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 recently, it is been 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 such that it satisfies a given formal specification of semantic 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* [?] to automatically compute programs [?]. 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 is funding the ExCAPE research project[1] (an amount of $3, 750, 000) to advance the theory and practice of program synthesis. In 2013 a unified framework for program synthesis was defined [?]. Since then, there is a yearly international competition (`http://www.sygus.org`) to compare the different approaches for program synthesis. Last but not least, program synthesis has already been deployed in the real world and it is part of the Flash Fill feature of Microsoft Excel that automatically generates programs for string transformation [?].

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

Algorithmic 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 [?]. On the other hand, inductive synthesis seeks to find a program matching a set of input-output examples by searching in a restricted space of programs [?, ?]. It is thus an instance of learning from examples, also termed as *inductive inference* or *Machine Learning* (ML). Many current approaches to synthesis blend induction and deduction [?]; syntax guidance is usually a key ingredient in these approaches.

Closely related to the aims of the project is the synthesis of *Finite State Controllers* (FSCs) [?]. The state-of-the-art algorithms for computing FSCs follow a *top-down* approach that interleaves *programming* the FSC with *validating* it [?, ?]. 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* [?], *conformant planning* [?], *CSP* [?] or a *Prolog program* [?]). The synthesis of programs from examples is also addressed in the classic AI field of *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 *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 given formal specification of its aimed semantics. 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 [?]. Current approaches for model checking reduces to graph search but, instead of enumerating reachable states one at a time, the state space is traversed 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* [?].

Program validation is also compilable into classical planning. Examples are the compilations for GOLOG *procedures* [?], *planning programs* [?], or *Finite State Controllers* [?, ?, ?]. Briefly, these compilations encode the *cross product* of a given planning instance and the automata corresponding to the program to validate. In this approach a *validation proof* is provided if a solution is found (the program is *correct*). Otherwise, if a classical planner proves that no solution exists, then the program is *incorrect* (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 the integration of *AI planning* into the *Test Driven Development* paradigm for the automatic synthesis and validation of programs**. In more detail, we use *test cases* to specify the semantic of the aimed programs, as in Test Driven Development. Then, we encode the *program synthesis* (and *program validation*) tasks as AI planning problems and finally, we use off-the-shelf AI planners to compute solutions to these problems.

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 [?, ?, ?, ?, ?, ?, ?]. Table 1 reports the time invested by the AI planner FD [?] 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 [?] 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 [?]. 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 [?]. The mainstream approach for AIP is *heuristic search* with heuristics derived automatically from the problem representation [?, ?]. Current planners add other ideas to this like *novelty exploration* [?], *helpful actions* [?], *landmarks* [?], and *multiqueue best-first search* [?] for combining different heuristics.

- **Test driven development (TDD)** [?] is a popular paradigm for software development that is frequently used in *agile methodologies* [?]. 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 Synthesis and validation of programs as AI planning

Here we explain the details of our AI method for program synthesis and validation. Given a *TDD programming* task, our approach is modeling and solving it as it were an *AI planning* problem.

2.2.1 The AI planning problem

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

- $V = \langle v_0, \ldots, v_n\rangle$ is the set of n *state variables* with finite domain. The respective domains are $D = \langle D_{v_0}, \ldots, D_{v_n}\rangle$ defining, 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.
  - $\theta(s, a) = s'$, that 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* bounding 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.

2.2.2 The TDD programming task

We formalize a *TDD programming task* as a programming task whose semantic correctness is specified with a set of input-output *test cases.* A *TDD programming* task is a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{T}\rangle$ where:

- $\mathcal{V}$ is the set of m *program variables.* The respective domains are $\mathcal{D} = \langle \mathcal{D}_{v_0}, \ldots, \mathcal{D}_{v_m}\rangle$ 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 $t = \langle \mathcal{I}_t, \mathcal{G}_t\rangle$ that represent the input value for the program variables and the aimed output for these variables.

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

2.2.3 Program synthesis and validation as AI planning

Here we describe how to encode a *TDD programming task* task as an AI planning problem. Our encoding defines **state variables** of three kinds:

- `program(line) := instruction`, that encode the instructions (from the instruction set $\mathcal{A}$) at the different program lines. Initialy 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 AIP task encode the initial/final values of the program variables. These values are specified by the test cases $\mathcal{T}$ of the TDD programming task. Finally, every program instruction ($w \in \mathcal{A}$) is encoded with two AIP **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 [**?**], so the AIP tasks resulting from our encoding can be solved with off-the-shelf planners, like the FD planning system [**?**]. Our encoding is *complete* and *correct* [**?**], which means that 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.

2.3 Evaluation

The performance of our AI method for the synthesis and validation of TDD programming tasks will be evaluated with regard to (1) **computation time** and (2), **memory** invested in the synthesize and the validation of the aimed programs.

The programming tasks used for the evaluation 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*: The 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 $\mathcal{A} = \{x = x + 1, y = y + 1, x = x + y, y = y + x\}$. The input test cases are $\{0 = \sum_0^0 x, 1 = \sum_0^1 1, 3 = \sum_0^2 x, 6 = \sum_0^3 x, 10 = \sum_0^4 x\}$ so the domains of variables x, y is $\mathcal{D}_x = \mathcal{D}_y = [0, 10]$ ($x = N$ and $y = N$ are Boolean variables whose value depends on the current value of x and y).

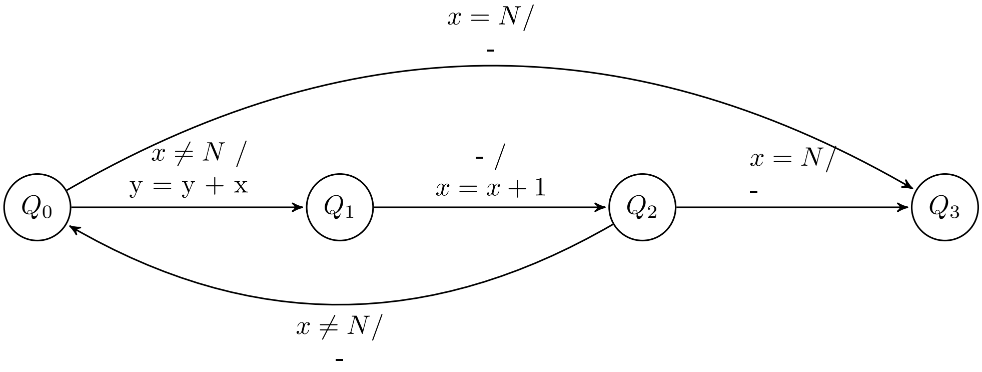

Fig. 1: Three-line program to solve the programming task of computing $y = \sum_0^N x$.

- *Real-world benchmarks*: 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 2 shows a five-line program (pictured as a finite state machine) that represents the sequence of instructions required for the activity of *preparing an orange juice*. We plan to evaluate our approach in synthesis and validation tasks coming from these real-world domains.

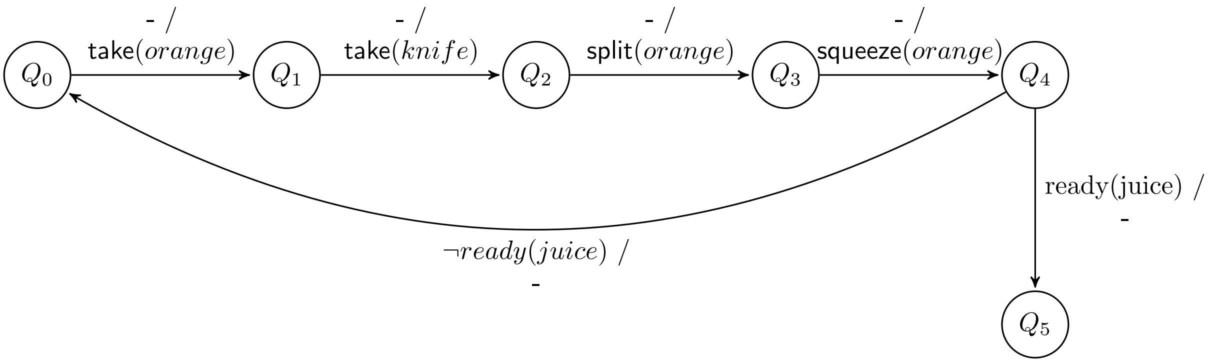

Fig. 2: Five-line program representing the activity of *preparing an orange juice*.

2.4 Specific objectives

We characterize the programs objects of study in this project 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.*

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

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 TDD programming tasks coming from the two kinds of domains described in the previous subsection. In further 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 [**?**, **?**, **?**, **?**].

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 (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.

1. **T1. System design (months 1-6)**

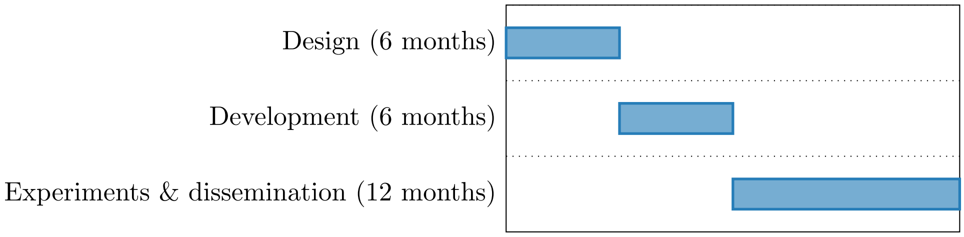

Fig. 3: Work-plan for developing a user-interactive program synthesizer/validator.

(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.

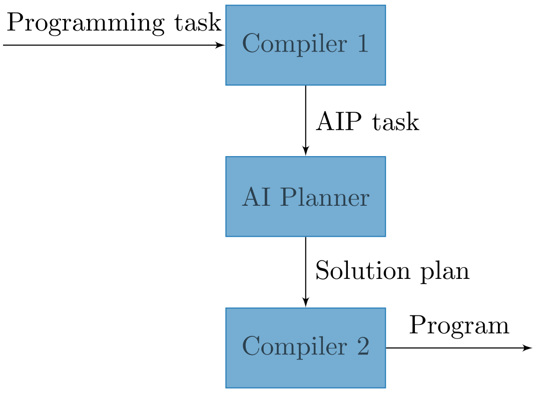

Fig. 4: System architecture for the synthesis and validation of TDD programs.

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 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

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 [**?**, **?**, **?**, **?**]. 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** [**?**, **?**, **?**, **?**, **?**, **?**] 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 [**?**]. 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).