Comparing Model Checkers Through Sokoban

Kasper Wijburg
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
k.m.wijburg@student.utwente.nl

ABSTRACT
Model checking is a prominent method for checking software systems for errors. This paper attempts to discern the strengths and weaknesses of the LTSmin, DiVinE and nuXmv model checkers. The model checkers are used to solve Sokoban puzzles. C code, DVE, and nuXmv are used as specification languages for the models of the puzzles. The run time and state space of each model checker and modeling formalism are measured and compared with one another. It is then concluded which modeling formalism is most suited to solve similar issues.

Keywords
Sokoban, Model Checking, Puzzle, LTSmin, DVE, nuXmv, DiVinE

1. INTRODUCTION
As the role of software in society increases, so does the importance of the validation of this software. Errors in software can be very costly: In 2002 the NIST (National Institute of Standards and Technology) reported that software bugs annually cost the U.S. $59.5 billion[10]. Additionally a number of fatal accidents have occurred in the past due to faulty software[13]. As the complexity of systems grows, the complexity of their testing tends to grow even more rapidly; in the worst cases this growth is exponential[11]. It is therefore important that testing is as efficient as possible.

Model checking is a method of testing software. Given the model of a system, a model checker can verify whether or not a certain property is violated. A trace leading to a state that violates this property can then be returned. This paper attempts to discern the optimal approach to testing through model checking. This is done through a case study of three model checkers: DiVinE[1], nuXmv[3], and LTSmin[7]. All three tools can be found on their respective websites1. In this study the model checkers are used to solve Sokoban puzzles. Solving Sokoban puzzles has been proven to be an NP-hard problem[6].

Model specifications of Sokoban puzzles are written in three modeling formalisms. These specifications are then run through the three model checkers. For each puzzle an optimised specification and an unoptimised specification is created. A large number of puzzles (523) is used to gather general statistics. This is combined with the three manually created puzzles which are altered in order to more closely study the effects of different variables on the model checkers. By measuring the run time required and states encountered when solving a puzzle the following questions are answered:

1. On average, which of the three model checkers, DVE, nuXmv, and LTSmin, can solve a Sokoban puzzle within the least amount of time?
   (a) What, if any, changes to a Sokoban puzzle influence the difference in run time required to solve a puzzle?
   (b) Are there aspects that cause a significant difference between the model checkers?

2. Using the LTSmin model checker, do the effects of using breath-first search or depth-first search differ for the two different modeling formalisms?

3. What is the effect of the optimisation of the Sokoban specification on the run time and state space of the model checkers.
   (a) Do these effects differ in strength for the different model checkers?

In this paper we first present some relevant information. The rules of Sokoban and the principle of model checking are explained. We then introduce the model checkers and the three modeling formalisms that are used in the research. After this, the setup on which the model checkers were run is described. The method of running Sokoban puzzles through model checkers is then described. Lastly the results are presented and discussed, followed by a conclusion.

2. BACKGROUND
2.1 Sokoban
Sokoban is a transport puzzle in which the player must push all the boxes to destination locations. The game is played on a board consisting of squares. Each square can either be a wall tile or a walkable tile. A number of these walkable tiles are marked as goal locations. The board contains a number of boxes equal to the number of goal locations (Figure 1). The player may move horizontally or vertically and is confined to the board. The player cannot move through wall tiles. When the player moves into a box, this box is pushed to the square beyond it. This only
A state within a machine representing a Sokoban puzzle is the basic modeling unit in DVE. These models are an abstract description of the system consisting of three layers: the frontend, the PINS layer and the backend. Modeling languages such as DVE are used for LTSmin in this paper, and already has a number of language modules connected, enabling the use of their modeling formalisms. The two modeling formalisms used for LTSmin in this paper are C code through the use of the POSIX/UNIX dlopen API, and DiVinE’s DVE language. The architecture of LTSmin consists of three layers: the frontend, the PINS layer and the backend. Modeling languages such as DVE are processed by the frontend to match the PINS interface. The C code specifications are processed directly by the PINS layers.

2.2 Model Checking
Model checking uses models of a software or hardware system to determine whether it meets a given specification. These models are an abstract description of the system they represent. These models are translated into finite-state machines so that queries can be performed on them. A state within a machine representing a Sokoban puzzle is defined by the location of the player and the boxes. The specification that a Sokoban puzzle can be solved, can be checked by determining whether a state can be reached in which all the boxes are located on goal tiles.

2.3 LTSmin
LTSmin is short for “Minimization and Instantiation of Labelled Transition Systems”. It is a tool-set used to model check state transition systems. It is language independent and already has a number of language modules connected, enabling the use of their modeling formalisms. The two modeling formalisms used for LTSmin in this paper are C code through the use of the POSIX/UNIX dlopen API, and DiVinE’s DVE language. The architecture of LTSmin consists of three layers: the frontend, the PINS layer and the backend. Modeling languages such as DVE are processed by the frontend to match the PINS interface. The C code specifications are processed directly by the PINS layers.

2.4 DiVinE and DVE
DiVinE defines itself as a parallel distributed LTL model checker[1]. The graph traversal algorithms implemented in DiVinE are pseudo breadth-first search. DVE is the specification language created for the DiVinE model checker. The philosophy of the DVE language as specified in the DiVinE manual[9] is as follows:

- The basic modeling unit in DVE is a system that is composed of processes. A process can use transitions to go from one process state to another. These transitions can be guarded by a condition which determines whether the transition can be activated.

Synchronisation of transitions can be achieved through (named) channels. Exactly two processes can be synchronised in a single step on a single channel. This synchronisation on a channel allows for the optional transmission of a value from one process to the other.

Transitions have so-called “effects”. Generally effects are assignments to local or global variables. It should not be possible for two processes undergoing a synchronisation to assign to the same variable.

2.5 nuXmv
nuXmv is a symbolic model checker for finite- and infinite-state synchronous transition systems[3]. It is the evolution of the NuSMV[4] model checker. NuSMV integrates Binary Decision Diagrams (BDD) and satisfiability (SAT) based symbolic model checking algorithms.

In this paper the nuXmv modeling language is used as input for the nuXmv model checker. This language extends the NuSMV language, which is described as follows: The language is designed in a manner that allows the description of finite state machines that range from completely synchronous to completely asynchronous. The language provides for modular hierarchical descriptions and for the definition of reusable components. The basic purpose of the language is to use expressions in propositional calculus to describe the transition relations of a finite Kripke structure.

In addition the nuXmv language extends the NuSMV language with the real and integer types.

2.6 dlopen
The dlopen function is used in UNIX-like operating systems to dynamically load a library.

Dynamic loading allows a program to load a library during run time. This can be particularly useful for the implementation of plugins or modules, permitting to not load the plugin until it’s needed [12].

3. RELATED WORK
To demonstrate the integration of the dlopen API into LTSmin’s language frontends, a Sokoban puzzle was implemented in C code[7]. This implementation proves that it is possible to solve Sokoban puzzles with the LTSmin tool. However this was merely a proof of concept.

Research has been done into solving Sokoban puzzles through the use of NuSMV[8]. The research focusses on optimising the models created from Sokoban puzzles as well as the efficiency of the solution generation by NuSMV. This is done in order to combat the state explosion that the research encountered while attempting to solve certain Sokoban puzzles with NuSMV. In our research we have implemented part of the optimisation referred to as “Abstraction” by Kwon et al.[8] As a box cannot be moved out of a corner, states where a box is located in a non-goal tile corner have been removed from the state space.

Sokoban has been used to compare NuSMV and LTSmin in previous work[2]. It was concluded that NuSMV was significantly faster than LTSmin. However the author admits that NuSMV may have had an unfair advantage due to the models being specifically designed for SMV and then translated to LTSmin’s supported languages. This paper creates both the NuSMV input and LTSmin input directly from a Sokoban input instead of creating one and then translating to another. Note that in our translations we do not optimise the specifications for nuXmv’s internal BDD representation. Additionally, different LTSmin input
Figure 2: An example of the Sokoban puzzle from Figure 1 depicted in ASCII. ‘@’ represents the player, ‘$’ the boxes, ‘.’ the goal locations, and ‘#’ the walls.

languages, DVE and C code, are used compared to the previous work.

4. METHOD

All three model checkers are run within a virtual machine running Ubuntu version 14.04, the virtual machine has 8GB of memory available and a single core\(^2\). The following versions of the model checkers were used: DiVinE 2.4, LTSMinn 2.1 (version tacas2015), and nuXmv 1.1.0.

The Sokoban puzzles used are depicted as a string of ASCII symbols (Figure 2).

This string is used as input for a program which then generates several model specifications for the given Sokoban puzzles; one in DVE, one in nuXmv and one in C code. The board of the puzzle is represented in the specification as an array. The location of the player is stored in two variables representing the player’s x and y coordinates where the top left corner of the board is the point of origin. A pseudo code example for the puzzle shown in Figure 1 can be found in Appendix A. The following rules are specified for transitions:

- The player can walk up, down, left or right if the respective destination is not a box or a wall.
- The player can walk up, down, left or right if the respective destination is a box and the tile behind that box is not a box or a wall. The box is then moved onto that tile.

For the more optimised specifications the following rule is added in addition to the aforementioned rules:

- A box may not be pushed into a tile that is a corner created by walls unless this tile is a goal tile.

This rule was added as boxes cannot be moved out of corners in a Sokoban puzzle. The puzzle is therefore no longer solvable if a box is pushed into a non-goal corner tile. Removing the states where a box is placed into a corner may greatly reduce the amount of possible states for certain Sokoban puzzles.

The C code specifications are run through the LTSMinn model checker. The nuXmv specifications are run through the nuXmv model checker. And the DVE specifications are run through both the DiVinE model checker and the LTSMinn model checker. Additionally both modeling formalisms used for LTSMinn are run through twice, once using breath-first search (BFS) and once using depth-first search (DFS). The nuXmv model checker was run with the dynamic flag as errors would otherwise occur when the system ran out of memory. Excluding this flag for nuXmv, and search strategy specifications for LTSMinn the model checkers were run with their default settings.

First 523 pre-existing Sokoban puzzles are run through the program. These puzzles are used to obtain general statistics for each of the modeling formalisms and model checkers.

Three additional Sokoban puzzles are created to further study the differences between the model checkers. These puzzles are designed to be easily alterable in the following ways:

- The amount of boxes and goal locations are increased or decreased.
- The amount of walkable tiles is increased or decreased.
- The amount of steps required to solve the puzzle is increased or decreased.

Each of these alterations made to one of the puzzles takes into consideration that the puzzle must still be solvable after the change.

The first of these puzzles can be found in Figure 3. The amount of boxes and steps required to solve the puzzle remain the same but the amount of walkable tiles increases with each iteration. This results in an increase of the total state space and may significantly increase the amount of states checked before a solution is found depending on the algorithm used.

The second puzzle can be found in Figure 4. The number of steps required to solve the puzzle increased by ten with each iteration. The amount of boxes and walkable tiles is consistent, causing the total state space to remain the same.

The last puzzle is depicted in Figure 5. For this puzzle the amount of boxes and goal states increases with each iteration. This results in a larger total state space and a larger amount of steps required to solve the puzzle.

Each of the puzzles created by the program is then run through its respective model checkers. Due to the possibility of the state explosion problem occurring a time limit is set before the puzzle is deemed unsolvable by the model checker. For the 523 pre-existing puzzles this time limit is set to one minute. This time limit is set to five minutes for the other three puzzles and their variations.

For each of the puzzles the run time and states found are noted. Additionally each puzzle is run through the model checkers with the goal states disabled in order to observe the total number of states the puzzle has. The gathered data is then analysed and conclusions are drawn from the relations between the run times and the states.

---

\(^2\)The cpu of the host system is an Intel i7-4770k.
Figure 4: The puzzle used to examine the effects of an increasing amount of steps required to solve a puzzle. The first, second, and eleventh puzzle are shown.

Figure 5: The puzzle used to examine the effects of an increasing number of boxes and goal locations. The first, second, and tenth puzzle are shown.

5. RESULTS

5.1 Unoptimised

The results obtained from the 523 pre-existing Sokoban puzzles are displayed in Table 1. The puzzles solved by a modeling formalism were always a subset of the puzzles solved by a modeling formalism with a larger number of solved puzzles. Except for one puzzle which was only solved by the C code model executed with a DFS search.

A comparison of the results obtained from DFS and BFS with the LTSmin model checker using C code models is shown in Figure 6. The Y-axis displays the run time required to solve a puzzle using a DFS algorithm. The X-axis displays the run time required to solve a puzzle using a BFS algorithm.

The run time for each solvable puzzle can be found in the graphs in Appendix B.

5.2 Optimised

The results obtained from the optimised versions of the 523 pre-existing Sokoban puzzles are displayed in Table 2. The puzzles solved by a modeling formalism were almost always a subset of the puzzles solved by a modeling formalism with a larger number of solved puzzles. Six puzzles were solvable by one or more formalisms that had less total solvable puzzles.

On average the state space of an unoptimised solvable puzzle decreased by 80.3% with the optimisation applied.

Figure 7 displays the same comparison as Figure 6 but for the optimised versions of the models.

Graphs depicting the run times of each unoptimised solvable puzzle in relation to its optimised version can be found.
Table 1: Results pre-existing Sokoban puzzles, unoptimised.

<table>
<thead>
<tr>
<th></th>
<th>Solvable puzzles (out of 523)</th>
<th>Average run time of solvable puzzles (sec)</th>
<th>Geometric mean average run time of solvable puzzles (sec)</th>
<th>The most states visited in a solvable puzzle</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiVinE</td>
<td>235</td>
<td>8.95</td>
<td>1.43</td>
<td>12,683,646</td>
</tr>
<tr>
<td>nuXmv</td>
<td>116</td>
<td>13.21</td>
<td>7.91</td>
<td>829,121</td>
</tr>
<tr>
<td>LTSmin-BFS C code</td>
<td>215</td>
<td>8.83</td>
<td>1.20</td>
<td>7,671,749</td>
</tr>
<tr>
<td>LTSmin-DFS C code</td>
<td>217</td>
<td>8.61</td>
<td>1.11</td>
<td>5,120,000</td>
</tr>
<tr>
<td>LTSmin-BFS DVE</td>
<td>236</td>
<td>9.35</td>
<td>1.08</td>
<td>12,496,259</td>
</tr>
<tr>
<td>LTSmin-DFS DVE</td>
<td>239</td>
<td>9.50</td>
<td>1.05</td>
<td>10,150,255</td>
</tr>
</tbody>
</table>

Table 2: Results pre-existing Sokoban puzzles, optimised.

<table>
<thead>
<tr>
<th></th>
<th>Solvable puzzles (out of 523)</th>
<th>Average run time of solvable puzzles (sec)</th>
<th>Geometric mean average run time of solvable puzzles (sec)</th>
<th>Average run time of puzzles that were solvable by unoptimised (sec)</th>
<th>Geometric mean average run time of puzzles that were solvable by unoptimised (sec)</th>
<th>The most states visited in a solvable puzzle</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiVinE</td>
<td>317</td>
<td>6.75</td>
<td>1.23</td>
<td>1.36</td>
<td>0.49</td>
<td>10,260,976</td>
</tr>
<tr>
<td>nuXmv</td>
<td>162</td>
<td>11.59</td>
<td>7.17</td>
<td>5.50</td>
<td>4.32</td>
<td>1,332,880</td>
</tr>
<tr>
<td>LTSmin-BFS C code</td>
<td>297</td>
<td>6.71</td>
<td>0.80</td>
<td>1.45</td>
<td>0.26</td>
<td>4,527,264</td>
</tr>
<tr>
<td>LTSmin-DFS C code</td>
<td>304</td>
<td>6.21</td>
<td>0.66</td>
<td>1.23</td>
<td>0.20</td>
<td>5,210,000</td>
</tr>
<tr>
<td>LTSmin-BFS DVE</td>
<td>318</td>
<td>6.50</td>
<td>0.57</td>
<td>1.08</td>
<td>0.18</td>
<td>9,123,972</td>
</tr>
<tr>
<td>LTSmin-DFS DVE</td>
<td>326</td>
<td>6.27</td>
<td>0.49</td>
<td>0.93</td>
<td>0.14</td>
<td>6,400,000</td>
</tr>
</tbody>
</table>

Figure 8: States versus run time
Figure 9: Steps versus run time

Figure 10: Boxes versus run time

Table 3: Total state space per amount of boxes.

<table>
<thead>
<tr>
<th>Number of Boxes</th>
<th>Total possible states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3906</td>
</tr>
<tr>
<td>2</td>
<td>119129</td>
</tr>
<tr>
<td>3</td>
<td>2382312</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

in Appendix B.

5.3 Altered Puzzles

The results of the puzzles where the number of states are increased but the amount of steps required to solve the puzzle remained the same (Figure 3) can be found in Figure 8. Three separate graphs are used as the scale of the run time greatly varied for several of the modeling formalisms. The results of the puzzles where the steps required to solve the puzzle increased but the total number of states remained the same (Figure 4) can be found in Figure 9. The results from nuXmv were omitted from the graph as nuXmv was only capable of solving the puzzle that only required one step to solve. This was done in 55.0 seconds. Lastly the results of the puzzles where the number of boxes and goal locations increased (Figure 5) can be found in Figure 10. When a line in this graph cuts off it indicates that a higher amount of boxes was not solvable by the puzzle. The total number of possible states for one of these puzzles can be found in Table 3. For puzzles with four or more boxes the total state space could no longer be calculated within the five minute time limit.

6. DISCUSSION

6.1 Unoptimised

DiVinE and LTSmin had very similar results. LTSmin using C code as input solved slightly less puzzles and had a considerable amount less states visited. The nuXmv model checker solves a significantly smaller number of puzzles than the other modeling formalisms. The average run times of the puzzles it does solve are higher than those of the other modeling formalisms. This contradicts the results found by Van Beek[2]. It is likely the nuXmv model implementation used for this paper is not suitable for the nuXmv model checker. As stated in the method section, the dynamic flag had to be used in order to prevent memory related errors. Our implementation saves the entire Sokoban board as an array. A more memory efficient implementation may yield better results for the nuXmv model checker.

When looking at the average run time it appears that after nuXmv, the DVE specification run through the LTSmin model checker is the slowest. However it has the lowest geometric mean average run time. It is likely the average run time is inflated by the higher number of puzzles solved. These extra puzzles are likely barely solvable and therefore have high run times, increasing the average. DiVinE also has a higher puzzles solved count than LTSmin running the C code specification. However both its average run time and geometric mean run time are higher than that of LTSmin. This suggests that DiVinE requires more time to solve simple puzzles than LTSmin.

In general a DFS approach will solve slightly more puzzles than a BFS approach. The graph in Figure 6 depicts this as well; points underneath the blue line are puzzles that were solved more quickly by DFS than by BFS. This can be explained by the DFS algorithm getting lucky and finding the goal state in one of the earlier branches it explores. This can also explain the lower number of states visited by DFS algorithms compared to BFS algorithms.

The one puzzle that was only solvable by LTSmin using C code implementation and a DFS approach can be explained by the DFS algorithm getting lucky. Upon inspection of the results obtained from the altered puzzles it was determined that the unoptimised implementation of puzzles in C code specified the possible transitions in a different order than the unoptimised implementation in DVE. This can explain the puzzle not being solvable by LTSmin using a DFS algorithm with the DVE specification as input.

6.2 Optimised

The optimisation has proven to be effective. The amount
of puzzles that could be solved by each modeling formalism greatly increased. The run time of the puzzles that were already solvable by the unoptimised implementation greatly decreased. The reduction in run time for these puzzles is similar for each of the model checkers indicating that the reduction in state space has a similar effect on each model checker.

The results of DiVinE and LTSmin still closely resemble each other. Except the geometric mean averages; the geometric mean averages of the run times required by DiVinE are more than twice as high as those of LTSmin using a DVE specification. Once again the LTSmin using C code as input solved slightly less puzzles and had a considerable amount less states visited.

The relation between the nuXmv model checker and the other model checkers remains mostly unchanged. As the implementation is similar to that of the unoptimised version this can be attributed to same aspects discussed in the previous section.

DFS algorithms benefited more from the optimisation than BFS algorithms. The DVE specifications run through the LTSmin model checker were faster on average with the use of a DFS algorithm than with the use of a BFS algorithm. The opposite is true for the unoptimised versions. The graph in Figure 7 displays this as well. Compared to the graph in Figure 6 the portion of points below the line has increased. Additionally the points tend to be further below the line. This is due to the DFS algorithm now no longer delving into branches where a box is within a corner. With the unoptimised implementation a DFS algorithm is more likely to visit all states where a box is in a specific corner than a BFS algorithm.

The graphs found in Appendix B depict the relation between the run times of optimised solutions and their unoptimised versions. In general spikes occur at similar puzzles for both versions, with the spikes being smaller for the optimised ones. It can also be noted that the DFS approach and BFS approach have spikes at similar locations. It differs per puzzle if the spike for DFS or for BFS is higher.

Another thing worth noting is the decrease in most states that have been visited. This is unexpected as the optimisation does not increase the speed at which the model checkers can search through states. It only decreases the amount of states each model has.

### 6.3 Altered Puzzles

An increase in states as shown in Figure 3 has a different effect on each model checker. The run time required by the nuXmv model checker greatly increases. This is likely due to it being a symbolic model checker and exploring a large number of states other than just the goal trace. For the DiVinE model checker the run time increases slightly as the number of states increases. For the LTSmin model checker the effect of the increase in states depends on the algorithm that is used. For DVE specifications both BFS and DFS approaches result in a nearly horizontal line with a few outliers. For the C code specification the BFS approach results in a line similar to that of the DVE specifications. For the DFS approach the line increases more rapidly than it did for DiVinE. As previously mentioned the order in which transitions were specified differed for the C code and DVE specifications. This results in the C code implementation exploring a portion of the puzzle before solving it, whereas the DVE specification appears to solve it near instantaneously.

The increase in steps required as shown in Figure 4 has a different effect depending on the search algorithm used. The model checking using BFS have a slight rise as the number of steps required increases, with DiVinE's pseudo BFS having the least sloped line. The model checking using DFS results in nearly horizontal lines, for these puzzles when the algorithm enters the correct branch it nearly instantly completes all the steps. The difference in height between the C code and DVE specifications run through LTSmin can once again be explained by the difference in the order of transition specification. The C code specification appears to first traverse a portion of the puzzle whereas the DVE specification appears to instantly push right. Lastly the nuXmv model checker appears to traverse a large portion of the puzzle before reaching the goal state. The puzzle where only one step is required took nearly a minute longer than any other modeling formalism. And puzzles requiring more than one step were not solvable within the time limit.

An increase in the number of boxes and goal locations greatly increases the number of total possible states a puzzle has. For the puzzles shown in Figure 5 the model checkers were no longer able to traverse the entire state space when the puzzle contained four or more boxes. The C code specification run with a DFS algorithm, DiVinE and nuXmv can no longer solve the puzzle when four or more boxes are present. It is likely they start traversing the puzzle in an incorrect direction and run out of resources before they find a solution. The C code specification run with a BFS algorithm can solve the puzzle update until it contains seven boxes. It is likely it would have been able to solve the puzzle with seven boxes if it were given a few more seconds, as the DVE specification run through LTSmin with a BFS algorithm is barely capable of solving it. The formalism that can solve the puzzle with the most boxes is the DVE specification run through LTSmin with a DFS algorithm. It appears it branches in the correct direction to solve the puzzle, however even with this correct direction it is not capable of solving puzzles with nine or more boxes.

### 7. CONCLUSION

On average the LTSmin model checker is capable of solving a Sokoban puzzle in the least amount of time. It achieved the shortest run times when using a DFS algorithm and a DVE specification.

The effect of alterations to a puzzle, as well as the strength of these effects differ per algorithm used. Additionally the order in which the transitions are specified can greatly effect the run time of a DFS algorithm.

The optimisation of the puzzles greatly increased the amount of puzzles that could be solved and greatly reduced the run time required for puzzles that were already solvable. This effect was of similar strength for each of the model checkers.

### 8. FUTURE WORK

The implementation used in this paper saves the state of the board in an array. This is a relatively memory heavy implementation. Other research can be done with a less memory consuming implementation of the models. For example the locations of the boxes and the player can be stored as coordinates combined with a static board.

The effects of other optimisations to the Sokoban puzzles remain to be studied. For example the disallowing of four boxes forming a square without all being in goal locations.
as this results in a deadlock.

This research was performed using a single core. However the multicore capabilities of the model checkers may vary. A similar research can be done utilising more than one core.

During this research several errors were produced by the model checkers. These errors did not occur when the situation was reproduced on a native linux system. It may therefore be desirable to determine the effects of a virtual machine on the model checkers.

9. REFERENCES


APPENDIX

A. IMPLEMENTATION EXAMPLE

Initialisation:

'0' is a walkable tile, '1' is a wall, and '2' is a box.

\[
\text{board}[] = \\
\{1,1,1,1,1,1,1,\\
1,1,1,0,0,1,\\
1,0,2,0,0,0,1,\\
1,0,2,1,0,0,1,\\
1,0,2,0,0,1,1,\\
1,0,2,1,0,0,1,\\
1,0,0,0,0,0,1,\\
1,1,1,1,1,1,1,\}
\]

\[x = 3\] //Players x coordinate
\[y = 4\] //Players y coordinate

Define:

\[\text{goal} = \forall \text{board}[\text{goal tile}] == 2\]

Transition rules:

//The players current location on the board is equal to: y*width + x

Player movement:

\[\text{if } (\text{board}[y*width + x-1] == 0) \text{ then:}\]
\[x = x-1\]
\[\text{if } (\text{board}[y*width + x+1] == 0) \text{ then:}\]
\[x = x+1\]
\[\text{if } (\text{board}[(y-1)*width + x] == 0) \text{ then:}\]
\[y = y-1\]
\[\text{if } (\text{board}[(y+1)*width + x] == 0) \text{ then:}\]
\[y = y+1\]

Pushing boxes:

\[\text{if } (\text{board}[y*width + x-1] == 2 \text{ and } \text{board}[y*width + x-2] == 0) \text{ then:}\]
\[\text{board}[y*width + x-1] = 0 \text{ and } \text{board}[y*width + x-2] = 2 \text{ and } x = x-1\]
\[\text{if } (\text{board}[y*width + x+1] == 2 \text{ and } \text{board}[y*width + x+2] == 0) \text{ then:}\]
\[\text{board}[y*width + x+1] = 0 \text{ and } \text{board}[y*width + x+2] = 2 \text{ and } x = x+1\]
\[\text{if } (\text{board}[(y-1)*width + x] == 2 \text{ and } \text{board}[(y-2)*width + x] == 0) \text{ then:}\]
\[\text{board}[(y-1)*width + x] = 0 \text{ and } \text{board}[(y-2)*width + x] = 2 \text{ and } y = y-1\]
\[\text{if } (\text{board}[(y+1)*width + x] == 2 \text{ and } \text{board}[(y+2)*width + x] == 0) \text{ then:}\]
\[\text{board}[(y+1)*width + x] = 0 \text{ and } \text{board}[(y+2)*width + x] = 2 \text{ and } y = y+1\]
B. UNOPTIMISED VERSUS OPTIMISED

Figure 11: DiVinE

Figure 12: muXmv

Figure 13: C code BFS

Figure 14: C code DFS

Figure 15: DVE BFS

Figure 16: DVE DFS