Alpaga: Two Examples

Dietmar Berwanger[1], Krishnendu Chatterjee[2], Martin De Wulf[3], Laurent Doyen[3,4], and Thomas A. Henzinger[4]

[1] LSV, ENS Cachan and CNRS, France
[2] CE, University of California, Santa Cruz, U.S.A.
[3] Université Libre de Bruxelles (ULB), Belgium
[4] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

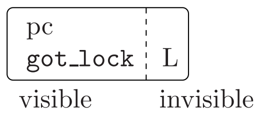
Abstract. We illustrate the use of imperfect-information games for the synthesis of distributed programs. We give two examples of distributed-system synthesis solved with Alpaga[1]. First, we demonstrate the need for imperfect information in the games that arise in the synthesis problem by considering a simple lock-based program. The specification requires that the lock is never acquired or released twice in a row, even if the status of the lock is not visible to the program. We also consider the design of a mutual-exclusion protocol for two processes. Using Alpaga, we have synthesised a winning strategy for a requirement of mutual exclusion and starvation freedom which corresponds to Peterson's protocol. We present two examples of .

1 Examples

1.1 A Locking Example

Consider the abstract program in Fig. 1 that acquires and releases a lock [2]. The `if`-statement in line 1 is a nondeterministic choice that abstracts different concrete conditions. The choice will be resolved by Player 2 in each iteration of the loop. The functions `lock()` and `unlock()` should be called in strict alternation in order to satisfy the assertions over the boolean variable `L`. However, this variable is not visible to the program. The program has an integer variable `got_lock` that can be updated in line 3 and 6 by four possible assignments `inc`, `dec`, `s0`, or `s1`.

Fig. 2 shows the game graph for this program. Each state has the following shape:



where pc is the program counter. The variable L is not visible to Player 1. We use ϵ-transitions from states with $pc = 1$ to emphasize that Player 2 actually chooses the successor state. Otherwise, the action a executes the next computation step, and the actions `inc`, `dec`, `s0`, or `s1` model the choices of Player 1. The error state corresponds to the violation of an assertion in `lock()` or `unlock()`. We have not constructed the full game graph as one can immediately see that some parts are not useful. For instance, from the four states 120, 110, $1-20$, and $1-10$, Player 2 should choose to skip the `if`-statement as this choice forces the game to the error state.

When solving the game with Alpaga, we find that Player 1 is winning, and should play s_0 in line 6. In line 3, all actions can be played except s_0.

[1] http://www.antichains.be/alpaga

```
   int got_lock = 0;                          void lock() {
   do {                                          assert(L == 0);
1.     if (*) {                                  L=1;
2.             lock();                        }
3.             | got_lock++ (inc);
               | got_lock-- (dec);            void unlock() {
               | got_lock=1 (s1);                assert(L == 1);
               | got_lock=0 (s0);                L=0;
       }                                      }
4.     if (got_lock != 0) {
5.             unlock();
       }
6.     | got_lock++ (inc);
       | got_lock-- (dec);
       | got_lock=1 (s1);
       | got_lock=0 (s0);
   } while(true)
```

Fig. 1. Locking.

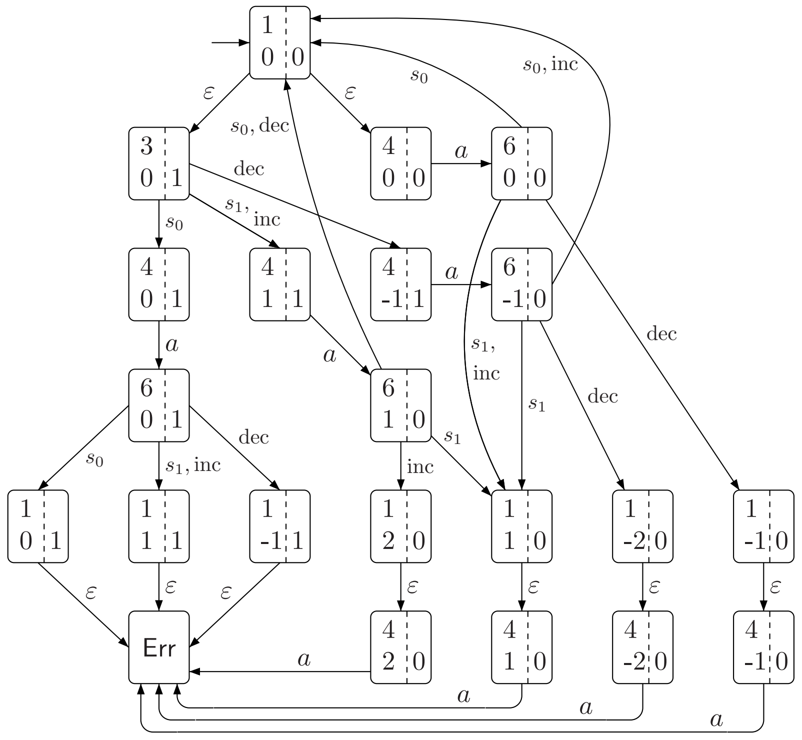


Fig. 2. Game graph for the locking example.

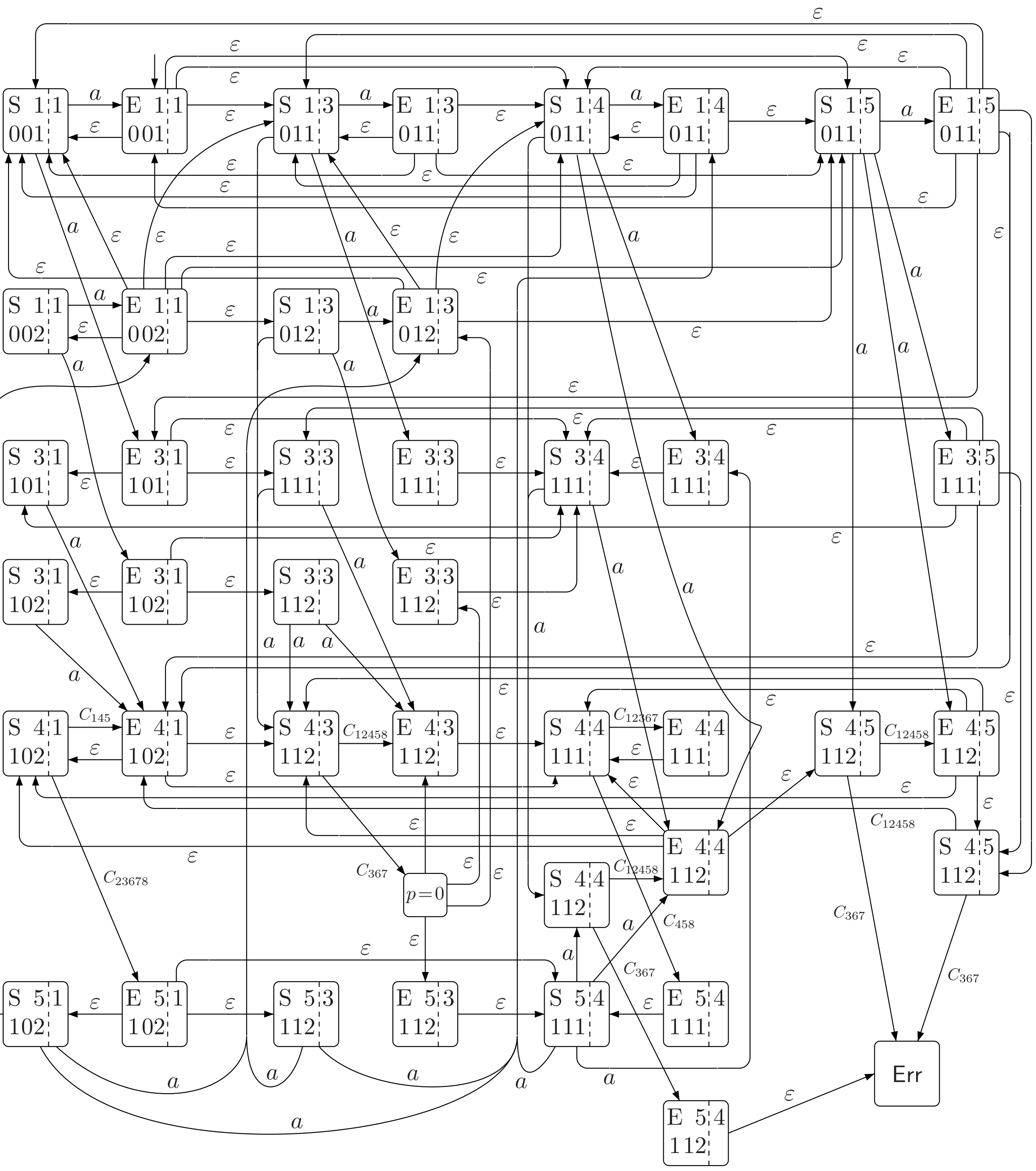


Fig. 3. Complete game abstraction for the synthesis of a mutual exclusion protocol.

1.2 Mutual-exclusion protocol

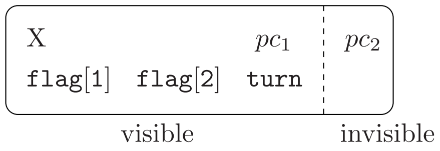
We illustrate the use of games with imperfect information as a model for the synthesis of reactive programs in distributed systems. For technical reasons (we have to encode the game manually) and because our aim is to illustrate the feasibility of the approach in principle, we present a simplified version of the problem, and we construct a model as simple as possible. To our knowledge, this is the first attempt to use a solver for imperfect information games to automatically synthesize distributed programs.

We consider the design of a mutual-exclusion protocol for two processes, following the lines of [1]. We assume that one process (on the right in Fig. 4) is entirely specified. The second process (on the left in Fig. 4) has freedom of choice in line 4. It can use one of 8 possible conditions `C1`–`C8` to guard the entry to its critical section in line 5. The boolean variables `flag`[1] and `flag`[2] are used to place a request to enter the critical section. They are both visible to each process. The variable `turn` is visible and can be written by the two processes.

There is also some nondeterminism in the length of the delays in lines 1 and 5 of the two processes. The processes are free to request or not the critical section and thus may wait for an arbitrary amount of time in line 1 (as indicated by `unbounded_wait`), but they have to leave the critical section within a finite amount of time (as indicated by `fin_wait`). In the game model, the length of the delay is chosen by the adversary.

Finally, each computation step is assigned to one of the two processes by a *scheduler*. We require that the scheduler is fair, i.e. it assigns computation steps to both processes infinitely often. In our game model, we encode all fair schedulers by allowing each process to execute for an arbitrary finite number of steps, before releasing the turn to the other process. Again, the actual number of computation steps assigned to each process is chosen by the adversary.

Fig. 5 shows the game structure for the two processes under a fair scheduler. Each state has the following shape:



where $X =$ `S`(ystem) if the next step is assigned to the first process, and $X =$ `E`(nvironment) otherwise. The program counters pc_1 and pc_2 take their values in the set $\{1, 3, 4, 5\}$ and it corresponds to the line of the next instruction to execute. To avoid too many states, we disallow 2 and 6 as possible values for the program counters since the lines 1 and 5 are delays, and we can assume that either the process waits there, or it executes (at least) the next line. The `flag`[·] and `turn` variables are boolean, with `flag`$[\cdot] \in \{0, 1\}$ (where `true` $= 1$ and `false` $= 0$) and `turn` $\in \{1, 2\}$. All variables are visible to the first process, except the program counter pc_2 of the second process. To keep Fig. 5 readable, we have only represented one step of computation for the system states (Fig. 3 shows the full game).

The specification for the first process consists of two parts: a safety part $\Phi_1^{\mathtt{mutex}} = \Box\neg(\mathtt{pc}_1 = \mathtt{5} \wedge \mathtt{pc}_2 = \mathtt{5})$ and a liveness part $\Phi_1^{\mathtt{prog}} = \Box(\mathtt{flag}[1] = \mathtt{true} \;\rightarrow\; \Diamond(\mathtt{pc}_1 = \mathtt{5}))$. The first part $\Phi_1^{\mathtt{mutex}}$ specifies that both processes are never simultaneously in the critical sections (*mutual exclusion*); the second part $\Phi_1^{\mathtt{prog}}$ specifies that if process P_1 wishes to enter its critical section, then it will eventually enter (*starvation freedom*). The specification for the first process is the conjunction $\Phi_1^{\mathtt{mutex}} \wedge \Phi_1^{\mathtt{prog}}$. Priorities can be assigned to each states to encode this specification. The states where $pc_1 = 5$ have priority 2, and the other states have priority 1 if $flag[1] = true$ and priority 0 if $flag[1] = false$. The state where both processes are in the critical section $pc_1 = pc_2 = 5$ is assigned an odd priority, and no outgoing transition is allowed. Notice that this assignment of priorities is correct because once the entry to the critical section is requested, the flag remains `true` until the request is granted. We should also consider the symmetric specification for the second process, but it would require a larger state space, so we omit it in this first example.

In Fig. 5, the action a represents a step of the first process, while ε represents one or more steps of the second process (this is for the sake of clarity, in the model we can view a and ε as a unique action). The nondeterminism encodes the choices of the adversary. For instance, in state $S11001$, when the system plays

```
do {                                              do {
1. unbounded_wait;                                1. unbounded_wait;
2. flag[1]:=true;                                 2. flag[2]:=true;
3. turn:=2;                                       3. turn:=1;

4. | while(flag[1]) nop;           (C1)           4. while(flag[1] & turn=1) nop;
   | while(flag[2]) nop;           (C2)
   | while(turn=1) nop;            (C3)
   | while(turn=2) nop;            (C4)
   | while(flag[1] & turn=2) nop;  (C5)
   | while(flag[1] & turn=1) nop;  (C6)
   | while(flag[2] & turn=1) nop;  (C7)
   | while(flag[2] & turn=2) nop;  (C8)

5. fin_wait;  // Critical section                 5. fin_wait;  // Critical section
6. flag[1]:=false;                                6. flag[2]:=false;
} while(true)                                     } while(true)
```

Fig. 4. Mutual-exclusion protocol synthesis.

a, there are two possible successors: either $E31101$ which means that the first process has executed one step, requesting the critical section, or $E11001$ which means that the first process was still waiting. In the states with $X = \mathtt{S}$ and $pc_1 = 4$, one of the choices `C1`–`C8` is expected.

When solving this game with Alpaga, we find that $E11001$ belongs to a winning cell, and that the strategy that chooses C_8 is winning.

References

1. K. Chatterjee and T. A. Henzinger. Assume-guarantee synthesis. In *Proc. of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 4424, pages 261–275. Springer, 2007.
2. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Proc. of CAV: Computer-Aided Verification*, LNCS 3576, pages 226–238. Springer, 2005.

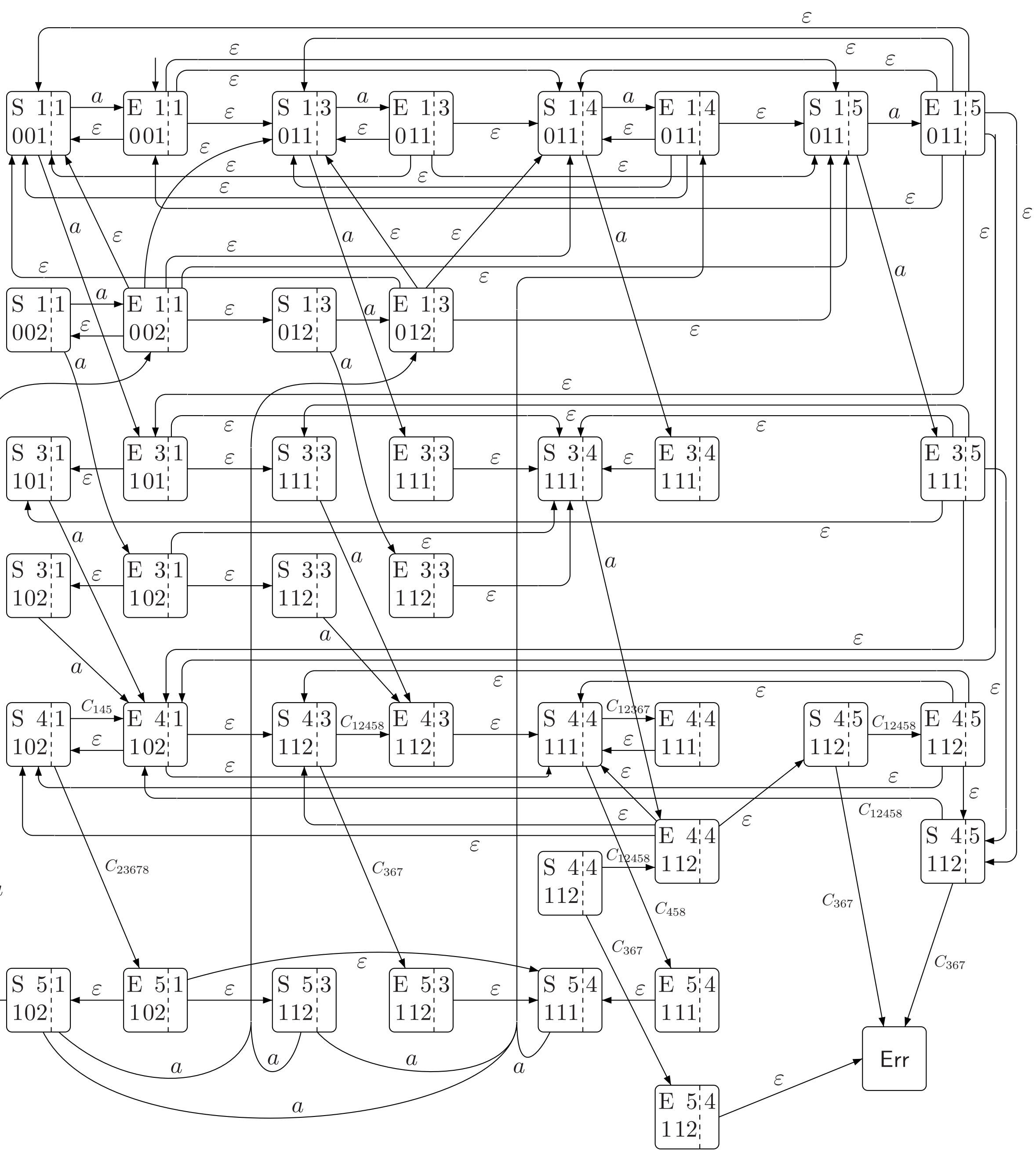


Fig. 5. Game abstraction for the synthesis of a mutual exclusion protocol.