Artificial Intelligence for the Automated Synthesis and Validation of Programs

Sergio Jiménez Celorrio

Universitat Politécnica de València, Spain

Area: Ciencias de la Computación y Tecnología Informática

Abstract

Programming errors, commonly known as *bugs*, cause undesired software behavior making a program crash or enabling malicious users to access private data. Just for 2017, the cumulative cost of software bugs is worldwide estimated in more than one trillion US dollars.

This research project aims to investigate a **novel approach for software development, that leverages Artificial Intelligence, to reduce the chances of introducing programming errors**. The project proposes to addresses automatic program synthesis and validation starting from *input-output* tests cases, and using *AI planning* as a problem solving engine. Our work on this particular research topic is the recipient of the *2016 distinguished paper award* at IJCAI, the main international conference on Artificial Intelligence.

1 Historial científico-técnico del Equipo de Investigadores (Apartado 5.1,bases de la convocatoria

1.1 Principal Investigator: Sergio Jiménez Celorrio

Sergio is a *Ramón y Cajal* fellow at the Universitat Politècnica de València (2017-2022). Previously, Sergio was a post-doc at the University of Melbourne and a *Juan de la Cierva* fellow at the Universitat Pompeu Fabra de Barcelona. From 2004 to 2013 Sergio was assistant lecturer at the Universidad Carlos III de Madrid where he obtained the *Distinguished Thesis Award 2011*. Sergio is co-organizer of the 7^{th} *International Planning Competition* and program committee at top international conferences on Artificial Intelligence.

His research addresses complex *decision-making* problems with innovative integrations of two *Artificial Intelligence* paradigms: *Machine Learning* and *AI Planning*. His research trajectory on this topic has produced a novel approach to the synthesis of programs, grammars and controllers for autonomous behavior and is the recipient of the *2016 distinguished paper award* at the International Joint Conference on Artificial Intelligence (IJCAI)[1], the main international conference on *Artificial intelligence*.

His top five scientific contributions are:

1. Inductive programming with classical planning. [1], [2], [3].

[1] IJCAI is ranked as $A+$ by the CORE rank and with *h5-index* 43 at Google Scholar

2. Generalization in AI planning. [4], [5], [6].
3. Learning to relieve the *knowledge acquisition bottleneck* in AI planning. [7], [8], [9], [10], [11].
4. Machine learning for addressing the *curse of dimensionality* in AI planning. [12], [13], [14].
5. Evaluation in AI planning. [15], [16], [17].

General quality indicators of scientific research: **373 citations** (287 since 2011) and **h-index 11**. *Source google scholar.*

1.2 Senior Researcher: Eva Onaindia

1.3 Researcher in trainnig period: Diego Aineto

2 Estado del arte y objetivos específicos (Apartado 5.2.i, bases de la convocatoria)

This research project will **investigate the integration of *AI planning* into the *Test Driven Development* for the automatic synthesis and validation of programs**. Remarkably, the programs synthesized with our approach are guaranteed to be bug-free over given sets of *input-output* tests cases.

Here we briefly introduce the technology we will rely on along the project:

- **AI Planning (AIP)** is the Artificial Intelligence component that studies the synthesis of sets of actions to achieve some given objectives [18]. AIP arose in the late '50s from converging studies into *combinatorial search*, *theorem proving* and *control theory* and now, is a well formalized paradigm for problem solving with algorithms that scale-up reasonably well. State-of-the-art planners are able to synthesize plans with hundreds of actions in seconds time [19]. The mainstream approach for AIP is *heuristic search* with heuristics derived automatically from the problem representation [20, 21]. Current planners add other ideas to this like *novelty exploration* [22], *helpful actions* [23], *landmarks* [24], and *multiqueue best-first search* [25] for combining different heuristics.
- **Test driven development (TDD)** [26] is a popular paradigm for software development that is frequently used in *agile methodologies* [27]. In TDD, test cases are created before the program code is written and they are run against the code during the development, e.g. after a code change via an automated process. Tests alert programmers of bugs before handing the code off to clients (the cost of finding a bug when the code is first written is considerably lower than the cost of detecting and fixing it later). When all tests pass, the program code is considered *complete* while when a test fails, it pinpoints a *bug* that must be fixed from the program code. Tests cases are a natural form of program specification, programmers often claim *'code that is difficult to test is poorly written'*. Further, writing a thorough set of tests cases forces programmers to think through inputs, outputs, and error conditions of programs.

Now we review the two most successful approaches for program synthesis:

- **Programming by Example (PbE)**. PbE techniques have already been deployed in the real world and are part of the FLASH FILL feature of Excel in Office 2013 that generates programs for string transformation [28]. In this case the set of synthesized programs are represented succinctly in a restricted Domain-Specific Language (DSL) using a data-structure called version space algebras [29]. The programs are computed with a domain-specific search that implements a divide and conquer approach. The approach posed in this research project falls into this category.

- In **Programming by Sketching (PbS)** programmers provide a partially specified program, i.e. a program that expresses the high-level structure of an implementation but that leaves low level details undefined to be determined by the synthesizer [30]. This form of program synthesis relies on a programming language called SKETCH, for sketching partial programs. PbS implements a counterexample-driven iteration over a synthesize-validate loop built from two communicating SAT solvers, the inductive synthesizer and the validator, to automatically generate test inputs and ensure that the program satisfy them. Despite, in the worst case, program synthesis is harder than NP-complete, this counterexample-driven search terminates on many real problems after solving only a few SAT instances [31].

Closely related to the aims of the project is the work on the synthesis and validation of *Finite State Controllers* (FSCs) [32]. The existing algorithms for computing FSCs follow a *top-down* approach that interleaves *programming* the FSC with validating it [33]. To keep the computation of FSCs tractable, they limit the space of possible solutions bounding the maximum size of the FSC. In addition, they impose that the instances to solve share, not only the domain theory (actions and predicates schemes) but the set of fluents [34] or a subset of *observable* fluents [35].

The computation of FSCs for generalized planning includes works that compile the generalized planning task into another forms of problem solving so they benefit from the last advances on off-the-shelf solvers (e.g. *classical planning* [34], *conformant planning* [35], *CSP* [36] or a *Prolog program* [37]). This last case requires a behavior specification of the FSC consisting on classified execution histories that (1) accept all legal execution histories leading to a goal-satisfying state, and (2) reject those that contain repeated configurations (indicating an infinite loop) and that cannot be extended (indicating a dead end) [37].

Finally the generation of programs from examples is a research question also addressed in inductive Logic Programming (ILP). ILP arises from the intersection of *Machine Learning* and *Logic Programming* and deals with the development of inductive techniques to learn a given target concept, expressed as a logic program, from examples and background knowledge, that are expressed as logic facts [38].

3 Metodología y Plan de Trabajo (Apartado 5.2.ii, bases de la convocatoria)

Our research shows that current AI planners can synthesize programs for non trivial tasks like sorting lists, traversing graphs or manipulating strings [3, 34, 39, 33]. To illustrate this, Table 1 reports the invested time to solve the following programming tasks: computing the n^{th} term of the *summatory* and *Fibonacci* series, *reversing* a list, *finding* an element (and the *minimum* element) in a list, *sorting* a list or traversing a binary tree.

Programming Task	Time (seconds)
Summatory	1
Fibonacci	5
Reverse	22
Find	336
Minimum	284
Sorting	30
Tree	165

Tab. 1: Time to synthesize the programs with the FD AI planner [24] on a processor *Intel Core i5 3.10GHz x 4* and with a 4GB memory bound.

Our approach is modeling and solving a given TDD programming task as an AI planning task. Briefly, the state variables of these AIP tasks are *fluents* of the kind `(var:=value)` while the initial/goal states of the AIP tasks encode the programming input/output tests. Finally, program instructions are encoded using two kinds of AIP actions:

1. *Programming actions*, that set an instruction on a given program line.
2. *Execution actions*, execute the instruction set on a given program line.

We implement this encoding using standard planning languages, such as PDDL [40], so the AIP tasks resulting from our encoding can be solved with off-the-shelf planners, like FD [24]. Interestingly, the encoding allows also program validation by (1), specifying the program to validate in the initial state of the AIP task and (2), disabling the mentioned *programming actions*.

3.1 Two-year workplan

We designed a 24-month workplan plan for the **development of a user-interactive program synthesizer** that (1), takes as input a set of test cases that specify the TDD programming task to solve and (2), outputs a program code that passes these tests with a bug-free guarantee. Figure 1 details the proposed 24-month timeline for the project.

1. **T1. System design (months 1-6**)
   (a) Design of the user-interaction for the test-case specification. Programming tasks are specified as a set of *input-output* tests cases and the available instruction set.

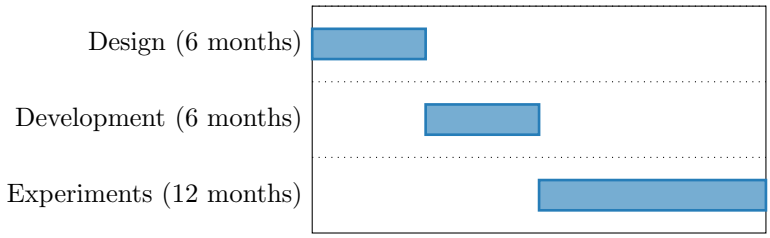

Fig. 1: Twentyfour-month workplan for the development of a user-interactive program synthesizer.

(b) Experimental design. The experiments will comprise taking time and memory measurements to evaluate the resources required by our approach to solve a given set of TDD programming tasks:

(c) Evaluation of different AI planners available, with special attention to the planners that get the best results at the IPC-2018.

Deliverable: Technical report with the specifications of the system design.

2. **T2. Development of the system architecture (months 7-12**)

(a) The programming-into-planning compiler (*Compiler 1*). This component parses the *TDD programming task* and produces an *AIP task* encoded in the standard planning language PDDL.

(b) The plan-into-program compiler (*Compiler 2*). This component extracts the program code from the solution plan produced by an off-the-shelf AI planner.

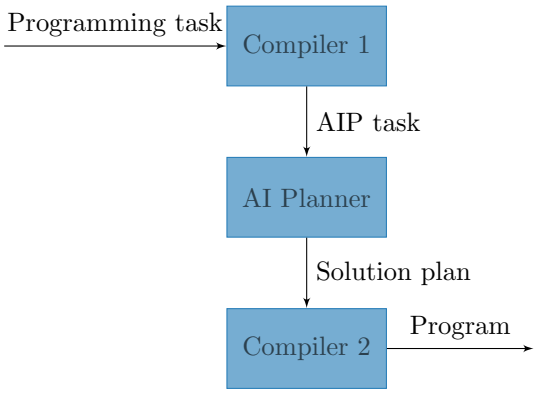

Fig. 2: The system architecture.

Deliverable: Open repository with the source code of the system architecture.

3. **T3. Experiments and dissemination of results (months 13-24**). **Deliverable:** Final report with the obtained conclusions and produced publications.

4 Objetivos específicos del Proyecto (Apartado 5.2.iii, bases de la convocatoria)

To precisely characterize the specific kind and size of the programs that we can synthesize and validate, we model programs as *finite state machines* [33]. Figure 3 shows a program for traversing linked lists pictured as a *finite state machine*. The machine nodes mount to the different program lines while edges are tagged with a *condition/instruction* label, that denotes the condition under which program instructions are taken.

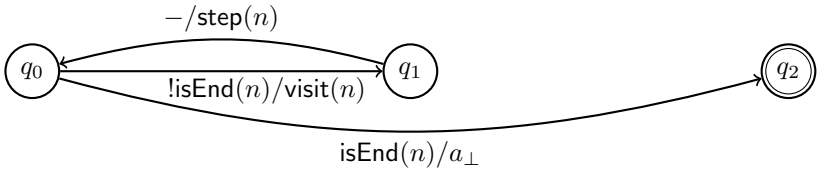

Fig. 3: *Finite state machine* that depicts a program to solve the programming task of traversing a linked list (n is a variable that points to a node of the linked list).

Formally, we define a program as a tuple $\langle \mathcal{Q}, q_0, Q_\perp, \mathcal{V}, , \mathcal{I}, \Omega, T, \Gamma \rangle$:

- $\mathcal{Q}$ is a finite set of program lines where $q_0 \in \mathcal{Q}$ is the *initial program line* and $Q_\perp \subseteq \mathcal{Q}$ is the subset of *terminal program lines*.
- $\mathcal{V}$ is the finite set of program variables, where the value of each variable $v \in \mathcal{V}$ is defined in the domain D_v.
- $\mathcal{I}$ is a finite instruction set where each instruction $a \in \mathcal{I}$ defines an *update* θ_a of a subset of the program variables.
- $\Omega : Q \times D_{v_1} \times \ldots \times D_{v_{|\mathcal{V}|}} \mapsto \{0, 1\}$ is an *observation function* that maps a program line and a condition over the program variables into a Boolean (1 if the condition holds 0 if it does not).
- $T : Q \times \{0, 1\} \to Q$ is a *transition function* mapping pairs of a program line and the result of a *condition* check into the next program line.
- $\Gamma : Q \times \{0, 1\} \to \mathcal{I}$ is a *transition function* mapping pairs of a program line and the *condition* result into the corresponding instruction.

In the example of Figure 3, the edge (q_0, q_2) with label $\mathsf{isEnd}(n)/a_\perp$ encodes that the condition is $\mathsf{isEnd}(n)$?, while $T(q_0, 1) = q_2$, $\Gamma(q_0, 1) = a_\perp$ encode the transition and associated instruction when $\mathsf{isEnd}(n)$ holds. The edge (q_0, q_1) with label $!\mathsf{isEnd}(n)/\mathsf{visit}(n)$ encodes the transition and instruction when $\mathsf{isEnd}(n)$ does not hold, i.e. $T(q_0, 0) = q_1$ and $\Gamma(q_0, 0) = \mathsf{visit}(n)$. The edge (q_1, q_0) with label $-/\mathsf{step}(n)$ denotes that, when in q_1, the transition and instruction are always the same no matter the value of the program variables.

The *state of a program* is a pair (q, s) that consists of the program line $q \in \mathcal{Q}$ and $s \in D_{v_1} \times \ldots \times D_{v_{|\mathcal{V}|}}$, an assignment to the variables compatible with their domains. Programs transition to a program state (q', s') that is computed as follows:

1. First we observe whether the condition associated to the current progam line holds (we denote the result of this observation as $O_q(s) \in \{0, 1\}$).

2. Then we compute the new program line $q' = T(q, O_q(s))$ and the instruction $a = \Gamma(q, O_q(s))$ to apply next. Each transition $T(q, O_q(s))$ is associated with a single observation of the variable values however, programs can represent expressive conditions including the no-op action ($a_\perp \in \mathcal{I}$) in the instruction set. The no-op action does not modify the program variables and affects only to the next program line which allows to concatenate conjunctions of conditions like in *decision trees*.

3. Finally the value of the program variables $s' = \theta_a(s)$ is updated executing the corresponding instrucntion $a = \Gamma(q, O(s, q)) \in \mathcal{I}$.

The specific objective of this project is the study of the performance of our approach for the automated synthesis and validation of programs that are characterized by these tree dimensions:

1. $|\mathcal{Q}|$, the maximum number of program lines.

2. $|\mathcal{V}|$, the maximum number of program variables.

3. $|\mathcal{I}|$, the maximum size of the available instruction set.

Note that despite setting bounds for $|\mathcal{Q}|$, $|\mathcal{V}|$ and $|\mathcal{I}|$, challenging programming tasks can still be addressed using *problem decomposition*. With this regard, our AIP encoding supports callable procedures to decompose a given programming task in simpler modules and to enable recursive solutions [34, 33].

In this project we aim to **study the performance of an Artificial Intelligence approach for the automated synthesize and validation of programs up to**: 5 program lines, 50 variables with binary domain and, an instruction set that comprises 10 instructions. The performance of our approach will be evaluated with regard to the *computation time* and *memory* invested in the synthesize and validation of the characterized programs.

5 Beneficios del proyecto. Difusión y Explotación en su caso de los Resultados.

This project will **provide new insights into the current understanding of how Artificial Intelligence can assist programmers in the software development**.

Research in AI algorithms is too often tested with laboratory problems and AIP is not an exception. Most of the new planning algorithms are only tested within the benchmarks of the International Planing Competition (IPC) [41]. This project will also help to **meet the computational and expressiveness limits of AI planners when addressing real-world programming tasks**.

Promising research opportunities come from the application of AIP to *program synthesis* given that, *program synthesis* with a tests base, can also be seen as a dual to *program testing*. AIP has recently shown successful in *program testing* to generate *attack plans* that completed non-trivial software security tests [42, 43, 44, 45].

Our current work on *program synthesis* with AIP already produced **publications at top international conferences on Artificial Intelligence** [46, 39, 33, 34] and is the recipient of the *2016 distinguished paper award* at the International Joint Conference on Artificial Intelligence, the main international conference on *Artificial intelligence*. The scientific results obtained during the development of the project will be submitted to top Artificial Intelligence conferences (such as IJCAI, AAAI, ICML and ICAPS) and to the main journals in the AI field (such as AIJ, JMLR and JAIR).

References

[1] J. Segovia-Aguas, S. Jiménez Celorrio, and A. Jonsson, "Generating context-free grammars using classical planning," in *International Joint Conference on Artificial Intelligence*, 2017.

[2] J. Segovia-Aguas, S. Jiménez Celorrio, and A. Jonsson, "Generalized planning with procedural domain control knowledge," in *International Conference on Automated Planning and Scheduling*, 2016.

[3] S. Jiménez Celorrio and A. Jonsson, "Computing plans with control flow and procedures using a classical planner," in *Eighth Annual Symposium on Combinatorial Search*, 2015.

[4] J. Segovia-Aguas, S. Jiménez Celorrio, and A. Jonsson, "Unsupervised classification of planning instances," in *International Conference on Automated Planning and Scheduling*, 2017.

[5] D. Lotinac, J. Segovia, S. Jiménez Celorrio, and A. Jonsson, "Automatic generation of high-level state features for generalized planning," in *International Joint Conference on Artificial Intelligence*, 2016.

[6] J. Segovia, S. Jiménez Celorrio, and A. Jonsson, "Hierarchical finite state controllers for generalized planning," in *International Joint Conference on Artificial Intelligence*, 2016.

[7] D. Aineto, S. Jiménez Celorrio, and E. Onaindia, "Learning strips action models with classical planning," in *International Conference on Automated Planning and Scheduling*, 2018.

[8] S. Jiménez Celorrio, F. Fernández, and D. Borrajo, "Integrating planning, execution and learning to improve plan execution," *Computational Intelligence*, vol. 29, no. 1, pp. 1–36, 2013.

[9] S. Jiménez Celorrio, F. Fernández, and D. Borrajo, "The pela architecture: integrating planning and learning to improve execution," *AAAI Conference on Artificial Intelligence*, 2008.

[10] S. Jiménez Celorrio, A. Coles, and A. Smith, "Planning in probabilistic domains using a deterministic numeric planner," *25th Workshop of the UK Planning and Scheduling Special Interest Group*, 2006.

[11] J. Lanchas, S. Jiménez Celorrio, F. Fernández, and D. Borrajo, "Learning action durations from executions," *Workshop on Planning and Learning ICAPS 2007*, 2007.

[12] S. Jiménez Celorrio, T. De La Rosa, S. Fernández, F. Fernández, and D. Borrajo, "A review of machine learning for automated planning," *The Knowledge Engineering Review*, vol. 27, no. 04, pp. 433–467, 2012.

[13] T. De la Rosa, S. Jiménez Celorrio, R. Fuentetaja, and D. Borrajo, "Scaling up heuristic planning with relational decision trees," *Journal of Artificial Intelligence Research*, pp. 767–813, 2011.

[14] T. De La Rosa, S. Jiménez Celorrio, and D. Borrajo, "Learning relational decision trees for guiding heuristic planning," *International Conference on Automated Planning and Scheduling*, 2008.

[15] C. L. López, S. Jiménez Celorrio, and Á. G. Olaya, "The deterministic part of the seventh international planning competition," *Artificial Intelligence*, vol. 223, pp. 82–119, 2015.

[16] C. L. López, S. Jiménez Celorrio, and M. Helmert, "Automating the evaluation of planning systems," *AI Communications*, vol. 26, no. 4, pp. 331–354, 2013.

[17] A. Coles, A. Coles, A. G. Olaya, S. Jiménez Celorrio, C. L. López, S. Sanner, and S. Yoon, "A survey of the seventh international planning competition," *AI Magazine*, vol. 33, no. 1, pp. 83–88, 2012.

[18] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[19] H. Geffner and B. Bonet, "A concise introduction to models and methods for automated planning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 8, no. 1, pp. 1–141, 2013.

[20] D. V. McDermott, "A heuristic estimator for means-ends analysis in planning.," in *International Conference on Artificial Intelligence Planning and Scheduling*, vol. 96, pp. 142–149, 1996.

[21] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1, pp. 5–33, 2001.

[22] G. Frances, M. Ramırez, N. Lipovetzky, and H. Geffner, "Purely declarative action representations are overrated: Classical planning with simulators," in *IJCAI*, 2017.

[23] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, p. 57, 2001.

[24] M. Helmert, "The fast downward planning system.," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[25] S. Richter and M. Westphal, "The lama planner: Guiding cost-based anytime planning with landmarks," *Journal of Artificial Intelligence Research*, vol. 39, no. 1, pp. 127–177, 2010.

[26] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[27] D. Cohen, M. Lindvall, and P. Costa, "Agile software development," *DACS SOAR Report*, no. 11, 2003.

[28] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGPLAN Notices*, vol. 46, pp. 317–330, ACM, 2011.

[29] T. M. Mitchell, "Generalization as search," *Artificial intelligence*, vol. 18, pp. 203–226, 1982.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 404–415, 2006.

[31] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[32] B. Bonet and H. Geffner, "Policies that generalize: Solving many planning problems with the same policy," *IJCAI*, 2015.

[33] J. Segovia-Aguas, S. Jiménez, and A. Jonsson, "Hierarchical finite state controllers for generalized planning," in *International Joint Conference on Artificial Intelligence*, pp. 3235–3241, 2016.

[34] J. Segovia, S. Jiménez, and A. Jonsson, "Generalized planning with procedural domain control knowledge," in *International Conference on Automated Planning and Scheduling*, pp. 285–293, 2016.

[35] B. Bonet, H. Palacios, and H. Geffner, "Automatic derivation of finite-state machines for behavior control," in *AAAI Conference on Artificial Intelligence*, 2010.

[36] C. Pralet, G. Verfaillie, M. Lemaître, and G. Infantes, "Constraint-based controller synthesis in non-deterministic and partially observable domains.," in *ECAI*, 2010.

[37] Y. Hu and G. De Giacomo, "A generic technique for synthesizing bounded finite-state controllers," in *International Conference on Automated Planning and Scheduling*, 2013.

[38] S. Muggleton, "Inductive logic programming," *New generation computing*, vol. 8, no. 4, pp. 295–318, 1991.

[39] D. Lotinac, J. Segovia-Aguas, S. Jiménez, and A. Jonsson, "Automatic generation of high-level state features for generalized planning," in *International Joint Conference on Artificial Intelligence*, pp. 3199–3205, 2016.

[40] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains.," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[41] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, *et al.*, "The 2014 international planning competition: Progress and trends," *AI Magazine*, vol. 36, no. 3, pp. 90–98, 2015.

[42] J. Hoffmann, "Simulated penetration testing: From "dijkstra" to "turing test++"," in *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015.

[43] M. Steinmetz, J. Hoffmann, and O. Buffet, "Revisiting goal probability analysis in probabilistic planning.," in *International Conference on Automated Planning and Scheduling*, pp. 299–307, 2016.

[44] D. Shmaryahu, "Constructing plan trees for simulated penetration testing," in *The 26th International Conference on Automated Planning and Scheduling*, vol. 121, 2016.

[45] M. Steinmetz, J. Hoffmann, and O. Buffet, "Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art," *Journal of Artificial Intelligence Research*, vol. 57, pp. 229–271, 2016.

[46] J. Segovia-Aguas, S. Jiménez, and A. Jonsson, "Generating context-free grammars using classical planning," in *International Joint Conference on Artificial Intelligence*, 2017.