Artificial Intelligence for the Automated Synthesis and Validation of Programs

Abstract

Programming is nowadays a manual handcraft task however, the need to automate programming tasks increases every day: Programming errors, commonly known as *bugs*, cause undesired software behavior making programs 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 investigates a novel approach for software development, that leverages *Artificial Intelligence* (AI), to increase the automation of the programming process and hence, reduce the chances of introducing software bugs. The project proposes to address **program synthesis and program validation, starting from *input-output* tests cases, and using *AI planning* as a problem solving engine**.

Our current work on this research topic is the recipient of the *2016 distinguished paper award* at IJCAI (the main international conference on AI) and it is accepted for publication at the *Artificial Intelligence Journal*, the premier international journal for research in AI.

Keywords: *Computer Science, Artificial Intelligence, Program Synthesis, Program Validation, AI planning.*

1 Introduction

Program Synthesis is the task of computing a program that satisfies a given semantic and formal specification of correctness.

The 2008 PhD Thesis work by Armando Solar-Lezama, at the University of California Berkeley, showed that it is possible to encode program synthesis as a *Boolean logic SAT* problem and therefore, use *Satisfiability Modulo Theories* [1] to automatically compute programs [2]. Since then, there has been a surge of practical interest in the idea of program synthesis in the formal verification community and related fields.

To illustrate this interest: Since 2012 the US National Science Foundation funds the ExCAPE research project[1] ($3, 750, 000) to advance the theory and practice of program synthesis. In 2013 a standard framework for program synthesis was defined [3]. Since then, two yearly international competitions are established (`http://www.sygus.org`), to compare the different approaches for program synthesis, and the sister competition (`http://www.syntcomp.org/`), to compare different approaches for the synthesize of reactive programs. Program synthesis has already been deployed in the real world and it is part of the Flash

[1] `https://www.nsf.gov/awardsearch/showAward?AWD_ID=1138996`

FILL feature of MICROSOFT EXCEL that automatically generates programs for string transformation [4].

Computational approaches to program synthesis range over a wide spectrum, from *deductive synthesis* to *inductive synthesis*. In deductive program synthesis, a program is synthesized by constructively proving a theorem, employing logical inference and constraint solving [5]. On the other hand, inductive synthesis aims to compute a program matching a set of input-output examples by searching in a restricted space of programs [6, 7]. It is thus an instance of *Machine Learning* (ML). Most of current systems for program synthesis blend induction and deduction [8] and usually syntax guidance is a key ingredient in these systems.

Closely related to the aims of the project is the synthesis of *Finite State Controllers* (FSCs) [9]. The state-of-the-art algorithms for computing FSCs follow a *top-down* approach that interleaves *programming* the FSC with *validating* it [10, 11]. To keep the computation of FSCs tractable, the space of possible solutions is bound by the maximum size of the FSCs. The computation of FSCs includes works that compile this task into another forms of problem solving so they benefit from the last advances on off-the-shelf solvers (e.g. *classical planning* [12], *conformant planning* [13], *CSP* [14] or a *Prolog program* [15]). Last but not least, the synthesis of programs from examples is also addressed in the classic AI field of *Inductive Logic Programming* (ILP) [16, 17]. ILP arises from the intersection of *Machine Learning* and *Logic Programming* and deals with the development of inductive techniques to learn *logic programs* from examples and background knowledge, that are expressed as logic facts.

Program Validation is the task of proving (or disproving) the correctness of a given program with respect to a formal specification of the aimed program semantics. Program validation is considered a *necessary step* for program synthesis and *model checking* is the mainstream AI approach for the formal validation of programs and controllers [18]. Current approaches for model checking reduces to graph search but, instead of enumerating reachable states one at a time, they traverse the state space considering large numbers of states at a single step. For instance, representing sets of states and transition relations as logical formulas or *binary decision diagrams*, like in *symbolic model-checking* [19].

Program validation is also compilable into classical planning. Examples are the compilations for GOLOG *procedures* [20], *planning programs* [21], or *Finite State Controllers* [13, 10, 11]. Briefly these compilations encode the *cross product* of a given planning instance and the automata corresponding to the program to validate. A *validation proof* is provided if a solution to the compiled instance is found (i.e. the program is *correct*). Otherwise, if a classical planner proves that no solution exists for the compiled instance, then the program is *incorrect* because its execution necessarily failed. When actions have non-deterministic effects, program validation becomes more complex since it requires proving that all the possible program executions reach the goals. In such a scenario, *model checking* is a more suitable approach.

2 Methodology

This research project will investigate a computation method that **integrates *AI planning* into the *Test Driven Development* paradigm for the automatic synthesis and validation of programs**.

Our current research already shows that this method can synthesize and validate programs for non trivial tasks like sorting lists, traversing graphs or manipulating strings [22, 12, 23, 10, 24, 11, 21]. Table 1 reports the time invested by the AI planner FD [25] 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, traversing a binary tree, or building a *parser* for simple arithmetic operations.

Programming Task	**Time (seconds)**
Summatory	1
Fibonacci	5
Reverse	22
Find	336
Minimum	284
Sorting	30
Tree	165
Parser	45

Tab. 1: Time to synthesize the programs with the AI planner FD [25] on a processor *Intel Core i5 3.10GHz x 4* and with a 4GB memory bound.

2.1 Background

First we briefly introduce the technology required by our AI method for the automated synthesis and validation of programs:

- **AI Planning (AIP)** is the Artificial Intelligence component that studies the synthesis of sets of actions to achieve some given objectives [26]. 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 [27]. The mainstream approach for AIP is *heuristic search* with heuristics derived automatically from the problem representation [28, 29]. Current planners add other ideas to this like *novelty exploration* [30], *helpful actions* [31], *landmarks* [25], and *multiqueue best-first search* [32] for combining different heuristics.

- **Test driven development (TDD)** [33] is a popular paradigm for software development that is frequently used in *agile methodologies* [34]. 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. When all tests pass, the program code is considered *correct* 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, 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).

2.2 Automated synthesis and validation of programs

We use input-output *test cases* to specify the semantics of the aimed programs, as in Test Driven Development. Then, we encode the *TDD programming* (or validation) task as an AI planning problem and finally, we use an off-the-shelf AI planner to compute solutions to these problems.

2.2.1 The TDD programming task

A *TDD programming task* is a programming task where the semantics of the aimed program is specified with a set of input-output *test cases*. Formally, a *TDD programming* task is a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{T} \rangle$ where:

- $\mathcal{V}$ is the set of n *program variables*. The respective domains are $\mathcal{D} = \langle \mathcal{D}_{v_0}, \ldots, \mathcal{D}_{v_n} \rangle$ defining the set of possible values for each variable $v \in \mathcal{V}$.
- $\mathcal{A}$ is the *instruction set*, i.e. the set of different instructions that can appear in a program.
- $\mathcal{T}$ is a set of input-output *test cases* where each test $t \in \mathcal{T}$ is a pair $t = \langle \mathcal{I}_t, \mathcal{G}_t \rangle$ indicating the input value for the program variables and the aimed output (i.e. after the program is executed) for these same variables.

A *solution* to TDD programming task is a program whose execution succeeds to solve every given test case.

2.2.2 The AI planning problem

An *AI planning problem* is formalized as a tuple $\langle V, D, A, I, G \rangle$ where:

- $V = \langle v_0, \ldots, v_m \rangle$ is the set of m *state variables* with finite domain. The respective domains $D = \langle D_{v_0}, \ldots, D_{v_m} \rangle$ define, for each variable $v \in V$, its set of possible values.
- A is a set of *actions* whose dynamics are specified with two functions:
  - $App(s) \subseteq A$, defines the subset of actions applicable in a state s.
  - $\theta(s, a) = s'$, defines the *successor state* s' that results of applying an action a in a state s.
- I is an *initial state*, i.e. a full assignment of values to the state variables.
- G is the set of *goal conditions* constraining the possible values of the state variables in the goal states.

A *solution* to an AI planning problem is a sequence of applicable actions such that its application, starting from the initial state, reach a state where all goal conditions are met.

2.2.3 Program synthesis and validation as AI planning

Here we introduce how to encode a *TDD programming task* as an *AI planning problem*. For further details on the encoding we refer the reader to [22, 12, 23, 10, 24, 11, 21].

Breifly, our encoding defines an *AI planning problem* with **state variables** of three kinds:

- `program(line) := instruction`, that encode the instructions (from the instruction set $\mathcal{A}$) at the different program lines. Initially all program lines are empty.
- `pcounter := line`, encoding which is the current program line. Initially the program counter points to the first program line.
- `var := value`, that encode the value of the program variables (the $\mathcal{V}$ set).

The **initial/goal states** of the AI planning problem encode the input/output values of the program variables (as specified by the test cases $\mathcal{T}$ of the TDD programming task). Finally, every program instruction ($w \in \mathcal{A}$) is encoded into the *AI planning problem* with two **actions**:

- *Programming actions* that assign an instruction in the *instruction set* to a given program line, i.e. change the value of variable `program(line)`.
- *Execution actions*, execute the instruction assigned to the current program line.

We implement this encoding using standard planning languages, such as PDDL [35], so the AIP tasks resulting from our encoding can be solved with off-the-shelf planners, like the FD planning system [25]. Our encoding is *complete* and *correct* [21], which means that the programs synthesized with this approach are guaranteed to be bug-free over the given set of *input-output* tests cases.

Interestingly, our PDDL encoding allows also program validation by (1), specifying the lines of the program to validate (i.e. the `program(line):=instruction` fluents) in the initial state of the AI planning problem and (2), disabling the mentioned *programming actions* so only *execution actions* are applicable.

2.3 Evaluation

The performance of our AI method for the synthesis and validation of programs will be evaluated with regard to (1) **computation time** and (2), **memory**. Further, the programming and validation tasks used for the evaluation of the performance of our approach are of two different kinds:

- *Theoretical benchmarks*: Classic programming tasks are a neat touchstone to assess the performance of our approach. For instance, programs for the computation of mathematical/logic series, string manipulation and for the management of data structures such as *lists*, *queues*, *stacks* or *trees*. Figure 1 shows a synthesized program for computing $y = \sum_0^N x$. The program is pictured as a *finite state machine*: Machine nodes mount to the different program lines while edges are tagged with a *condition/instruction* label, that denotes the condition (over the program variables) under which program instructions are taken. The program in Figure 1 assumes that the program variables are $\mathcal{V} = \{x, y, x = N, y = N\}$ and that the value of x and y is initially 0. The instruction set is $\mathcal{A} = \{x := x+1, y := y+1, x := x+y, y := y+x\}$. For the synthesis of this program we are using these five test cases $\{0 = \sum_0^0 x, 1 = \sum_0^1 x, 3 = \sum_0^2 x, 6 = \sum_0^3 x, 10 = \sum_0^4 x\}$ so the domains of variables x and y are $\mathcal{D}_x = \mathcal{D}_y = [0, 10]$. Note that $x = N$ and $y = N$ are Boolean variables whose value depends on the current value of x and y.

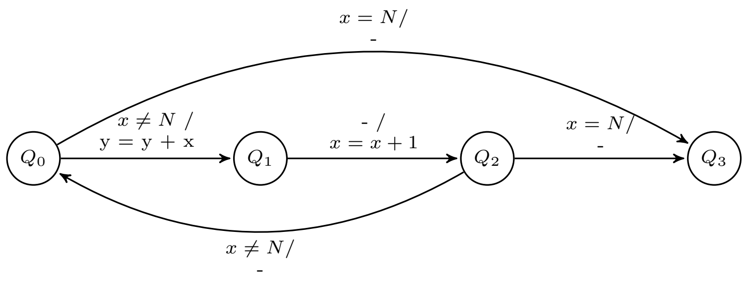

Fig. 1: Three-line program to solve the programming task of computing $y = \sum_0^N x$.

- *Real-world benchmarks*: We are involved in the four-year research project TIN2017-88476-C2-1-R from the *Spanish national plan* in which *AI Planning* and *activity recognition* is applied to different real-world domains such as *domotics*, *tourism*, *traffic control* and *robotics*. Interestingly many activities from these domains can be understood as simple programs. For instance, Figure 2 shows a five-line program (pictured as a finite state machine) that represents the sequence of instructions required for *making an orange juice*. We plan to evaluate our approach in synthesis and validation tasks coming from these real-world domains.

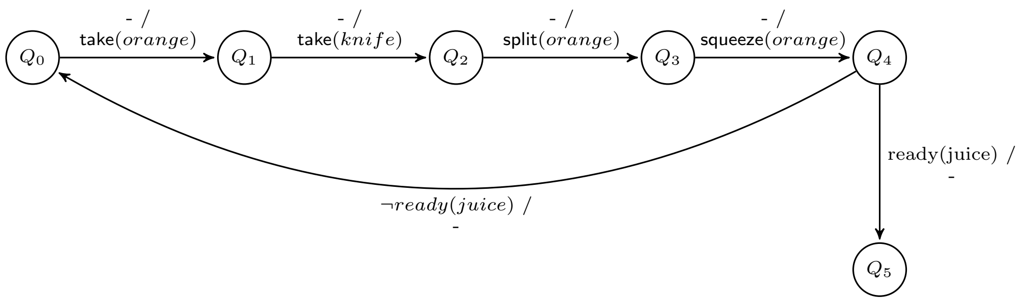

Fig. 2: Five-line program representing the *making an orange juice* activity from the *domotics* domain.

2.4 Specific objectives

We characterize the programs objects of study in this project by these three dimensions:

- Number of *program lines*.
- Size of the available *instruction set*.
- Number and domain of *observable* program variables. That is, the subset of program variables whose value can condition the program flow.

To illustrate this, the program of Figure 1 for computing $y = \sum_0^N x$ have three program lines, an instruction set that comprises four instructions and a single observable Boolean variable $(x = N)$. Likewise, the program of Figure 2, representing the activity of *preparing an orange juice* have five program lines, an instruction set with four instructions (namely $\mathsf{take}(orange)$, $\mathsf{take}(knife)$, $\mathsf{split}(orange)$ and $\mathsf{squeeze}(orange)$) and a single observable Boolean variable, `ready(juice)`, that holds when the orange juice is recognized to be ready.

The objective of this project is to study the performance of our approach (in terms of **computation time** and **memory**) in the synthesis and validation of *TDD programming tasks* that come from the two kinds of domains described in the previous subsection. This objective can be itemized into two specific objectives:

1. To study the performance of our Artificial Intelligence approach for the **synthesis of programs** up to: **10 program lines**, **10 observable variables** with binary domain and, an instruction set that comprises **10 instructions**.
2. To study the performance of our Artificial Intelligence approach for the **validation of programs** up to: **15 program lines**, **15 observable variables** with binary domain and, an instruction set that comprises **15 instructions**.

Note that despite setting bounds for the programs size and kind, challenging programming tasks can be addressed using *problem decomposition.* With this regard, our AIP encoding already supports callable procedures to decompose a given programming task into simpler modules and to enable recursive solutions [12, 10, 36, 11, 21].

3 Workplan and Budget

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 outputs a program source code that passes these tests with a bug-free guarantee.
2. The *program synthesizer* must also work in *validation a mode.* In this particular setting, the *program synthesizer* receives an additional input specifying the program source code and it outputs a *validation certificate* guaranteeing that the input program passes (or fails) at the given test cases.

Figure 3 details the proposed 24-month timeline for the project. A deliverable is provided at the end of each task.

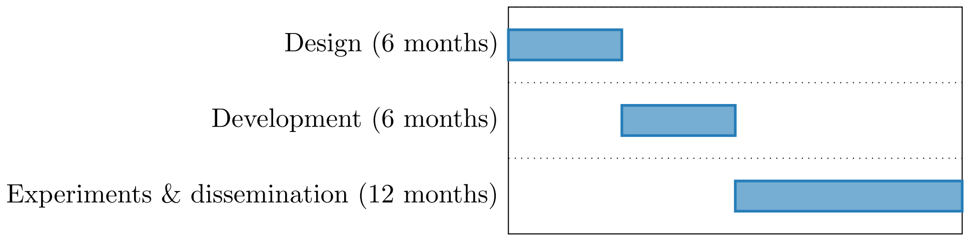

Fig. 3: Work-plan for developing a user-interactive program synthesizer/validator.

1. **T1. System design (months 1-6)**
   (a) Design of the test-case specification. Programming tasks are specified as a set of *input-output* tests cases plus the available instruction set.
   (b) Experimental design. Experiments will comprise taking time and memory measurements to evaluate the resources required by our approach to solve the given TDD programming/validation tasks.
   (c) Evaluation of the different AI planners available, with special attention to the planners that get the best results at the IPC-2018.

   Deliverable T1: 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 system 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 system component extracts the program code and the corresponding validation certificate from the solution plan produced by an off-the-shelf AI planner.

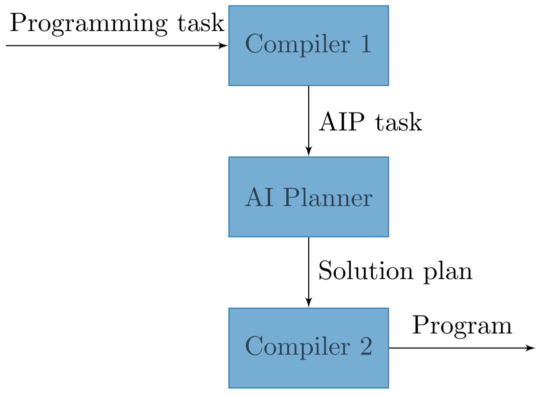

Fig. 4: System architecture for the synthesis and validation of TDD programs.

   Deliverable T2: Open repository with the source code of the system architecture and the corresponding benchmarks.
3. **T3. Experiments and dissemination of results (months 13-24)**.

(a) Reporting the experimental performance of our AIP approach for solving diverse TDD programming tasks. This task will follow an iterative workflow over the following subtasks:
   i. Executing the system architecture in the *theoretical benchmarks* described in Section 2.3.
   ii. Analysis and validation of the obtained results.
   iii. Tuning and repairing the system components, evaluation metrics and benchmarks according to the obtained results.
   iv. Executing the system architecture in the *real-world benchmarks* introduced in Section 2.3.
   v. Analysis and validation of the obtained results.
   vi. Tuning and repairing the system components, evaluation metrics and benchmarks according to the obtained results.

(b) Dissemination of the obtained theoretical and empirical results by submitting papers to top international conferences and journals in AI.

Deliverable T3: Final report with the obtained conclusions and produced publications.

3.1 Budget

Here we show the detailed desctiption of the two-year budget for the development of the proyect.

Priority	Description	Term	Euros
1	Difusión de las actividades del grupo	2018, 2019	2,000
2	Viajes, manutención y alojamiento grupo de investigación	2018	1,500
3	Viajes, manutención y alojamiento grupo de investigación	2019	1,500
4	Viajes, manutención, alojamiento y ponencias investigadores invitados	2018	1,500
5	Viajes, manutención, alojamiento y ponencias investigadores invitados	2019	1,500
			8,000

Tab. 2: Two-year budget for the development of the proyect.

4 Disemination and exploitation of the research results.

AIP has recently shown successful in *program testing* to generate *attack plans* that completed non-trivial software security tests [37, 38, 39, 40]. Promising research opportunities come from the application of AIP to *program synthesis* given that, *program synthesis* with a tests base, can be seen as the *program testing* dual.

In fact, our current work on *program synthesis* with AIP already produced several **publications at top international conferences and journals on Artificial Intelligence** [24, 23, 10, 12, 11, 21] 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 main benefit of this project is to provide new insights into the current understanding of how AI can assist programmers in the software development. Automated program synnthesis can produce huge cost savings in comparison to conventional programmming and increase the quality, trustbility and modifiability of programs. A longer term objective is the deeper and deeper embedding of intelligence in all manner of software systems, as well as the design and management of organizational processes by automated means.

In more detail, the expected benefits for this particular research project are four-fold:

1. A new **evaluation methodology for the *Synthesis and Validation of Programs***. The application of the exiting AIP technology to program synthesis can provide new evaluation metrics that assess how well a program covers a set of *input-output* test cases.

2. An empirical **study on the performance of the state-of-the-art AI planners for the *Synthesis and Validation of Programs***. 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 [41]. This project will help to meet the computational and expressiveness limits of AI planners when addressing real-world programming tasks.

3. The **development of open software and open benchmarks for the *Synthesis and Validation of Programs***. We strongly belief that reproducibility and open knowledge are essential to the advance of the research on computer science. With this regard, we plan to develop a *github* repository where we make available the developed source code and benchmarks.

4. **International dissemination of the obtained scientific results**. 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] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, *et al.*, "Satisfiability modulo theories.," *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.

[2] A. S. Lezama, *Program synthesis by sketching.* PhD thesis, Citeseer, 2008.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 1–8, IEEE, 2013.

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

[5] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," in *Readings in artificial intelligence and software engineering*, pp. 3–34, Elsevier, 1986.

[6] P. D. Summers, "A methodology for lisp program construction from examples," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 161–175, 1977.

[7] E. Y. Shapiro, *Algorithmic program debugging*, vol. 44. MIT press Cambridge, MA, 1983.

[8] S. A. Seshia, "Combining induction, deduction, and structure for verification and synthesis," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2036–2051, 2015.

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

[10] 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.

[11] J. Segovia-Aguas, S. Jiménez, and A. Jonsson, "Computing hierarchical finite state controllers with classical planning," *Journal of Artificial Intelligence Research*, vol. 62, pp. 755–797, 2018.

[12] 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.

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

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

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

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

[17] L. De Raedt, *Logical and relational learning*. Springer Science & Business Media, 2008.

[18] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[19] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*, pp. 25–60, Springer, 1993.

[20] J. A. Baier, C. Fritz, and S. A. McIlraith, "Exploiting procedural domain control knowledge in state-of-the-art planners.," in *ICAPS*, pp. 26–33, 2007.

[21] J. Segovia-Aguas, S. Jiménez, and A. Jonsson, "Computing programs for gener-alized planning using a classical planner," *Artificial Intelligence*, 2019.

[22] 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.

[23] 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.

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

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

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

[27] 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.

[28] 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.

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

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

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

[32] 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.

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

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

[35] 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.

[36] S. Jiménez, J. Segovia-Aguas, and A. Jonsson, “A review of generalized planning,” *The Knowledge Engineering Review*, 2018.

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

[38] 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.

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

[40] 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.

[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.