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 work on this particular research topic is the recipient of the *2016 distinguished paper award* at IJCAI (the main international conference on AI) and has recently been accepted for publication at the *Artificial Intelligence Journal*, the premier international journal for publishing research results in AI.

Keywords: Computer Science, Artificial Intelligence, Automatic Programming, Program Validation, AI planning.

1 Introduction

Program synthesis is the task of computing a program that satisfies a given formal specification. In 2008 the PhD Thesis work by Armando Solar-Lezama, at the University of California Berkeley, showed that it is possible to encode program synthesis problems in Boolean logic 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 [3]. To illustrate this, in 2013, a unified framework for program synthesis problems was proposed and since 2014 there is a yearly program synthesis competition, comparing the different algorithms for program synthesis in a competitive event (`http://www.sygus.org`). Further the US National Science Foundation is funding the ExCAPE research project[1], to transform the way programmers develop software by advancing the theory and practice of software synthesis (since 2012 the ExCAPE research project is funded with a total amount of $\$3,750,000$).

Now we review the two most successful approaches for automated program synthesis.

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

- **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 [4]. 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 [5]. The programs are computed with a domain-specific search that implements a divide and conquer approach.

- In **Programming by Sketching (PbS)** programmers provide a partially specified program, i.e. a program with the high-level structure of an implementation but that leaves low level details undefined to be determined by the synthesizer [6]. 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 [7].

On the other hand, *program validation* is the task of proving, or disproving, the correctness of a given program with respect to a certain formal specification or property. Program validation is considered a necessary step for program synthesis. *Model checking* is the mainstream AI approach for the formal validation of programs and controllers [8]. In model checking a given model of a system (the program or controller to validate), is exhaustively and automatically checked to verify whether this model meets a given specification. Typically the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Both the model of the system and the specification are formulated in a formal language. Current approaches for model checking reduces to graph search. Instead of enumerating reachable states one at a time, the state space can sometimes be traversed more efficiently by considering large numbers of states at a single step. For instance, representing set of states and transition relations as logical formulas or binary decision diagrams, like in *symbolic model-checking* [9].

Closely related to the aims of the project is the work on the automated synthesis of *Finite State Controllers* (FSCs) [10]. The state-of-the-art algorithms for computing FSCs follow a *top-down* approach that interleaves *programming* the FSC with validating it [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]). The validation and synthesis of programs from examples are research questions 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.

2 Methodology

This research project will **investigate the integration of *AI planning* into the *Test Driven Development* paradigm for the automatic synthesis and validation of programs**. Here we briefly introduce this technology:

- **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. 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, 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).

Our current research already shows that AI planners can synthesize programs for non trivial tasks like sorting lists, traversing graphs or manipulating strings [28, 12, 29, 11, 30, 31]. Table 1 reports the time invested by the AI planner FD [24] 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 and 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 AI planner FD [24] on a processor *Intel Core i5 3.10GHz x 4* and with a 4GB memory bound.

2.1 Synthesis and validation of TDD programs as AI planning

Given a *TDD programming* task, our approach is modeling and solving it as it were an *AI planning* task.

Briefly the **state variables** of this AIP task are *fluents* of the kind:

- `var:=value`, that encode the program variables.
- `program(line):=instruction`, that encode the instructions at the different program lines.
- `pcounter:=line`, encoding the current program line.

The **initial/goal states** of the AIP task encode the initial/final values of the program variables and are given by the input/output tests of the TDD programming task.

Finally, program instructions are encoded using two kinds of AIP **actions**:

- *Programming actions*, that assign an instruction to a given program line.
- *Execution actions*, execute the instruction assigned to the current program line.

We implement this encoding using standard planning languages, such as PDDL [32], so the AIP tasks resulting from our encoding can be solved with off-the-shelf planners, like the FD planning system [24]. The programs synthesized with this approach are guaranteed to be bug-free over given sets 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 AIP task and (2), disabling the mentioned *programming actions* so only *execution actions* are applicable.

2.2 Evaluation

The performance of our approach for the synthesis and validation of TDD programs will be evaluated in two kinds of programming tasks:

- *Real-world benchmarks*: The GRPS group is currently leading 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 in these domains can be understood as simple programs. For instance, Figure 1 shows a four-line program (pictured as a finite state machine) that represents the sequence of instructions required for the *making a buttered toast* activity. We plan to evaluate our approach in synthesis and validation tasks coming from these real-world domains.
- *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 2 shows a synthesized program for traversing a linked list. The program is pictured as a *finite state machine*: The machine nodes

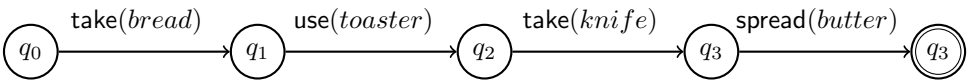

Fig. 1: Four-line program representing the activity of *making a buttered toast*.

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.

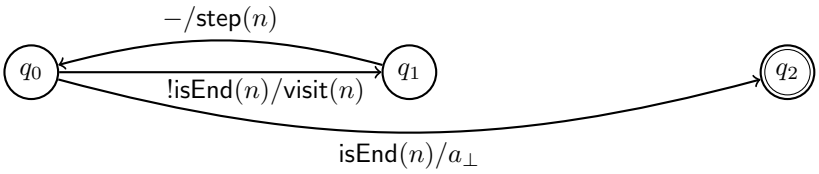

Fig. 2: Synthesized program to solve the programming task of traversing a linked list (n is a variable that points to a node of the linked list).

In both cases the performance of our approach will be evaluated with regard to the **computation time** and **memory** invested in the synthesize and the validation of the aimed programs.

2.3 Specific Objectives

The specific objective of this project is to study the performance of our approach (in terms of computation time and memory) for the automated synthesis and validation of programs that are characterized by these tree dimensions:

1. Number of *program lines*.
2. Number and domain of the observable *program variables*.
3. Size of the available *instruction set*.

For instance the program of Figure 2, for traversing linked lists, is generated using three program lines, one observable variable (isEnd(n) that has Boolean domain and indicates when n points to the end of the given linked list), and an instruction set that comprises two instructions (visit(n), that marks the list node assigned to n as *visited* and step(n), that advances n to the next node in the linked list).

In more detail, the two specific objectives of this project are:

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**.

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, 11, 30, 31].

3 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 source code that passes these tests with a bug-free guarantee.

In the particular case that the *program synthesizer* receives an additional input specifying the program source code, it outputs a validation certificate guaranteeing that the input program passes 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.

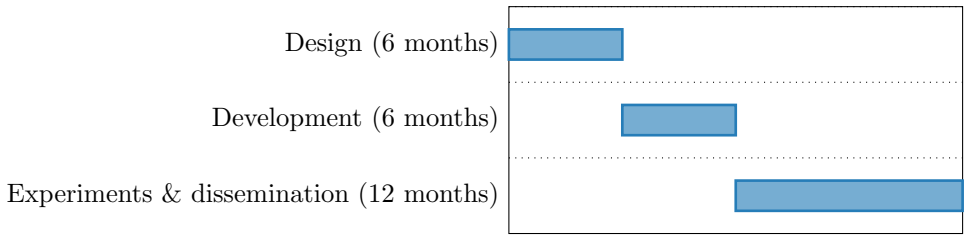

Fig. 3: Work-plan for developing a user-interactive program synthesizer/validator.

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.

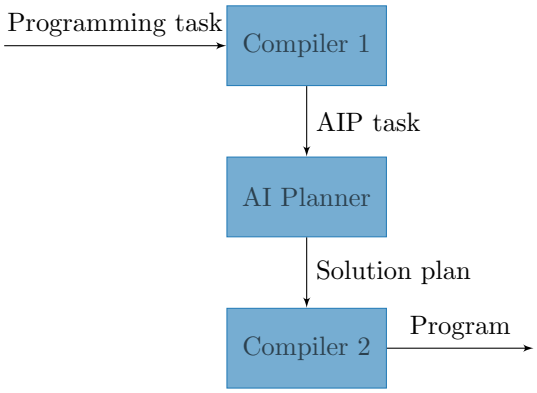

Fig. 4: System architecture for the synthesis and validation of TDD programs.

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 2.2.
      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 2.2.
      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 Workload

*Tasks (***T1-T3***)* will be developed by the three mentioned members of the research group: Sergio Jiménez, Eva Onaindia and Diego Aineto.

In addition, we plan to hire a master student for 6 months to assist in the completion of two tasks, **T3(a,i)** and **T3(a,iv)**. The detailed responsability of the student will be:

1. Executing the scripts of the system architecture.
2. Collecting the output data.
3. Presenting the output data in a easy-readable format.

3.2 Budget

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
2	Viajes, manutención y alojamiento grupo de investigación	2019	1,500
3	Viajes, manutención, alojamiento y ponencias investigadores invitados	2018	1,500
3	Viajes, manutención, alojamiento y ponencias investigadores invitados	2018	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 [33, 34, 35, 36]. 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** [37, 29, 11, 12, 30, 31] 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. 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 [38]. 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] T. M. Mitchell, "Generalization as search," *Artificial intelligence*, vol. 18, pp. 203–226, 1982.

[6] 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.

[7] 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.

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

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

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

[11] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

[38] 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.