ScAmPER: generating test suites to maximise code coverage in interactive fiction games

ScAmPER: Generating Test Suites to Maximise Code Coverage in Interactive Fiction Games

Martin Mariusz Lester[0000−0002−2323−1771]

University of Reading, Reading, United Kingdom
m.lester@reading.ac.uk

Abstract. We present ScAmPER, a tool that generates test suites that maximise coverage for a class of interactive fiction computer games from the early 1980s. These games customise a base game engine with scripts written in a simple language. The tool uses a heuristic-guided search to evaluate whether these lines of code can in fact be executed during gameplay and, if so, outputs a sequence of game inputs that achieves this. Equivalently, the tool can be seen as attempting to generate a set of test cases that maximises coverage of the scripted code. The tool also generates a visualisation of the search process.

Keywords: reachability · coverage · explicit state · interactive fiction

1 Introduction

A common complaint concerning tools in automated verification is that they are inadequate for handling the complex software of today, written in modern programming languages. What about the software of yesterday?

Interactive fiction or *text adventure* games are a genre of computer game that peaked in popularity in the 1980s, although a small but active community continues to create and play new games. The games take the form of a textual dialogue between a player, who gives commands, and the computer, which executes the commands and describes their effect in a game world.

These games are conceptually easy to understand, but making progress within them often involves a mixture of high-level planning (such as deciding in which order to solve in-game puzzles) and low-level execution (such as moving objects to specific locations). For this reason, they present an appealing case study for the application of automated verification tools.

An occasional complaint concerning research in automated verification is that tools are sometimes benchmarked either against large, inaccessible, incomprehensible pieces of commercial software, or against unrealistic toy examples that have been chosen to showcase the strengths of a tool. Using older software as a benchmark addresses some of these concerns, as while it is still covered by copyright, it is often freely available and small enough to be comprehensible, yet still realistic.

In terms of implementation detail, two aspects of interactive fiction games make them relevant to modern software. Firstly, the structure of an interpreter

loop, which is used to execute scripted actions in many games, is commonly used in software that features scripting. Secondly, the pattern of repeatedly reading user input and responding by updating internal state or occasionally triggering events is also used in GUI applications and some control software.

We investigated whether tools from automated verification and formal methods could be used to explore, test and solve interactive fiction games, considering specifically the Scott Adams Grand Adventures (SAGA) series. When we discovered that they were unsuccessful, we wrote a specialised tool to tackle this problem.

Our tool, ScAmPER (Scott Adams exPlicitly Evaluating Reachability), is an explicit-state on-the-fly model checker that uses a heuristic-guided search to determine whether certain lines of script code within these games are reachable. When it finds that they are, it outputs a sequence of game inputs to witness this. Equivalently, this sequence of game inputs is a test for the reached line of script code. Taken together, these tests form a suite that aims to maximise both branch coverage and modified condition/decision coverage (MC/DC) of the script code. In addition, the tool generates a visualisation of the search process. The tool is available online [9,3].

Appendix A gives some background on SAGA games. Section 2 outlines our initial attempts to analyse them with existing tools. Section 3 demonstrates how our tool works, describes how it works and evaluates its performance. We consider related work on interactive fiction games and more generally in Section 4, concluding with lessons learned and suggestions for future work in Section 5.

2 Preliminary Investigation

We thought it was plausible to use automated tools on 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 data file. Many of these interpreters have now been rewritten in C, allowing the games to be played on modern computers. This meant we could use off-the-shelf tools for C programs.

We initially attempted to use off-the-shelf tools for C programs to determine reachability of scripted events within a game. Bounding loops with constants allowed bounded model checkers such as CBMC to unroll them. We replaced the input of commands with a nondeterministic choice of a sequence of verb/noun pairs and tested reachability by adding an assertion that a line of script would not be executed. Then, a counterexample trace would give the inputs necessary to reach the assertion. However, neither of the tools we tested could produce plays of the game longer than a single command. We concluded that this was because a single command involved executing every line of script, each of which executed

the script interpreter, meaning that the unrolled straight-line code could be tens of thousands of lines per game command.

We also tried using the fuzzing tool American Fuzzy Lop (AFL). While its genetic algorithms generated plays of reasonable length, most of the commands in them did not progress through the game. Movement commands or commands to pick up or drop objects usually fail if their sequence is changed, so combining two interesting plays is unlikely to produce an interesting new play. AFL instruments the code it is fuzzing to identify when it has found a new path of execution. However, in an interpreter loop, almost all executions will pass through the same lines of source code, just with different data, making this technique ineffective.

3 ScAmPER

Usage. ScAmPER takes as input the database that defines a game. It uses a heuristic search to find sequences of commands that reach *novel* states. *Novel* states are those in which, for the first time during play, the player enters a room, triggers execution of a line of script (*action*), or sees a message. States where the conditions in the guard of an action take on a new permutation of truth values, thereby increasing MC/DC coverage, are also novel. The tool outputs the commands needed to reach these novel states and a visualisation of the search process. ScAmPER stops when it has explored an initial pre-defined number of states *and* has not recently encountered any novel states. It prints statistics showing what percentage of rooms, actions and messages it could reach, as well as MC/DC coverage.

Fig. 1 shows a still from a visualisation, which is rendered using GraphViz's `dot` and animated using `gifsicle`. The main part of the image shows the game map, with rooms that have been reached filled in. The two grids show the IDs of the game's scripted actions and messages. Again, those that have been reached are filled in. Next to the still is an example of a sequence of input commands needed to trigger a scripted action.

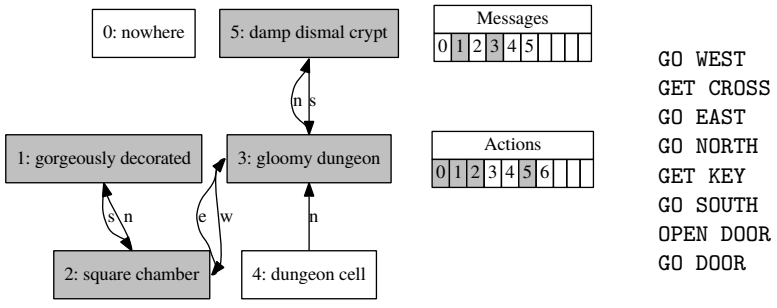

Fig. 1. Left: Still image from visualisation of Tutorial 4 of Scott Adams Compiler [14]. Right: A sample test input trace generated by ScAmPER to trigger action 4.

Game	Rooms reached	Messages shown	Actions reached	MC/DC	Time (s)	Mem (MB)
1	88 % / 34	80 % / 77	75 % / 170	76 %	209	431
2	77 % / 27	75 % / 90	66 % / 178	64 %	25	37
3	54 % / 24	65 % / 82	54 % / 162	64 %	18	32
4	92 % / 26	80 % / 100	83 % / 190	83 %	96	105
5	86 % / 23	89 % / 89	84 % / 220	78 %	39	51

Fig. 2. Benchmarks of ScAmPER on 5 SAGA games.

Benchmarks. We tested ScAmPER on the first 5 original SAGA games [1]. Results are shown in Fig. 2. By various metrics, coverage is typically 60–90%. Timings are for an Intel Core i7-7500U at 2.70 GHz with 16 GB of RAM.

Architecture. Fig. 3 shows the architecture of ScAmPER. In order to ensure that it accurately models the behaviour of a game, it makes heavy use of the existing ScottFree interpreter [2]. The first component of ScAmPER is a modified version of the interpreter that, rather than running a game, loads its database, then dumps it as a series of variable definitions in a C header file. The second component, which actually performs the search, links against a modified version of the interpreter that includes the header file, effectively hard-coding the game's database.

The second modified version of the interpreter is heavily optimised for use in the search. All startup code and any code handling message display has been removed. Code to get user input and parse it has also been removed. Instead, a "next state" function has been added that takes as arguments the game state and the numeric IDs of a verb and a noun; these are passed directly to the code that evaluates the user's commands. In order to support this, the game's mutable data has been separated from its immutable data and bundled in a single, fixed-size C struct. As the game's database is hard-coded, variables referring to the sizes

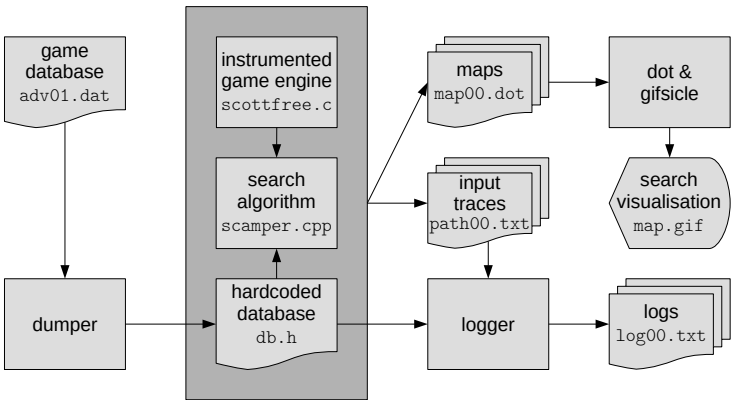

Fig. 3. The architecture of ScAmPER.

of arrays of rooms, items and so on have been replaced with defined constants, enabling more optimisation by the compiler.

Search algorithm. Before the search starts, the tool uses a simple static analysis to determine all possible commands accepted by the game. A few commands are hard-coded into the game engine. These are navigation commands ("GO X", for any direction X) and commands to pick up or drop an item ("GET X" or "DROP X", for any movable item X). The possible directions are hardcoded into the game engine and the movable items are marked with a flag in the game database, so these are easy to determine. All other valid commands are handled by scripted actions. These are also easy to determine, as the verb/noun combination that triggers an action must be stored in a specific field in the game database.

The search algorithm itself is relatively straightforward. A game tree is initialised with a node containing the starting state of the game. On each iteration of the search, a state is picked (according to heuristics) to expand as follows. Firstly, a breadth-first search using navigation commands finds all rooms accessible using "GO X" only. From each of these rooms, the search tries: picking up or dropping any item ("GET X" or "DROP X"); and any combination of verb and noun that is listed as a trigger in the game's scripts. This is sufficient to solve a large number of in-game puzzles, many of which involve taking an item to a certain location and entering a specific command.

Rather than attempting to determine in advance whether these commands will be fruitful, ScAmPER simply tries them all, as that is likely to be faster. Most of the commands will fail with an error, for example if the command was "GO NORTH", but there was no exit to the north, or the command was "GET KEY", but the key was not in the player's current room. In order to detect this, the interpreter has been instrumented with a flag that is set when a command fails. If a command fails, the resulting state is discarded. Otherwise, it is added to the game tree. If the command triggered a previously unseen message or unexecuted line of script, this is recorded. Again, the interpreter is instrumented with flags to detect this. The goal of this expansion strategy is to find all currently accessible scripted actions, without wasting time moving back and forth aimlessly between locations in the game's map.

The interpreter is also instrumented to evaluate and record the value of all conditions in the guards of scripted actions after every move. This is necessary in order to determine MC/DC coverage, as if any condition in a guard is false, the subsequent conditions would not normally be evaluated.

Explored states are stored in a hash table for easy lookup. If a duplicate state is ever encountered, it is immediately discarded. This cuts out many pointless sequences of commands, such as returning to the room from which the player just came, or dropping an object and immediately picking it up again.

A small number of scripted actions occur with random probability. For the sake of reproducibility, we decided to use a simplified random number generator and store its state as part of the game state. However, we ignore it when hashing states or comparing them for equality in order to avoid filling the game tree with

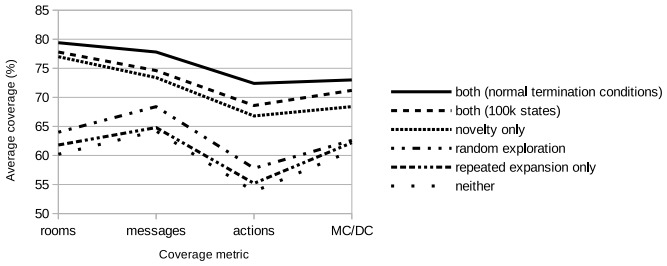

Fig. 4. Average coverage over first 5 SAGA games with different search heuristics.

many copies of the same state that differ only by random number generator state. Similarly, we also ignore the state of the counter that records how much longer a light source will last for. These simplifications aside, the search process is complete, in the sense that it will eventually explore all possible states of a game, although in practice this only happens within reasonable time/memory limits for extremely small examples.

ScAmPER uses two main heuristics to pick the node to expand. Firstly, novel states (as described above) are preferred. Secondly, new states encountered during expansion are themselves immediately expanded, as are those encountered during this second round of expansion; this ensures that paths from any state chosen for expansion are explored to a reasonable depth. The rationale is that games usually feature a number of dependent puzzles. Completion of one puzzle grants access to a new item or location, or sets a flag, which enables other puzzles to be completed. In order to complete the later puzzles (and execute their associated lines of script), the search must focus on states where the earlier puzzles have already been solved; these are more likely to be the novel states and deeper in the game tree.

We evaluated both heuristics on the first 5 SAGA games, stopping the search after 100,000 states had been explored. Our results are shown in Fig. 4. Both heuristics improved coverage, but novelty was far more significant. For comparison, we also evaluated a search strategy that randomly picks a state to explore and tries all available commands when it does so. This performed acceptably and was comparable to our expansion strategy without the novelty heuristic.

4 Related Work

Machine learning for playing video games. 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. There are several key differences in the problem we chose to tackle. Firstly, in many interactive fiction games, there is no equivalent of the player's score for assessing the success or progress. Secondly, we allow tools to access the internal state and code of the program we are

analysing. Thirdly, the tree of game states has a higher degree of branching (as there are more possible actions at any point), but successful or interesting paths are shorter while potentially more intricate.

Narasimhan and others [12] investigated the use of deep learning to solve interactive fiction using only the text output of the game. Its success was limited in games without a score to guide play.

Model checking for solving interactive fiction games. The idea of applying model-checking to interactive fiction was first investigated by Pickett, Verbrugge and Martineau [13]. 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 [15], which was able to solve several NFGs that were infeasibly large for NuSMV.

Our main criticisms of this work concern the choice to re-encode the games as PNFGs. Firstly, the manual translation of games into PNFGs might not be entirely accurate. Looking at the C code for the script interpreter in ScottFree, the behaviour of some scripted events is quite subtle, and it is likely that a manual translation would not capture all the details correctly. 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.

Automated test case generation. A key problem with automated test generation is generating tests that cover difficult-to-reach parts of a system, in particular when tests are generated randomly. Testar [16], a tool for automated generation of GUI test cases, addresses this using Q-learning. The tool identifies possible actions in a GUI application using its accessibility API. It runs the program being tested and uses instrumented versions of GUI libraries to gather information about the state of GUI widgets; it uses this to determine whether an action caused the application to enter a new state. Q-learning is used to tune the frequency with which different GUI actions are taken in order to maximise the chances of reaching a new state. In contrast, as ScAmPER has access to the internal state of the game and is not limited to constructing whole paths of execution, tuning the frequency with which different actions are taken is less important. All actions from a state of interest can be taken with ease. Nonetheless, this approach could lead to better heuristics for choosing which state to expand.

Fuchs presents a tool [5] for generating test cases for Web applications written in Java. These applications provide a user interface to a database, which is built by combining and customising ready-made components. In that regard, they are somewhat similar to SAGA games. The tool constructs tests using simulated

interactions with the application, making use of the compiled application code. Their tool is augmented with a symbolic execution engine.

Julliand and others [8] consider test generation for abstracted event systems. Their approach, called concrete exploration, is based on models written in B, but applicable to other settings. The games we consider could be viewed as event systems, with the obvious abstraction being to group concrete game states in which the player is in the same room; the scripted actions in a game would correspond to guarded actions in the event system. Their work proposes a strategy for covering the abstract transitions of a system by trying to extend existing concrete paths to cover new abstract transitions.

5 Future Work and Conclusions

Despite their relative simplicity, SAGA games were extremely popular and ported to at least 10 different home computer systems. After source code for two games and their interpreter were published in magazines in 1980, the style of the interpreter was copied by other developers. This led to the creation of game authoring packages such as The Quill and Graphic Adventure Creator. Combined, over 500 games were published using these packages. We are highly confident that our techniques are applicable to these engines too, but we leave that for future work. We suspect that adapting our work to more complex and dynamic game engines, such as Infocom's Z-Machine, would prove much more difficult.

Old computer and video games present an appealing challenge for program analysis and automated verification, with relevance to current problems, such as GUI test case generation. Explicit state methods have proven to be successful in model-checkers such as SPIN [7] and FDR [6]. ScAmPER shows once again that explicit state methods can beat symbolic methods when the "next state" function of a system is complex, but can be evaluated cheaply.

One blind spot of ScAmPER is that it only finds when an action can be executed; it never proves that it cannot. Our tool could be improved with better heuristics, by using techniques from model-checking such as CEGAR, or by using symbolic execution to gather constraints and help to guide the search. But we are more interested in how to encode the problem so that it can be handled better by existing tools, or in finding improvements for those tools.

Off-the-shelf verification tools cannot yet handle games from 40 years ago without help, and even then they struggle. ScAmPER shows that a custom tool can tackle these examples. It is unsurprising that a specialised tool should outperform a generic tool on a particular problem. However, it is not clear whether a clever generic tool or a stupid specialised tool, such as ours, is likely to perform better. Our work provides one datapoint to suggest that, for the moment, a stupid specialised tool is better. We hope that it will motivate future developments in verification tools and that its performance on SAGA games will provide a benchmark for others to beat.

A Scott Adams Grand Adventures

In an interactive fiction game, the computer displays a textual description of the player's location. The player types a command, such as "GO NORTH", "GET KEY" or "OPEN DOOR". The computer parses the command, evaluates its effect on the game world, and displays a textual description of any outcomes. The process repeats until the player wins (for example, by defeating a monster or finding a treasure) or loses (often by dying).

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 automated analysis. Instead, we decided to tackle an earlier and simpler format, namely Scott Adams Grand Adventures (SAGA) and the corresponding open-source interpreter ScottFree [2]. This format supports only very limited scripting, which makes the behaviour of the games far less dynamic.

Games consist of a fixed number of rooms and objects (usually 20–40 each). Commands consist of one or two words (a verb and a noun), which are taken from a fixed list. The player can move between rooms and pick up and drop objects; this is hard-coded into the engine. A limited scripting system allows the behaviour of each game to be customised. Scripts allow the player to trigger special events by entering certain rooms or typing certain commands, provided a guard consisting of a conjunction (AND) of conditions is satisfied. Examples of conditions that can be checked include the location of a certain object or the value of a finite number of flags and bounded counters. Examples of events include moving objects, moving the player, displaying messages and adjusting the values of flags and counters.

Fig. 6 shows pseudocode illustrating the structure of the game engine and some possible scripted events. Despite the relatively simple structure of the engine, the open source implementation ScottFree is around 1,500 lines of C, of which around 600 implement an interpreter for the scripting language. Scripts can check around 20 different kinds of condition and trigger around 40 different kinds of event. The state space of a game is finite but too large to enumerate. Its size is dominated by the potential for any movable object to be in any room, ranging from roughly 2^{40} combinations in adventure 11 to 2^{114} in adventure 9.

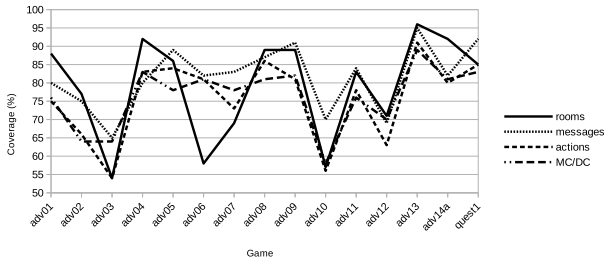

Fig. 5. Coverage achieved by ScAmPER with different metrics on SAGA games.

```
// 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!);
}
```

Fig. 6. 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 [14]. 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.

References

1. Adams, S.: Scott adams grand adventures, http://www.msadams.com/downloads.htm
2. Cox, A.: Scottfree interpreter, https://www.ifarchive.org/indexes/if-archiveXscott-adamsXinterpretersXscottfree.html
3. Dietsch, D., Jakobs, M.C.: Tap 2020 virtual machine (Apr 2020), https://doi.org/10.5281/zenodo.3751284
4. Dubois, C., Wolff, B. (eds.): Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings, Lecture Notes in Computer Science, vol. 10889. Springer (2018), https://doi.org/10.1007/978-3-319-92994-1
5. Fuchs, A.: Automated test case generation for java EE based web applications. In: Dubois and Wolff [4], pp. 167–176, https://doi.org/10.1007/978-3-319-92994-1_10
6. Gibson-Robinson, T., Armstrong, P.J., Boulgakov, A., Roscoe, A.W.: FDR3 - A modern refinement checker for CSP. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014. Proceedings. pp. 187–201 (2014), https://doi.org/10.1007/978-3-642-54862-8_13
7. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
8. Julliand, J., Kouchnarenko, O., Masson, P., Voiron, G.: Under-approximation generation driven by relevance predicates and variants. In: Dubois and Wolff [4], pp. 63–82, https://doi.org/10.1007/978-3-319-92994-1_4
9. Lester, M.M.: ScAmPER: Scott Adams exPlicitly Evaluating Reachability (Mar 2020), https://doi.org/10.5281/zenodo.3724977
10. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing atari with deep reinforcement learning. CoRR **abs/1312.5602** (2013), http://arxiv.org/abs/1312.5602
11. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529 (2015)
12. Narasimhan, K., Kulkarni, T.D., Barzilay, R.: 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 (2015), http://aclweb.org/anthology/D/D15/D15-1001.pdf
13. Pickett, C.J., Verbrugge, C., Martineau, F.: 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 (2005)
14. Taylor, M.: Scott adams compiler (sac), http://www.miketaylor.org.uk/tech/advent/sac/
15. Verbrugge, C., Zhang, P.: Analyzing computer game narratives. In: Entertainment Computing - ICEC 2010, 9th International Conference, ICEC 2010, Seoul, Korea, September 8-11, 2010. Proceedings. pp. 224–231 (2010), https://doi.org/10.1007/978-3-642-15399-0_21
16. Vos, T.E.J., Kruse, P.M., Condori-Fernández, N., Bauersfeld, S., Wegener, J.: TESTAR: tool support for test automation at the user interface level. IJISMD **6**(3), 46–83 (2015), https://doi.org/10.4018/IJISMD.2015070103