Back to Basics: Solving Games with SAT

Stefano QUER
Dip. di Automatica e Informatica, Politecnico di Torino, Turin, Italy
stefano.quer@polito.it

Abstract—Games became popular, within the formal verification community, after their application to automatic synthesis of circuits from specifications, and they have been receiving more and more attention since then. This paper focuses on coding the "Sokoban" puzzle, i.e., a very complex single-player strategy game. We show how its solution can be encoded and represented as a Bounded Model Checking problem, and then solved with a SAT solver. After that, to cope with very complex instances of the game, we propose two different ad-hoc divide-and-conquer strategies. Those strategies, somehow similar to state-of-the-art abstraction-and-refinement schemes, are able to decompose deep Bounded Model Checking instances into easier subtasks, trading-off between efficiency and completeness. We analyze a vast set of difficult hard-to-solve benchmark games, trying to push forward the applicability of state-of-the-art SAT solvers in the field. Those results show that games may provide one of the next frontier for the SAT community.

***Index Terms*—artificial intelligence, algorithm design and analysis, Boolean algebra, formal verification, partitioning algorithms.**

I. Introduction

Since their application to automatic synthesis of circuits from specifications [1], games have become popular in several formal fields. These include control of discrete event systems [2], realizability and synthesis, model checking [3], planning [4], μ-calculus [5-6], and the analysis of systems where the distinction among the choices controlled by different components are made explicit.

Research and related applications have led to a variety of game formulations [6-10], such as infinite games on finite graph, concurrent multi-player games, etc. [5]. Though the theoretical complexity of solving various games is well understood, there has been relatively less effort in identifying how modern symbolic strategies can be applied and perform on them.

The simplest games, that most solutions computationally rely on, are single and two-player reachability and safety games on finite graphs.

Single-player games are played by one single entity, usually referred to as the system. The target of the system is to reach a fixed goal starting from the initial configuration of the game. This problem is usually analyzed as a reachability quest, i.e., as a task where it is necessary to check the existence of a sequence of moves that will force the game from the initial configuration to the final goal.

Two-player games are played between two entities, the system and the environment. The problem is to check whether the system has a winning strategy, i.e., a strategy that will force the game from the initial configuration to some target position, no matter how the environment plays. As far as single player games are concerned, these problems are often analyzed in terms of reachability, as the system must be able to reach a winning position, starting from the initial configuration, in all possible scenarios.

In this paper, we will consider only perfect-information and positional (or memory-less) games, i.e., games in which the player(s) has (have) perfect information about the game. In those puzzles the strategy is based only on the current state of the game.

Recent works, such as the one from Alur et al. [6], showed how Binary Decision Diagrams (BDDs) are more efficient than Satisfiability (SAT) and Quantified Boolean Formulas (QBF) tools to solve a few selected games. Anyhow, given the increasing efficiency of SAT and QBF solvers, the effort in coding problems in different ways, and the variety of available games and encodings to experiment with, this issue is still open.

This paper focuses on encoding the "Sokoban" puzzle", i.e., a very complex strategy game, and to encode it as a Bounded Model Checking (BMC) problem. As far as we know, this sort of encoding, and the subsequent resolution schemes, has never been presented before.

Sokoban is a "transport" puzzle, that is, a game in which the player has to push boxes in a labyrinth, in order to put them in designed places. The BMC formulation encodes the existence of a winning strategy, i.e., a sequence possible legal moves done by the player to reach the final configuration (the final goal). As a consequence, the length (or depth) k of the BMC unrolling coincides with the number of moves of the player. Checking the existence of a solution with increasing bound delivers the shortest sequence of moves leading to success.

We show that, even with modern state-of-the-art satisfiability solvers, this straightforward "standard" BMC formulation is able to solve only extremely simple, and shallow, instances of the game.

To alleviate this problem, we present two abstraction and refinement techniques. Both methods decompose the game into several easier sub-problems, usually one sub-problem for each box present in the initial configuration scheme. In the first strategy, those sub-problems are solved in a loosely coupled way, i.e., they are analyzed in a specific order, one at a time, such that the solution of one of them is used to constrain all subsequent ones. This strategy, albeit more efficient and scalable than the original straightforward algorithm, is able to solve the game only when there exists a complete serial decomposition of the solution. In the second algorithm, sub-games are more tightly coupled. They evolve in lock-step, one step at a time, such that any partial solutions is used to restrict the search spaces of all others sub-games. Albeit less effective to decompose the game into easier sub-problems, this strategy is sound and complete,

Digital Object Identifier 10.4316/AECE.2016.03013

and it allows a better trade-off between efficiency and completeness.

Although our resolution strategies are not intended as a serious attack on solving the Sokoban puzzle, they are indicative of the performance of a SAT using standard techniques, without heuristics or propagation rules specifically realized and tuned for the game.

A. Contributions

Given the previously described overall flow, our contributions are the following.

We present a Conjunctive Normal Form (CNF) encoding of a real challenging game, and we solve it through proper divide-and-conquer decomposition, adopting two abstraction and refinement methods. Those strategies target the decomposition of very deep BMC instances into several easier sub-problems. The presented problems are challenging benchmarks for the SAT community. We compare the most efficient SAT solvers available today, and we contrast them on different hardware architectures.

To sum up, our experiments show that, albeit SAT tools can be very efficient on simple games, they are not yet mature enough to deal with complex games, which still provide extremely hard-to-solve instances.

B. Road-Map

The paper is organized as follows. Section II introduces our notation and the model we adopted. Section III presents the Sokoban game and our solution procedures. Section IV concentrates on benchmarks and related experimental results. Finally, Section V concludes the paper with some summarizing remarks.

II. Background

A. Model and Notation

In our notation, B indicates the Boolean space. Symbols such as $\wedge$, $\vee$, $\neg$, $\leftrightarrow$, and $\rightarrow$ are used for Boolean conjunction (AND), disjunction (OR), negation (NOT), exclusive-nor (XNOR) and implication

$$x \leftrightarrow y = \neg x \vee y \tag{1}$$

respectively.

We frequently adopt an indexed set of Boolean variables $V = \{v_1, v_2, ..., v_n\}$, and the notation V^i to indicate such a set V referred to time frame i. We use both v ($\neg v$) and $v = 1$ ($v = 0$) to express the same meaning, that is, a variable in its uncomplemented or complemented form.

A state predicate (SP) is a Boolean formula over V. We write $SP(x)$ to denote $SP|_{v_i / x_i}$, i.e., the predicate SP with each variable v_i replaced by x_i.

B. The Game Structure

We model single- and two-player games using the structure $S = (W, X, TR)$, where W, X, TR indicate the input variables, the state variables, and the transition predicate, respectively. Among the X variables we make distinction between the present state variables, denoted with P, and the next state variables, denoted with N. The transition relation $TR(\alpha, \omega, \beta)$ is true if and only if the game moves from state α to state β when the player selects the move ω.

Given a game structure S, we define a game G as a tuple $G = (S, I, T, F)$, where I is the initial state of the game, T is the target predicate (the goal), and F is a safety predicate.

The behavior of a two-player game G is the following one. G starts in its initial state I. At each time frame, the two players (the system and the environment) make their move, such that the game evolves according to the rules given by its transition predicate TR. Given a state P, then as long as $F(P)$ is true the game stays in its safe region; whereas whether $T(P)$ is true the goal of the game has been reached. If a move makes the game reach the goal region, the player playing that move wins. If a move makes the game going outside the safe region, the player playing the move loses. Thus a winning strategy for a player consists of a set of moves which, starting from the initial configuration I, allows the player to make the move entering the goal region T for every possible move done by the other player. A reachability game (or guarantee game) is a game in which the problem is to check whether one player has a winning strategy, forcing the game from I to T. A safety game is a game in which F must always be false.

C. Satisfiability

Given a Boolean formula f depending on V, the Boolean Satisfiability problem [11], usually known as SAT, consists in finding, if it exists, an assignment to the variables of f, namely $\{v_1, v_2, ..., v_n\}$, such that f is true, i.e., $f(v_1, v_2, ..., v_n) = 1$. SAT solvers work on Boolean formulas usually expressed in Conjunctive Normal Form (CNF), even if several extension have been proposed over the year (see for example [12-14]). A CNF formula is a conjunction of *clauses*, each of which is a disjunction of *literals*. Each literal is either a variable or its complementation.

D. Bounded Model Checking

Given a sequential system, SAT based Bounded Model Checking (BMC) [15] builds a propositional formula that is satisfiable iff there is a path from I to T of bounded length k. More specifically, a BMC run of depth k unfolds the transition relation of the system k times

$$TR^k(V^0, ..., V^k) = TR_1(V^0, V^1) \wedge ... \wedge TR_k(V^{k-1}, V^k) \tag{2}$$

and uses a SAT solver to check the satisfiability of

$$f_k = I(V^0) \wedge TR^k(V^0, ..., V^k) \wedge T(V^k) \tag{3}$$

If f_k is unsatisfiable, there is no path of length k connecting I and T, and a larger value of k should be tried.

A special version of the BMC process exploits the concept of incrementality [16], whose main idea is re-using the set of conflict clauses (generated by the solver) across several calls to the SAT solving routine. We also exploit this notion by means of the unit assumptions mechanism [17].

III. THE SOKOBAN GAME

Sokoban was created by Hiroyuki Imabayashi in 1980. The game is a transport puzzle, as perfectly described by its name "Sokoban", meaning "warehouse keeper" in Japanese. Fig. 1 reports a graphical representation of the game board.

The goal of the game is simple. The player (the black dot) has to push the boxes (gray squares) in a labyrinth (black walls), in order to put them in designed places (crossed cells in the grid). The rules that have to be respected are simple as well. The player may move on the grid only horizontally or vertically, one cell at a time; boxes may be pushed but they cannot be pulled; only one box at a time may be pushed, provided that the cell it will occupy is empty.

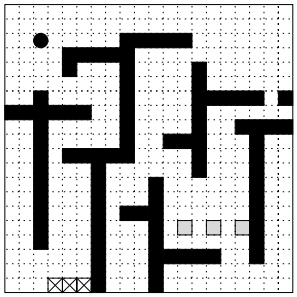

Figure 1. The Sokoban puzzle: A correct initial configuration on a square board of size 20.

The game can be encoded as an automaton, implicitly represented through its transition relation TR. An instance of TR represents exactly one move of a player, where the move is specified through the system primary inputs. The set of allowed moves is up, down, left, right. Such a set can be encoded by means of two Boolean variables. However, we will actually use a non-deterministic transition relation TR_{ND}, formally defined as:

$$TR_{ND}(P,N) = \exists_w TR(P,W,N) \tag{4}$$

We do this because the expression of TR_{ND} is easier to specify, thus leading to a smaller set of clauses. The sequence of moves the player has to perform in order to solve the game can be then retrieved by examining the solution itself. Once the transition relation is generated, our target is to find a sequence of states $(s_0, s_1, ..., s_k)$ connecting the initial game configuration ($I(s_0)$) to the goal one ($T(s_k)$), without violating the game rules ($TR_{ND}(s_i, s_{i+1})$). We perform this task resorting to SAT-based BMC. In our approach, each cell rc of the Sokoban game board is encoded with two Boolean variables, denoted as high (h) and low (l), with the following meaning:

- The cell is empty.

$$p^h = 0 \wedge p^l = 0 \tag{5}$$

- The cell is occupied by the player

$$p^h = 0 \wedge p^l = 1 \tag{6}$$

- The cell is occupied by a box.

$$p^h = 1 \wedge p^l = 0 \tag{7}$$

In other words, the two bits can be viewed as representing the presence of a box or the player respectively. In other words the cell is:

- Occupied by a box iff the high bit is true.
- Occupied by the player iff the low bit is true.
- Empty iff both bits are false.

The two bits cannot both be true at the same time.

Given this representation, the rules for the case of a cell not surrounded by wall cells (constraints coming from walls can be directly used to simplify the equations) are encoded as follows:

- Rule number 1: In every reachable state, the bits encoding the cell cannot be asserted at the same time. Since this condition obviously holds for the initial state, the rule is expressed only on the next state variables:

$$\neg n^l_{rc} \vee \neg n^h_{rc} \tag{8}$$

- Rule number 2: If a cell will be occupied in the next state by the player, the player must be in one of the adjacent cells in the current state (see Fig. 2):

$$n^l_{rc} \rightarrow (p^l_{(r-1)c} \vee p^l_{(r+1)c} \vee p^l_{r(c-1)} \vee p^l_{r(c+1)}) \tag{9}$$

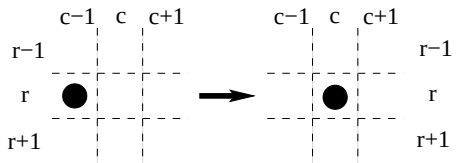

Figure 2. Rule number 2: Player position for a left-to-right move. All other moves have similar graphical representations.

- Rule number 3: The player may only move in one of the adjacent cells (see Fig. 3):

$$\begin{aligned} & p^l_{rc} \rightarrow \\ & [(n^l_{(r-1)c} \wedge \neg n^l_{(r+1)c} \wedge \neg n^l_{r(c-1)} \wedge \neg n^l_{r(c+1)}) \vee \\ & (\neg n^l_{(r-1)c} \wedge n^l_{(r+1)c} \wedge \neg n^l_{r(c-1)} \wedge \neg n^l_{r(c+1)}) \vee \\ & (\neg n^l_{(r-1)c} \wedge \neg n^l_{(r+1)c} \wedge n^l_{r(c-1)} \wedge \neg n^l_{r(c+1)}) \vee \\ & (\neg n^l_{(r-1)c} \wedge \neg n^l_{(r+1)c} \wedge \neg n^l_{r(c-1)} \wedge n^l_{r(c+1)})] \end{aligned} \tag{10}$$

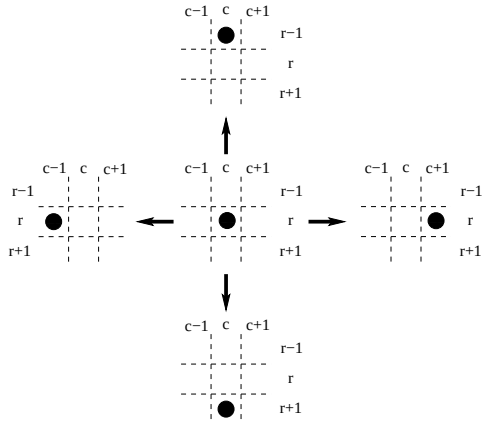

Figure 3. Rule number 3: The player may move horizontally or vertically of one single cell.

- Rule number 4: If a cell will be occupied by a box in the next state, the box was already there or it has

just been pushed there. In this case, the player must be behind the box in the current time step, and he will take the box position in the next time step (see Fig. 4):

$$
\begin{aligned}
& n^h_{rc} \rightarrow \\
& [p^h_{rc} \vee \\
& (p^h_{(r+1)c} \wedge p^l_{(r+2)c} \wedge n^l_{(r+1)c}) \vee \\
& (p^h_{(r-1)c} \wedge p^l_{(r-2)c} \wedge n^l_{(r-1)c}) \vee \\
& (p^h_{r(c+1)} \wedge p^l_{r(c+2)} \wedge n^l_{r(c+1)}) \vee \\
& (p^h_{r(c-1)} \wedge p^l_{r(c-2)} \wedge n^l_{r(c-1)})]
\end{aligned}
\tag{11}
$$

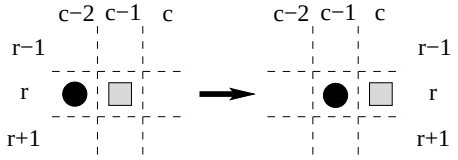

Figure 4. Rule number 4: Box and player positions for a left-to-right move. All other moves have similar graphical representations.

Notice that the walls of the Sokoban game board are not explicitly represented: In our framework, constraints coming from walls are directly used to simplify the given formulas.

Once all these expressions are in place, the automaton transition relation is simply given by their conjunction. The system description is then completed by expressing the initial and the target states, which is straightforward to do given the encoding previously described.

In Section IV we will show that the game, encoded as we have just described, is extremely difficult to solve, even for the fastest SAT tools available today. Although this limitation is due to several factors, our analysis shows that one of the main problems is the depth of the unrollings that have to be considered to find a solution with BMC. In turn, the solution depth strongly depends on the number of boxes that have to be moved onto the right place (schemes with just one box are usually trivial, and having several boxes on the grid, let us say up to 10 boxes, is normal). From the one hand, it is understandable that the number of moves tends to increase as the number of boxes increases. From the other one, we have observed that, with several boxes, most of the moves are performed just to relocate the player (not the boxes) from one place of the board to another one, without making any real progress.

In order to alleviate this problem, we present a divide-and-conquer approach, in which the player has the ability to duplicate himself whenever necessary, so that he (they) can push more than one box at a time. The approach can also be seen as an abstraction and refinement strategy, as the initial solution is found on an abstract model, and it has to be refined to be compliant with the concrete model. This approach can be implemented in different ways. We will present two different decomposition techniques in the following two subsections. We refer to these decomposition strategies as “Game by Game” and “Frame by Frame”, rephrasing a terminology firstly introduced by Cho et al. [18] while performing over-approximate reachability analysis with BDDs.

A. Game by Game Decomposition

Let us deal with a game G, as defined in Section II-B. In the Game by Game (GBG) approach, the player is conceptually duplicated up-front, before even starting any move. The number of player copies is equal to the number of boxes that have to be moved to the right place. We then divide the game into several sub-games, one for each box pushed by one copy of the player. All sub-games are analyzed in turn using BMC. Every solution found is used to constrain the following sub-games.

This situation is depicted in Fig. 5. In this case, the original puzzle of Fig. 5(a) is decomposed into the three sub-problems of Fig. 5(b), one for each box. Once the first sub-problem (the one on top) is solved, its solution is used to constrain the subsequent two sub-problems, as the presence of the first box and the first player in the labyrinth (in specific places at specific times) may be used to restrict the resolution spaces of all other sub-problems. If it is possible to solve all sub-problems, the global solution can be derived from those solutions. Otherwise, if one of the problem does not have a solution given the restrictions imposed by previous sub-problems (or because a global solution requires a more fine-grained interaction among sub-problems), these restrictions have to be ruled-out and that particular sub-problem re-analyzed to find a different solution.

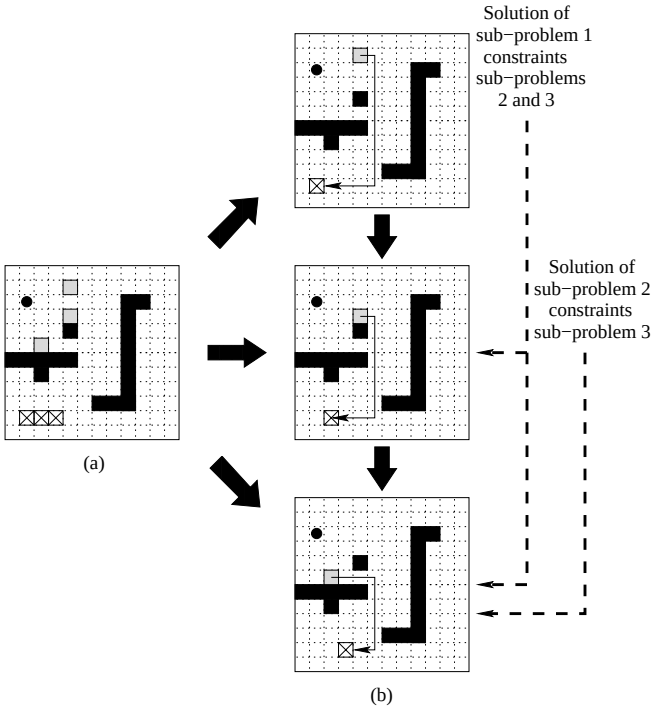

Figure 5. Sokoban decomposition following the GBG strategy: The original problem (a) is decomposed into three sub-problems (b). Sub-problems solved in sequence one after the other. Each partial solution is used to restrict the search space of all subsequent sub-problems.

As the main idea of the method is to process each sub-game separately, we call this method Game by Game. Fig. 6 shows the pseudo-code of the procedure applying this strategy.

The *GBG* routine receives the game G as parameter. This game G is then decomposed into the necessary sub-games $\{G_i\}$ by the *decomposeGame* function (line 2).

```
1  GBG (G)
2     {G_i} = decomposeGame (G)
3     do
4        Trace = φ
5        {G_i} = sort ({G_i})
6        for each g ∈ {G_i}
7           trace = BMC (g)
8           if (trace = φ)
9              break
10          // Constrain {G_i} with trace
11          {G_i} = constrain ({G_i} \ g, trace)
12          Trace = Trace ∪ trace
13       // Find Global Trace
14       if (trace ≠ φ)
15          Ĝ = constrain (G, Trace)
16          trace = BMC (Ĝ)
17    while (trace = φ ∧ ¬globaLimit)
18    return (trace)

19 BMC (g)
20    {I, TR, T} = extract (g)
21    k = 0
22    do
23       f = I(V^0) ∧ (∏_{i=0}^{i=k-1} TR(V^i, V^{i+1})) ∧ T(V^k)
24       trace = SAT( f )
25       k = k + 1
26    while (trace = φ ∧ ¬localLimit)
27    return (trace)
```

Figure 6. Serial decomposition of the Sokoban game, using the Game by Game (GBG) strategy.

Decomposing G into the set of sub-problems $\{G_i\}$ (one for each box on the game board) implies partitioning the initial and target state sets, so that each element of $\{G_i\}$ is described by a proper initial state set I_i and a proper target state set T_i. In particular, I_i specifies an initial condition in which only one player and one box are on the board, whereas all other boxes are considered as walls. Furthermore, T_i generally encodes a situation in which the current box may be moved to one among all possible final destinations.

Once this decomposition is performed, all sub-games have to be traversed independently. The order in which the sub-games are traversed is very important to obtain a good performance of the algorithm. This order is computed by function $sort$ on line 5. One strategy we use in this context is to assign to each sub-game a priority depending on the physical distance between the initial position of the box and the closest target not yet assigned.

After that, we perform a BMC analysis for each sub-problem $g \in \{G_i\}$. The BMC routine (called on line 7 and defined on lines 19-27) is used to find a trace from I_i to T_i for the current sub-game. To solve a sub-problem, a Boolean formula f is built by unrolling k times the game transition relation, in conjunction with the system initial and target states (line 23). Function SAT, called on line 24, performs the satisfiability analysis of the formula, possibly returning the satisfying assignment. If the formula is false, k is increased and the process restarts from line 23.

The main iteration of function BMC is also controlled by a local threshold on the available resources (CPU and memory). This threshold is tighter than the global one ($globalLimit$) used in line 17. Controlling the resource limit is necessary because, in the GBG decomposition, sub-games are not guaranteed to have a solution. When a trace is not found (i.e., the local resource limit is exceeded) the break command on line 9 forces a jump to line 17, where the global resource limits (CPU and memory) are checked, and the entire process is possibly restarted by reordering the sub-games on line 5. When a trace is found, it is used to constrain all other sub-games (line 11) using function $constrain$. In this way, though each sub-problem is traversed individually, the current trace is used to pose constraints on all sub-games still to be analyzed. This means that the solution eventually found for each sub-game will be compatible, as far as the position of the boxes is concerned, with the ones obtained up to that moment. Once all single sub-games $g \in \{G_i\}$ have been analyzed, all single traces have to be checked for congruence. This step is performed in lines 15-16, where the original game G is constrained with the global trace ($Trace$), and then it is checked for a global solution.

This approach is relatively simple, but it does not guarantee to find a solution for complex games. This is due to two factors:

- The game may not be solvable with a serial decomposition of its simple components, as it may require their interaction during the solution phase. For example, there are situations in which one box can be moved to one of the final destinations only when another one has been moved around “to free” a passage for it.
- Even if the game is solvable adopting a serial decomposition, the existence of a solution depends on the order in which the components are analyzed. To be complete, given n components we should try $n!$ possible permutations, which is obviously feasible only for very small values of n. This possibility is taken into consideration by the do-while loop starting in line 3, by which we may try another sub-game order to solve the problem, until the available resources are exhausted.

B. Frame by Frame Decomposition

In this section, we present an algorithm based on a quite different strategy from the one adopted by the GBG algorithm of Section III-A. The approach is more complex and somehow less efficient in terms of its decomposition power, but it is guaranteed to find a solution, given enough time and memory, when a solution exists. As for the GBG algorithm, the new approach adopts an abstraction and refinement strategy, as an initial over-approximation of the original problem is subsequently refined to deliver a correct

result. However, this decomposition is closer than the GBG approach to the original method in its algorithmic construction.

The idea is simple. Instead of traversing each sub-game completely before processing the next one, the new method handles all sub-games in parallel. Interaction between sub-games is more fine-grained, since the base unit of interaction is a time frame rather than an entire BMC verification phase. These considerations led us to call the new technique "Frame by Frame" (FBF).

This is represented in Fig. 7. Fig. 7(a) shows the initial situation. After that, Fig. 7(b) illustrates the analysis done for the first time frame, and Fig. 7(c) for the second one. Whenever a conflict is detected, the conflict has to be ruled-out before proceeding along with the analysis.

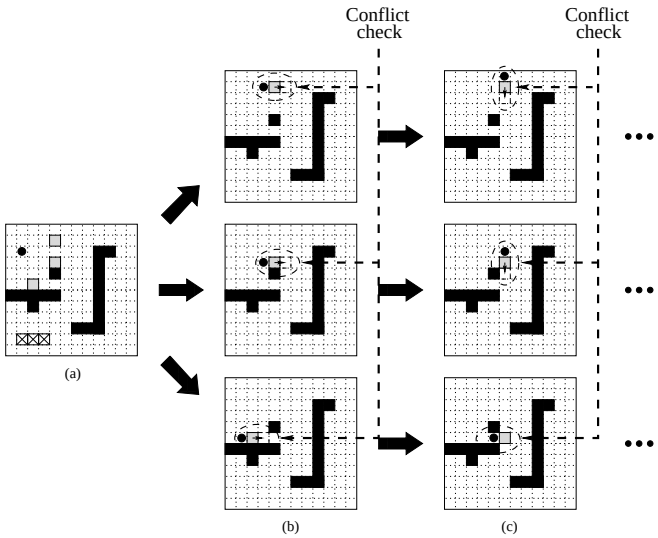

Figure 7. Sokoban decomposition following the FBF strategy: The original problem (a) is decomposed into three sub-problems (b) which are analyzed in parallel for one step and then validated with respect to each other. After the validation a further step (c) is performed on the three of them, till resolution.

```
1  FBF (G)
2    do
3      Ĝ = changeGame (G)
4      Trace = BMC (Ĝ)
5      if (Trace = φ)
6        return (φ)
7      // Constrain G with trace Frame by Frame
8      trace = φ
9      for i = 1 to length (Trace)
10       g = extract (G, timeFrames (Trace, i − 1, i))
11       trace_i = BMC (g)
12       if (trace_i = φ)
13         G = constrain (G, ¬Trace)
14         break
15       else
16         trace = trace ∪ trace_i
17   while (trace_i = φ ∧ ¬globalLimit)
18   return (trace)
```

Figure 8. Parallel decomposition of the Sokoban game, using the Frame by Frame (FBF) strategy.

Fig. 8 presents the pseudo-code illustrating the entire procedure.

Function *changeGame*, in line 3, generates a new game structure $\hat{G}$ from the original game G. In $\hat{G}$ the player's rules are modified. More specifically, moving from time frame t to time frame $t+1$ one single copy of the player can generate multiple copies of himself or, vice-versa, more copies of the player can converge in a single copy.

All copies have to respect the same set of rules as far as movements, or position incompatibility with walls and boxes are concerned.

The new game $\hat{G}$ is a strict over-approximation of the original game G, and it coincides with G when no player is duplicated. From a practical point of view, the game $\hat{G}$ is obtained from G by:

- Modifying the rule of Equation (9) such that the player is allowed to remain in the same cell.
- Deleting the rule of Equation (10).
- Modifying the rule of Equation (11) so that no box duplication may occur. Notice that according to Equations (8-11), no box duplication may occur when a single player is on the board. However, this is no more true when multiple players are available.

On this new game structure, function BMC looks for a solution trace. The BMC procedure, called in line 4, is the one defined in Fig. 6, and, for this reason, it is not duplicated here. If no solution is found, the function just returns an empty result (line 6), otherwise the solution has to be refined. Refinement is performed in the following way. For each time frame of the trace with multiple players (line 9), we generate a new game g, with the same rules of G (i.e., only one player), but with the target of performing a single step of the trace itself (line 10). Then, we look for a concrete trace on g, using again function BMC (line 11). If this trace exists, the process goes on to the next time frame. If the trace does not exist, we have to constrain the original game G such that the trace found in line 4 will be excluded in the future (line 13). The entire process is then restarted until the global resource limits are exceeded.

Theorem. Function FBF is sound and complete.

Sketch of Proof. Soundness. Every trace discovered on the abstract game $\hat{G}$ is verified on the concrete model G before being returned. Completeness. As $\hat{G}$ is an abstraction of G, including all behaviors of G itself, no solution is omitted.

IV. Experimental Results

In this section, we present our results to solve the Sokoban game. We compare the basic "monolithic" method (beginning of Section III) with the decomposed ones (Sections III-A and III-B).

We present results using several state-of-the-art SAT solvers, and we present the ones obtained with the fastest one plus the one gathered with Minisat (version 2.0). In particular, Minisat is used both as an external tool (as all the other solvers), and as an internal package, directly linked with the top-level procedure. In this last case, we adopt

incremental SAT in order to evaluate the strength of incrementality (see Section II-D) on these verification problems.

We ran all experiments on an AMD Phenom 9850 Quad Core 2.4 GHz, 64 bits machine, with 16 GBytes of main memory (limited to 8 GBytes), running Ubuntu Linux (version 14.04). The time limit was set to 3600 seconds (one hour) for all experiments.

We performed experiments on 20 home-made grids (named $home_i$) and on 100 instances (called $push\_push_i$ and $soko_i$) publicly available.

The home-made benchmarks include the less constrained configurations, whereas the soko suite is made up of quite large puzzles, with a remarkable number of boxes (sometimes more than 20). We provide data only for those instances on which we were able to complete the game within the given time limit (1 hour). In all the other schemes we ran out of time.

We present two sets of experiments.

In the first set, we applied the monolithic technique presented at the beginning of Section III. All results are reported in Table I. The performance of the different SAT solvers (such as Riss, Picosat, Lingeling, Minisat, etc.) seems to vary with the instance more than in other cases, and there is no clear winner among them. Incrementality also seems to help, sometimes drastically, but not in all the cases. Memory usage (that we do not detail) does not seem to be a problem, as it is limited to less than 0.5 GB in all cases. On the contrary, running time is the major hurdle for all unsolved instances. Indeed, we can complete, with the original approach, only 13 instances out of 120, all of them with relatively small value of the BMC bound.

In the second set of experiments, we ran the GBG and FBF techniques described in Sections III-A and III-B.

The GBG approach was able to solve 8 puzzles among the home-made instances, whereas it completed only 3 benchmarks taken from the WEB. The reason for this is that, for almost all these benchmarks, the solution cannot be obtained through a simple serial decomposition, since reaching a solution requires alternate movements of different boxes. Anyway, on the solved instances the GBG technique outperformed both the original method and the FBF technique in terms of time and memory.

The FBF approach is usually fast at finding a solution on the abstract model, but often these solutions have to be refined to hold on the concrete game. Overall, the FBF strategy could complete 87 abstract schemes, and it was able to refine 35 of them to a true concrete solution within the time limit. Given the large amount of data collected, Table 2 reports results only for the FBF strategy, and only for those instances that the monolithic method was able to solve (i.e., the ones presented in Table 1). We also report data only for the Minisat solver adopting incremental SAT. For the monolithic approach the meaning of the columns is the same of Table I. For the FBF strategy the meaning of the columns is the following. The abstract bound (column Abs) is the depth of the trace on the abstract model (named Trace in Fig. 8). The concrete bound (column Con) is instead the length of the final trace, i.e., the one valid on the concrete model. Column #Ref indicates the number of refinements done (the number of iterations of the loop at line 2 of Fig. 8).

Analyzing the data of Table II, we can draw the following conclusions:

- First of all the (exact) bound obtained with the monolithic method indicates the length of the shortest possible solution on the concrete model. As it can be noticed, the abstract bound is always smaller than this value, and the concrete bound is always larger. This is not surprising, as the FBF method decomposes a single BMC quest into several shortest and easier searches. If from the one side, this choice avoids erratic behavior and movements (which are really inconvenient while looking for a very deep solution), on the other one it focuses the player toward the target, often forcing him to push in turn almost all boxes by one position. These considerations explain the advantages in terms of time and scalability but also the possible overheads of our decomposition scheme.
- The number of refinements needed by the technique is quite small. This means that the number of "adjustments" needed to concretize the abstract model is limited.
- The time needed by the FBF approach can be larger than the one required by the original encoding. This is mainly due to the refinement process, which represents the main current limit of the method. Anyhow, we would like to highlight that the FBF strategy allowed us to find a solution for 21 instances not solvable with the monolithic approach, and not reported in the table for the sake of readability.

TABLE I: BOUNDS AND RUNTIMES FOR THE (STANDARD) MONOLITHIC APPROACH

Puzzle Name	Bound	Minisat	Minisat (linked)	Best Solver
$home_{01}$	16	8.0	4.2	3.4
$home_{02}$	25	39.5	24.5	22.5
$home_{04}$	36	94.6	78.9	69.2
$home_{08}$	37	594.6	481.9	363.5
$home_{12}$	41	993.9	738.9	591.6
$push\_push_{01}$	10	0.4	0.2	0.1
$push\_push_{02}$	89	1900.8	999.6	886.6
$push\_push_{04}$	33	4.1	3.8	3.1
$push\_push_{05}$	21	0.6	0.6	0.6
$push\_push_{09}$	34	100.8	71.4	70.7
$push\_push_{10}$	57	862.7	457.9	371.8
$push\_push_{11}$	29	31.1	38.7	26.7
$push\_push_{43}$	35	ovf	1234.8	736.9
$soko_{15}$	48	159.2	151.5	147.6
$soko_{31}$	53	1559.2	1351.5	1247.6

V. CONCLUSIONS

Games are receiving increasing attention from the formal verification community. This paper analyses a very complex single-player game, namely the "Sokoban puzzle". This puzzle is a transport game in which a player has to move boxes from their initial positions into specific final locations. First of all, we concentrate on how to encode the puzzle in a logic (Boolean) form, and on how to solve it

using Bounded Model Checking. Then, we focus on how to efficiently apply two divide-and-conquer (abstraction-and-refinement) strategies to decompose very deep Bounded Model Checking instances into easier sub-problems. Experimental results show that, although SAT tools are becoming more and more efficient, complex games may provide very difficult and hard-to-solve benchmarks to the SAT community.

TABLE II: RESULTS FOR THE MONOLITHIC AND THE FRAME BY FRAME (FBF) DECOMPOSITION APPROACHES

Puzzle Name	**Monolithic Approach**		**FBF Approach**			
	Bound	**Time [s]**	**Abs**	**Con**	**#Ref**	**Time [s]**
$home_{01}$	16	3.4	7	21	0	2.3
$home_{02}$	25	22.5	11	38	0	17.4
$home_{04}$	36	69.2	15	59	1	48.5
$home_{08}$	37	363.5	18	71	2	221.2
$home_{12}$	41	591.6	21	83	3	333.7
$push\_push_{01}$	10	0.1	3	16	7	0.3
$push\_push_{02}$	89	886.6	22	142	5	339.6
$push\_push_{04}$	33	3.1	13	75	2	2.1
$push\_push_{05}$	21	0.6	13	56	3	1.1
$push\_push_{09}$	34	70.7	14	92	6	77.9
$push\_push_{10}$	57	371.8	14	118	6	245.7
$push\_push_{11}$	29	26.7	9	69	5	33.6
$push\_push_{43}$	35	736.9	5	91	4	854.2
$soko_{15}$	48	147.6	21	88	4	131.8
$soko_{31}$	53	1247.6	27	91	4	831.8

REFERENCES

[1] A. Church, “Logic, arithmetic, and automata,” International congress of mathematicians, Djursholm, Sweden, pp. 23-35, August 1963. doi: 10.2307/2270398.

[2] P. J. G. Ramadge and W. M. Wonham, “The control of discrete event systems,” Proceeding of the IEEE, 77(1), pp. 81-98, January 1989. doi: 10.1109/5.21072.

[3] T. Wolfgang, “Infinite games and verification,” in Ed Brinksma and Kim Guldstrand Larsen editors, in Proceedings of the Computer Aided Verification Confernce (CAV), London, UK, Springer, pp. 58-64, September 2002. doi: 10.1007/3-540-45657-0_5.

[4] H. Kautz and B. Selman, “Planning as satisfiability,” in Proceedings of the 10th European Conference on Artificial Intelligence (ECAI’92), Vienna, Austria, John Wiley & Sons Inc. publisher, New York, NY, USA, pp. 359-363, 1992. ISBN: 0-471-93608-1.

[5] W. Thomas, “Infinite games and verification,” in Ed Brinksma and Kim Guldstrand Larsen editors, in Proceedings of the Computer Aided Verification Conference (CAV), volume 2102 of LNCS, Springer-Verlag, Copenhagen, Denmark, pp. 58-64, July 2002. doi: 10.1007/3-540-45657-0.

[6] R. Alur, P. Madhusudan, and W. Nam, “Symbolic computational techniques for solving games,” International Journal on Software Tools for Technology Transfer (STTT), pp. 118-128, January 2005. doi: 10.1016/S1571-0661(05)82544-7.

[7] I. P. Gent and A. G. D. Rowley, “Encoding connect-4 using quantified Boolean formulae,” in Proceedings of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, Kinsale, Ireland, pp. 78-93, 2003. doi: 10.1.1.5.9915.

[8] C. Ansòtegui, C. P. Gomes, and B. Selman, “The Achilles’ heel of QBF,” in Proceedings of the 20th National Conference on Artificial Intelligence (AAAI’05), Pittsburg, Pennsylvania, AAAI Press, pp. 275-281. July 2005. ISBN: 1-57735-236-x. doi: 10.1.1.147.714.

[9] I. Lynce and J. Ouaknine, “Sudoku as a SAT problem,” in Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, January 2006. doi: 10.1.1.331.4584.

[10] A. Sabharwal, C. Ansòtegui, C. P. Gomes, J. W. Hart, and B. Selman, “QBF modeling: exploiting player symmetry for simplicity and efficiency,” in Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT’06), volume 4121 of LNCS, Springer-Verlag, Seattle, Washington, pp. 382-395, August 2006. doi: 10.1007/11814948_35.

[11] J. P. Marques-Silva and K. A. Sakallah, “GRASP - a new search algorithm for satisfiability,” in Proceedings of the International Conference on Computer Aided Design (ICCAD), November 1996. doi: 10.1109/iccad.1996.569607.

[12] H. Jin and F. Somenzi, “CirCUs: a hybrid satisfiability solver,” in Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, Vancouver, BC, Canada, May 2004. doi: 10.1007/11527695_17.

[13] T. Walsh, C. Thiffault, and F. Bacchus, “Solving non-clausal formulas with DPLL search,” in Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, volume 2919 of LNCS, Springer-Verlag, pp. 663-678, 2004. doi: 10.1007/978-3-540-30201-8_48.

[14] L. Zhang, “Solving QBF with combined conjunctive and disjunctive normal form,” in Proceedings of the 21th National Conference on Artificial Intelligence (AAAI’06), Boston, Massachusetts, AAAI Press, pp. 143-149, 2006. ISBN: 978-1-57735-281-5. doi: 10.1.1.527.444.

[15] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “ Symbolic model checking using SAT procedures instead of BDDs,” in Proceedings of the 36th Design Automation Conference, New Orleans, Louisiana, IEEE Computer Society, pp. 317-320, June 1999. doi: 10.1145/309847.309942.

[16] J. Whittemore, J. Kim, and K. Sakallah, “SATIRE: a new incremental satisfiability engine,” in Proceedings of the 38th Design Automation Conference, New York, NY, USA, ACM Press, pp. 542-545, June 2001. ISBN: 1-58113-297-2. doi: 10.1145/378239.379019.

[17] N. Eén and N. Sörensson, “Temporal induction by incremental SAT solving,” in First International Workshop on Bounded Model Checking (BMC’03), Boulder, Colorado, pp. 58-64, July 2003. doi: 10.1016/s1571-0661(05)82542-3.

[18] H. Cho, G. D. Hatchel, E. Macii, B. Plessier, and F. Somenzi, “Algorithms for approximate FSM traversal based on state space decomposition,” IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 15(12), pp. 1465-1478, 1996. doi: 10.1109/43.552080.