Gossiping Girls Are All Alike

Theo C. Ruys[1] and Pim Kars[2]

[1] RUwise, The Netherlands, theo.ruys@gmail.com
[2] Ordina, The Netherlands, pim.kars@ordina.nl

Abstract. This paper discusses several different ways to model the well-known *gossiping girls* problem in PROMELA. The highly symmetric nature of the problem is exploited using plain PROMELA, TOPSPIN (an extension to SPIN for symmetry reduction), and by connecting SPIN to *bliss* (a tool to compute canonical representations of graphs). The model checker SPIN is used to compare the consequences of the various modelling choices. This – tutorial style – paper is meant as a road map of the various ways of modelling symmetric systems that can be explored.

1 Introduction

In the early 1970s, the following puzzle (popularized by Paul Erdös) was circulated among mathematicians [9]:

> There are n girls, each of whom knows a unique piece of initial information. They communicate by telephone calls, and whenever two speak they share all the gossip that they know. The goal is to determine the minimum number of calls necessary for all of the girls to learn all of the initial information.

A number of researchers have independently proven that $2n-4$ calls are necessary and sufficient to achieve the goal. See [9] for an overview. A solution for $2n-4$ is straightforward. From [6]: "For $n \geq 4$, $2n-4$ conversations suffice. For four persons A, B, C and D, say, take conversations AB, and CD, followed by AC and BD. For every additional person P, schedule one conversation AP, before A, B, C and D interchange their knowledge, and another conversation AP afterwards."

Given the optimal $2n-4$ solution, it is clear that we are not really interested in using a model checker to compute this optimal solution. But given the highly symmetric nature of this problem, it is interesting to see whether the model checker SPIN can cope with this. And if not, what options do we have to improve SPIN's performance? We also want to explore different ways to model this problem. The first thing that comes to mind is to model each girl by a process and a telephone call by sending and receiving messages. But, as we will see, there are many ways to model the exchange of gossips between the girls.

Using the gossiping girls problem as a running example, we will show how a symmetric model can be analyzed with SPIN. Our goal is to inspire other verification engineers – when facing modelling challenges – to explore alternative

modelling routes, especially in the case of a problem with apparent symmetries. The paper tries to retain the tutorial style of presentation of [13,15] to make the techniques easy to be adopted by intermediate to advanced SPIN users. The effectiveness of the techniques is illustrated by some experiments.

Related Work. In [17], Frits Vaandrager uses the gossiping girls problem to introduce the UPPAAL model checker. In [8], Joost-Pieter Katoen describes the gossiping girls problem as an example of gossiping networks. The paper itself focuses on the performance evaluation of such networks. Curiously, Katoen writes in a footnote: "The solution to instances of the gossiping girls example, e.g., can easily be computed using off-the-shelf model checkers in a few seconds." This might be true for small instances of the problem, i.e., $n \leq 6$, but does not scale up for slightly larger n, as we will see in this paper.

In [11], Arend Rensink reports on experiments with the GROOVE tool set on various highly symmetrical problems, including the gossiping girls problem. We will discuss GROOVE in more detail in Sec. 6, where we compare our results with GROOVE.

Two extensions to SPIN which exploit symmetry reductions for PROMELA models should be mentioned: SYMMSPIN [1], a tool which lets the user specify symmetries through scalarsets and TOPSPIN [2,23], a tool which can automatically detect symmetries in PROMELA models. In Sec. 4, we will use TOPSPIN on a PROMELA model of the gossiping girls.

Overview. Sec. 2 presents two straightforward PROMELA specifications which model the problem. We also explain how to obtain the optimal solution to the problem with SPIN. In Sec. 3 we report on our most optimal model in *vanilla* PROMELA, i.e., without using external tools and/or embedded C code. Sec. 4 discusses a PROMELA model which we feed to TOPSPIN. In Sec. 5 we use SPIN's embedded C extensions to connect our PROMELA model to *bliss*, a tool which computes canonical representations of graphs. Sec. 6 discusses the experiments that we performed with SPIN. Sec. 7 concludes the paper.

Source code. This paper presents several PROMELA specifications which model the gossiping girls problem. All PROMELA models consider the problem for $n = 4$ girls. Of course, the models that we have used for our benchmark experiments have been parameterized in n, i.e., using the `m4` macro processor [16]. All PROMELA models and the verification results as discussed in this paper are available from http://ruwise.nl/spin-2012/.

2 Initial Attempts

In this section we will discuss two straightforward approaches to model the gossiping girls problem in plain PROMELA. We will also explain how to find the minimal solution of the problem with SPIN.

Girl processes Fig. 1 shows the PROMELA model `girl-processes`. The proctype `Girl` models the behavior of a single girl. The knowledge of a girl i is

```
byte knows[4] ;                  /* bit vector of gossips     */
chan phone = [0] of { byte } ;   /* phone connections         */
byte calls ;                     /* number of telephone calls */

proctype Girl(byte x) {
  byte y;
  do
  ::  atomic { phone ! x }
  ::  d_step { phone ? y ->
        if
        ::  knows[x] != knows[y] ->
                knows[x] = knows[x] | knows[y];
                knows[y] = knows[x];
                calls++;
        ::  else
        fi;
        y=0;
      }
  od
}

init {
  atomic {
    knows[0]=1<<0; knows[1]=1<<1;
    knows[2]=1<<2; knows[3]=1<<3;
    run Girl(0); run Girl(1); run Girl(2); run Girl(3);
  }
}
```

Fig. 1. Promela model `girl-processes`.

stored in the global bitvector `knows[`i`]`. If the j-th position (from the right) in `knows[`i`]` is set, it means that girl i knows the j-th gossip. Within `init`, the initial gossips are set using the bitshift-operator `<<`.

Each girl uses the channel `phone` to call one of the other girls. When two girls are connected, the mutual knowledge in `knows[x]` (the callee) and `knows[y]` (the caller) is exchanged and the total number of `calls` is updated; but only when new information has been exchanged. Exchanging the gossips themselves is easy: a simple bitwise-or (`|`) of the bitvectors suffices.

Due to the interleaving semantics of Spin all possible sequences of calls between the `Girl` processes will be considered.

Single process A drawback of the `girl-processes` model is that, for each proctype instance, Spin will allocate memory in the state vector. This is not really necessary as we are only interested in all possible configurations of the `knows` array.

In the model `single-process` as listed in Fig. 2 we resolve this drawback by exploiting Spin's non-deterministic choice to generate all possible scenarios. There is only a single process `allgirls` which contains a `do`-loop. Each choice in the `do`-loop represents a call between two girls that do not yet share the same knowledge about the gossips. In this model the array of bitvectors is named `k`.

```
byte k[4];   /* bit vector of gossips     */
byte calls;  /* number of telephone calls */

active proctype allgirls() {
  k[0]=1<<0; k[1]=1<<1; k[2]=1<<2; k[3]=1<<3;
  do
  :: d_step { k[0] != k[1] -> k[0]=k[0] | k[1]; k[1]=k[0]; calls++; }
  :: d_step { k[0] != k[2] -> k[0]=k[0] | k[2]; k[2]=k[0]; calls++; }
  :: d_step { k[0] != k[3] -> k[0]=k[0] | k[3]; k[3]=k[0]; calls++; }
  :: d_step { k[1] != k[2] -> k[1]=k[1] | k[2]; k[2]=k[1]; calls++; }
  :: d_step { k[1] != k[3] -> k[1]=k[1] | k[3]; k[3]=k[1]; calls++; }
  :: d_step { k[2] != k[3] -> k[2]=k[2] | k[3]; k[3]=k[2]; calls++; }
  :: else -> break
  od;
}
```

Fig. 2. Promela model `single-process`.

Another drawback of `girl-processes` is that we cannot easily express and exploit the fact that an exchange between girls i and j is not really different from an exchange between girls j and i. The single-process model easily allows to express this by always choosing girls i and j with $i < j$. This cuts away half of the outgoing transitions of a state, and consequently the number of states matched.

Finding the optimal solution If we feed one of the PROMELA models to SPIN, a safety run will generate all possible states of the models. And due to SPIN's smart static analysis, the statement `calls++;` and the variable `calls` will even be removed from the model, as the variable `calls` is never used in the model.[1]

We can also use SPIN to find an optimal sequence of calls in the sense that the number of `calls` is minimal [15]. From Sec. 1 we know that for this optimal sequence the following holds: `calls == 2*N-4`. For the Promela models, we have defined a boolean expression `ALLKNOW` which is *true* when all girls know all gossips. For example, for the `single-process` model, this expression could be defined as follows (remember that $n = 4$, and `15` is decimal for `1111` binary):

```
#define ALLKNOW (k[0] == 15 && k[1] == 15 && k[2] == 15 && k[3] == 15)
```

The logical formula '`ALLKNOW` $\Rightarrow$ `calls >= 2*N-4`' is invariant and holds for all states. To get an error trail to a state where `calls == 2*N-4` *and* `ALLKNOW` both hold we could define the following PROMELA process:

```
#define P (ALLKNOW && calls == 2*N-4)
active proctype monitor() { atomic { P -> assert(!P)} }
```

[1] One should not replace the statement `calls++;` by the semantically equivalent `calls=calls+1;` though. SPIN regards this latter statement as a usage of the variable `calls` and will thus retain the variable and statement in the system. With a blow-up of the number of states as a result.

The property `P` corresponds with the goal state. The process `monitor` is blocked on this condition `P`. At the moment `P` becomes *true*, the `atomic` sequence can be taken and the `assert` of the negation of `P` will trigger an error. If we had used `assert(!P)` instead of the `atomic` sequence, this would have doubled the number of states. See [14] for details.

The safety check using the proctype with the `assert` will potentially visit *all* states, as SPIN has to check the property for all states. We can do better though. Instead of a safety check, we can also use SPIN's liveness mode and check the following LTL property:

```
(!ALLKNOW) U (calls > 2*N-4)
```

The property corresponds with paths where condition `ALLKNOW` remains *false* until we reach a state where `calls > 2*N-4`. This property will be violated on paths where we reach a state where `ALLKNOW` is *true* and `calls > 2*N-4` is *false*, i.e., our goal state. The reason why this liveness property should be preferred over the safety check is the following: for each execution path for which the until-formula holds, SPIN will stop searching as soon as it hits a state for which `call > 2*N-4` holds.

Care should be taken when choosing the options for checking this liveness property: the until-operator in SPIN is the strong until-operator. This means that the property p `U` q is violated on an infinite path where q never becomes *true*. In our PROMELA models there are infinite paths where the property `call > 2*N-4` never becomes *true*: paths containing 'calls' where no new information is being exchanged. For this reason, we should check the above LTL formula with 'acceptance cycle checking' disabled. Due to the monotonic nature of the variable `calls`, it is safe to do so: the state space has the shape of a tree, with self-loops for the calls that do not exchange information.

Preliminary Results. Table 1 shows the verification results of all PROMELA models of this paper and will be discussed in full in Sec. 6. For now, we are only interested in the first two columns of Table 1 which show the results of verifying the `girl-processes` and `single-process` model. It is interesting to see that the number of states grows very fast as the number of girls increase. SPIN still manages to generate all states for $n \leq 6$. For greater n, the problem is infeasible for SPIN (when limited to 2GB of memory). This is remarkable as the problem at hand seems rather simple.

It should be clear that there is a lot of symmetry in these models. In fact, if we have found a solution, any permutation on the set of girls gives rise to new solution that is isomorphic to the original one. The tool to exploit this is symmetry reduction, which we will further explore in Sec. 4. Before we do that, we explore another line of reasoning to abstract from girl identity.

3 Channel of knowledge

In the initial phase of our study, we constrained ourselves to 'pure' PROMELA models and 'vanilla' SPIN. When experimenting with SPIN trying to solve the

problem more effectively, we noticed that there are many ways to incorporate some notion of symmetry reduction into the model. In this section we discuss our best approach – in vanilla PROMELA – so far. The central observation for this model is that if girls i and j have the same knowledge,

- a conversation between girl i and girl j is not going to add anything
- a conversation between girl i and any girl k with different knowledge is the same as a conversation between girl j and girl k

So we might as well model equivalence classes of girls with the same knowledge. If we do so, we only have to record the number of girls in the equivalence class. Denoting the size of an equivalence class with knowledge k by $size(k)$, an exchange between girls with knowledge k_i and k_j then results in $size(k_i)$`--`, $size(k_j)$`--` and $size(k_i|k_j) = size(k_i|k_j)$`+2`. From an error trace with SPIN, we can easily produce a trace of girls within each equivalence class.

Fig. 3 lists the PROMELA model `chan-groups`. Central to this solution is the channel `groups`. The channel `groups` is a compressed representation of the bitvector array `knows` of `girl-processes`: `groups` contains tuples (*knowledge*, *size*). The *knowledge*-part is the bitvector representation of the gossips. The *size*-part specifies how many girls have this knowledge. For example, the tuple `(5,2)` means that two girls have the knowledge `101`. The channel `groups` is used as a *sorted channel*. In this way we get a *canonical* representation of the knowledge information.

As with `single-process`, the `do`-loop is responsible for generating all possible conversations between the girls. In the two `select`-statements[2], the variables `i` and `j` are non-deterministically set to indices of elements within the `groups` channel.

The subsequent invocations of the `getgroup` macro remove these tuples from the channel and set the information of the tuples to `(ki,si)` and `(kj,sj)`, respectively. After exchanging the gossips, the most elaborate part of the process is storing the new knowledge back into `groups`. Note that due to fact that `groups` is sorted, `kc` will always be different from `ki`. When both the knowledge associated with i and j have changed, we use SPIN's *random receive* operator (`??`) to inspect whether there is already a tuple in `groups` which has the knowledge `kc`. If this is the case, two girls will be added to this tuple. Otherwise, a new tuple `(kc,2)` is added to `groups`.

Fig. 4 lists the definitions of the inline macros `getgroup` and `remove`. The macro `getgroup` is hairy as we need to get the i-th element out of the sorted FIFO-channel `groups`. The first $0 \ldots (\texttt{ix} - 1)$ elements of `groups` are copied to the end of `groups` using SPIN's regular send operation. Then, the `ix`-th element is stored into `(kx,sx)`. Finally, the remaining elements are copied to the end of `groups`. In this way we keep the channel sorted without the need of an extra

[2] The statement `select` is new in SPIN version 6. The statement `select(i: n..m);` sets `i` non-deterministically to a value in the range $\texttt{n} \ldots \texttt{m}$.

```
#include "chan-inlines.h"

chan groups = [4] of {short,byte};    /* fields: (knowledge, size)  */
byte maxgroup = 3;                    /* is equal to len(groups)-1  */
byte calls;                           /* number of calls            */

/* local, temporary variables of the process Girls: */
hidden byte i, j, ii, ki, kj, si, sj, kk, sk, kc, sc;

proctype Girls() {
  do
  ::  if
      ::  len(groups) == 1 -> break;
      ::  else -> atomic {
            select(i: 0..maxgroup-1);
            select(j: i+1..maxgroup);
            getgroup(i,   ki, si);
            getgroup(j-1, kj, sj);

            kc = ki | kj;
            if
            ::  kj == kc ->           /* only i has changed  */
                  remove(ki,si);
                  groups !! kj,sj+1
            ::  else ->             /* i and j have changed */
                  remove(ki,si);
                  remove(kj,sj);
                  if
                  ::  groups ?? [eval(kc),sc] ->  /* existing group */
                        groups ?? eval(kc),sc;
                        groups !! kc,sc+2;
                  ::  else ->                     /* new group      */
                        groups !! kc,2;
                        maxgroup++
                  fi
            fi;
            calls++;
            i=0; j=0; ii=0; ki=0; si=0; kj=0; sj=0;
            kk=0; sk=0; kc=0; sc=0;
          }
      fi
  od
}

init {
    groups !! 1<<0,1; groups !! 1<<1,1; groups !! 1<<2,1; groups !! 1<<3,1;
    run Girls();
}
```

Fig. 3. Promela model `chan-groups`.

channel.[3] The macro `remove` removes one girl from the tuple `(kx,sx)`. That is, if there are still remaining girls with knowledge `kx`, the tuple will be put back with `sx-1`. If `sx==1`, nothing will be put back into `groups`.

[3] For our experiments we replaced the two calls to `getgroup` in `chan-groups` by a call to a more efficient (but even more verbose) macro `getgroups` which retrieves both tuples in a single walk over `groups`.

```
inline getgroup(ix, kx, sx)
{
  ii=0;
  do
  ::  ii < ix  -> groups ? kk, sk;
                  groups ! kk, sk;  /* put at the end */
                  ii++;
  ::  ii == ix -> groups ? kx, sx;
                  do /* move remaining elements to end */
                  :: ii < len(groups) -> groups ? kk, sk;
                                         groups ! kk, sk;
                                         ii++
                  :: else -> break
                  od;
                  break;
  od;
  ii=0; kk=0; sk=0;
}

inline remove(kx, sx)
{
  if
  ::  sx > 1  -> groups !! kx,sx-1
  ::  else    -> maxgroup--
  fi
}
```

Fig. 4. Include file `chan-inlines.h`.

Semantically, the `hidden` variables `i` . . . `sc` are local variables of the proctype `Girls`. They are only used within the `atomic` clause to compute the new groups. By defining these variables as `hidden` variables, they will not be stored in the state vector, and thus reducing each state by 12 bytes.

The model `chan-groups` is more effective than the previous attempts in the sense that it minimizes the outgoing transitions of states. Given two tuples (k_1, s_1) and (k_2, s_2) in `groups`, only a single conversation between a girl with knowledge k_1 and a girl with knowledge k_2 will be considered. Whereas in the corresponding state of the previous models, $s_1 \times s_2$ conversations would be considered.

Although this model outperforms the models of the previous section, it still does too much work. Even though we have abstracted from the identity of girls in the exchange part, we still are left with a symmetry in the identity of the gossips exchanged, which is encoded in the knowledge bitvectors.

4 TopSPIN

In this section we will report on our ventures with TopSPIN to solve the gossiping girls problem. TopSPIN [2,23] is an automatic symmetry reduction tool for the SPIN model checker. TopSPIN is applied to a PROMELA model, and – provided that the model adheres to some restrictions – uses computational group theory to automatically determine a group of component symmetries associated with

```
chan phone = [0] of { chan };
chan k1    = [4] of { pid };  /* ki = knowledge of Girl with pid i    */
chan k2    = [4] of { pid };
chan k3    = [4] of { pid };
chan k4    = [4] of { pid };
chan tmp   = [4] of { pid };  /* temporary channel                    */
byte calls;                   /* number of telephone calls            */

proctype Girl(chan know) {
  pid  p;
  bool newinfo;
  chan friend = know;         /* outside of d_step, friend == know    */

  know ! _pid;
  do
  ::  atomic { phone ! know; }
  ::  d_step {
        phone ? friend;
        newinfo=false;
        do  /* 1. move elements from friend to tmp */
        ::  empty(friend)  -> break
        ::  nempty(friend) -> friend?p; tmp!p;
                              if
                              ::  know ?? [eval(p)] -> skip
                              ::  else -> newinfo=true
                              fi
        od;
        do  /* 2. move elements from know to tmp */
        ::  empty(know)    -> break
        ::  nempty(know)   -> know?p;
                              if
                              ::  tmp ?? [eval(p)] -> skip
                              ::  else -> tmp!p; newinfo=true;
                              fi;
        od;
        do  /* 3. move elements from tmp to friend and know */
        ::  empty(tmp)      -> break;
        ::  nempty(tmp)     -> tmp?p; friend!!p; know!!p
        od;
        if
        ::  newinfo -> calls++
        ::  else
        fi;
        p=0; friend=know; newinfo=false;
      }
  od
}

init { atomic { run Girl(k1); run Girl(k2); run Girl(k3); run Girl(k4); } }
```

Fig. 5. Promela model `symm-girls`.

the specification. The tool automatically modifies the model checking algorithm employed by SPIN to exploit these symmetries during verification. We also considered the older SYMMSPIN package [1] but preferred TOPSPIN because the tool is fully automatic and gives helpful explanations in cases when the tool cannot find symmetries in the model.

TOPSPIN uses the GAP [19] computational algebra system to effectively detect state space symmetry from the associated PROMELA model. Furthermore,

TopSPIN uses a prototype extension of the `saucy` program [21], which is used to compute symmetries of directed graphs. As mentioned, TopSPIN places some restrictions on the PROMELA model, including:

- All processes should be instantiated using `run` statements within `init`.
- The current version of TopSPIN does not support arrays of channels.
- TopSPIN is not yet compatible with the verification of liveness properties.
- For symmetry to be detected, it is important for proctypes to use their built-in `_pid` variable rather than a user-defined process identifier.
- TopSPIN does not support bitvectors in the sense that it recognizes a `byte` being used as an array from `pid` to `bool`.

Especially the last two restrictions forced us to rethink the representation of knowledge within our model. Fig. 5 lists the PROMELA model `symm-girls`. This model is a variation of the original `girl-processes` model. The behavior of each girl is represented by a `Girl` process. However, instead of using an arbitrary number to identify the girls, we now use the *process identifier (pid)* of a `Girl` process to identify the gossip that she originally knew.

Instead of using bitvectors to store the knowledge of the girls, we now encode the knowledge in sorted channels of *pid*'s. For each `Girl` with pid i, a channel `k`i is defined, which holds the pids of all gossips that this girl knows. When the processes are instantiated in the `init` process, the channel `k`i is passed to the `Girl`-process which will get pid i. This channel is stored in the `Girl`'s `know` variable. Now, in combination with the symmetric version of the process `Girl`, TopSPIN will recognize that these channels can safely be permuted. As with the `chan-groups` model, we keep these channels sorted to obtain a canonical representation of the knowledge.

As before, the girls exchange information through the rendez-vous channel `phone`. The girls offer their own knowledge channel `know` on this channel `phone`. The *receiving* girl makes sure that knowledge is exchanged and that both her own knowledge channel (`know`) and the knowledge channel of the other girl (`friend`) get updated. Updating `know` and `friend` requires three steps. Firstly, all elements of `friend` are copied to the temporary channel `tmp`. Secondly, all elements of `know` which are not yet in `tmp` are copied to `tmp`. Now `tmp` contains the combined knowledge of the original `friend` and `know` channels. Thirdly, all elements of `tmp` are copied to `friend` and `know` using SPIN's sorted send operation. Note that we only update the variable `calls` if the conversation between the two girls revealed new information. At the end of the `d_step` we again reset all local variables.

The symmetric model for TopSPIN comes with a price, though. With respect to the information in the state vector, the `symm-girls` model is quite expensive, compared to the previous models. Instead of using an efficient array of bitvectors, $n+1$ channels of length of n are being used. Furthermore, exchanging information has become an expensive operation. Instead of using a single bitwise operator, a single conversation requires the copying of three of those channels.

```
c_decl { \#include "bitmatrix.h" }
c_code {
  unsigned int automorph_partition[4];
  BitMatrix stack_adjmat;
  BitMatrix canonical_adjmat;
}
c_track "&automorph_partition"  "sizeof(unsigned␣int)*4"    "StackOnly"
c_track "stack_adjmat.pbv"      "((4*4)/8+((4*4)%8>0))"     "StackOnly"
c_track "canonical_adjmat.pbv"  "((4*4)/8+((4*4)%8>0))"

#define SINGLE_CALL(x,y)                                        \
  c_expr { designated_partners(&stack_adjmat,                   \
                               automorph_partition, x, y) } ->  \
    calls++;                                                    \
    c_code {                                                    \
      merge_rows(&stack_adjmat, x, y);                          \
      comp_canonical(&stack_adjmat, &canonical_adjmat);         \
    };

byte calls=0;
active proctype allgirls() {
  c_code {
    init_bit_matrix(&stack_adjmat, 4);
    init_bit_matrix(&canonical_adjmat, 4);
  };

  do
  ::  c_code { comp_automorph_partition(&stack_adjmat,
                                        automorph_partition); };
      if
      ::  d_step { SINGLE_CALL(0,1) }
      ::  d_step { SINGLE_CALL(0,2) }
      ::  d_step { SINGLE_CALL(0,3) }
      ::  d_step { SINGLE_CALL(1,2) }
      ::  d_step { SINGLE_CALL(1,3) }
      ::  d_step { SINGLE_CALL(2,3) }
      ::  else -> break
      fi;
  od;
}
```

Fig. 6. Promela model `bliss`.

As mentioned above, the current version of TopSPIN cannot yet deal with liveness properties. Therefore, for the verification runs to find the optimal solution to the problem, we had to resort to the less effective `assert` statement.

5 Bliss

In this section we exploit SPIN's embedded C code extensions to encode the symmetries ourselves. This section assumes a working knowledge of SPIN's C code extensions, i.e., chapter of 17 of [3]. Basic knowledge of graph isomorphism theory is also assumed. The approach taken for this model is based on the abstraction techniques of [4].

The original PROMELA models of Sec. 2 use an array `knows` of bitvectors to store the knowledge within the system. This array `knows` is essentially the

adjacency matrix of a graph representation of the knowledge: if a girl i knows the gossip of girl j, there is a directed edge from i to j. For this final PROMELA model, we exploit this graph representation. We will use the *bliss* tool to find the canonical representatives for these graphs.

bliss [7,18] is a tool for computing automorphism groups and canonical forms of graphs. It has both a command line user interface as well as C/C++ programming language APIs. Compared to other well-known tools for checking graph isomorphisms (e.g., `nauty` and `saucy`), *bliss* seems to perform quite well [7].

Two features of *bliss* which make the tool ideal for our purposes are the following: (i) *bliss* can work with directed graphs, and (ii) *bliss* can compute a canonical representation of the automorphism group of a graph. Given a graph G and its canonical representative graph $\rho(G)$ the following holds: for all graphs G' that are isomorphic with G, *bliss* will return the same representative graph, i.e., $\rho(G) = \rho(G')$.

Fig. 6 shows the PROMELA model `bliss`. The model is a variation of the `single-process` model: there is a single process `allgirls`, which enumerates all possible calls between the girls in a `do`-loop. We use an efficient bitmatrix to store the knowledge of the girls. In fact, we use two bit matrices: `stack_adjmat` and `canonical_adjmat`. The `stack_adjmat` is the adjacency matrix of the directed graph that we work with in the body of the proctype `allgirls`. When a conversation takes place between two girls, this adjacency matrix gets updated. However, this adjacency matrix is never saved in the state space: it is only used in the current state. Hence the decoration `"StackOnly"` next to the `c_track` declaration of `stack_adjmat`. Instead of `stack_adjmat`, we store the adjacency matrix of its canonical graph: `canonical_adjmat`. Note that `canonical_adjmat` does not have the `"StackOnly"` decoration within its declaration. In terms of [4], `canonical_adjmat` is the *abstract* graph.

Let us look at the macro `SINGLE_CALL`. The knowledge of the two girls x and y is merged using `merge_rows`, which performs the familiar bitwise-or operation on the rows of x and y. The function `comp_canonical` computes the canonical representation of `stack_adjmat`. First, in this function, the adjacency matrix of `stack_adjmat` is converted to *bliss*'s internal graph representation. Then, *bliss*'s function to compute the canonical representation of a graph is called. Finally, the internal graph representation is converted back to our `BitMatrix` representation.

An important part of `SINGLE_CALL` is the call to `designated_partners`. Converting the adjacency matrix to *bliss*' graph representation (and vice-versa), and especially the computation of the canonical representation of a graph are time-consuming procedures. To improve the running time of the verification with SPIN, it is important to eliminate, when possible, calls between girls that are guaranteed to lead to a canonical graph that has or will be covered by connecting two other girls. Given two potential partners x and y, the function `designated_partners` returns `true`, when:

– the knowledge of x and y is different
  (i.e., the rows of the bitvectors x and y differ), *and*

- if x and y belong to the *same* cell of the partition of the automorphism group, then x and y should both have the lowest indices of this cell, *or*
- if x and y belong to *different* cells of the partition of the automorphism group, they both should have the lowest indices within their respective cells.

A partition of a set V is a set of disjoint non-empty subsets of V whose union is V. The elements of a partition are called cells. Given a graph G, the automorphism group $Aut(G)$ of G can be represented by a partition $P(G)$ of the nodes V of G: the nodes in each cell of $P(G)$ can be *permuted*. For example, in the initial state of the model, when all girls only know their own gossip, the partition of the automorphism group consists of a single cell containing all nodes. It is enough to only consider a single conversation between two nodes of this cell. After the first transition, the partition consist of two cells: cell c_1 contains the two girls which just exchanged their gossips and cell c_2 contains the other girls. It is enough to only consider a conversation between two nodes of c_2 and a conversation between a node of c_1 and a node of c_2.

The partition is computed in `comp_automorph_partition`. Within this function, *bliss*' `find_automorphisms` function is called, which returns a set of generators for the automorphism group of the graph. Given this set of generators the cells can be computed. In Fig. 6 these cells are stored in the `"StackOnly"` array `automorph_partition`. If two nodes n_1 and n_2 belong to the same cell, `automorph_partition[`n_1`] == automorph_partition[`n_2`]`.

Implementation. Although *bliss* has a well documented API, connecting the tool to SPIN took some considerable effort. Roughly 300 lines of C code were needed for this interface.

6 Experiments

This section discusses the experiments that we conducted with SPIN on the various PROMELA models. During our study, we carried out two types of experiments with SPIN: 'exhaustive' experiments where we used SPIN's standard *safety* mode to generate all reachable states for the given PROMELA model, and 'find the optimal sequence' experiments where we used SPIN's *liveness* mode to find a $2n-4$ solution to the gossiping girls problem.

We only report on the 'exhaustive' experiments. The 'find the optimal sequence' experiments show similar results though; except for the fact that in some cases SPIN could still find the optimal sequence where the corresponding 'exhaustive' experiment would run out of memory. However, these results depend on SPIN's default DFS exploration order. Since version 5.1.x, SPIN supports several C compile time options (e.g., `-DREVERSE, -DRANDOMIZE`) which change the order of selecting outgoing transitions. Running the 'find the optimal sequence' experiments with any of these options will thus show different results. Therefore we decided that including the results of the second type of experiments was not very meaningful. Instead, we report on our experiments with the SWARM tool [5,22] to 'find the optimal sequence'.

#girls		girl-processes	single-process	chan-groups	symm-girls SPIN	symm-girls TOPSPIN	bliss
$n = 4$	sv (bytes)	64	20	44	104	104	20
	states, stored	190	192	140	251	42	45
	states, matched	2,080	670	83	2,391	394	33
	time (sec)	0	0	0	0	0	0
	memory (MB)	131	131	131	131	131	131
$n = 5$	sv (bytes)	72	20	44	120	120	20
	states, stored	9,153	9,155	2,468	10,234	540	321
	states, matched	173,889	67,139	3,659	186,098	9,897	753
	time (sec)	0.08	0.01	0.01	0.36	0.06	0.03
	rate (s/sec)	114k	no data	no data	28k	9.0k	11k
	memory (MB)	131	131	131	132	131	131
$n = 6$	sv (bytes)	80	20	52	136	136	20
	states, stored	1,092,474	1,092,476	91,348	1,150,478	18,316	6,505
	states, matched	31,681,718	13,311,818	252,039	32,818,752	525,940	31,400
	time (sec)	13.4	3.0	0.8	70.5	4.8	1.2
	rate (s/sec)	81k	369k	114k	16k	3.8k	5.2k
	memory (MB)	197	172	137	272	133	131
$n = 7$	sv (bytes)	88	20	52	216	216	20
	states, stored	$> 28\,e6$	$> 37\,e6$	6,588,212	$> 13\,e6$	1,334,922	279,985
	states, matched	$> 11.4\,e8$	$> 6.6\,e8$	29,214,527	$> 5.3\,e8$	54,525,101	2,201,294
	time (sec)	541	195	96	1460	815	87
	rate (s/sec)	52k	191k	69k	9.0k	1.6k	3.2k
	memory (MB)	o.o.m.	o.o.m.	533	o.o.m.	325	142
$n = 8$	sv (bytes)			60		240	20
	states, stored			$> 27\,e6$		$> 11\,e6$	23,461,597
	states, matched			$> 17\,e7$		$> 62\,e7$	267,185,640
	time (sec)			595		14,300	11,100
	rate (s/sec)			47k		0.8k	2.1k
	memory (MB)			o.o.m.		o.o.m.	1026
exhaustive, solution found		$n \leq 7$	$n \leq 7$	$n \leq 7$	$n \leq 7$	$n \leq 7$	$n \leq 8$
swarm, solution found		$n \leq 8$	$n \leq 8$	$n \leq 10$	n.a.	$n \leq 6$	$n \leq 10$
modelling time		$<$ 1 hour	$<$ 1 hour	$\approx$ 4 days	$\approx$ 2 days		$\approx$ 8 days

Table 1. Summary of exhaustive benchmark experiments.

Settings. All experiments were run on an Apple MacBook 2.66GHz Intel Core 2 Duo, with 8GB of RAM, running Mac OS X 10.6.8 (in single-user, console mode) and gcc 4.2.1. We used SPIN version 6.1.0 for all experiments, except for the TOPSPIN experiments for which we used SPIN version 5.2.5. We used TOPSPIN version 2.2.5, *bliss* version 0.50 and GROOVE version 4.4.5. For the TOPSPIN experiments we used TOPSPIN's default *fast* symmetry reduction strategy (which is indeed fast, but does not yield canonical representations). For each verification run we reserved 2GB of memory. To compile SPIN's `pan` verifiers for safety runs, we used the following options for gcc:

```
gcc -O2 -o pan -DSAFETY -DNOCLAIM -DMEMLIM=2048 pan.c
```

Furthermore, for TOPSPIN and *bliss*, we linked `pan` to the respective libraries, of course. We executed *all* `pan` verifiers using the following command-line:

```
./pan -m50000 -c1 -w24
```

Due to `-w24`, `pan` will always reserve 128MB of memory for the hash table.

Effort. The last row of Table 1 shows an estimate of the modelling time for the various PROMELA models. Parameterizing the models in n (using the `m4` macro processor) and setting up all experiments took us more than a week. Finally, *all* experiments can be executed in roughly 28 hours. All PROMELA models and verification results are available from http://ruwise.nl/spin-2012/.

Exhaustive experiments Table 1 lists the results of letting SPIN compute all reachable states for the various models. The columns correspond with the PROMELA models that have been discussed in this paper. The `symm-girls` model has been verified with both vanilla SPIN and with TOPSPIN.

The rows 'time (sec)' list the *elapsed time* as reported by SPIN. The rows 'rate (s/sec)' list the number of states per second, again as reported by SPIN. When SPIN does not have enough memory to finish the verification, this is indicated with 'o.o.m.' (out-of-memory). We still report the output by SPIN though. For TOPSPIN and *bliss* the 'rate (s/sec)' column needs clarification. Symmetry reduction involves computing a canonical representative for *every* state that is encountered. Every time TOPSPIN or *bliss* processes a state, the tool is really processing a whole equivalence class of states.

From Table 1 we learn that our two initial PROMELA models are fast. Especially, the rate of the `single-process` model is high, compared to the rates of the other models. This is obvious as a transition in `single-process` only consists of a bitwise-or operation and two assignments. Moreover, the state vector of `single-process` consists of only 20 bytes.

Our best vanilla PROMELA model `chan-groups` behaves quite well. Compared to the two straightforward models, it succeeds in lowering the number of states without becoming too slow. It succeeds for $n = 7$.

Due to n processes and the $n + 1$ channels of length n, the state vector of the `symm-girls` model is large. We see that `symm-girls` exhibits poor results when veryfying the model with vanilla SPIN. Due to the expensive computation involving the knowledge channels, the rate is also quite low. On the other hand, in combination with TOPSPIN, the `symm-girls` model fares quite well for $n \leq 6$. The rate is even lower, but this is no surprise due to the computation of canonical representatives. For $n \geq 7$ this computational overhead becomes significant.[4]

The 'winner' of our experiments is the `bliss` model. It is the only model which succeeds in generating all states for $n = 8$, albeit slowly. SPIN needs only slightly more than 1GB of memory to verify this model for $n = 8$. As with the TOPSPIN model, we see that with increasing n, the verification rate for this model deteriorates. This is not surprising given all the operations that are executed within a single transition. Of these operations, with increasing n, the automorphism detection becomes the most dominant factor.

[4] We have discussed these verification results with Alastair Donaldson, the developer of TOPSPIN. He suspects that the current version of TOPSPIN does not handle *sorted send* channels in the most efficient way. A future version of TOPSPIN should fix this.

	gossip-priorities				*gossip-nested*			
#girls	states	trans.	time	memory	states	trans.	time	memory
$n = 4$	55	253	0.3s	6kB	13	66	0.2s	0.2MB
$n = 5$	382	1,852	1.1s	0.5MB	52	418	1.0s	0.3MB
$n = 6$	4,458	22,492	4.0s	1.5MB	361	4,749	3.2s	0.5MB
$n = 7$	80,459	409,372	44.7s	17.1MB	3,703	71,804	18.9s	2.6MB
$n = 8$	4,157,679	22,684,164	3099s	899MB	62,982	1,790,230	519s	54.9MB

Table 2. Summary of experiments with GROOVE.

Swarm SWARM [5,22] is a tool, which – given a PROMELA model and some verification settings – generates a script that performs many small verification jobs with SPIN in parallel, that can increase the problem coverage for very large verification problems. SWARM's verification jobs use SPIN's bitstate hashing mode.

We have used SWARM on the PROMELA models to 'find the optimal sequence' using the LTL property as discussed in Sec. 2. For each PROMELA model (and for each $n \in \{4, \ldots, 10\}$) we first performed a test run with SPIN to get an estimate for the speed and the hash factor. These settings were then used to let SWARM run 60 minutes on each n-th version of the PROMELA models. We let SWARM terminate as soon as one of the parallel verifications would find an error trail (i.e., option `-e`). For $n \leq 6$ swarm would come up with an error trail within a few seconds. For $n \geq 7$ it took longer.

The one-but-last row of Table 1 summarizes our experiments with SWARM. For both `chan-groups` and `bliss`, SWARM succeeds in finding an optimal sequence for $n = 10$. Curiously, on `symm-girls`, the combination SWARM and TOPSPIN only finds an error for $n \leq 6$. Perhaps the large state vector in combination with the slow speed are responsible for this. Obviously, if SWARM would have been given more time, it would probably have found more errors trails. This experiment just illustrates the power of SWARM in dealing with PROMELA models where exhaustive verification is not an option.

Groove As mentioned in Sec. 1, the GROOVE tool set [10,20] has also been used for experiments with the gossiping girls problem [11]. GROOVE [20] is a project centered around the use of simple graphs for modelling several structures object-oriented systems, and graph transformations as a basis for model transformation and operational semantics. The GROOVE tool set includes an editor, a simulator, a generator for automatically exploring state spaces and a model checker.

One of the key features of the GROOVE tool set is its underlying, very efficient algorithm for isomorphy checking of graphs. With a symmetric model, such as the one for the gossiping girls, you get your symmetry reductions for free.

On the same machine that we ran our SPIN experiments, we have also ran the GROOVE generator to compute the state space. We allocated 2GB of memory for the JVM and used GROOVE's DFS mode. Table 2 shows the results of the GROOVE experiments. The results for *gossip-priorities* correspond with GROOVE models which use rule priorities [11], whereas the *gossip-nested* results correspond with improved GROOVE models which use a rule system with nested rules [12]. This

rule system with nested rules has recently been added to GROOVE. We see that GROOVE does not have any problems with $n = 7$ and clearly outperforms our PROMELA models for the exhaustive experiments.

7 Conclusions

We have explored several ways to model the gossiping girls problem along different points of view, trying to exploit the inherit symmetries of the problem. The main point of the paper is that alternative modelling routes with SPIN can lead to substantial better verification results.

We have seen that straightforward PROMELA models are easy to construct and show reasonable results, but are not very efficient in the terms of the number of states. However, we have also seen that even in vanilla PROMELA we can implement symmetry reductions to lower the number of states. Remember though that it took several modelling efforts to come up with the final `chan-groups` model that we discuss in this paper.

TOPSPIN is a fully automatic, user-friendly tool, but it requires some basic knowledge of group theory. And although TOPSPIN outperforms vanilla SPIN on the gossiping girls problem, for larger n one has to have patience. TOPSPIN requires a truly symmetric model. This might mean that you either have to rework your model or that you have to identify the symmetries yourself.

Although our PROMELA model which uses *bliss* shows the best results, it took quite some programming effort to get there. For the occassional symmetric model this route is therefore not recommended. However, whenever a very memory efficient model is required, this approach could be an option.

Due to its efficient algorithm for checking isomorphy, GROOVE performs best on the gossiping girls problem. The GROOVE tool set also includes CTL and LTL model checkers. So, if you have to verify a highly symmetric model – and you are not afraid to learn a new modelling formalism – GROOVE should be high on your list of potential candidates.

Finally, we witnessed the power of SWARM. The tool succeeds in finding the optimal sequence in PROMELA models where exhaustive verification surely would have failed. The results are not uniform for all PROMELA models though. More research with SWARM is needed.

We also intend to apply our modelling approach to other, larger symmetric problems to see in howfar our results can be generalised. Furthermore, we intend to extend our comparison with other model checkers.

Acknowledgements We would like to thank Arend Rensink for providing us with the GROOVE models of the gossiping girls problem and with his assistance in running GROOVE. Gerard Holzmann is thanked for his help with SWARM. We would like to thank Joost-Pieter Katoen and especially Alastair F. Donaldson for their valuable comments and suggestions to improve the paper.

References

1. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *STTT*, 4(1):92–106, 2002.
2. A. F. Donaldson and A. Miller. A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In M. Johnson and V. Vene, editors, *Proc. of AMAST 2006*, LNCS 4019, pages 374–380. Springer, 2006.
3. G. J. Holzman. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, USA, 2003.
4. G. J. Holzmann and R. Joshi. Model-Driven Software Verification. In S. Graf and L. Mounier, editors, *Proc. of SPIN 2004*, LNCS 2989, pages 76–91. Springer, 2004.
5. G. J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Proc. of SPIN 2008*, LNCS 5156, pages 134–143. Springer, 2008.
6. C. Hurkens. Spreading gossip efficiently. *Nieuw Archief voor Wiskunde*, 5/1(2):208–210, June 2000.
7. T. Junttila and P. Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proc. of ALENEX 2007*, pages 135–149. SIAM, 2007.
8. J.-P. Katoen. How to Model and Analyze Gossiping Protocols? acm sigmetrics' *Performance Evaluation Review*, 36(3):3–6, Dec 2008.
9. D. Liben-Nowell. Gossip is Synteny: Incomplete Gossip and the Syntenic Distance between Genomes. *Journal of Algorithms*, 43(2):264–283, 2002.
10. A. Rensink. The groove Simulator: A Tool for State Space Generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Proc. of AGTIVE 2004)*, LNCS 3062, pages 479–485. Springer, 2004.
11. A. Rensink. Isomorphism Checking in groove. In A. Zündorf and D. Varró, editors, *Proc. of Graph-Based Tools (GraBaTs) 2006, Natal, Brazil*, volume 1 of *Electronic Communications of the EASST*, 2006.
12. A. Rensink and J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In A. Boronat and R. Heckel, editors, *Proc. of Graph Transformation and Visual Modeling Techniques, York, UK, March 2009*, volume 18 of *Electronic Communications of the EASST*, 2009.
13. T. C. Ruys. Low-Fat Recipes for Spin. In K. Havelund, J. Penix, and W. Visser, editors, *Proc. of SPIN 2000*, LNCS 1885, pages 287–321. Springer, 2000.
14. T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, Enschede, The Netherlands, March 2001.
15. T. C. Ruys. Optimal Scheduling Using Branch and Bound with SPIN 4.0. In T. Ball and S. K. Rajamani, editors, *Proc. of SPIN 2003*, LNCS 2648, pages 1–17. Springer, 2003.
16. R. Seindal. *GNU m4, version 1.4.16*. Free Software Foundation, Inc., 1.4.16 edition, 2011. Available from http://www.gnu.org.
17. F. W. Vaandrager. A First Introduction to uppaal. In G.J. Tretmans, editor, *Quasimodo Handbook*. Springer, 2012. To appear.
18. *bliss* – A Tool for Computing Automorphism Groups and Canonical Labelings of Graphs. http://www.tcs.hut.fi/Software/bliss/.
19. gap – system for computational discrete algebra. http://www.gap-system.org/.
20. groove – graphs for object-oriented verification. http://groove.cs.utwente.nl/.
21. Saucy2 – Fast Symmetry Discovery. http://vlsicad.eecs.umich.edu/BK/SAUCY/.
22. swarm - verification script generator for Spin. http://www.spinroot.com/swarm/.
23. topspin. http://www.doc.ic.ac.uk/~afd/topspin/.