not hold.

In all these cases, we generated counter examples, which were typically shorter than 50 steps. However, we also noticed some long traces: for White-Box `Problem51.c`, LTSMIN came up with counter examples of 117,754 steps, and for Gray-Box `Problem74.c` the longest counter example even consisted of 181,555 steps. We successfully tested these traces on the original C code. Note that due to DFS, these are probably not the shortest possible counter examples.

Table 4 shows the statistics on the reported LTL properties for the White-Box and the Gray-Box problems, split over the subcategories plain, with arithmetic, and with arrays. Apparently, the plain problems were slightly easier than the arithmetic and array problems, and we have solved more Gray-Box instances than White-Box instances.

After the challenge, we were suprised to learn that out of the 240 reported "true" answers, actually 4 results were wrong, despite the claimed completeness of CNDFS. We looked into these problems together with the organizers, and found that the deviations were due to a different interpretation of LTL formulae over the executions of a RERS problem. Note that the interpretation of LTL formulae is rather fragile with respect to the granularity of what constitutes an atomic step, even when the neXt-operator is not used. We viewed one iteration of the main loop as an atomic step, incorporating both an input and an output. This interpretation avoids intermediate states, as illustrated by the following trace:

$$\begin{pmatrix} input = 1 \\ output = 25 \end{pmatrix} \rightarrow \begin{pmatrix} input = 2 \\ output = 25 \end{pmatrix} \rightarrow \begin{pmatrix} input = 3 \\ output = 26 \end{pmatrix}$$

The formula ($output = 25$ U $output = 26$) would evaluate to *true* on this trace following our approach. The organizing team had a different interpretation in mind, separating input and output symbols [13, Section 5]. The trace above would then be:

$$iA \rightarrow oY \rightarrow iB \rightarrow oY \rightarrow iC \rightarrow oZ$$

The corresponding formula ($oY\ U\ oZ$) evaluates to *false* on this trace, because the oY outputs are separated by the iB input. Luckily, this potentially harmful deviation in semantic interpretation only caused 4 errors out of 4,186 reported answers.

4 Conclusions

In this paper, we described our attack of the RERS 2013 Challenge. We connected the explicit state model checkers SPIN and LTSMIN to the C versions of the problems. Due to the existing C code facilities of SPIN, the former proved easier than the latter, where we had to develop a complete language module, including some boiler plate code.

Based on our thoughtful brute-force approach with SPIN and LTSMIN, we won the 1st place in the overall category, as well as in the White-Box and Gray-Box categories. We could, however, not handle problems in the Black-Box category. Besides, our team was awarded the Method Combination Award, and got 9 gold, 11 silver, and 9 bronze achievements. Note that a single wrong answer for some problem (as caused by our interpretation of the LTL formulae) leads to an immediate disqualification from an achievement for that problem.

Crucial to our approach is that we can observe the complete state, i.e., all global variables of a `ProblemN.c` file. This was true for the White-Box and Gray-Box problems. More complex C programs—e.g., pointers with aliasing, dynamic memory allocation—might be more problematic. It is clear that we cannot use our approach on the Black-Box problems.

Compression of the state vector might be straightforward, but still it proved to be an important optimization: without it we would not have been able to use SPIN on the larger models.

In retrospect, it took us too much time to get up and running with the RERS problems. It required considerable software engineering effort to patch and glue the problems to our verification tools. We also believe that the problems were too big. If state-of-the-art C compilers have problems with compiling the enormous source files, it is not to be expected that

experimental analysis tools will be able to deal with these source files.

Future work We regarded each call to `calculate_output` as a single, atomic transition during which all variables might be read or written. For a future RERS Challenge, we plan a fine-grained handling of the C code by splitting the transition relation and variable dependencies. This would open optimization techniques such as symbolic model checking and partial order reduction.

Due to the nature of our approach, we were not able to prove the absence of reachability errors, except for small problems. We envision expansion of our verification approach with further analysis, e.g., slicing, static analysis, invariants, or CEGAR.

Connecting the RERS problems to our verification tools took considerable time and effort. And although we managed to automate a large part of the verification process, still a lot of manual interventions were needed. We want to further automate the verification process, e.g., in the realm of resource scheduling of our verification with LTSMIN.

Finally, we believe it is crucial to get more research groups involved in competing within future RERS Challenges. It should be made easier to start with the problems. This means that the problems should be better documented and the interfaces to other tools should be improved.

References

1. Blom, S.C.C., van de Pol, J.C., Weber, M.: ltsmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) Proceedings of CAV 2010 LNCS 6174, pp. 354–359. Springer, New York (2010)
2. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) Proceedings of ATVA 2012, LNCS 7561, pp. 269–283. Springer, New York (2012)
3. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Paris, vol. 2102. Lecture Notes in Computer Science, pp. 53–65. Springer, New York (2001)
4. Holzman, G.J.: The Spin Model Checker—Primer and Reference Manual. Addison-Wesley, Boston (2003)
5. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: Graf, S., Mounier, L. (eds.) Proceedings of SPIN 2004 LNCS 2989, pp. 76–91. Springer, New York (2004)
6. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: analysis of event-condition-action systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, vol. 7609. Lecture Notes in Computer Science, pp. 608–614. Springer, Berlin (2012)
7. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Pasareanu, C.S.: Rigorous Examination of Reactive Systems. The RERS Challenges 2012 and 2013. Software Tools for Technology Transfer. doi:10.1007/s10009-014-0337-y (2014)
8. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the First Symposium on. Logic in Computer Science, pp. 332–344. IEEE Computer Society (1986)
9. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Bloem, R., Sharygina, N. (eds.) Proceedings of FMCAD 2010, pp. 247–255. IEEE (2010)
10. Laarman, A.W., van de Pol, J.C., Weber, M.: Multi-core LTSmin: marrying modularity and scalability. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) Proceedings of NFM 2011, LNCS 6617, pp. 506–511. Springer, New York (2011)
11. Laarman, A.W., van de Pol, J.C., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) Proceedings of SPIN 2011, LNCS 6823, pp. 38–56. Springer, New York (2011)
12. Ruys, T.C., Kars, P.: Gossiping girls are all alike. In: Donaldson, A.F., Parker, D. (eds.) Proceedings of SPIN 2012, LNCS 7385, pp. 117–136. Springer, New York (2012)
13. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: synthesizing programs of realistic structure. Softw. Tools Technol. Transf. doi:10.1007/s10009-014-0336-z (2014)
14. van der Vegt, S., Laarman, A.W.: A parallel compact hash table. In: Kotásek, Z., Bouda, J., Cerná, I., Sekanina, L., Vojnar, T., Antos, D. (eds.) Proceedings of MEMICS 2011, LNCS 7119, pp. 191–204. Springer, New York (2011)
15. LTSmin—Minimization and Instantiation of Labelled Transition Systems. http://fmt.cs.utwente.nl/tools/ltsmin/
16. RERS—Rigorous Examination of Reactive Systems. http://rers-challenge.org/
17. The Spin model checker. http://spinroot.com/