Proceedings Paper:
Arnold, James and Alexander, Rob orcid.org/0000-0003-3818-0310 (2013) Testing Autonomous Robot Control Software Using Procedural Content Generation. In: Computer Safety, Reliability, and Security. 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), 24-27 Sep 2013 Lecture Notes in Computer Science . Springer , pp. 33-44.

https://doi.org/10.1007/978-3-642-40793-2_4

Testing Autonomous Robot Control Software using Procedural Content Generation

James Arnold[1], Rob Alexander[2]

[1]BAE Systems Detica, UK
james.arnold@detica.com
[2]Department of Computer Science, University of York, UK
rob.alexander@york.ac.uk

Abstract. We present a novel approach for reducing manual effort when testing autonomous robot control algorithms. We use procedural content generation, as developed for the film and video game industries, to create a diverse range of test situations. We execute these in the Player/Stage robot simulator and automatically rate them for their safety significance using an event-based scoring system. Situations exhibiting dangerous behaviour will score highly, and are thus flagged for the attention of a safety engineer. This process removes the time-consuming tasks of hand-crafting and monitoring situations while testing an autonomous robot control algorithm. We present a case study of the proposed approach – we generated 500 randomised situations, and our prototype tool simulated and rated them. We have analysed the three highest rated situations in depth, and this analysis revealed weaknesses in the smoothed nearness-diagram control algorithm.

Keywords: autonomy, robots, faults, simulation, procedural content generation

1 Introduction

There is much work afoot to bring autonomous robots (AR) into public spaces, both on the ground and in the air. We therefore need high confidence that their behaviour will be safe. This is difficult, however, given the complexity of environments that the robots must interact with, and the scope of authority that they need in order to effectively respond to those environments. There have already been minor accidents caused by the control software of autonomous vehicles, for example in the DARPA Urban Challenge [1], and there are likely to be more as their numbers increase. We need ways to verify and validate such control software, finding ways in which it could cause an accident, whether due to intrinsic algorithmic limitations or implementation-specific faults.

One approach would be testing AR by putting them through linear scenarios, represented as a sequence of stimuli, and checking the safety of the resulting behaviour. This is inadequate, however: AR will influence their environment, and that will influence their own future behaviour. This is true, of course, for any reactive system, but AR may have many ways to influence their environment (e.g. movement, manipulator

arms, communication) and many ways to respond to those changes (through sensing, model-building and prediction). We thus need to understand how the behaviour of the AR will develop from a given starting point. We therefore need to test AR by generating *situations* – combinations of maps, peer entities, and missions or objectives, into which a (simulated) AR can be placed. Rather than providing a fixed sequence of stimuli, simulated situations can react to the AR's actions. Given such a situation, we can assess whether the AR achieves safe behaviour.

For example, one situation might have an autonomous ground vehicle driving down an empty motorway to deliver a package. Another might modify the mission to have a tight time limit and stationary traffic jam. A third situation might instead pepper the motorway with abandoned cars and spilt fuel. Each of these would test different aspects of the vehicle's control software.

We can observe that a situation-based approach can provide a degree of both verification (of the implementation against a vehicle behaviour specification) and validation (of the behaviour specification against the behaviour that we, in retrospect, want from it, now that we've seen the consequences).

There are two major obstacles for situation-based testing. First, useful situation-based testing will require a wide variety of situations, and creating those in a simulation requires considerable effort – for example, engineers need to list the precise position, heading and plan of each vehicle involved in each situation. This involves a lot of work, perhaps engineer-weeks for a complex situation, and this thus limits the range of situations that can be studied. Martin and Hughes report similar problems with scenarios for training simulations [2].

Second, the *diversity* of generated situations is important. In particular, we do not want biases in the generation of situations that mirror the biases of the designers of the control algorithms; we do not want our test set to have the same 'blind spots' as the software. Research on n-version programming suggests that this is likely – in Knight and Leveson's classic experiment [3], different programmers working from the same specification produced programs that had largely similar errors. It may be that different situation designers produce situation sets which miss similar challenges.

A response to these two problems is to generate situations automatically. This is the response taken in the simulation-for-training field, where it is called Automatic Situation Generation (ASG) e.g. Martin and Hughes [2] and in the video games field, where it is called Procedural Content Generation (PCG) e.g. Togelius et al [4, 5]. PCG is often used in situations where the manual creation of content would be prohibitively time-consuming or expensive [6]. It also allows highly detailed content to be stored extremely efficiently—only the parameter values input into the algorithms need to be stored. PCG has the potential to produce output that surprises the PCG engine developers; this has often been observed in video game uses [7].

In this paper we show how PCG techniques can be adapted to test AR control software, and demonstrate this by application to a small case study. In section 2 we identify the requirements for such an approach, in section 3 we explain what we have implemented, and in section 4 we describe an initial experimental evaluation. Section 5 compares our approach against previous work; section 6 summarises the paper and identifies future possibilities.

2 Requirements for a PCG Testing Approach

The major criterion for evaluating a situation-based testing approach is its ability to find the kind of problem behaviours that AR may exhibit. Here, for simplicity, we have limited our scope; we only consider how the combination of sensing, sensor processing and driving algorithms can lead to dangerous movement behaviour. To support this, our situation generator needs the ability to generate static terrain, including plausible variations of terrain type such as rocky deserts, cave systems, and urban areas. It also needs to generate moving obstacles, such as peer robots, as these are particularly difficult to sense and respond to appropriately.

A key requirement is that the PCG technique generates a *wide range of diverse situations*. Beyond mere diversity, it is also important to generate situations that have a *high likelihood of finding dangerous behaviour*; it would be easy for PCG to spend most of its time generating safe situations, which is computationally inefficient.

Regardless of the efficiency of the generator, an effective PCG technique will generate a great many situations, necessitating a great many simulation runs, and hence generating a very large amount of data. We thus need a way to rationalise the output before it is presented to human engineers. At a minimum, this must filter out runs that are safe and as expected. Beyond that, the tool must prioritise the runs in some way, allowing the humans to focus first on the most interesting ones.

Ideally, we would want to gain confidence that the set of situations generated is in some sense 'complete', at least for the scope of situations being considered. This could be approached by measuring the "situation coverage" achieved by a set of situations – given the range of situations that could potentially be generated, the proportion that actually have been generated. This could be composed from a variety of components such as potential for pair-wise interactions between entity types, where entities include (for example) the AR, peer vehicles, roads, walls, and solid obstacles, and interactions include (for example) sensing, proximity, collision avoidance and "moving in formation with". Previous work on agent interaction ontologies (e.g. Nguyen [8]) and on road-traffic interaction patterns and accident proximity measures (e.g. Archer [9]) will be relevant here.

Given the proof-of-concept nature of the work described here, we have not attempted to address the completeness issue. Instead, we have limited ourselves to the assertion that if the method discovers any specification or implementation flaws then it has some value. The decision to use it then becomes a return-on-investment question.

3 Proposed Method

The basic idea is to use PCG to create situations, and then run them in the simulator to observe how the robot (specifically, its control software in command of the robot) behaves in them. The process takes place in three stages.

The first stage uses PCG algorithms to create a binary terrain map, i.e., terrain is either navigable or obstructed. A mission generator then reads the map and places a

number of robots within the environment, ensuring they are initially unobstructed. It then assigns each robot a randomised route, which the robot control algorithm must follow during the second stage. The output of the first stage is a complete situation: a fully-populated environment with robots, obstructions and mission allocations.

The second stage involves executing a situation within a simulated environment. During its execution, the progress and behaviour of the robots within the simulation are monitored to determine whether any hazardous behaviour is exhibited. Monitoring is done by a daemon process which interfaces with the simulation. The daemon inspects the state of the robots to detect whenever an event occurs that corresponds to a member of a defined set of unwanted behaviours, such as colliding with a wall. The occurrence of such an event would indicate a failure of the control algorithm.

Once a run is complete, a third stage processes the event log produced by the daemon. The processing analyses the sequence of events within the run and ranks its overall significance, based on the likelihood that faults or weaknesses of the robot control algorithm were exhibited.

The output of the third stage is a ranked list of situations, ordered by their significance. A domain expert can use this to indicate which situations are most likely to be worth investing further time and resources in, using more labour-intensive techniques.

In the current work, we used Player/Stage [10] as our implementation platform. Player is an abstraction library that provides a standardised interface to robot sensor and actuator hardware. Player also features a client/server architecture, allowing control code to execute across a network connection, rather than only running on-board a robot. It is widely used by robotics researchers as it enables a seamless transition between simulation and physical hardware. This allows developers to, for example, identify a weakness in a control algorithm in simulation, then directly test it in the real world to verify that there is really a problem.

Stage is an open-source robot simulation environment that allows one or more robots to explore and interact within a 2D world. It is often used in conjunction with the Player project, as it provides virtualised hardware for many of the interfaces Player defines. Stage accurately models a range of sensors often found on physical hardware, such as laser rangefinders, and it simulates realistic physics with accurate collision detection. The limitation to a 2D world is pertinent, but is not a major limitation for this proof-of-concept study, and Player is fully compatible with 3D simulations including Gazebo (http://www.gazebosim.org/).

To generate an environment representing terrain and built structures the tool first creates a 2D noise map using a Perlin noise process [11]. As with most PCG techniques, this is used as a base for subsequent post-processing. Using a simple pipe-and-filter architecture, filter effects can be applied and chained arbitrarily to produce the final output that represents terrain. In the current tool we use two filters: a pixelisation filter to make the noise more granular and a thresholding filter to convert the noise into binary occupied/unoccupied. The latter is a wrapped by a coverage constraint algorithm that randomly varies the threshold used by the filter until the coverage of the space reaches a suitable mix of occupied and unoccupied space (as defined in a parameter file). Fig. 1 shows the effects of the filters.

Fig. 1. – Effects of post-processing filters available during environment generation

The tool is configured by a situation file, which specifies various parameters to use during generation, such as the size of the map and the settings for the filters. The combination of these parameters and random variation gives rise to several different types of environment – examples are given in Fig. 2.

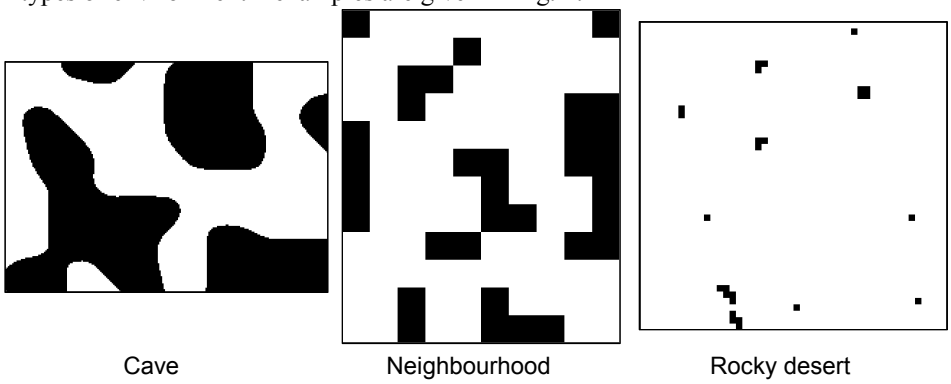

Fig. 2. – Range of possible environments (with suggestive names)

The mission planning tool takes as input the map generated by the environment generator and a second configuration file. The configuration file specifies the number of robots that need missions, including any (optional) minimal route length constraints. The mission planner then places each robot within the environment, ensuring they are not wholly or partially within an obstruction. It then generates a randomised route, defined by an ordered sequence of waypoints, for each robot.

To generate a route, the planner picks an unobstructed point at random, then applies the A* path-finding algorithm [12] between a robot's initial position and the selected point. If no navigable route can be found then the end point is discarded and a new one selected. If a minimum route length is specified in the configuration file, additional end points are appended until the constraint is satisfied. In practice, this lead to routes that closely traced the edges of terrain features, which is not sensible given that the robot needs to maintain useful lines of sight. For the purpose of route planning, therefore, we therefore performed a binary dilation on the map to exagger-

ate the size of terrain objects and thus force the routes to be plotted through more open space.

The pathfinding algorithm generates complete, navigable routes with a great many waypoints. The Ramer–Douglas–Peucker[1] algorithm is used to reduce the number of waypoints, both to reduce storage space for the situation and to force robots to do some online global path-planning to avoid obstacles, thus increasing the challenge for their movement algorithms. The algorithm's epsilon parameter, which determines the amount by which the simplified path can deviate from the original (and thus the amount of global planning that can be required), can be set in the configuration file.

Failures are detected by a daemon that continuously monitors the simulation environment. The daemon is independent of the robot controller code and can detect loitering, route completion, stalls (which are produced by Stage when the robot collides with terrain or another robot), waypoint arrival and unsafe proximity (between two robots). These are a mixture of safety, mission and performance events; in future work we plan to expand the range of safety-specific measures. In effect, it filters the fine detail of the simulation to produce a simpler textual representation (an event log) of each run.

The logs are human readable, but very long, and one log is produced for every situation that is run. In order to guide engineers to the most interesting runs in the log, we developed a process for scoring runs according the interestingness (i.e. safety-significance) of each the log events in that run. The score for each run is derived by applying event-specific penalties each time an event is triggered.

Appropriate values to use for these penalties are subjective and highly dependent on both the features of the control algorithm being assessed and the context under which a robot will operate. The values used during the case study for this project were tuned using trial and error. For example, a penalty of 10 points is added if two robots enter an unsafe proximity, and a further penalty of 1 point per second is applied for the duration of the proximity. An idea for evolving or learning weights is presented in section 6.

We can note that any controller that this implementation was applied to would have to support the Player/Stage-supplied virtual hardware and 2D environment, but otherwise the robot software is independent of our approach and the tool implementation is completely independent of the specific control software used. It could easily be applied to alternative algorithms or rival software designs.

The full source code for our implementation is available at `http://www-users.cs.york.ac.uk/~rda/arnold_sitgen_src.zip`, under a BSD licence.

[1] Also known as the "split-and-merge" algorithm

4 Experiment & Results

4.1 Experiment

For the case study, we implemented a simple robot controller that combined simple path-following with smoothed nearness diagrams (SNDs) [13] for collision avoidance and basic reactive navigation. SNDs are well-understood and have been designed to work in both open and confined environments, including those with dynamic obstacles. This meant that they were suitable for any of the environments created by the environment generator and mission planner. SNDs are often used in robotics research, thus an implementation is bundled with the Player distribution. Finding any weaknesses or faults in that would be evidence in favour the approach taken by this project on a real world and well-tested control algorithm, and it would also provide useful feedback for the Player developers.

The controller was implemented on a virtual Pioneer 3-AT robot (`http://www.mobilerobots.com/ResearchRobots/P3AT.aspx`) equipped with a SICK LMS200 laser rangefinder; a model for this is provided by Stage. The robot is four-wheel drive, uses skid-steering and is capable of speeds of up to 0.8m/s. Although equipped with a sonar array, we used the laser range finder for obstacle detection as it gave us a forward-facing 180° field of view, a maximum sense range of 10m and precision to the nearest cm (surpassing the capabilities of the sonar array).

For the experiment, we generated and ran 500 unique situations, each with randomised parameters such as map size, obstacle density and minimum route lengths. Each situation was run as fast as Stage could simulate, before being terminated after 120 seconds of wall-clock time[2]. Additionally, each situation contained a single AR and between zero and five 'dumb' robots; the latter acted as dynamic obstacles. The dumb robots had collision avoidance disabled and were only allocated routes that did not require any path-planning, i.e., adjacent waypoints could always be reached as the crow flies. Failure detection was only performed for the AR; events concerning only dumb robots did not contribute to the rating of a situation.

The source code and Player/Stage configuration files used for the case study are included in the code distribution linked earlier. Further details are given in [14].

4.2 Results

The risk scores for the different runs were widely distributed; indeed they appeared to follow a power-law distribution with a long tail of low-scoring (low-risk) runs. We investigated several of them in detail, and will discuss the highest-scoring three here. These scored 3064, 1574 and 988 respectively, versus a mean of 61.0. All of these high-scoring runs involved collisions between the AR and one of the dumb robots. The maps and initial mission plans for these runs are shown in figures 3-5. Robot starting points are shown by squares, robot goals are shown by circles; those for the

[2] The final simulation time of each run varied, but it tended to be on the order of 10 minutes, a simulation performance of around 5x real time.

AR are filled shapes, those for the dumb robot are hollow. The green lines represent the route plans provided to the robots; those for the AR have been simplified (as described in section 3) and thus cannot be completed without obstacle avoidance.

In run 207 (ranked first), a repeat collision occurred because the AR rounded a corner and collided with a dumb robot that was outside its field of view. As the dumb robot was still out of view after the collision, the AR continued to try to drive through it, thus continually pushing against it (which was logged as a very large number of collisions). This is a simple example of a hazard.

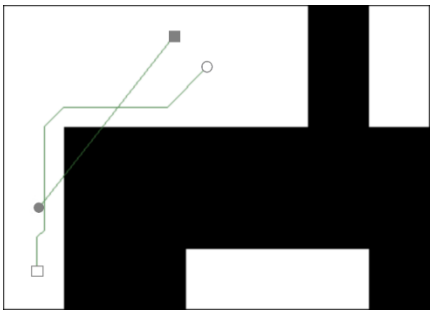

Fig. 3. – environment and mission plan for situation 207

In run 219 (ranked second), the AR and a dumb peer were started very close to each other on routes that took them through each other. There *was* space for them to avoid each other and pass successfully, but they started too close for the AR's avoidance algorithms to work properly. The AR therefore oscillated between trying to pass to the left and trying to pass to the right, colliding repeatedly with the dumb robot.

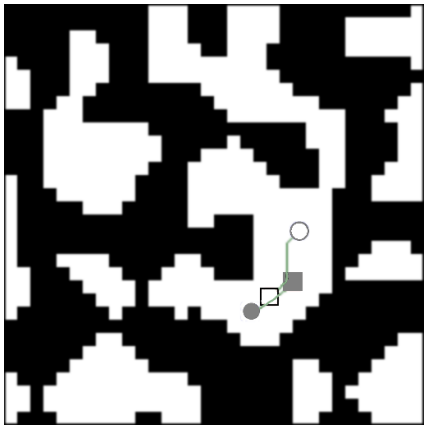

Fig. 4. – environment and mission plan for situation 219

This is interesting because *smoothed* nearness diagrams were meant to fix the oscillation proneness of the original nearness diagram concept, but here this was defeated because the AR could not see enough free space. The problem could be solved if

the AR could comprehend the problem and back up slightly, but the AR's control system does not have that feature.

Run 231 (ranked third) was similar to run 207, except that the AR *could* see the dumb robot. It could have avoided a collision by moving at full speed, thus getting out of the way before the dumb robot reached it. SND, however, is designed to command low speed in "high risk" areas (ones near obstacles). In this case, the obstacle was itself moving towards the SND-equipped robot so high speed was necessary to avoid it. SND guided the AR to move slowly, thus colliding with the dumb robot.

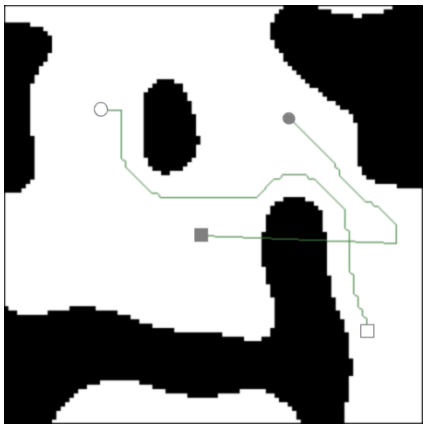

Fig. 5. – environment and mission plan for situation 231

Run 231 is perhaps the most interesting of the three runs, as it reveals a weakness in the established SND concept, not just with the robot's perception (run 207) or with its specific control implementation (run 219). Despite this, it is only scored at one-third the value of run 207. It is therefore clear that some fine-tuning of the scoring system is desirable; with this in mind, we looked at the overall picture of whether the scoring had sorted runs into roughly the right order.

The three runs described above are examples of true positives – runs scored highly that contain accidents. To search for false positives, we randomly sampled 10 runs from the top-rated 50 (excluding the three already discussed) and studied them in some detail. All of the 10 exhibited accidents or other abnormal behaviour, suggesting that the false positive rate was low.

True negatives (runs that have no accidents and produce a low score) are perhaps the least significant category. However, the higher the proportion of true negatives, the more situations need to be generated (and runs performed) before problems are identified. In total, 227 runs (45% of the total) were assigned a score of less than ten points. 10% of these were sampled, and in all cases the low ratings were justified. This sampling also suggests that the false negative rate was low.

5 Related Work

As noted earlier, there are many applications of procedural generation in video games and virtual environment training (e.g. [2, 4-7]). These, however, aim for an optimum appearance or a slightly varied play experience, rather than the highly diverse challenges that concern us here. The use of Parish and Müller [15] in the CityEngine urban planning system is closer to our concerns, but still not directly applicable to creating challenging environments for robots.

Ashlock et al. [16] presented a technique for generating mazes to test the effectiveness of robot path planning algorithms. They evolved mazes which they then statically analysed to check they meet specific constraints, such as a minimum number of turns or a minimum optimal route length. Unfortunately, they did not evaluate any path-planners in the generated mazes – the merely conclude that high fitness scores were achieved and that the visualised output was varied and (intuitively) challenging.

Nguyen et al. [17] apply a similar approach, using evolutionary optimisation to create challenging test situations for an AR. They then take it one step further, putting simulated AR within the situations to test how well they perform. The robot's performance is measured using a fitness function based on stakeholder-derived "soft goals", and this is used to guide the optimisation. In this their technique is successful (it produces high fitness values), but they do not explore whether the runs are revealing diverse faults or merely exploiting progressively worse consequences of a single fault.

There are a variety of approaches to runtime safety of AR (e.g. Wardziński [18]). These are orthogonal to the design-time analysis described in this paper; both are necessary for safe AR. Indeed, there would be interesting further work in applying our approach to analysis of a system that included such a runtime safety system, looking for situations that could cause the runtime system to dangerously fail.

6 Conclusions

We have described a PCG approach for generating situations to find faults in robot control software, and shown through a small case study that the basic approach can find faults in some cases. The case study found faults in the standard SND algorithm, without being tuned specifically for application to SND. The method is not fully automated – it requires a human engineer to study accident runs and discover exactly what is causing the problems – but the tool generates situations, executes them, and prioritises the results for human attention.

Because the method tests purely by generating environments, it could be used to test any robot that is written for Player/Stage. Similarly, it can be used on a simple prototype in order to test and algorithm or an overall control strategy – a lot can be learned (fundamental algorithm flaws can be found) without needing a fully developed and mature implementation.

As further work, this approach could be implemented in a higher-fidelity simulation. This is perhaps best done by industry or as a commercialisation effort; further academic work should focus on refining the situation generation and result prioritisa-

tion algorithms, and understanding the space of environments that cause problems for major AR algorithms and strategies.

An empirical evaluation of the method's fault-finding power would be valuable. It could be compared to conventional testing approaches, and to manual situation generation. The comparative testing work by Nguyen et al [19] is useful template – in particular, note the way that they assess proportion of known faults found rather than relying on arbitrary measures such as run scores or search fitness functions.

As further evaluation, there are many robot algorithms and control strategies that this approach could be applied to. One interesting case would be to apply it to the initial formulation of Velocity Obstacles, as presented by Fiorini and Shiller [20], and check whether it reveals the known flaws in that algorithm (see [21], [22]).

Although the case study showed the run scoring system producing useful results, it is not clear whether it is well-calibrated to the scores that a human expert would assign had they studied the situation in detail. It may be possible to learn or evolve the parameters of the scoring system (event weightings) by having human experts rate a large set of runs, then letting the learner or optimiser derive a scoring system that reproduces those scores for those runs. Such scoring system could be used as a fitness function for an optimisation technique that tuned the parameters of the situation generator. Although trying to maximise the scores achieved might lead to a narrow focus on a few high-impact faults, tuning the generator to minimise the number of *low* scores achieved might greatly increase the performance of the approach without compromising the diversity.

Acknowledgements The authors would like to thank Ibrahim Habli for his comments on an earlier draft of this paper.

References

1. Fletcher, L., Teller, S., Olson, E., Moore, D., Kuwata, Y., How, J., Leonard, J., Miller, I., Campbell, M., Huttenlocher, D., Nathan, A., Kline, F.-R.: The MIT – Cornell Collision and Why it Happened. Journal of Field Robotics 25, 775-807 (2008)
2. Martin, G.A., Hughes, C.E.: A Scenario Generation Framework for Automating Instructional Support in Scenario-based Training. Proceedings of the Spring Simulation Multiconference, (2010)
3. Knight, J.C., Leveson, N.G.: A Large Scale Experiment In *N*-Version Programming. Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing, pp. 135-139, Ann Arbor, MI (1985)
4. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games (2011)
5. Togelius, J., Preuss, M., Yannakakis, G.N.: Towards multiobjective procedural map generation. Proceedings of the Workshop on Procedural Content Generation in Games (PGGames '10), Monterey, CA (2010)

6. Roden, T., Parberry, I.: From Artistry to Automation: A Structured Methodology for Procedural Content Creation. Entertainment Computing – ICEC 2004, pp. 301-304 (2004)
7. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-Based Procedural Content Generation Applications of Evolutionary Computation, (2010)
8. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based test generation for multiagent systems. Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems, pp. 1315-1320 (2008)
9. Archer, J.: Indicators for traffic safety assessment and prediction and their application in micro-simulation modelling. PhD thesis, KTH, Stockholm (2005)
10. Gerkey, B.P., Vaughan, R.T., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. Proceedings of the Intl. Conf. on Advanced Robotics (ICAR), pp. 317-323, Coimbra, Portugal (2003)
11. Perlin, K.: An Image Synthesizer. SIGGRAPH Comput. Graph. 19, 287-296 (1985)
12. Hart, P.E.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths IEEE Transactions on Systems Science and Cybernetics 4, 100-107 (1968)
13. Durham, J., Bullo, F.: Smooth Nearness-Diagram Navigation. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008), pp. 690-695. IEEE (2008)
14. Arnold, J.: Robot Hazard Analysis using Procedural Content Generation. MEng thesis, University of York (2012)
15. Parish, Y.I.H., Müller, P.: Procedural Modeling of Cities. Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 301-308, New York, USA (2001)
16. Ashlock, D.A., Manikas, T.W., Ashenayi, K.: Evolving A Diverse Collection of Robot Path Planning Problems. IEEE Congress on Evolutionary Computation, Vancouver, BC, Canada (2006)
17. Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary testing of autonomous software agents. Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09), (2009)
18. Wardziński, A.: The Role of Situation Awareness in Assuring Safety of Autonomous Vehicles. Proceedings of the 25th International Conference on Computer Safety, Reliability and Security (SAFECOMP '06), pp. 205-218 (2006)
19. Nguyen, C.D., Perini, A., Tonella, P.: Constraint-based Evolutionary Testing of Autonomous Distributed Systems Proceedings of the Software Testing Verification and Validation Workshop, pp. 221 - 230 Lillehammer (2008)
20. Fiorini, P., Shiller, Z.: Motion Planning in Dynamic Environments Using the Relative Velocity Paradigm. Proceedings of the IEEE International Conference on Robotics and Automation, pp. 560 –565 (1993)
21. Large, F., Laugier, C., Shiller, Z.: Navigation among moving obstacles using the NLVO: Principles and applications to intelligent vehicles. Autonomous Robots 19, 159–171 (2005)
22. Fulgenzi, C., Spalanzani, A., Laugier, C.: Dynamic Obstacle Avoidance in uncertain environment combining PVOs and Occupancy Grid. Proceedings of the IEEE International Conference on Robotics and Automation, pp. 1610-1616 (2007)