Program Transformations Enable Verification Tools to Solve Interactive Fiction Games

Martin Mariusz Lester

Department of Computer Science
University of Reading

m.lester@reading.ac.uk

We present a work-in-progress case study on using program verification tools, specifically model-checkers for C programs, to solve simple interactive fiction games from the early 1980s. Off-the-shelf model-checking tools are unable to handle the games in their original form. In order to work around this, we apply a series of program transformations that do not change the behaviour of the program. An interesting aspect of these games is that they use a simple, interpreted language to script in-game events. This turns out to be the most difficult part of the program for verification tools to handle. Our case study thus provides some insights that may be applicable more generally to verification and analysis of programs that interpret scripting languages.

1 Introduction

Interactive fiction games, also known as text adventures, are an early form of computer game. A game takes the form of a dialogue between the computer and a human player. The player controls a character in a virtual world, often inspired by science-fiction or fantasy literature. The computer describes the player's location. The player gives textual commands to the computer, such as "GET KEY", "GO NORTH" or "OPEN DOOR". The computer executes these commands in the game world and describes their outcome. The sequence repeats until the player wins the game (for example, by finding some treasure) or loses the game (often by dying). We present a case study on using automated program verification tools to solve these games.

In outline, we insert an assertion into the game that is violated on completion of the game, then ask a model-checker to prove that the violated assertion is unreachable. If the game can be completed, the model-checker ought to fail, at which point the counterexample it produces shows how to complete the game. However, despite the age and apparent simplicity of the games we considered, we found that leading program model-checkers were unable to solve even very simple games. To address this, we applied a sequence of program transformations, which did not change the behaviour of the game programs, but enabled the model-checkers to handle them.

We explain our motivation for this case study, including why we believe this is an interesting problem, in Section 2. Because the game engine we consider uses an embedded, interpreted scripting language, we believe some of the techniques may be applicable to other programs with scripted behaviour.

WPTE 2020 informal proceedings

© M M Lester
This work is licensed under the Creative Commons Attribution-No Derivative Works License.

In Section 3, we give some background information on the problem, including some details about the class of games we consider, namely Scott Adams Grand Adventures [1] (SAGA). Of particular relevance is that the games have a large but finite state space.

Next, in Section 4, we explain how we posed solution of a game as a verification problem and how we iteratively transformed the game program to enable its solution using automated verification tools. We focus on application of the established bounded model-checker CBMC [4], but we also considered tools that ranked highly in the recent SV-COMP 2020 Competition on Software Verification [2]. As a result, we were able to solve a small tutorial game using CBMC.

We consider related work, primarily on use of automated tools and techniques on interactive fiction games, in Section 5. This includes our custom tool ScAmPER, which tackles the same class of games using a very fast but stupid heuristic search.

Finally, in Section 6, we conclude with a summary of what we know so far, what we still have to do, and where this will lead. Our hope is that some of the transformations we applied by hand can be fruitfully automated within existing tools.

2 Motivation

The successful application of machine learning to playing old Atari video games [10, 11] has captured the interest of both researchers and the general public. They are an appealing application of research for a number of reasons:

- the public understand what a computer/video game is and what its objective is;
- the player's score is an easily available metric for assessing success automatically;
- older games are simple enough that machine learning techniques can be very successful;
- older games, while still under copyright, are easy to obtain for use in research and provide a wide range of realistic examples for evaluating a new technique.

However, there has been relatively little research of a comparable nature that uses techniques from formal verification. With access to the internal workings of a game, verification techniques ought (in theory) to be able to produce better results, although plays produced by verification tools might seem strange to a human observer, as they could use information about the game state that is invisible to a human player. One of the main problems in verification is state space explosion, but the limited memory of older computer systems mitigates this, meaning that verification techniques are more likely to be effective in this domain than with modern applications software.

Solutions to the problem of using verification techniques to play video games have broader applications for two main reasons. Firstly, solution of games by humans typically involves a mixture of high-level and low-level reasoning. For example, navigating through a maze of rooms involves high-level reasoning about the connections between rooms and low-level reasoning about how to move from one end of a room to another. An effective verification technique for this kind of problem will need either to combine these kinds of reasoning, or to be

sufficiently good at low-level reasoning that high-level reasoning is not needed. The same is true of many problems in program analysis.

Secondly, source code for older games is not usually available, so they must be analysed at the binary level. The binary code may be for a processor for which there are no existing tools, so either a new tool must be made, requiring great expenditure of effort, or an existing tool can be applied to an emulator for the processor, which introduces a further layer of complexity to the verification process. Even then, the game might have been written in BASIC or another interpreted language, resulting in multiple layers of emulation or interpretation to analyse. Yet the same problems arise when trying to analyse a program written in an unusual language or when trying to analyse compiled programs for security purposes; consider, for example, analysis of a program written in OCaml but compiled to JavaScript.

3 Choice of Case Study

We chose to tackle interactive fiction games for two reasons. Firstly, measured by number of player actions, solutions to interactive fiction games are often relatively short. A game might be solvable with a hundred commands or under, compared with thousands of joystick inputs for an arcade game. This limits the depth of search tree that must be considered (although the range of possible commands means the branching factor may be large). Secondly, many interactive fiction games published by the same company consisted of a generic interpreter and a game-specific database. The database describes the contents of the game world and contains script code that is executed when certain events are triggered. Many of the interpreters have now been rewritten in C, allowing the games to be played on modern computers. This meant that we only had to deal with one level of emulation or interpretation, and could use off-the-shelf C verification tools.

The most popular of the generic interactive fiction interpreters was Infocom's Z-Machine. However, its flexibility, which allows games to include their own customised command parsers, makes it a difficult target for verification. Instead, we decided to tackle an earlier and simpler format, namely SAGA games and the corresponding open-source interpreter ScottFree [5]. This format supports only very limited scripting, which makes the behaviour of the games far less dynamic. Pseudocode for a simplified version of the game engine and some examples of scripted events from a specific game are shown in Figure 1.

4 Program Transformation and Verification

We attempted to find a solution to a SAGA game using the bounded model-checker CBMC. We chose CBMC for two reasons. Firstly, it is a relatively mature tool; unusually for a verification tool, it is now included in the stable release of Debian Linux. Secondly, bounded model-checkers tend to be very good at finding counter-examples where a significant amount of low-level reasoning is required, which we judged would be important in this case. We also tested CPAchecker [3], which came second in the ReachSafety category in SV-COMP 2020.

```
// Game engine
while (not game_over) {
    print(current_room.description());
    execute_automatic_scripts();
    (verb, noun) = parse_player_input();
    if (scripted_action(verb, noun)) {
        execute_scripted_action(verb, noun);
    }
    else if (verb == "go") {
        current_room = current_room.exit[noun];
    }
    else if (verb == "get" and items[noun].location == current_room) {
        items[noun].location = carried;
    }
    else if (verb == "drop" and items[noun].location == carried) {
        items[noun].location = current_room;
    }
}

// Example scripted actions
// Action 0:
if (verb == "score") {
    print_score();
}
// Action 1:
else if (verb == "inventory") {
    print_inventory();
}
// Action 2:
else if (verb == "open" and noun == "door" and items["locked door"].location == current_room and
    items["key"].location != carried and items["key"].location != current_room) {
    print("It's locked.");
// Action 3:
else if (verb == "open" and noun == "door" and items["locked door"].location == current_room) {
    swap(items["locked door"].location, items["open door"].location);
}
// Action 4:
else if (verb == "go" and noun == "door" and items["open door"].location == current_room) {
    current_room = rooms["cell"];
}

// Example automatic scripts
// Action 5:
if (items["vampire"].location == current_room and items["cross"].location != carried) {
    print("Vampire bites me! I'm dead!);
    exit();
}
// Action 6:
if (items["vampire"].location == current_room and items["cross"].location == carried) {
    print("Vampire cowers away from the cross!);
}
```

Figure 1: Pseudocode for the structure of the game engine and some scripted events. Scripted events are taken from Tutorial 4 of Mike Taylor's Scott Adams Compiler [16]. The functions invoked by actions 0 and 1, which display the player's score and inventory (list of items carried), are built into the game engine.

CPAchecker can be configured to use a range of techniques, including bounded model-checking and predicate analysis. The tool which came first, VeriAbs, uses a portfolio of other tools, including CBMC, so we did not think it would add any further insights.

Posing the Problem Beginning with the C source code for the ScottFree interpreter, the first step was to insert an assertion stating that the game could not be won. For the purposes of this case study, we focused on games in which the objective is to find a number of pieces of treasure and move them to a fixed "treasure room". The player wins the game by running the "SCORE" command in this location, which causes the game to print a congratulatory message and terminate. We asserted that this message was unreachable.

The interpreter has a main loop that repeatedly describes the player's current location, asks for input and interprets it. We placed a bound on the number of times this loop would run, limiting our search to solutions below a fixed number of moves.

Fixing the Game The interpreter usually reads the game from a text file, which encodes the game as a sequence of integers and fixed strings. CBMC and other model-checkers usually abstract file reads as nondeterminism. In this case, we wanted the file to be fixed to a specific game, so we replaced calls to `fscanf` with calls to a function that returned in sequence the integers and strings from the file. This worked, but unrolling of the loop was still very slow. So instead, we modified the interpreter to dump a C header file immediately after reading the game; when included by the original interpreter, this header file initialised the arrays that held the game database with the relevant constants.

Removing Display Code CBMC was unable to infer loop bounds for much of the code that displayed the description of the player's current location. The code only modifies local variables and makes calls to the terminal library to move the on-screen cursor or print strings. This clearly did not alter the game state, so we removed it.

Bounding Loops with Constants The interpreter includes a number of loops bounded by constants read from the database. For example, the game includes a list of scripted events and the conditions under which they may occur, which is fixed for a particular game. CBMC was unable to determine that the bound was constant, so we augmented our game-specific header file with defined constants for this and other similar values. We replaced references to the variables holding the constant with the defined constant.

Bounding Player Input Commands from the player are read into a buffer before being parsed by the interpreter. The buffer is large (256 bytes) but fixed-size, so each command introduced 2,048 nondeterministic bits. However, SAGA games only permit commands of one or two words (a verb and noun) and only consider the first three characters of each. Thus we could safely replace the input routine with a non-deterministic assignment to the first 7 characters and a terminating zero, reducing this to 56 bits, or 35 bits if we allowed only upper-case characters and space.

This is still quite large and CBMC was unable to infer a bound for unrolling the use of `scanf` to lex the commands. So instead we bypassed lexing entirely and nondeterministically set the verb and noun to an integer index from the game's fixed list of verbs and nouns, reducing the choice of verb and noun to 16 bits of nondeterminism. There was a slight complication in that the code for taking an object uses the name of the noun for the object, not its index, in order to check whether it is in the current room and available to pick up. We resolved this using an assumption that the object picked up matched the object whose noun had been chosen.

Tracking Extra State By this point, CBMC was able to find bounds for all loops in the program and could attempt to unwind it. However, the unrolling was very slow. Two of the culprits were loops that iterated through every object in the game in order to check how many objects the player was carrying and how many treasures the player had found. In order to mitigate this, we introduced integer variables to track these values and update them whenever an object was moved. This increased the state space, but made unrolling much faster. (Pedantically, one would wish to verify that the variables tracked these counts correctly.)

Initial Infeasibility Having made all these changes, we applied CBMC to our modified interpreter and the smallest SAGA game we could find, the Adventure Sampler. It took about an hour to unwind just a single loop of the interpreter, which made it infeasible even to construct a solution that would find one treasure, which takes about ten unwindings and an invocation of a SAT solver on the (presumably gigantic) unwound program. However, we were able to verify that it was possible to leave the starting room of the game. To ensure that our problems were not specific to our choice of tool, we also tried using CPAchecker. It was able to verify that the player could leave the starting room more quickly, but its SMT solver was unable to handle anything larger in under an hour.

Unrolling Interpreter Loop At this point, we wondered if we could construct a smaller game that would be feasible. We discovered that Mike Taylor's Scott Adams Compiler, a tool for creating SAGA games, included a number of very small tutorial games. Tutorial 4 was the smallest game with any puzzles, but that was still infeasible.

The problem appeared to be that unrolling the interpreter loop created a very large program, as CBMC was unable to determine statically which script instruction would be executed on each iteration of the loop. Our solution was to unroll the interpreter loop by hand for every line of scripted code and simplify it by removing the branches that would never be executed. Effectively, we had compiled the script program into C! At this point, setting a limit of 15 commands, CBMC was able to solve the game in under 20 seconds.

Discussion With the exception of replacing player input with nondeterminism, the transformations we applied to make the game amenable to solution with a model-checker could mostly be summarised as follows:

1. Remove code that is redundant because it only displays output.

2. Where loop bounds and other values can be calculated statically, replace them with a constant.
3. Unroll loops with constant bounds.
4. If calculation of a function is costly, but could be done incrementally, add and maintain a variable that caches its value.

These are all variants of classical compiler optimisations, namely dead code elimination, constant folding, loop unrolling and strength reduction. Therefore it should be possible in principle to automate the majority of the transformation, although it may be difficult to identify where and when to apply each step.

Ultimately, CBMC's most significant difficulty was that it could not identify statically which instruction would be executed on each iteration of the interpreter loop. Would a different tool, based on different techniques, fare any better? Probably not. A major difficulty is that execution of each line of script code corresponds to execution of the same block of interpreter loop code. So a tool based on abstract interpretation, for example, would need a highly context-sensitive analysis of the control flow of the program to maintain any reasonable level of precision.

The next step in this project will be to automate the transformations we used in this case study, so that they may be easily applied to other games that use the same engine. We suspect that a partial evaluation tool, such as LLPE [15], may be the best way to approach this. In order for the approach to be applied to other game engines without prior knowledge, we would need a way of identifying the location and structure of an interpreter loop. It may be possible to do this heuristically by using a form of run-time program analysis on plays of the game with random input.

5 Related Work

Narasimhan and others [13] investigated the use of deep learning to solve interactive fiction using only the text output of the game. However, we are more interested in the use of techniques from automated verification.

The idea of applying model-checking to interactive fiction was first investigated by Pickett, Verbrugge and Martineau [14]. They introduced the NFG (Narrative Flow Graph), a formalism based on Petri nets, and the higher-level PNFG (Programmable Narrative Flow Graph). They manually encoded some existing games as PNFGs, used a compiler to translate those into NFGs, then used the model-checker NuSMV to attempt to find solutions. They also discussed other properties of such games that one might check, such as the player being unable to reach a state in which the game had not been lost, yet was no longer winnable. Verbrugge and Zhang later addressed the problem of scalability using a heuristic search [17], which was able to solve several NFGs that were infeasibly large for NuSMV.

From an automated verification perspective, there are two possible criticisms with this approach. Firstly, the manual translation of games into PNFGs might not be entirely accurate. It is quite possible that some subtlety in the way certain actions are handled has not been translated accurately. Secondly, it is often the case that expressing a problem in a different format makes

it easier for automated tools to solve, as it reveals the high-level structure of the problem. If the translation is done manually, the translator may have introduced this structure, and it is not clear that it could be derived automatically.

Model-checking techniques have been applied to a range of other computer puzzle games. Kwon [8] considers encoding and solving Sokoban puzzles using NuSMV. Like interactive fiction games, Sokoban involves moving a player and discrete objects around discrete locations on a map, so it also suffers from state space explosion. Moreno-Ger and others [12] consider encoding point-and-click adventures written for the engine eAdventure and using NuSMV to check that they have no dead ends. These games do not suffer from state space explosion to the same extent, as objects cannot usually be dropped after they are picked up, so they are either in their initial location, held by the player, or no longer in the game. Like interactive fiction games, they usually feature a limited form of scripting. In both of these cases, the translation from the puzzle or game instance to NuSMV is automated, but the translation was designed manually. To our knowledge, there is no previous work that takes the code of a game engine as its starting point. Teeny Tiny Mansion [18] is a toy example of an interactive fiction game whose source code can be verified to have no dead ends using CBMC, but it was written specifically for this purpose.

We recently developed a tool called ScAmPER [9] that uses a heuristic-guided, explicit-state search to analyse SAGA games. The tool outputs a suite of game inputs that serve as test cases to maximise coverage of lines of scripted code within a game. It is able to achieve a high level of coverage fairly quickly, but is unable to prove when a line of code is unreachable.

Partial evaluation of programs has been studied extensively as a method of optimisation. For example, the literature on partial evaluation suggests applying a partial evaluator to the combination of an interpreter and a program in an interpreted language as one way of producing an optimising compiler [7]. It has been suggested as a way of improving the effectiveness of verification techniques [6], but does not seem to have gained much prominence.

6 Future Work and Conclusions

Using program transformation, we were able to solve a small interactive fiction game using a model-checker for C programs. As the final step of transformation, namely unrolling the interpreter loop, was done by hand and specific to the game under investigation, we have not yet been able to test our approach on larger games. Automating the final unrolling will enable us to compare this technique against our previous work; this will be our next step. We expect and hope that the combination of clever, general automated program verification tools and domain-specific knowledge applied by a human expert will ultimately outperform a stupid but specific tool, at least in terms of what they can prove, if not in terms of speed.

Looking beyond this case study, the next logical step would be to attempt to automate the entire sequence of transformations within an existing verification tool or framework. The main technical difficulty here would likely be how to identify the structure of an interpreter loop, so that the transformations could be applied in the right place.

Verification of old computer and video games is an appealing application of formal veri-

fication with relevance to current problems. We hope that our case study will motivate future developments in verification tools.

References

[1] Scott Adams: *Scott Adams Grand Adventures*. Available at `http://www.msadams.com/downloads.htm`.

[2] Dirk Beyer (2020): *Advances in Automatic Software Verification: SV-COMP 2020*. In Armin Biere & David Parker, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, *Lecture Notes in Computer Science* 12079, Springer, pp. 347–367, doi:10.1007/978-3-030-45237-7_21. Available at `https://doi.org/10.1007/978-3-030-45237-7_21`.

[3] Dirk Beyer & M. Erkan Keremoglu (2011): *CPAchecker: A Tool for Configurable Software Verification*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, *Lecture Notes in Computer Science* 6806, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16. Available at `https://doi.org/10.1007/978-3-642-22110-1_16`.

[4] Edmund M. Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In Kurt Jensen & Andreas Podelski, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, *Lecture Notes in Computer Science* 2988, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15. Available at `https://doi.org/10.1007/978-3-540-24730-2_15`.

[5] Alan Cox: *ScottFree interpreter*. Available at `https://www.ifarchive.org/indexes/if-archiveXscott-adamsXinterpretersXscottfree.html`.

[6] Matthew B. Dwyer, John Hatcliff & Muhammad Nanda (1998): *Using Partial Evaluation to Enable Verification of Concurrent Software*. *ACM Comput. Surv.* 30(3es), p. 22, doi:10.1145/289121.289143. Available at `https://doi.org/10.1145/289121.289143`.

[7] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science, Prentice Hall.

[8] Gihwon Kwon & Taehoon Lee (2004): *Solving Box-Pushing Games via Model Checking with Optimizations*. In Farn Wang, editor: *Automated Technology for Verification and Analysis: Second International Conference, ATVA 2004, Taipei, Taiwan, ROC, October 31-November 3, 2004. Proceedings*, *Lecture Notes in Computer Science* 3299, Springer, pp. 491–494, doi:10.1007/978-3-540-30476-0_43. Available at `https://doi.org/10.1007/978-3-540-30476-0_43`.

[9] Martin Lester (In press): *ScAmPER: Generating Test Suites to Maximise Code Coverage in Interactive Fiction Games*. In: *Tests and Proofs 2020*. Available at `http://lal.mariusz.co.uk/scamper/`.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra & Martin A. Riedmiller (2013): *Playing Atari with Deep Reinforcement Learning*. *CoRR* abs/1312.5602. Available at `http://arxiv.org/abs/1312.5602`.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski et al. (2015): *Human-level control through deep reinforcement learning*. Nature 518(7540), p. 529.

[12] Pablo Moreno-Ger, Rubén Fuentes-Fernández, José Luis Sierra-Rodríguez & Baltasar Fernández-Manjón (2009): *Model-checking for adventure videogames*. *Inf. Softw. Technol.* 51(3), pp. 564–580, doi:10.1016/j.infsof.2008.08.003. Available at `https://doi.org/10.1016/j.infsof.2008.08.003`.

[13] Karthik Narasimhan, Tejas D. Kulkarni & Regina Barzilay (2015): *Language Understanding for Text-based Games using Deep Reinforcement Learning*. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pp. 1–11. Available at `http://aclweb.org/anthology/D/D15/D15-1001.pdf`.

[14] Christopher JF Pickett, Clark Verbrugge & Felix Martineau (2005): *NFG: A language and runtime system for structured computer narratives*. In: *Proceedings of the 1st Annual North American Game-On Conference (GameOn'NA 2005)*, pp. 23–32.

[15] Christopher Smowton (2015): *I/O optimisation and elimination via partial evaluation*. Ph.D. thesis, University of Cambridge, UK. Available at `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708547`.

[16] Mike Taylor: *Scott Adams Compiler (sac)*. Available at `http://www.miketaylor.org.uk/tech/advent/sac/`.

[17] Clark Verbrugge & Peng Zhang (2010): *Analyzing Computer Game Narratives*. In: *Entertainment Computing - ICEC 2010, 9th International Conference, ICEC 2010, Seoul, Korea, September 8-11, 2010. Proceedings*, pp. 224–231, doi:10.1007/978-3-642-15399-0_21. Available at `https://doi.org/10.1007/978-3-642-15399-0_21`.

[18] Claire Xen: *Teeny Tiny Mansion (TTTM)*. Available at `http://svn.clifford.at/handicraft/2017/tttm/README`.