Science of Computer Programming 201 (2021) 102536

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A family of multi-concept program synthesisers in Alloy*

Alexandre Correia [a,b,*], Juliano Iyoda [b], Alexandre Mota [b]

[a] Instituto Federal Sertão Pernambucano, Petrolina-PE, 56.316-686, Brazil
[b] Centro Informática, Universidade Federal de Pernambuco, Recife-PE, 50.740-560, Brazil

ARTICLE INFO

Article history:
Received 16 July 2019
Received in revised form 21 August 2020
Accepted 24 August 2020
Available online 2 September 2020

Keywords:
Program synthesis
Alloy*
SAT solving
Genetic Algorithm

ABSTRACT

Program synthesis aims to mechanise the task of programming from the user intent (using pre and post condition, examples and sketches). There are many approaches (or concepts) in program synthesis that are usually implemented in isolation: deductive, syntax-based, inductive, etc. In this paper, we present a characterisation of program synthesis as model finding, using Alloy*. Such a characterisation unifies several of these concepts in a single model. Through model finding, we obtain a general framework for rapid development of a program synthesiser accommodating denotational semantics based synthesis, simultaneous deductive and inductive synthesis, software reuse, syntactic ingredients (the Alloy* scope of entities), and a new one: a *soft sketch* (a set of commands that must appear in the synthesised program but in no particular order of execution). Our family of synthesisers produce general purpose programs in the Java language. As the Alloy* synthesiser requires several rounds of user assistance to set scope, sketches, etc., particularly for complex problems, we integrated the model finder to a genetic algorithm module, where candidate solutions and user inputs are generated and mutated automatically. We carried out empirical evaluations on program synthesis successfully. As results, we verified that: (i) we can synthesise thirteen programs (`Maj5`, `Maj8`, `IntSQRT`, `Max4`, `Modu`, `Fact`, `Fib`, `aMax`, `aDouble`, `aSum`, `eCount`, `aBubSort`, `aSelSort`); (ii) inductive synthesis was faster than deductive synthesis; (iii) synthesis with reuse was faster; and (iv) Genetic Algorithm is better than user trial and error approach.

1. Introduction

Program synthesis typically performs a search over the space of programs to find a source code that is consistent with a variety of constraints (for instance, input-output examples, specifications, and incomplete programs — or sketches [1]). Program synthesis is considered the holy grail of Computer Science since the beginning of Artificial Intelligence in the 1950s [2]. Automatic program synthesis is gaining attention nowadays thanks to the advances in Artificial Intelligence and SAT/SMT theories (together with the development of efficient solvers). Such advances have led to recent developments in different trends in program synthesis like *deductive synthesis* and *inductive synthesis*.

In deductive synthesis, a program that satisfies a formal specification must be produced by deduction. One way to do that is to model the problem of program synthesis as a SAT/SMT problem so that a solver is employed to automatically find a program that satisfies a formal specification [2]. As a SAT/SMT solver can now check the satisfiability of very complex logical expressions in reasonable time, fully automatic deductive synthesisers have currently produced promising results [3–6].

* Corresponding author.
E-mail addresses: alexandre.correia@ifsertao-pe.edu.br (A. Correia), jmi@cin.ufpe.br (J. Iyoda), acm@cin.ufpe.br (A. Mota).

https://doi.org/10.1016/j.scico.2020.102536
0167-6423/

In the last decade, a new trend in program synthesis has appeared. Several applications of synthesis in the so called programming-by-examples field have been deployed in mass-market industrial products [7–9]. This kind of synthesis is known as inductive synthesis, where the formal specification is replaced by a set of input-output examples. The usage of examples instead of specifications comes from the Artificial Intelligence culture [10–13].

In our previous study [14], we presented a deductive synthesiser which was built on top of the Alloy* [15] model finder. Here, the term *deductive synthesiser* means that the user intention is represented by a pre and post condition (like in a deductive approach). Alloy* subsequently tries to find a program that satisfies this contract. Note, however, that Alloy* is, at the end, always inductive. By *inductive synthesiser* we mean the usual definition: input and output examples are generalised into programs.

The syntax and the denotational semantics of Winskel's IMP (erative) language [16] were our starting point to build an Alloy*-based Program Synthesiser (APS for short), whose syntax and semantics are based on Alloy language elements (such as signatures, relations, predicates, etc). We then applied the Alloy* model finder (the Alloy* Analyser) to search for a program that satisfies a contract written in terms of pre and post condition. Alloy* is a variant of the Alloy [17] model finder that implements a convenient Counter Example-Guided Inductive Synthesis (CEGIS) [18] algorithm. This facility, in combination with the Alloy* modelling language, made the construction of the Alloy* synthesiser interesting. By using the primitive implementation of CEGIS and the high-level modelling language of Alloy*, we could encode the synthesis problem at a very abstract level (synthesis as a search over the states constrained by a denotational semantics). Our Alloy*-based Program Synthesiser (APS) was able to synthesise high-level program constructs like sequential compositions, if-then-elses and while loops. Such results were similar to the synthesiser developed by Srivastava et al. [4], where a SAT/SMT solver was used directly. Alloy* is related to a large domain of applications such as program verification, software testing, fast prototyping, as well as teaching. Our goal is to exploit program synthesis in these domains in the future.

As Alloy* provides us with this platform for a rapid development of a synthesiser, the original synthesiser was easily extended and adapted to deal with examples instead of specifications (that is, instead of using a formal contract we can just use input-output examples). This allowed us to produce an inductive programming-by-example synthesiser with an almost effortless endeavour. The inductive version of the synthesiser was achieved by realising that Alloy* treats a set of input-output examples just as any regular constraint. The inductive version modified 4 lines of the deductive model and added 5 new lines. This new version takes as input not only examples but also *sketches* (programs with holes) [1]. It also possesses the ability to reuse functions by simply reusing the way the language was embedded into Alloy*. For instance, a program that returns the maximum of 3 numbers can be implemented by calling the maximum of 2 numbers twice: `max(x,max(y,z))`. With the ability to call functions, it can (hopefully) avoid doing all programs from scratch.

In the attempt to synthesise more real-world related problems, we introduced `arrays` in the Alloy* specification. With this, we were able to synthesise well-known sorting algorithms like Bubble sort and Selection sort. In terms of arrays, there are synthesisers capable of: (i) mapping functions to search/sort over lists [4,19,20]; (ii) handling strings by extracting specific data [21–23]; or (iii) simulating finite size arrays [24]. APS outputs arrays written as Java's source code, such as:

```
int i = 0;
int[] myArray = new int[3];
while(i<3) {
  myArray[i] = 2 * i;
  i++;
}
```

In order to reduce the repetitive tasks of interacting with Alloy* to change the scope of the model entities and the task of verifying the correctness of the synthesised program, we enveloped APS around the concepts of Genetic Algorithms [25]. This extension gave rise to the Alloy* Program Synthesiser with Genetic Algorithm (APS-GA, for short), a variant of the APS that can generate user inputs, mutate candidate solutions, and automatically test the results, thus reducing both the user mental effort and amount of interactions. APS-GA was first introduced in our previous work [25]. In this article we describe it in more detail.

To explore the inductive aspect of APS, the user provides a small set of examples (to guide the synthesiser on producing candidate programs) and a larger set of test cases that is as comprehensive as possible to capture all expected input and output values (in order to enable APS-GA to check whether the solution is correct). Examples must be smaller than test cases in order to prevent state explosion. We detail this issue in the following sections.

By providing only examples and test cases as inputs, the synthesiser can be effective for simple problems. Although our sketches are optional, complex cases need them in general. Many synthesisers use sketches to guide and speed up the search for a solution. Our synthesiser is no different than the others but uses a more flexible solution. Our sketches are called *soft sketches* because they represent a set of commands that must appear in the synthesised program, but in no particular order (the order of execution of a traditional sketch is hard coded). It is worth noting that a soft sketch can also define (optionally) pieces of hard code as well. It is up to the user's needs. Also, pieces of soft sketches can occasionally

occur more than once in the synthesised program if needed. Such a feature is impossible with a traditional sketch as it denotes a fixed template of the synthesised program.

Moreover, the Alloy* scope is produced automatically. This scope defines the amount of operations and commands that must appear in the synthesised program. For instance, a user can inform Alloy* to search for a program that contains exactly one while loop and at most 5 assignments. This information, which we call *syntactic ingredients*, was provided interactively by the user when using the APS, but is generated automatically with the aid of Genetic Algorithm (APS-GA). Finally, with Genetic Algorithm we also verify automatically (by producing Java programs and running tests on them) whether a synthesised program is correct or not with respect to the test cases provided. All these features reduce both the mental effort of the user and the interactions needed. With Genetic Algorithm, we were able to synthesise thirteen programs: integer square root (`IntSQRT`), majority of 5 (`Maj5`), majority of 8 (`Maj8`), maximum of 4 (`Max4`), modulo (`Mod`), factorial (`Fact`), Fibonacci (`Fib`), maximum element of an array (`aMax`), assignment `a[i] = 2*i`, for each `i` and array `a` (`aDouble`), sum of all elements of an array (`aSUm`), counting of occurrences of a given number in an array (`eCount`), Bubble Sort (`aBubSort`) and Selection Sort (`aSelSort`). They are found in the SyGuS competition [26], iJava and IntroClass, and Genetic programming communities.

In summary, the main contributions of this article are:

- The characterisation of synthesis of general-purpose programs as model finder that acts as a multi-concept platform for rapid development of a program synthesiser that features both inductive and deductive synthesis simultaneously;
- An empirical evaluation showing that an inductive synthesis is as efficient as (and, most of the times, better than) a deductive synthesis;
- The introduction of a synthesiser that, after a simple adjustment in the Alloy* specification, is able to: (i) reuse functions, where we have to define their syntax and well-formedness rules, and define their semantics in the Alloy* relational language; and (ii) deal with arrays, where we have to define the arrays themselves (their names and contents in terms of Alloy* sequences), the array well-formedness rules and an updated syntax and semantics for expressions and assignments as they were affected by the inclusion of arrays;
- An empirical evaluation about the performance of the synthesis of 7 well-known non-array programs (`IntSQRT`, `Maj5`, `Maj8`, `Max4`, `Mod`, `Fact` and `Fib`) and 6 well-known array programs (`aMax`, `aDouble`, `aSUm`, `eCount`, `aBubSort`, and `aSelSort`);
- An empirical analysis measuring the time taken: (i) to translate from an Alloy* model to a SAT solver language, (ii) to find a solution, and (iii) to synthesise with reuse. Results showed that deductive synthesis takes less time to translate to the SAT language than inductive synthesis but takes longer to find a solution, and that synthesising with reuse is faster.

This article is organised as follows. Section 2 presents an Alloy* specification (syntax, well-formedness rules, semantics and the synthesis predicate) corresponding to a deductive synthesiser. Section 3 shows how to extend this specification to implement an inductive synthesiser as well as the facility of reusing functions. Section 4 describes an extension to deal with arrays. Section 5 augments the Alloy*-based Program Synthesiser (APS) with Genetic Algorithm aspects (APS-GA) and other facilities to ease the user experience. Section 6 performs an empirical evaluation discussing: deductive vs inductive approaches; reuse; the use of Alloy* model combined with Genetic Algorithm strategy; programs with arrays, and the effectiveness of the Genetic Algorithm. Section 7 compares our work with some related studies, and Section 8 concludes and addresses future work.

2. An Alloy* deductive program synthesiser

Our Alloy*-based synthesiser was inspired by Winskel's IMP(erative) language [27] (syntax and the denotational semantics). In this section we present its syntax, well-formedness rules (defined by ourselves), and its semantics as well as a synthesis predicate in Alloy*.

2.1. The syntax of IMP in Alloy*

In his book [27], Winskel defines a set of locations (or memory addresses) by Loc. In Alloy*, an abstract signature **sig** defines such a given set.

```
abstract sig Loc {}
```

We define another abstract syntactic class to represent commands (Cmd).

```
abstract sig Cmd {}
```

Concrete commands are defined as subclasses of Cmd, such that they partition Cmd in distinct subclasses. The most basic is Skip, which is the command that does nothing.

```
lone sig Skip extends Cmd { }
```

The relation **extends** Cmd establishes an inheritance relationship between Cmd and Skip. In this case, **extends** models the grammatical dependency between the non-terminal Cmd and the terminal Skip. The term **lone** constrains the signature to have at most one instance.

An arithmetic expression AExp is either an integer constant (IntVal), integer variable (IntVar), addition (Add), subtraction (Sub) or multiplication (Mult).

```
abstract sig AExp { }
sig IntVal extends AExp { val: one Int }
sig IntVar extends AExp { name: one Loc }
sig Add extends AExp { op1: one AExp, op2: one AExp } ...
```

Each class of expression has fields. For instance, an IntVar expression contains the field name that belongs to Loc, and the fields op1 and op2 of an addition are its operands. Subtraction and multiplication are defined similarly to addition.

An assignment $X := a$ is a command whose left-hand side is an integer variable and the right-hand side is an arithmetic expression.

```
sig Assign extends Cmd { lhs: one IntVar, rhs: one AExp }
  { (IntVar<: rhs) ≠ lhs and rhs ∉ IntVal and lhs.name ∉ XLoc }
```

Let `R` be a relation and `S` be a set. The expression `S <: R` filters `R` with respect to `S`. For instance, `(IntVar <: rhs) != lhs`, means that any `IntVar` signature (integer variable) that appears in the `rhs` relation (right side hand of an expression, such as assignment) does not occur in the `lhs` relation (left side hand of an expression). This semantic rule prevents a Java source-code like `anyVar = anyVar;` to be produced. This signature has some restrictions. The right-hand side: (i) must be different from the variable on the left-hand side; (ii) cannot be a constant (`rhs ∉ IntVal`) and (iii) must not be a read-only variable (`XLoc` extends `Loc` and denotes read-only variables).

The sequential composition of commands curr (for the current command) and next becomes the entity SComp.

```
sig SComp extends Cmd { curr, next: one Cmd }
```

A Boolean expression BExp is encoded in the form lhs OP rhs, where lhsand rhs are arithmetic expressions such that they must be different from auxiliary variables (ALoc). The ALoc signature extends a Loc signature to denote auxiliary variables that are forbidden to be used in conditions. This helps the synthesiser in reducing the state space.

```
sig ALoc extends Loc { }
```

An operator OP ∈ {EQ, NEQ, LEQ, LTH, GEQ, GTH} where EQ, NEQ, LEQ, LTH, GEQ, and GTH denote $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$, respectively.

```
abstract sig BExp { lhs, rhs: one AExp }
                  { lhs.name ∉ ALoc and rhs.name ∉ ALoc }
sig EQ extends BExp { } ...
```

The operators NEQ, LEQ, LTH, GEQ, and GTH are defined in a similar way to EQ.

A conditional statement **if** b **then** C_0 **else** C_1 becomes the entity CondS, where we had to change **else** to elsen because the former is a reserved Alloy* keyword.

```
sig CondS extends Cmd { cond: one BExp, then, elsen: one Cmd }
                      { then ≠ elsen }
```

We restrict then to be different from elsen to prevent the synthesis of commands of the form **if** b **then** C **else** C, which is the same as C [28]. This restriction reduces our search space.

The most difficult statement to encode is the while statement. In addition to the usual Boolean condition (cond) and body (wbody), we consider an auxiliary structure (unfold) to unfold (or "run") the body over the iterations.

```
1  sig While extends Cmd {
2      cond: one BExp, wbody: one Cmd, unfold: set Expansion }
3      { (#unfold ≥ 2 ⇒ (∀ disj e1,e2: unfold •
4          e1.exp.first.curr.bind ≠ e2.exp.first.curr.bind)) }
5  sig Expansion { exp: seq StChg }
6      { #exp = #exp.elems and (∀i: exp.inds • i ≠ exp.lastIdx
7          ⇒ exp[i].next = exp[add[i,1]].curr) }
8  sig StChg { curr, next: one State }
```

A While is a command (Cmd) composed of a condition (cond), the body of the while (wbody), and a set of expansions (Expansion) of that body (unfold) (Line 2). The expansion denotes the minimum repetition of the body of the while before its condition becomes false. We require at least two expansions (#unfold ≥2), such that two disjoint expansions do not reuse the same starting state to expand the while body (Lines 3 and 4). An Expansion is a sequence $\langle(s_0, s_1), (s_1, s_2), ..., (s_{n-1}, s_n)\rangle$ of state changes (Line 8) where s_i is a state (Lines 5 and 7). The size of an expansion exp is the length of its sequence of state changes exp.elems (Line 6). Each pair (s_i, s_{i+1}) denotes a state changing StChg (Line 10) from s_i to s_{i+1} by running the body of the loop one time. A state change at position i must have its next state equals to the current state at position i+1 (Lines 6 and 7). Thus, instead of finding the semantics of while by a fix-point computation, we ask Alloy* to produce expansions that correspond to such a computation.

A program is a unique atom from which all other statements must be linked to (or reachable from).

```
one sig Prog { body: one Cmd }
```

2.2. The well-formedness of IMP in Alloy*

Signatures do not fully capture the grammar of the IMP language. Thus we need to add some extra constraints to assure the synthesis of well-formed programs.

A program must have all its locations associated to a variable name.

```
∀v: Loc • v ∈ IntVar.name
```

The remaining constraints are listed below. For conciseness we omit their Alloy* code.[1]

- All commands must belong to the body of the synthesised program;
- All relational operators must belong to conditions of an if-then-else or a while;
- All arithmetic expressions must be in the right-hand side of an assignment;
- An illegal cycle must not be produced. An illegal cycle is a program that executes the command c and, without being in the scope of a while loop, returns to c.

2.3. The semantics of IMP in Alloy*

Another Alloy* characterisation of IMP is the model of its denotational semantics. Except for the conditional and while statements, the embedding of the semantics comes directly from the semantics of IMP.

The semantic clauses are defined in terms of the relations evalC, evalA and evalB that evaluates a command, an arithmetic expression and a Boolean expression, respectively, based on a corresponding state.

A state has a field bind that maps a location to an integer.

```
sig State { bind: Loc → one Int }
```

The relation evalC relates a command, an initial state and a final state. The relations evalA and evalB relate an expression e, a state s, and the value of e in s.

The final state of Skip is equal to its initial state. Thus the semantics of Skip is an identity relation.

```
evalC[Skip] ∈ iden
```

An assignment overrides the binding of the initial state with the new binding established by the assignment.

[1] All details can be found in the work by Maranhão [29] (Chapter 3).

```
∀c: Assign , iSt: evalC[c].univ • evalC[c][iSt].bind = iSt.bind ++ {c.lhs.name→evalA[c.rhs][iSt]}
```

The override operator ++ updates the initial binding (iSt.bind) with the mapping from the left-hand side c.lhs.name to the value of the right-hand side evalA[c.rhs][iSt] of the assignment.

A sequential composition is the composition (denoted by . in Alloy*) of the final state of the first command with the initial state of the second command.

```
∀c: SComp • evalC[c] = (evalC[c.cur]).(evalC[c.next])
```

The semantics of the conditional command is defined below.

```
∀c: CondS, iSt: evalC[c].univ •
    (evalB[c.cond][iSt] = BitTrue ⇒
        evalC[c.then][iSt] = evalC[c][iSt]
        else evalC[c.elsen][iSt] = evalC[c][iSt])
```

If the condition evaluates to true, the final state is the final state of the then branch (evalC[c.then][iSt]). Otherwise, the final state is the final state of the elsen branch (evalC[c.elsen][iSt]).

Concerning the semantics of a while loop, the first point is the evaluation of its condition in the initial state (evalB[c.cond][iSt]). If it evaluates to false (BitFalse), then nothing happens: the final state equals the initial state (Line 2). Otherwise (Line 4 to 10), the body of the loop expands (or "runs"). Such expansions must satisfy the following properties: (i) the first state of the expansion (st.exp.first.curr) is the initial state (Line 4); (ii) each next state (st.exp.last.next) must correspond to the evaluation of the current state (evalC[c.wbody][ic.curr]=ic.next) (Line 6); (iii) each intermediate iteration evaluates its condition to true (Line 7); and, (iv) the final state (ic=st.exp.last) has the last state (condition) false (Line 8), otherwise true (Line 9). The last clause assures that the initial states of all expansions are valid (Line 10).

```
1  ∀c: While, iSt: evalC[c].univ •
2    (((evalB[c.cond][iSt]=BitFalse ⇒ evalC[c][iSt]=iSt
3    else
4      (one st: c.unfold • st.exp.first.curr=iSt and
5      evalC[c][iSt]=st.exp.last.next and
6      (∀ic: st.exp.elems • evalC[c.wbody][ic.curr]=ic.next and
7      evalB[c.cond][ic.curr]=BitTrue and
8      (ic=st.exp.last ⇒ evalB[c.cond][ic.next]=BitFalse else
9          evalB[c.cond][ic.next] = BitTrue)))))
10   and (∀ st: c.unfold.exp.first.curr • evalB[c.cond][st] = BitTrue))
```

2.4. The synthesis predicate of IMP in Alloy*

The following predicate is an adaptation of the predicate reported by Milicevic et al. [30] in which we take into account states and state changes.

```
1  pred Synt[p: Prog] {
2      ∀iSt: State, evalC: Cmd → (State → lone State),
3          evalB: BExp → State → Bit,
4          evalA: AExp → State → Int
5          when { PreC[iSt] and Semantics[evalC, evalB, evalA] }
6          { PosC[iSt, evalC[p.body][iSt]] }
7  }
```

The synthesis predicate is defined in terms of a pre condition (PreC), a post condition (PosC) and the semantics (Semantics). The semantics defines the behaviour of the evaluation functions. The pre and post condition are predicates that are instantiated for each case study. This predicate states that we want a representative program (`p: Prog`) such that in all initial state (`iSt`), which satisfies the precondition of the problem (`PreC[iSt]`) and the semantics of the modelled language (`Semantics[evalC, evalB, evalA]`), its postcondition holds in the final state given by `evalC[p.body][iSt]`. More details about these predicates can be found in the work by Maranhão [29].

3. Extending the Alloy* model towards inductive synthesis and reuse

Following the study by Gulwani et al. [2], creating formal specifications (deductive synthesis) needs an effort comparable to creating the source code itself. Thus inductive synthesis seemed to be an interesting starting point to users with few (or

even none) programming skills. It is worth observing that by *deductive* we mean a synthesis based on a formal contract. Thus, in this work, the term *deductive* applies only to the style of the user intent, namely, pre and post condition. But internally, the Alloy* Analyser engine uses instances of entities to find a solution instead of deducting it from axioms and inference rules. That is, from a formal contract, the Alloy* Analyser instantiates all input and output (I/O) examples (in a finite domain) to find a solution. On the one hand, the computational effort is considerably high to find all these examples. On the other hand, a classical deductive system needs human guidance.

We extend our Alloy* model towards inductive synthesis by simply adapting the state to take into account already provided input-output examples.

```
sig IState extends State {}
sig FState extends State {}
one sig Example {
  pairs: IState one → one FState
}
```

The synthesis predicate changed slightly to deal with the provided I/O examples from the user (instead of a formal contract). In summary, we threw away the pre and post condition and created a new predicate named `Exmp` (Line 6). Also, for each synthesis, a soft-sketch is written down into the new predicate `Exmp` (Lines 10 to 12).

```
1  pred Synt[p: Prog] {
2      ∀ iSt: IState, evalC: Cmd → (State → lone State), evalB: BExp → State → Bit,
3          evalA: AExp → State → Int when {
4              Semantics[evalC, evalB, evalA]
5          }{
6              Exmp[iSt, Example.pairs[iSt]]
7          }
8  }
9  ...
10 pred Exmp[iSt: one IState, fSt: one FState] {
11     // for each synthesis problem the user has to
12     // provide examples or soft-sketches here.
13 }
```

This simple adaptation allowed us to see another benefit of inductive synthesis that may be even more worthy than the fact that examples[2] are easier to be provided than a formal contract.

3.1. Reuse of functions calls

Another aspect that is used by some synthesisers is the reuse of function calls. Inspired by the study by Polozov and Gulwani on PROSE [7], we noticed that it is basically built on top of reuse. Our study instead was based on the synthesis from scratch of the IMP language by Winskel [16]. These functions are, in general, domain specific (for example, to handle strings). Thus, the user may have an idea of what it is needed.

So we decided to investigate what happens when we allow the reuse of function calls instead of always trying to synthesise from the primitive commands originally available in the IMP language. In this section we illustrate the case of the Max3 problem.

By using our original formulation we synthesised a code like this (with **no function calls**):

```
1  int Max3(int num1, int num2, int num3) {
2      int max;
3      if(num1 > num2) {
4          if(num1 > num3) { max = num1; }
5          else { max = num3; }
6      }
7      else {
8          if(num2 > num3) { max = num2; }
9          else { max = num3; }
10     }
11     return max;
12 }
```

[2] An example for us is a pair (input, output). Although we agree with Gulwani *et al.* [2] that examples are easier to produce than contracts, we have not performed a controlled experiment with real users in order to assess the impact of using the kind of examples we adopt.

Then, we provided the synthesiser with a new syntactical construct named `Max2`:

```
sig Max2 extends Cmd {
     arg1: one Loc, arg2: one Loc, res: one Loc
  }
  {  arg1 ≠ arg2 and res ∉ XLoc }
```

`Max2` semantics is defined as:

```
∀ c: Max2, iSt: evalC[c].univ •
       evalC[c][iSt].bind = iSt.bind ++
           { c.res → max[iSt.bind[c.arg1] + iSt.bind[c.arg2]] }
```

Then we have got the following code (that is, a code **with reuse**), which is more readable than the previous one. As we already pointed out in the beginning of this section, reuse is domain related and then each domain has its own specific opportunities.

```
int Max3(int num1, int num2, int num3) {
    return Max2(Max2(num1, num2), num3);
}
```

The syntactic ingredients used to get a reused-solution shown before were the following: three `IntVar` that represent those 3 variables (`num1, num2, num3`); and two `Max2`, the new syntactic construct that represent reuse. All those remaining syntactic ingredients (detailed in Section 5) were not mentioned in this example because they were set to automatic mode. It is worth observing that the synthesis becomes faster with reuse because we direct the search to try to solve the problem with the reusable syntactic ingredients first. If that reuse is not possible at first, then the synthesis continues with the other syntactic ingredients.

We used this facility when dealing with the synthesis involving arrays (Section 4). We simply noticed that all sorting algorithms need a `swap` statement as a composition of three assignments over three variables (a traditional solution to swapping the content of two variables, say `X` and `Y`, uses a temporary variable, say `T`, to avoid losing the initial value of one of the variables).

```
T := X;
X := Y;
Y := T;
```

This is equivalent of having a *multiple assignment* [31] of two variables like `X, Y := Y, X` available in our synthesiser. Thus similar to the `Max2` function, we introduce a `Swap` construct.

```
sig Swap extends Cmd {
    src: one (IntVar+IntArrVar),
    dst: one (IntVar+IntArrVar) } { src ≠ dst }
```

Such a construct is used to represent a multiple assignment like this one `src, dst := dst, src`. The semantics of `Swap` is very similar to that of a traditional assignment. The first part is for integer variables. Recall that the `bind` relation is an association between names and values. Thus, the following Alloy* snippet is stating that the `bind` relation will be updated with two new pairs: (i) the first associates the name of `src` (`c.src.name`) with the value of `dst` (`iSt.bind[c.dst.name]`); and (ii) the second associates the expected complement: the name of `dst` (`c.dst.name`) with the value of `src` (`iSt.bind[c.src.name]`). As arrays are not involved in such an assignment, we must ensure they do not change: `evalC[c][iSt].bindA = iSt.bindA`).

```
∀ c: Swap,
  iSt: evalC[c].univ •
    (c.src ∈ IntVar and c.dst ∈ IntVar
      ⇒ (evalC[c][iSt].bind = iSt.bind ++
           { c.src.name → iSt.bind[c.dst.name] +
             c.dst.name → iSt.bind[c.src.name] } and
           evalC[c][iSt].bindA = iSt.bindA))
```

4. Extending the Alloy* model towards arrays

After dealing with the main concepts synthesisers can offer to the user, we decided to check whether our Alloy* synthesiser could find the solution of well-known array-based programs. To do that, we just needed to introduce the concept of an array in our Alloy* specification.

4.1. Extending the syntax

Our first effort was to extend the syntax of IMP with arrays. To use an array inside expressions and assignments we needed to add the array itself (its name) as a new element in the Alloy* specification.

```
sig Array { }
```

By having the `Array` entity, we could extend the expressions allowed.

```
sig IntArrVar extends AExp { base: one Array, index: one AExp }
    { no (IntArrVar <: index) }
```

The above definition was needed to permit expressions like `A[i]` (that is, `A[i] ∈ IntArrVar`), where `A` is an `Array` (through the element `base`) and `i` is an usual arithmetic expression (through the element `index`). Note that we forbid using an array expression as an index (`no (IntArrVar <: index)`).

Besides expressions, arrays can also occur inside assignments (in the left-hand side). Thus we extend the entity `Assign` as follows.

```
sig Assign extends Cmd { lhs: one (IntVar+IntArrVar),
            rhs: one AExp
       }
       { rhs ≠ lhs and lhs.name ∉ XLoc }
```

That is, the left-hand side (`lhs`) of an assignment can now be an integer variable or an integer-based array variable (`lhs:one (IntVar+IntArrVar)`).

Finally, we need to store the content of an array in the state of the system (its semantics). The array's content (`ArrayContainer`) is indeed an alias for an Alloy* sequence (of integers). Thus

```
sig ArrayContainer {
    cells: seq Int
}
```

And the state used by the semantics of IMP (`State`) must have a reference to the array's content. This is accomplished by the map `bindA`, which links an array name with its content.

```
sig State { bind: Loc → one Int,
            bindA: Array → ArrayContainer }
```

4.2. Extending the well-formedness conditions and the semantics

Concerning well-formedness, we need just to assure that if an array name is used it must be used by some expression.

```
some Array ⇒ some IntArrVar and
      (∀ eA: IntArrVar • eA.base ∈ Array)
```

The semantics was extended in just two points: expressions and assignments. For expressions, we characterised in Alloy* the evaluation of something like `A[e]` (or `a: IntArrVar`), where `A` (or `a.base`) is the name of an array and `e` (or `a.index`) is some integer expression used as index of this array. The index is usually evaluated by `evalA`, but the array content has to be recovered from the sequence (`s.bindA[a.base].cells`).

```
∀ a: IntArrVar, s: State •
   evalA[a][s] = (s.bindA[a.base].cells)[evalA[a.index][s]]
```

For assignments, we just had to add the formalisation corresponding to arrays (`c.lhs ∈ IntArrVar`).

```
∀ c: Assign,
  iSt: evalC[c].univ •
      ...
  and
  (c.lhs ∈ IntArrVar
    ⇒ some ct: ArrayContainer • ct.cells =
        iSt.bindA[c.lhs.base].cells ++ { evalA[c.lhs.index][iSt] → evalA[c.rhs][iSt] }
        and (evalC[c][iSt].bindA = iSt.bindA ++ { c.lhs.base → ct })
        and evalC[c][iSt].bind = iSt.bind)
```

The `...` is the original formalisation for integer variables as left-hand sides, but with an extra condition to assert that array state variables do not change. For arrays as left-hand sides, we need to ensure that non-array variables do not change (`evalC[c][iSt].bind = iSt.bind`) and show what happens to the array variables. First we state that the content is updated by overriding, storing this new state information in `ct`. Then we use `ct` to update (by overriding as well) the mapping of array variables maintained by `bindA`.

And the next part deals with array-based variables. For arrays, we have a multiple assignment like `A[i], A[j] := A[j], A[i]`, where the goal is to change the elements positions inside an array. We have a similar but complementary predicate. We ensure non-array variables do not change (`evalC[c][iSt].bind = iSt.bind`) and we assume a new array content (name `ct`) such that its content is the same of the original one except at the positions whose elements are exchanged (`iSrc -> vDst + iDst -> vSrc`). Finally, this new array content is updated in the bindings of arrays (`evalC[c][iSt].bindA = iSt.bindA ++ c.src.base -> ct`).

```
((c.src ∈ IntArrVar and c.dst ∈ IntArrVar)
  ⇒ let iSrc = evalA[c.src.index][iSt] •
    let iDst = evalA[c.dst.index][iSt] •
    let vSrc = iSt.bindA[c.src.base].cells[iSrc] •
    let vDst = iSt.bindA[c.dst.base].cells[iDst] •
      (c.src.base = c.dst.base ⇒ (some ct: ArrayContainer •
        (ct.cells = iSt.bindA[c.src.base].cells ++
                    { iSrc → vDst + iDst → vSrc } and
      evalC[c][iSt].bindA = iSt.bindA ++ { c.src.base → ct }
      and evalC[c][iSt].bind = iSt.bind))))
```

5. Augmenting the Alloy* model with Genetic Algorithm

This section presents an integration of a genetic algorithm with our Alloy*-based Program Synthesiser (APS). Our genetic algorithm integration with the APS is referred to as *APS-GA*. To better understand this integration, we start by describing the syntactic ingredients, the control flow of APS and the user interactions with it.

We call the Alloy* scope *syntactic ingredients*. The synthesiser manipulates up to 15 syntactic constructors, described as follows: `IntVar` (integer variables), `IntVal` (integer values), `Assign` (assignment operations), `SComp` (sequential compositions), `CondS` (conditional statements), `While` (loop statements), `Add` (+ operations), `Sub` (− operations), `Mult` (× operations), `EQ` (= operators), `NEQ` (≠ operators), `LTH` (< operators), `LEQ` (≤ operators), `GEQ` (≥ operators), `GTH` (> operators). Each syntactic constructor must have a range, that is, a pair of integer values $\{lb : ub \mid 0 \leq lb \leq ub\}$ defining a lower bound (lb) and an upper bound (ub) of the amount of such a constructor to occur in the synthesised program. For example: (i) a `0:2` pair for `Add` means that either no `Add` operator is needed, or a single or double `Add`s are needed; and (ii) a `0:0` pair for `CondS` means no conditional statement (`if-then-else`) must occur in the synthesised program. By default, all syntactic constructors are initially set to automatic (auto), which means the synthesiser mechanically assigns values to the constructors. Here is an example of syntactic ingredients set manually: the statement `run Synt for 7 but exactly 4 IntVar, exactly 1 While, exactly 1 CondS` runs the synthesiser with at most 7 occurrences of each syntactic constructor, except for integer variables (it must have exactly four), while (it must have exactly one), and if-then-else (it must have exactly one).

Fig. 1 shows an Activity Diagram [32] illustrating the control flow of the APS in 6 steps. The user prepares the pre and post condition, the syntactic ingredients and the sketch (1). The synthesiser searches for a solution (2). If the synthesiser does not generate an instance, then the user needs to make changes (solution not found). If the synthesiser generates an instance (a candidate program), then the Alloy* Analyser generates an XML file (3) that is subsequently transformed into source code (4). The user has to check its correctness either by inspection or by compilation and testing (5). The user analyses the results and decides whether the candidate program is a correct solution (6). If it is not a correct solution, more adjustments are made to the input (1). Otherwise, the synthesis is finished.

In the diagram shown in Fig. 1, all activities contained in the user swim lane are done manually, while all activities in the APS swim lane are done mechanically. Also, the user has to interact with the APS dozens of times into the loops 1-2-1

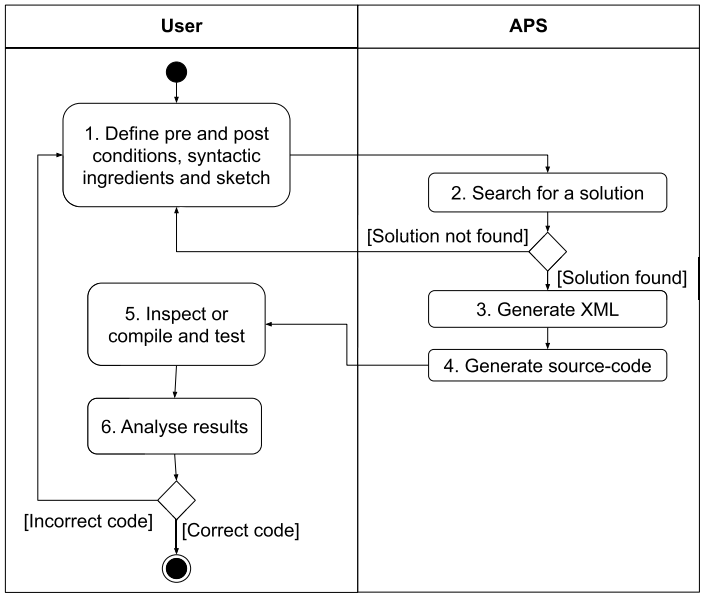

Fig. 1. The control flow of the APS.

and 1-2-3-4-5-6-1 until finding, by a trial and error method, a correct program. For example, the synthesis of Fibonacci takes a few hundreds of such iterations to find out a desirable solution.

One of the ideas of the program synthesis is to discover programs in which the user does not know a complete solution. Therefore, the user would rely on the outcome of the synthesiser. The user may have clues about the inputs (pre and post condition, syntactic ingredients, *sketch*, etc.), but would not know how to implement a complete source code and perhaps would be unable to judge whether the generated code is correct. The state-of-the-art does not yet allow us to rely fully on synthesisers. Therefore, the scenarios shown in Fig. 2 (and other figures in this document) are within an experimental context, where the user either knows if the synthesiser has produced something correct (and does the code inspection) or performs test cases to increase its confidence.

Fig. 2 shows the control flow of the APS-GA in 12 steps.

The user must provide input/output examples, syntactic ingredients (optionally as the Genetic Algorithm can generate them automatically), a soft sketch (optional) and a test suite (that is, a comprehensive set of input-output pairs) (1)→(2). A fitness function (represented by a distance metric, such as discrete, Manhattan, etc [33]) can be optionally provided to evaluate how better is a candidate program. The discrete metric is the default. Recall that a soft sketch is a set of commands that must appear in the synthesised program, but in no particular order (in contrast to the order of execution of a traditional sketch, which is hard coded). The user then invokes the APS-GA (2) → (3). From this point on, the APS-GA does steps (3) to (10) mechanically. First, APS-GA generates the syntactic ingredients (3). A chromosome is generated based on the input/output examples, the *soft sketch* and the syntactic ingredients (4). The Alloy* Analyser then is executed to search for instances (5). If the analyser does not generate an instance, APS-GA discards that chromosome and generates a new chromosome by repeating steps (3), (4) and (5). If the analyser generates an instance (a candidate program), then an XML model is generated (6) and is transformed into source code in the Java language (7). APS-GA then compiles, executes and tests that source code against the test suite (8). It then calculates the fitness function of the candidate program (9). If the program has not passed 100% of tests and has not yet reached the timeout, then a mutation is generated (10). Our mutation is a random change in the program's syntactic tree and is implemented by the Alloy* `next()` command, responsible for producing a new model different from the current one. If the mutation is invalid (if there is no instance), APS-GA tries again N times, where default $N = 32$ (the user can change N). If `next()` results in a new instance, steps (6), (7), (8), and (9) are repeated. If it pass 100% of the tests or reach the timeout, APS-GA halts. If APS-GA is halted by timeout, then it means that the best code generated did not pass 100% of the tests. In this case, the user must inspect (11) and analyse (12) the source code to decide if it does (or does not) satisfy her intention. Note that the values of the syntactic ingredients found in the first instance (first execution of step (3)) is used as the upper limit of the $0, 1, .., m$ range of each syntactic ingredient to be generated in future generations.

The examples are used to synthesise the program, while the test suite is a different set of examples used to evaluate the correctness of a candidate solution. Ideally, examples must capture the user intention while preventing a state explosion. For example, we have chosen the 3rd number of the Fibonacci sequence (whose output is the number 2). If we had chosen the 10th number (whose output is 55), the number of loops at the `While` syntactic ingredient would lead the synthesiser to state explosion. On the other hand, if we had chosen the first number of the Fibonacci sequence (the well-known 1), it would not be expressive enough to help the synthesiser to find out a solution. More precisely, examples are used at

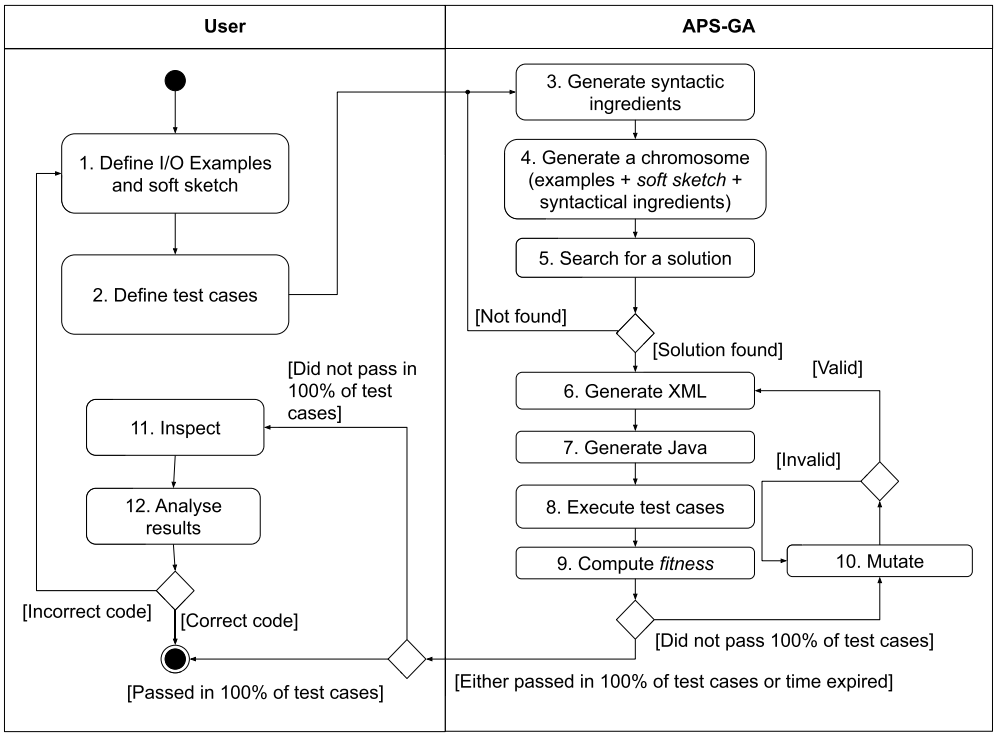

Fig. 2. The control flow of the APS-GA.

two different moments: (i) A very small set of examples is used in the core of the synthesiser to generate the candidate programs. Less than 6 examples were used for the synthesis of the programs in this study; and (ii) A set of tests (as many as necessary to represent all possible inputs and expected outputs, henceforth *critical cases*) composes the test suite, so that the generated candidate program that passes in all tests of the suite, is delivered as a source code solution. For example, in Max3 the number of critical cases to compose the test suite would be 6 tests (let $num1, num2, num3$ be integer variables, such that $num1 > num2 > num3$. Consider $in \rightarrow out = \{num1, num2, num3 \rightarrow num1; num2, num1, num3 \rightarrow num1, \ldots, num2, num3, num1 \rightarrow num1\}$), in Max4 it would be 24 tests. It would be an ideal scenario to always have a test suite that once a candidate program is going to pass 100%, it guarantees a correct source code by construction, but that depends on the user experience on informing a correct set of critical cases to compose the test suite, for each problem.

Regarding a soft sketch, here is an example of it to synthesise Fibonacci:

```
?? = Res − v3; ?? = Res + v3; ?? = v2 + 1;
```

where `??` means a hole, to be filled by the synthesiser. Also `v2`, `v3` and `Res` are integer variables and this soft sketch is written in Alloy* as follows:

```
//?? = Res − v3
Assign3.rhs = Sub1
Sub1.op1.name = Res
Sub1.op2.name = v3
//?? = Res + v3
Assign2.rhs = Add2
Add2.op1.name = Res
Add2.op2.name = v3
//?? = v2 + 1
Assign1.rhs = Add1
Add1.op1.name = v2
Add1.op2.val = 1
```

This soft sketch specifies that the program must have 3 assignments to unknown variables and, most importantly, the assignments do not need to occur in the particular order as shown in the soft sketch. A soft sketch has the advantage of leaving the ordering, occurrences and composition of the constructs to be synthesised. However, if needed, the user can restrict such freedom for a faster convergence.

APS-GA's evolutionary overview			Initial population					2nd generation					..	Gth generation				
			Chr 01	Chr 02	..	Chr 19	Chr 20	Chr 01	..	Chr 12	..	Chr 20	..	Chr 01	Chr 02	..	Chr 09
Chr	Gene 01	IntVar	3	3	..	3	3	4	..	3	..	3	..	4	4	..	3
	Gene 02	Add	1	0	..	0	1	3	..	1	..	0	..	3	3	..	0
	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
	Gene 15	GTH	2	2	..	2	2	1	..	3	..	1	..	3	3	..	3
Fitness	Boolean	Found instance?	No	Yes	..	Yes	Yes	No	..	Yes	..	Yes	..	Yes	No	..	Yes
	Int	Passed test (%)	NA	50	..	76	43	NA	..	84	..	78	..	47	NA	..	100

Fig. 3. The Genetic Algorithm way.

5.1. Genetic Algorithm on Alloy* instances

With a genetic algorithm we can now generate populations of candidate solutions that evolve (mutating their syntactic ingredients) over time. In particular, the population is produced mainly by generating syntactic ingredients automatically, while the mutation of the source code of the candidate programs is captured by the `next()` operation of the Alloy* Analyser.

Syntactic ingredients: Fig. 3 illustrates the genetic algorithm integrated with the APS. A gene is a syntactic constructor (like an addition operator or an `if-then-else`). In the case of APS-GA, there are 15 genes that make one chromosome, however this number will increase when implementing reuse or arrays. Note that, in the initial population, `Chromosome 01` has failed (`Found instance?` equals `No`) and thus it is not converted into Java, its fitness value is not computed and its "Passed test (%)" is marked as NA (Not Available). This chromosome is discarded. On the other hand, `Chromosome 19` finds a candidate program (`Found instance? Yes`), which is converted into source code and passes on 76% of the test cases (`passed test (%) 76`). The better the fitness value of the chromosome, the larger its probability to be selected to the next generation.

In the second generation, for instance, `Chromosome 12` had the best fitness of its generation and coincidentally the best so far. That process continues until the `G`th generation, where `Chromosome 09` achieves the *fitness=100%* (`passed test (%) 100`). This is a candidate program that has passed all tests and thus is a solution for the problem.

In parallel to the evolutionary process, the synthesiser now records the best fitness values achieved so far, that is, the synthesiser stores, along the way, all candidate programs sorted by their fitness values. This allows the synthesiser to show, at the end of the execution, the best program synthesised across all generations.

Our synthesiser stops when a program passes 100% of the tests (or we run out of time). So, although multiple correct programs could be produced, we only return the program that passed in most of tests or passed 100% of them. For instance, if an implementation of bubble sort is found, it is the one that is returned by the synthesiser regardless whether another implementation of a sorting algorithm could be produced few iterations later.

Mutation: In APS-GA, mutation is applied in two ways: (i) to syntactic ingredients; and (ii) to candidate programs. The former uses a genetic algorithm strategy that changes syntactic ingredients as follows. First, it uses an elitist selector that clones 90% of the best-fitness candidate programs and throws away the remaining 10% of the worst-fitness programs. It then fills up these 10% by repeating the best-fitness programs. Next, mutations are applied to each gene of the program population with a $1/12 = 0.083$ probability. To simplify, we have decided not to use crossovers along the evolutionary process.

In the latter case, APS-GA invokes the Alloy* Analyser to make changes in the source code of a candidate program by calling the `next()` method. This is how we implement mutation to candidate programs because it correspondingly changes some pieces of the source code, acting as the expected conceptual mutation operation as described in Genetic Algorithms. In other words, when the Alloy* Analyser finds an instance (a candidate program), it allows the APS-GA to call the Analyser's `next()` method to get another instance under the same constraints (or the same syntactic ingredients). The Alloy* Analyser then looks for a new instance through the search space by using its built-in enumeration strategy. Such a new instance happens to be a variation of the original instance where some parts of the candidate program change (usually it changes the position of variables, expressions or commands of the current candidate program).

Fig. 4 shows four consecutive mutations on candidate program, from the up-left-most to the down-right-most hand-side. From Fig. 4(a) to (b), we can see that the mutation changed the while loop condition (from `v2 < v3` to `v1 > v2`) as well as exchanged Lines `L3` and `L4`. From Fig. 4(b) to (c) it has exchanged Lines `L2` and `L3`. From Fig. 4(c) to (d) it has changed the left hand-side at three Lines (`L2: Res` with `v2`, `L3: v3` with `Res`, and `L4: v1` with `v3`). The `next()` function is always called a fixed number of times (say n) after an instance is found. Each call means a mutation. By default $n = 32$. But this value can be set by the user.

```
//(a)a%-test passed
while (v2 < v3) {//L1
  v3  = Res + v3; //L2
  v1  = Res - v3; //L3
  Res = v2 + 1; //L4
}
```

```
//(b)b%-test passed
while (v1 > v2) {//L1
  v3  = Res + v3; //L2
  Res = v2 + 1; //L3
  v1  = Res - v3; //L4
}
```

```
//(c)c%-test passed
while (v1 > v2) {//L1
  Res = v2 + 1; //L2
  v3  = Res + v3; //L3
  v1  = Res - v3; //L4
}
```

```
//(d)100%-test passed
while (v1 > v2) { //L1
  v2 = v2 + 1; //L2
  Res = Res + v3; //L3
  v3 = Res - v3; //L4
}
```

Fig. 4. (a) Excerpt of current Fib candidate; (b) after 1 call, (c) 2 calls, and (d) 3 calls to the `next()` method.

5.2. User inputs and interaction

Synthesisers that produce general purpose imperative programs usually take as input either a DSL (specific or subset of a general purpose language) [34], a sketch [35], or a formal specification [6].

APS [14] takes as input: (i) a formal contract (pre and post condition), (ii) syntactic ingredients (optional, but provided by hand by the user), and (iii) sketches (optional). On the other hand, the APS-GA takes as input: (i) examples, (ii) syntactic ingredients (optional at the start point, but they are generated mechanically during the evolutionary computation), (iii) a soft sketch (optional), (iv) a test suite and (v) a distance metric.

The default value of each syntactic ingredient in the APS is 3, i.e. each syntactic ingredient might occur at most 3 times in the synthesised program. It is very unlikely that this default value produces a correct program. Therefore, the user is forced to interact with the APS several times adjusting by hand the values of the syntactic ingredients in order to get more candidate programs. Moreover, the APS user must validate the candidate programs by hand (either by running test cases or by inspecting the source code). The amount of manual interactions the APS user must employ is much higher than when considering its combination with Genetic Algorithm (we mean the APS-GA), where the user can input (optionally) initial syntactic ingredients, but once the synthesis starts, no user interaction takes place. In each population, new syntactic ingredients are generated and all candidate programs are verified via testing in a fully automatic way.

6. Empirical evaluation

In order to evaluate whether APS and APS-GA are able to synthesise a program, three criteria have been used to select the problems: (i) similar problems found in the field and proposed by other synthesisers[3]; (ii) Turing complete problems, which means programs with states (variable data read/write), conditional branches, loops, arrays, etc.; (iii) problems that require several manual interaction with APS; for example, Max4 requires the user to call the `next()` 32 times followed by testing/inspection of all these instances before finding out a desired solution; and (iv) a few programs that represent different categories of problems (for example: Max2, 3 and 4 to Maximum/Minimum problems; GCD, Fib, IntSQRT exercise synthesis of while loop; aBubSort, aSelSort exercise array sorting problems; and so on).

The computer setup was: a notebook with an Intel 2.60 GHz i7 processor, 8 GB RAM, 256 GB SSD and Windows 10 Home operating system.

6.1. Deductive and inductive approaches and Reuse

Fig. 5 compares the time taken to translate Alloy* to the solver language (e.g. MiniSAT) with 4 different subjects: Swap (swaps the values of two variables), Max2 (returns the maximum of two numbers), Max3 (returns the maximum of three numbers) and GCD (returns greatest common divisor of two numbers). Such a translation from Alloy* to the solver's language is done automatically by the Alloy* Analyser.

According to Fig. 5 we can see that inductive synthesis takes more time than deductive synthesis in the task of translating from Alloy* to the solver's language. The difference, where deductive was faster than inductive, in terms of percentage of time were: Swap=26.6%; Max2=20.0%; Max3=17.6%; and GCD=7.2%. This is because, in the deductive case, the Alloy* Analyzer has to translate a generic predicate — the contract — whereas in the inductive case, each input-output example has to be translated.

The time taken for the solver to find a solution was measured under the configuration shown in Table 1. The synthesis process makes use of examples while the verification process makes use of test cases (the set of examples and the set of test cases are disjoint). Moreover, all subjects synthesised have passed in 100% of the test cases and have been inspected by the authors to further guarantee their correctness.

[3] White et al. [36] have reported a few benchmarks for general purpose automatic program synthesisers in the GP community. The authors have surveyed the EuroGP and the GECCO GP track conferences from 2009 to 2012 and have revealed that only 3.8% (8 out of 210) of the papers are in this field. On the other hand, programs presented at this paper were tested already and they fall into the "Synthetic problems" and "Algorithmic programming problems" sections into the same paper. Also they were tested into Sygus Competition, where some of them fall into the "Integer arithmetic" track [26].

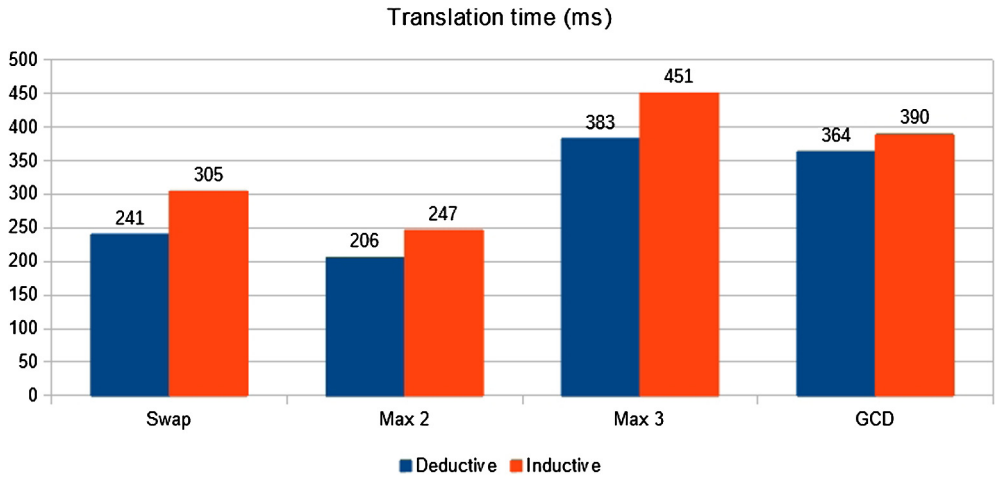

Fig. 5. Deductive vs inductive translation time.

Table 1
Number of examples used for synthesis and test cases used for verification for each subject..

Subject	Examples	Test cases
Swap	2	3
Max2	2	3
Max3	6	18
GCD	2	20

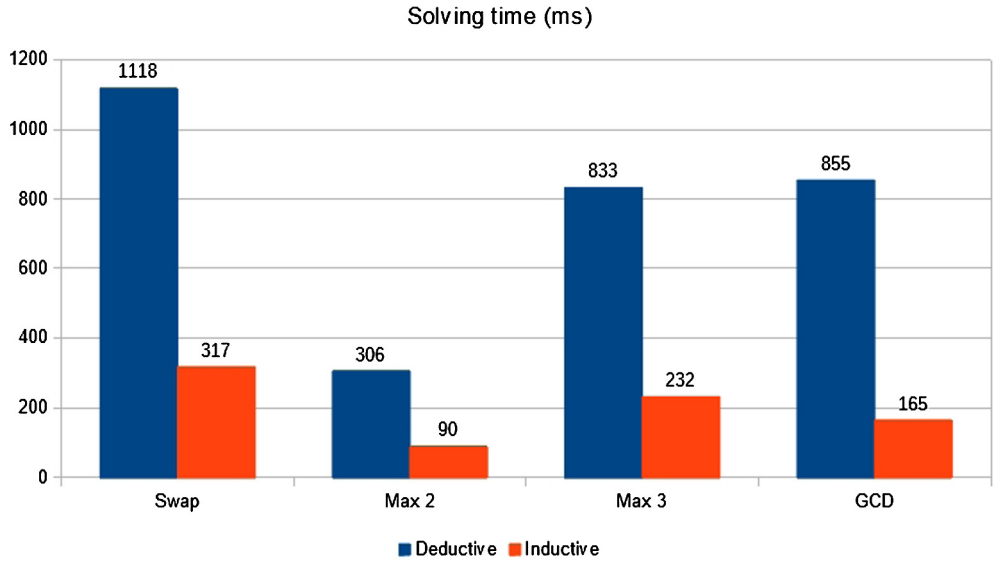

Fig. 6. Deductive vs inductive solving time.

So, the time taken for the solver to find a solution under this configuration was considerably shorter for the inductive case than the deductive case according to Fig. 6. The difference, where inductive was faster than deductive, in terms of percentage of time were: Swap=352.7%; Max2=340.0%; Max3=359.8%; and GCD=518.2%. This means that inductive synthesis can be 3 to 5 times faster than the deductive synthesis. For more complex problems, we may infer that these differences are going to be monotonically larger. This is because, in the deductive case, the generic predicate — the contract — takes an additional time to search for all but minimum set of input-output examples that satisfy the contract whereas in the inductive case, input-output examples are already given.

On the other hand, in the classical deductive synthesis based on theorem proving, the implementation is the final solution to the problem whereas in the inductive case it is a partial solution that needs rounds of verification to converge into the final solution. So the use of inductive or deductive synthesis depends mostly on the problem at hand.

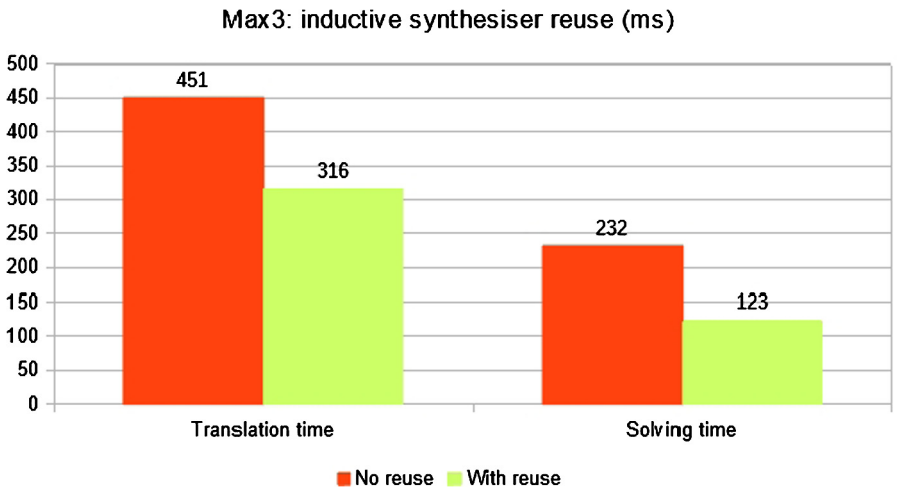

Fig. 7. Reuse translation and solving times of the Max3 problem.

Table 2
Execution time (seconds) of the synthesis task.

Prog.	Mean (s)	St. Dev.	Coeff. of Var (%)	# Cand.
Maj5	56.4	2.8	4.95	6
Maj8	56.6	2.7	4.77	8
IntSQRT	141.6	6.3	4.48	3
Max4	155.6	3.6	2.34	32
Modu	242.8	9.2	3.79	2
Fact	276.2	13.1	4.76	3
Fib	959.8	35.7	3.71	6

One can see that the effort taken by an inductive synthesiser is smaller than that of a deductive synthesiser (see Fig. 6). This is somewhat natural since the deductive case requires the synthesiser to create an implementation that must satisfy all input-output pairs whereas the inductive case uses a subset (used considerably smaller) of these input-output pairs.

In terms of reuse of function calls instead of trying to synthesise from the primitive commands, Fig. 7 presents the time taken by the Alloy* Analyser to translate the Alloy* model into the solver's language as well as the time taken by the solver to find a solution to the Max3 problem. It shows that the reuse of functions saves considerable time. It is worth recalling that the above code for Max3 is not reused as is: an Alloy* version of its syntax and semantics was embedded in the synthesiser by hand.

6.2. Using Alloy* and Genetic Algorithm

Using APS-GA, Table 2 shows the mean time of 10 runs (in seconds), its standard deviation, the coefficient of variation, and the number of candidate programs the system has tested before delivering a solution (`#Cand`) for the following subjects: Fibonacci (Fib), integer square root (IntSQRT), Majority of 5 (Maj5), Majority of 8 (Maj8), Maximum of 4 (Max4), modulo operation (Modu) and factorial (Fact).

The experimental setup was: population size (10 individuals); the total number of times the population can evolve (5 times).

In our empirical evaluation, we set the syntactic ingredients by giving priority at the following order: automatic, sketch, and manual (ranging from fully automatic to fully manual). In general, such a decision is up to the user.

Also, to achieve the performance shown in Table 2, we had to do some manual guesses,[4] described as follows: Fibonacci had 4 signatures defined manually out of 15, IntSQRT had 2 out of 15, Maj5 had 4 out of 15, Maj8 had 6 out of 15, Max4 had 0 out of 15, Modu had 4 out of 15, and Fact had 4 out of 15. The main limitation of using Genetic Algorithm is to get to a convergence when all syntactic ingredients are left in the automatic mode.

Table 3 shows I/O examples, soft sketches and the synthesised program for the same subjects from the Table 2. It is important to mention that the synthesised source codes for the majority programs (Maj5 and Maj8) are simplified versions that contain a single if-then-else block.

[4] We start by considering the number of variables by the elements used in I/O examples or the contract. By considering X variables, we assume at least X assignments. Conditionals and loops are inferred by thinking a bit about the problem at hand. That is, by creating some mental model of the problem (A very preliminary solution).

Table 3
APS-GA inputs and synthesised program.

Subject	I/O Examples	Soft Sketch	Synthesised Program

Fib	3→2	?? = Res+v3
		?? = Res-v3
		?? = v2+1
			v1=read(); v2=Res=0; v3=1;
			while (v1>v2) {
			v2 = v2 + 1;
			Res= Res + v3;
			v3= Res - v3;
			}

Int SQRT	6→2	?? = v2 + 1;
		?? = v3 + 2*v2+1;
		?? = v2 - 1;
			v1=read(); v2=v3=Res=0;
			while(v3<=v1) {
			v3 = v3 + 2 * v2 + 1;
			v2 = v2 + 1;
			}
			Res = v2 - 1;

Maj5	1;2;1;1;1→1
	2;1;1;2;1→1
	1;2;2;1;2→2
		?? = v1;
		?? = v2;
		?? = v4+v5+v6 +v7+v8;
			v1=1; v2=2; v3=7; Res=0;
			//Read v4,v5,v6,v7,v8
			Res=v4+v5+v6+v7+v8;
			if(Res >v3) {
			Res = v2;
			} else {
			Res = v1;
			}

Maj8	1,2,1,1,1,1,1, 1,1→1;
	2,2,1,2,1,2, 1,1→1;
	1,2,2,1,2,2, 1,2→2;
		?? = v1;
		?? = v2;
		??=v4+v5+...+v11;
			v1 = 1;
			v2 = 2; v3= 12; Res = 0;
			// Read v4,v5,..,v11
			Res=v4+v5+...+v11;
			if (Res >v3) {Res = v2;}
			else {Res = v1;}

Max4	5;7;2;1→7
	7;2;5;1→7
	1;5;7;2→7
	5;1;2;7→7
		v1 >v2
		v1 >v3
		v1 >v4
		v2 >v3
		v2 >v4
		v3 >v4
		?? = v1;
		?? = v2;
		?? = v3;
		?? = v4
			Res=0;//Read v1,v2,v3,v4
			if(v1 >v2) {
			if(v1 >v3) {
			if(v1 >v4) {Res = v1;}
			else {Res = v4;}
			} else {
			if(v3 >v4) {Res=v3;}
			else {Res = v4;}
			}
			} else {
			if(v2 >v3) {
			if(v2 >v4) {Res = v2;}
			else {Res = v4;}
			} else {
			if(v3 >v4) {Res = v3;}
			else {Res = v4;}
			}
			}

Modu	3;2→1	?? = v2+v1 -v3;
		?? = v4+1;
		?? = v2*v4;
			//Read v1,v2
			v3=v4=Res=0;
			while(v3 <= v1) {
			v4 = v4 + 1;
			v3 = v2 * v4;
			Res = v2 + v1 - v3;
			}

Fact	3→6	?? = v2;
		?? = v2 * v3;
		?? = v3 + 1;
			v1=read();v2=1;v3=Res=0;
			while(v1 >v3) {
			v3 = v3 + 1;
			v2 = v2 * v3;
			}
			Res = v2;

The total time spent to find out a solution varied from 56.4 seconds (Maj5) to 15.9 minutes (Fib). The coefficients of variation are under 5% of their own mean time, which show that the experimental setup performance was homogeneous. Finally, the synthesiser had to check from 2 (Modu) to 32 (Max4) candidate programs to find a solution.

Table 4
Syntactic ingredients for Fib defined mechanically: gray cells-sketch/automatic; white cells/manual.

Subject	IntVar	IntVal	Assign	SComp	CondS	While	Add	Sub	Mult
Fib	skt	skt	skt	2:2	0:0	1:1	skt	skt	0:0

EQ	NEQ	LEQ	GEQ	LTH	GTH
auto	auto	auto	auto	auto	auto

At first sight, it seems that the synthesis time is too long. But the question is that in general we do not take into account the time used by the user to interact with a synthesiser in a try and error approach. In our case, the Genetic Algorithm part performs this task automatically. Obviously that, if one considers just the time taken by the Alloy* synthesiser when the input is precisely defined, the synthesis takes considerable less time. For instance, in our previous study [14], the synthesiser was very competitive because the time taken by adjusts of the user input was not considered.

6.2.1. The synthesis of Fibonacci

To better understand how the Alloy* synthesiser interacts with the Genetic Algorithm part, we describe the inputs to synthesise Fibonacci.[5]

Input → output examples: For each I/O example, the user needs to provide to the synthesiser a pair of initial (`IState`) and final (`FState`) state values for all variables involved in the synthesis task. We named `v1`, `v2`, `v3`, `v4`, ..., and so on, for input and temporary variables, and `Res` for the output variable. For example, the synthesis of the Fibonacci program has required one input (`v1`), a pair of temporary variables (`v2` and `v3`), and one output (`Res`). For certain synthesis used in this article, we need to provide examples related to an expected well-known solution to the problem. In the case of Fibonacci, the iterative solution uses the previous two Fibonacci elements to compute the next one. This is the reason we used variables `v1`, `v2` and `v3` as input. The initial and final states are declared as shown below.

```
one sig IState1 extends IState {} {bind = {v1→3}+{v2→1}
                                         +{v3→0}+{Res→0}}
one sig FState1 extends FState {} {bind = {v1→3}+{v2→1}
                                         +{v3→3}+{Res→2}}
```

One I/O example was enough to perform a successful synthesis. The initial state of the variable `Res` is zero and it is expected that the program will end up with `Res` equals to two. Also, the variable `v1` must keep the same value 3 from the initial to the final state, the variable `v2` also keeps the same value 1 at any set of examples for the Fibonacci's synthesis, and the variable `v3` will vary from zero (at the initial state) to 3 (at the final state).

Moreover, the Fibonacci test suite contains pairs of the form `inp→out`, where `inp` is the input and `out` is the expected result. The test suite contains the following pairs: $1 \rightarrow 1$, $2 \rightarrow 1$, $4 \rightarrow 3$, $5 \rightarrow 5$, $6 \rightarrow 8$, $7 \rightarrow 13$, $8 \rightarrow 21$, $9 \rightarrow 34$, $10 \rightarrow 55$. Also, note that here we skip the pair $3 \rightarrow 2$ as it has already been used as an I/O example of the synthesis process.

Soft Sketch: The synthesis of Fibonacci has required the following soft sketch: `?? = Res + v3; ?? = Res - v3; ?? = v2 + 1;`, where `??` means a hole [35]. Note that the semicolon (`;`) only separates the assignments, without forcing any order in their executions.

Syntactic Ingredients: Table 4 shows how syntactic ingredients are defined for Fibonacci. The user may want to reduce the time spent to find out a solution by constraining the search space and guess the bounds of the scope of the signatures. There are two ways to do that: (i) by writing down a soft sketch that set up the scopes indirectly (see cells filled with `skt` in Table 4); and (ii) by setting up the bounds of the syntactic constructors manually.

Although the syntactic ingredients have default values (set to auto), if the genetic algorithm does not converge to a correct answer, we indeed need more user guidance. In order to provide such arguments, the user needs some knowledge of programming to guess the values of the ingredients. We have not carried out a controlled experiment to measure the effort to provide syntactic ingredients, but we think it is somewhere in between programming and providing examples.

Once a *soft sketch* (cells filled with `skt` in Table 4) and/or I/O examples are defined, the user has to define the syntactic ingredients as signatures in the Alloy* model. For instance, for the Fibonacci soft sketch, mentioned before, the syntactic ingredients (variables, assignments, addition and subtraction operations) have to be written as signatures in the Alloy* model, as shown on the next page

[5] For conciseness, we only report the Fibonacci synthesis in detail. The scripts for all other examples can be found in the following website: https://github.com/PSMFg/psmf/wiki.

```
...
// definition — v1 was mention on I/O example
one sig v1, v2, v3, Res extends DLoc {}
one sig Assign1, Assign2, Assign3 extends Assign {}
one sig Add1, Add2 extends Add {}
one sig Sub1 extends Sub {}
...
// Soft sketch
Assign3.rhs = Sub1 //?? = Res - v3
Sub1.op1.name = Res
Sub1.op2.name = v3
Assign2.rhs = Add2 //?? = Res + v3
Add2.op1.name = Res
Add2.op2.name = v3
Assign1.rhs = Add1 //?? = v2 + 1
Add1.op1.name = v2
Add1.op2.val = 1
...
```

After that, there is no further user intervention, APS-GA will count and will set ranges of the syntactic ingredients automatically. First APS-GA counts how many definitions exist in the Alloy* model, for the Fibonacci problem mentioned before, the count will be: 4 `DLoc` (named `v1, v2, v3, Res`), which is a kind of `IntVar`; 3 `Assigns` (named `Assign1, Assign2, Assign3`); 2 `Adds` (named `Add1, Add2`); and 1 `Sub` (named `Sub1`). Secondly, APS-GA sets the bounds of previous counted syntactic ingredients (`Invar=4:4`, `Assign=3:3`, `Add=2:2`, and `Sub=1:1`).

In the synthesis of the Fibonacci program, we defined manual values $\{lb : ub\}$ for a sequential composition (`SComp`), a conditional constructor (`CondS`), a while loop (`While`) and a multiplication (`Mult`) (see Table 4).

6.3. Considering programs with arrays

The subjects used are described below:

- Maximum/minimum element found in an array (`aMax`);
- Assignment of all elements of an array (e.g. `myArray[index]= 2*index;`) with some expression (`aDouble`);
- Addition of all elements of an array (`aSUm`);
- Counting the number of occurrences of a value in an array (`eCount`);
- Bubble sorting (`aBubSort`);
- Selection sorting (`aSelSort`).

For each program synthesised, Table 5 shows the mean time of 10 runs (seconds), its standard deviation, the coefficient of variation, and the number of candidate programs the system has tested before delivering a solution (`#Cand`).

Table 5
Execution time of the synthesis task.

Prog.	Mean	St. Dev.	Coeff. of Var (%)	#Cand.
aMax	14.0	0.4	2.97	1
aDouble	18.0	0.8	4.31	1
aSum	151.0	6.1	4.07	16
eCount	347.3	7.2	2.08	15
aBubSort	271.0	1.0	0.37	3
aSelSort	407.0	5.6	1.37	10

6.3.1. The synthesis of an array's Maximum element (aMax)

In what follows, we describe in detail the inputs given to synthesise aMax.[6]

Input → output examples: For each I/O example, the user needs to provide to the synthesiser a pair of initial (`IState`) and final (`FState`) state values for any variables involved in the synthesis task. For example, the synthesis of the aMax program has required a pair of input (`InLoc1` and `InLoc2`), an input array (`InArray1`) pointing out to a container (`ArrayC1` and their pairs of `index`→`value`-cells), and one output (`OutLoc`). The container, initial and final states are declared as follows on the next page:

[6] For conciseness, we only report the aMax synthesis in detail. The scripts for all other examples can be found in the following website: https://github.com/PSMFg/psmf/wiki.

```
one sig ArrayC1 extends ArrayContainer{}
    {cells = {0 → 2 + 1 → 3 + 2 → 1 + 3 → 0} }
one sig IState1 extends IState{}
    {bind = {OutLoc → 0 + InLoc1 → 0 + InLoc2 → 3}
        and bindA = {InArray1 → ArrayC1} }
one sig FState1 extends FState{}
    {bind = {OutLoc → 3 + InLoc1 → 3 + InLoc2 → 3}
        and bindA = {InArray1 → ArrayC1} }
```

The initial state of the variables `InLoc1` and `OutLoc` is `0` and it is expected that the program will end up with both values equals to `3`. Also, the variable `InLoc2` must keep the same value (3) from the initial to the final state, as so the `InArray1` array (with values shown at column `Examples` from Table 6). The input variables `InLoc1` and `InLoc2` represent the lowest and highest indices of the array. As `InLoc1` is incremented during the program, its final value must be equal to `InLoc2` in the final state. Moreover, the aMax test suite is quite different. Now the user needs to provide, for any variable involved in the synthesis task, its category (only input `_in`, only output `_out`, or both `_io`), its type (only integer `_int`, or only array of integers `_intArr`), its name (such as `_InLoc1` or `_IOArray2`), and its value (a single, a pair, or more) that is related with category and type the user can set. For example, the synthesis of the aMax program requires a single integer (`OutLoc`) for output (`_out`) variable, a pair of single integer (`_InLoc1` and `_InLoc2`), and an array (`_InArray1`) for input (`_in`) variables and the test suite must have lines such as:

```
_out_int_OutLoc=13;
_in_int_InLoc1=0;
_in_int_InLoc2=5;
_in_intArr_InArray1=[0]2=[1]13=[2]1=[3]4=[4]3=[5]5};
```

Finally, aMax, aSum, and eCount examples use arrays only for inputting purposes and their initial and final states remain the same. On the other hand, aDouble, aBubSort, and aSelSort use arrays for inputting and outputting purposes, which means their initial and final state will change, and that fact matters regarding the state explosion of the Alloy* synthesiser.

Soft Sketch: There are several possible solutions (source code) for the `aMax` program. One of them is shown at the `Synthesised program` column in Table 6. In this case, the synthesised program skeleton comprises 10 possible holes to be filled and they are denoted by `?1, ..., ?10` symbols, as shown at the `Possible holes` column in the same table. Notice that the `Possible holes` column is not required by the synthesiser. It has been shown just to explain which holes the synthesiser filled in. Have a look at the `sketch` column[7] to notice the sketch line `//s1` may fit at hole `?2` or `?4`, as so sketch line `//s2`. Also, sketch line `//s3` may fit at hole `?5=?6`, `?8`, or `?9=?10`, as so sketch line `//s4`. Moreover, hole `?8` may be any arithmetic expression, including an empty one. As our synthesiser deals with soft sketches, the order of execution (`//s2→//s1→//s4→//s3`) at the `synthesised Program` was different than the one listed at the `Sketch` column `//s1→//s2→//s3→//s4`. Finally, the sketch we have provided to run the experiments has fitted 40% (notice that the APS-GA have to decide which hole each sketch went to fit in). The 60%-remain (6 of 10) of the possible holes were filled by the APS-GA, in other words, the APS-GA has decided to fill `?1` with `while`, `?3` with `if`, `?5` with `OutLoc`, leaving `?8` empty, and so on. As the program gets more complex, we may infer that it becomes more difficult to synthesise its source code keeping the same amount of sketch. Thus, APS-GA has filled up to: 57%-remain (4 of 7) of the possible holes, including the array's index for aDouble and aSum problems; and 60%-remain (6 of 10) of the possible holes for eCount. From the aBubSort (43%-remain, 7 of 16) and aSelSort (25%-remain, 5 of 20) problems, the amount of the sketches are quite similar to the final source code, similarly to what has happened to Solar-Lezama's study [35].

Syntactic Ingredients: Without arrays we have up to 15 syntactic constructors, described in Section 5. Using arrays, we have included more three: `Skip` (a command that does nothing), `Swap` (swaps a value between two variables) and `IntArrVar` (number of arrays of integer variable). All eighteen syntactic constructors are working the same way described at Section 5. For aMax, the synthesis performance results shown in Table 5. APS-GA has setup the following syntactic constructors to: Sketch (`Assign, Add, IntArrVar, GTH`); Manual (`Swap 0:0`); and Automatic (`IntVar, IntVal, SComp, CondS, Skip, While, Sub, Mult, EQ, NEQ, LEQ, GEQ, LTH`). Table 6 shows I/O examples, possible holes, soft sketches and the synthesised source code for the subjects aMax, aSum, aDouble, eCount, aBubSort, and aSelSort.

6.4. Genetic algorithm effectiveness

This section describes how a change in the APS-GA inputs affects the searching time. In particular, we are interested in evaluating the genetic algorithm effectiveness when the range of the syntactic ingredients is enlarged.

We have used the same arguments described in Section 6, except for those below.

[7] `j=IONum[InLocJ]`, `i=IONum[InLocI]`, and `m=IONum[InLocMin]`. The holes `?2, ?6, ?8` into `aBubSort`'s sketch can be filled with any of these three sketch expressions: (i) `InLocJ<InLimJ`; (ii) `j<i`; or (iii) `InLocI<InLimI`.

Table 6
Inputs and synthesised programs using APS-GA with arrays.

Subject	Possible holes	I/O examples	Soft sketches	Synthesised program

aMax	?1(?2) {
	?3(?4) {
	?5=?6;
	} ?7 { ?8 }
	?9=?10;
	}
		[0]2; [1]**3**;
		[2]1→**3**
			InArray1[InLoc1] >
			OutLoc //s1
			InLoc2 >
			InLoc1 //s2
			??=InLoc1+1 //s3
			??=InArray1[InLoc1] //s4
				OutLoc=0; //Read InLoc1,
				//Read InLoc2, inArray1
				while(InLoc2 >InLoc1) {
				if(InArray1[InLoc1]
				>OutLoc) {
				OutLoc=InArray1[In-
				Loc1];
				} else { }
				InLoc1=InLoc1+1;
				}

aDouble	?1(?2) {
	?3= ?4;
	?5= ?6;
	}
		[0]2; [1]3;
		[2]1→
		[0]4; [1]6;
		[2]2
			InLoc1<InLoc2 //s1
			??= InLoc1+1 //s2
			??= 2*IOArray1[??] //s3
				OutLoc=0; InLoc1=0;
				//read InLoc2; IOArray1;
				while(InLoc1 < InLoc2) {
				IOArray1[InLoc1]=
				2*IOArray1[InLoc1];
				InLoc1= InLoc1+1;
				}

aSum	?1(?2) {
	?3= ?4;
	?5= ?6;
	}
		[0]2; [1]3;
		[2]1→6
			InLoc2 >InLoc1 //s1
			??=InLoc1+1 //s2
			??=OutLoc+
			InArray1[??] //s3
				OutLoc=0; //Read InLoc1,
				//Read InLoc2, inArray1
				while(InLoc2 > InLoc1) {
				OutLoc= OutLoc+
				InArray1[InLoc1];
				InLoc1= InLoc1+1;
				}

eCount	?1(?2) {
	?3(?4) {
	?5=?6;
	} ?7 { ?8 }
	?9= ?10;
	}
		[0]**2**; [1]3;
		[2]**2**; [3]1;
		[4]**2**→3
			InArray1[InLoc1]==
			InLoc3 //s1
			InLoc2 >InLoc1 //s2
			InLoc1=InLoc1+1 //s3
			OutLoc=OutLoc+1 //s4
				OutLoc=0; //Read InLoc1,
				InLoc2
				//Read InLoc3, inArray1
				while(InLoc2>InLoc1) {
				if(InArray1[InLoc1]==In-
				Loc3) {
				OutLoc=OutLoc+1;
				} else { }
				InLoc1= InLoc1+1;
				}

aBubSort	?1(?2) {
	?3=?4;
	?5 (?6) {
	?7 (?8) {
	swap(?9,
	?10)
	} ?11 { ?12
	}
	?13=?14;
	}
	?15=?16;
	}
		[0]2; [1]3;
		[2]1→[0]1;
		[1]2; [2]3
			while (?2) {
			?3=InLocI+1;
			while (?6) {
			if (?8) {
			swap(j, i);
			} else { ?12 }
			?13=InLocJ+1;
			}
			?15=InLocI+1;
			}
				InLocI=0;
				InLimI=IONum.length-1;
				InLocJ=0;
				InLimJ=IONum.length;
				while (InLocI<InLimI) {
				InLocJ=InLocI+1;
				while (InLocJ<InLimJ) {
				if (j<i) {
				swap(j, i);
				} else { } //skip
				InLocJ= InLocJ+1;
				}
				InLocI=InLocI+1;
				}

aSelSort	?1(?2) {
	?3=?4;
	?5=?6;
	?7(?8) {
	?9(?10) {
	?11=?12;
	} ?13 { ?14
	}
	?15=?16;
	}
	swap(?17,
	?18)
	?19=?20;
	}
		[0]2; [1]3;
		[2]1→[0]1;
		[1]2 [2]3
			while (InLocI<
			InLocN-1) {
			?3 = InLocI;
			?5 = InLocI+1;
			while (InLocJ<
			InLocN) {
			if (j<m) {
			?11= InLocJ;
			} else { }
			?15=InLocJ+1;
			}
			swap(i, m)
			?19=?20;
			}
				InLocN = IONum.length;
				InLocI=InLocMin=InLocJ=0;
				while (InLocI<
				InLocN-1) {
				InLocMin = InLocI;
				InLocJ = InLocI+1;
				while (InLocJ<InLocN) {
				if (j<m) {
				InLocMin= InLocJ;
				} else { }
				InLocJ=InLocJ+1;
				}
				swap(i, m)
				InLocI=InLocI+1;
				}

Table 7
Genetic Algorithm effectiveness.

Problem	Time w/tight syntactic ingredients	Enlarge search space	Time w/relaxed syntactic ingredients	Chromosome (generation)	Discard rate %
Fib	00h16m00s	16x	12h45m00s	96th (4th)	05%
Maj5	00h00m56s	64x	03h55m00s	729st (3rd)	87%

1. The candidate program population size has been increased from 3 to 30 individuals;
2. The Maximum Evolutionary Population Process has been increased from 5 to 20 generations;
3. The Manual guesses from some syntactic ingredients were relaxed from values $\texttt{lb}_{\texttt{from}}\texttt{:ub}_{\texttt{from}}$ to values $\texttt{lb}_{\texttt{to}}\texttt{:ub}_{\texttt{to}}$, in other words, $\texttt{lb}_{\texttt{from}}\texttt{:ub}_{\texttt{from}} \rightarrow \texttt{lb}_{\texttt{to}}\texttt{:ub}_{\texttt{to}}$, which means a larger search space was set;
4. We have enlarged by 16 times the search space for Fib and by 64 times for Maj5 by changing the range of following syntactic ingredients:
   - Fib={`SComp 2:2→0:3`, `While 1:1→0:1`, `GTH 1:1→0:1`};
   - Maj5={`SComp 1:1→0:3`, `CondS 1:1→0:3`, `GTH 1:1→0:3`}

By looking at the Fib problem, the `SComp` syntactic ingredient has increased from 1 to 4 possible values {2} → {0, 1, 2, 3}, denoted by 1 → 4. The same idea is applied to `While` and `GTH`, which holds `4(SComp) · 2(While) · 2(GTH) = 16`. The same concept is applied to Maj5, which now holds `4(SComp) · 4(CondS) · 4(GTH) = 64`.

The candidate program population size and the Maximum Evolutionary Population Process arguments were increased to allow the APS-GA to try a larger diversity of possible candidate programs, due to a larger search space (and, consequently, less manual guesses).

With those changes described before, the running time of the APS-GA to find out a solution has increased:

- From 16 minutes (at the 6th chromosome in the 1st generation) to 12 hours and 45 minutes (at the 96th chromosome in the 4th generation) for the `Fib` problem; and
- From 56 seconds (at the 6th chromosome in the 1st generation) to 3 hours and 55 minutes (at the 729th chromosome in the 3rd generation) for the `Maj5` problem.

The effectiveness of the Genetic Algorithm is shown in Table 7. As the population size comprises 30 individuals and, roughly speaking, there were approximately 90 chromosomes that have turned into candidate programs, the 729 attempts represent a discard rate $\approx 87\%$ of the invalid chromosomes. By using the same idea to Fib problem, the discard rate was smaller, only $\approx 5\%$ (5 from 96) chromosomes were discarded.

With these results (summarised in Table 7), we observe that, by integrating genetic algorithm with APS, it might take hours or even several days to find out a desired solution, as long as more syntactic ingredients are left on the automatic mode or the user guesses are relaxed. Nevertheless, combining a genetic algorithm with APS is still a better choice than using just APS because APS-GA avoids a lot of manual interactions in several trial-error attempts.

6.5. Choice of solver

Because the Alloy Analyser uses the Sat4Java solver as default, another important aspect to improve performance of the synthesis is to use the most appropriate solver. As already reported in literature, our synthesis were faster when using the MiniSAT solver available in the Alloy* Analyser. To give an idea of the importance of choosing an appropriate solver, Fig. 8 shows that, in the synthesis of the Bubble Sorting algorithm, if one chooses the Sat4J solver instead of MiniSAT, the synthesis time increases around 6 times. For conciseness, just Bubble sort was presented. However previous empirical runs have shown that, for all other array problems, the MiniSAT solver outperformed Sat4J.

By using a similar solver to MiniSAT, the MiniSAT UnSat Core solver, the time difference reduces significantly. Thus all examples in this paper were synthesised using the MiniSAT solver.

6.6. Threats to validity

The main threat to us is the external validity, which regards to the generalisation of the synthesiser efficiency and efficacy to other programs or when we relax the search parameters (considering two perspectives: successfully convergence to a solution, and synthesis times). Indeed, the set of programs used as sample examples allows us to have a good idea of the efficiency of the APS and APS-GA, although it is challenging to say that it will work well if we get the parameters of the genetic algorithm strategy more relaxed or if we apply the APS to other programs that deals with a higher level of complexity (such as nested loops, nonlinear expressions, etc). Another threat to generalisation is the number of subjects used. Although we chose them to possess varied different features, we still need to experiment with a larger set of subjects.

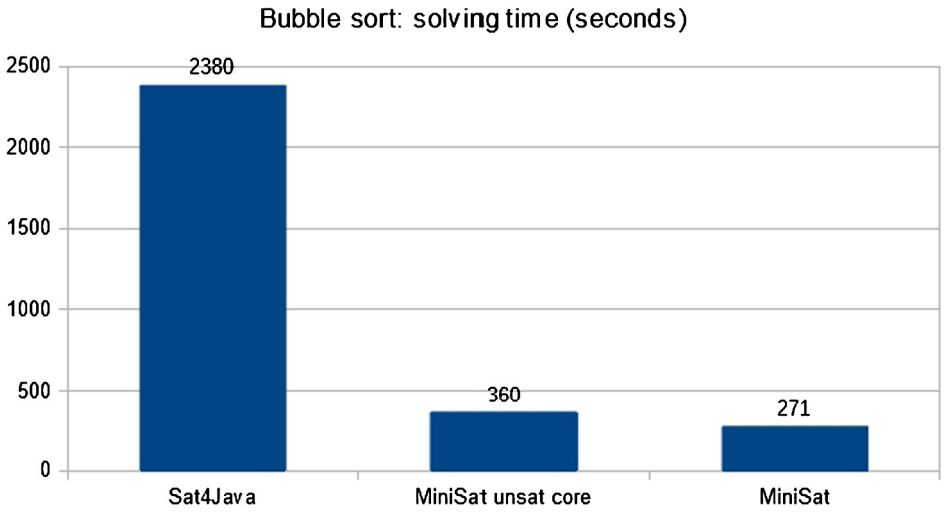

Fig. 8. Choice of solver.

7. Related work

This section presents synthesis strategies currently found in literature that are related (but different) to this study.

A classic study applied the deductive approach to generate synthesis from a complete specification through mathematical induction proof [37]. One purpose of our study is to employ genetic algorithms to drive the synthesis, requiring the user to provide as naturally as possible examples and soft sketches. We assume that such task is less challenging than constructing predicates, theorems, and formal proofs.

A synthesis approach by SAT/SMT solvers was proposed where, from the construction of constraints and sketches, the synthesiser is able to generate as a result the source code and the respective proof [3]. This approach requires the user to define a scaffold (which resembles sketches) so that the user needs to inform the syntactic structures, such as loops, assignment, and operators (which and in what order). Our approach allows the user to set constraints as soft sketches, where there is no need to define in what order its commands must occur.

A tool was created to design solver-aided languages with a set of predefined program synthesis resources (language constructors) that allow solving symbolically problems of synthesis through sketches [34]. One purpose of our study is to mitigate the user needs on defining detailed sketches.

A framework was created by Polozov and Gulwani [7] to facilitate rapid development of efficient, robust, and maintainable inductive program synthesis (called Program-By-Example) of industrial quality. That framework proposes an approach called data-driven domain-specific deduction (D4), which unifies the strengths of deductive, syntax-guided, and domain-specific inductive strategies in one meta-algorithm. They present synthesis case studies of string transformations in spreadsheet software and scripts for extracting data from semi-structured documents (such as custom fields from log files). Our study aims at producing general purpose imperative programs where we synthesise conditional branches, sequential compositions, arithmetic and relational expressions and loops.

Parisotto et al. [38] proposed the Neuro-Symbolic Program Synthesis technique, which can mechanically construct computer programs in a domain-specific language that are consistent with a set of examples provided at testing time. This method is based on two neural modules. First, the cross correlation I/O network produces a continuous representation of a given set of examples. Second, the Recursive-Reverse-Recursive Neural Network (R3NN) synthesises a program by incrementally expanding partial programs. They present synthesis case studies based on string transformations. Our approach does not use a training set and aims to produce general purpose programs.

Krawiec [39] proposed a synthesis of programs based on the behaviour of candidate programs so that (in addition to the examples) the values of the intermediate states of a candidate program are explored. Krawiec extends the traditional fitness function to find out a solution through genetic programming and machine learning techniques, presenting examples in functional language. Our study addresses the synthesis of general purpose but imperative programs.

Natural synthesis [40] is a technique that extends program synthesis by SAT / SMT solvers in an approach that uses natural evidence to automate the synthesis of correct construction programs extracted from rich and complex specifications, which in principle establishes an intractable synthesis problem. Then the user needs to identify a set of natural proofs to help on finding a program that supports natural proofs. In doing so, the original (intractable) synthesis problem becomes a natural synthesis problem (treatable and that can be manipulated by inductive synthesisers such as Sketch or Rosette, with strong specification, so that the program that meets the new specification also serves the original. Qiu *et al.* [40] present a synthesiser of data structures (iterative or recursive) that manipulate imperative programs with dynamically allocated stacks. The inputs for the synthesiser are: (i) a program sketch that describes the program skeleton at a high level and leaves the

Table 8
A comparison between synthesisers.

Proposal	Criteria	Solver	State	Loops	Sketch	Pointer	#Probl.
DeduSynt [37]	Ded.	SMT	No	No	No	No	Several
Sketch [18]	Ded.	SMT	No	No	Need	No	Several
PROSE [7]	Ind./Ded.	SMT^{+}	No	Yes	Need	No	Few
Rosette [34]	Ind.	SAT	No	No	Need	No	Few
VS^{3} [3]	Ded.	SMT	Yes	Yes	Need	No	Few
NeuroSymbol [38]	Ded.	SMT	Yes	Yes	Need	No	Few
BPSynt [39]	Ind.	Decision Tree	No	No	No	No	Few
Alloy* [15]	Ded.	SAT	No	No	Maybe	No	Several
NaturalSynt [40]	Ded.	SMT	Yes	Yes	Need	Yes	Few
SUSLIK [6]	Ded.	SMT	Yes	Yes	Free	Yes	Few
APS [14]	Ded./Ind.	SAT	Yes	Yes	Maybe	Yes	Few
APS-GA [25]	Ind.	SAT	Yes	Yes	Maybe	Yes	Several

implementation details as syntactic gaps (unknowns); (ii) Pre and post condition (uses the reserved words requires and ensures, respectively) where formulas can be built using a DSL.

The work by Polikarpova *et al.* [5,6] (about Synquid and Suslik) goes in a similar direction to that of Srivastava [3,4]. Indeed Solar-Lezama *et al.* [5] are also exploring features in such a direction as well. That is, of using contracts (pre and post condition) to aid on synthesis. But in these studies, the authors do a manipulation that we do not. They work in a true deductive approach, finding intermediate predicates that are related to Hoare's logic. And from these, they find the programming constructs. We always perform syntactic search using the Alloy*'s SyGUS infrastructure.

None of the related studies has combined a model finder with Genetic Algorithm to synthesise general purpose imperative programs.

Table 8 summarises the most representative synthesisers found in literature, comparing them with respect to:

- The synthesis **Criteria**: If it is deductive (Ded.) or inductive (Ind.) or both (Ded./Ind.);
- The kind of **Solver**: SAT, SMT, and SMT+;
- If it deals with **State** change (commands) instead of just expressions;
- If it deals with programs with **Loops**;
- If it needs (can use) a **Sketch**;
- If it handles pointers (**Pointer**) as we follow Winskel's denotational semantics for IMP, where we have a set of locations Loc (or memory addresses), introduced by `abstract sig Loc {}`. Based on Loc, a state is defined as a binding from Loc to Int (`sig State { bind: Loc -> one Int }`). Finally, an integer variable is defined indirectly (via a pointer) by `sig IntVar extends AExp { name: one Loc }`. That is, an integer variable is not directly associated with an integer but with a location from which an integer is associated with;
- The amount of synthesised problems (**#Probl.**).

8. Conclusion

In this article we have shown how to build a program synthesiser inspired by the imperative programming language IMP [27] using Alloy* [30]. The high level of abstraction of the Alloy* language in combination with its higher-order model finder allowed us to quickly develop a multi-concept synthesiser on top of the denotational semantics of IMP. Our Alloy* Program Synthesiser (APS) produces programs with the notions of state and with elaborate control flow commands like while loops and arrays.[8] The APS scales reasonably well once the right syntactic ingredients are found. The performance of the APS, when considering only the synthesis of expressions, can be comparable to Solar-Lezama's study [41] as can be seen in [42]. For statements synthesis (synthesis of commands like sequential composition and while loops), it is comparable to Srivastava study [3] in terms of expressiveness.

Many synthesisers [3,41] require the design of a sketch and of a specification language together with the development of a translator to a constraint solver. An additional benefit from using Alloy* is the level of confidence of its correctness as it inherits several years of use, development and maturity of Alloy [17].

We introduce a new kind of user intent: soft sketches, which are more flexible than sketches as the commands have no fixed order of execution.

However, as more complex problems are considered, user input becomes difficult. And this was exactly the main goal of considering Genetic Algorithm in this context, which automatically generates such syntactic ingredients.

In what follows we list our future work:

- We intend to use an axiomatic instead of a denotational semantics in the synthesiser. This will allow us to synthesise micro-contracts (smaller functionality) from macro-contracts (larger functionality of a real-world system) by refinement.

[8] Our synthesiser can be found at: https://github.com/PSMFg/psmf/wiki.

We already have preliminary results towards this goal. The axiomatic based synthesiser is very fast compared to denotational version, but currently we are facing problems to deal with the logic substitution operator $[\cdot/\cdot]$, which is absent in Alloy*. This initiative goes in the direction of the works [3–6], but still using the syntax-guided search engine provided by the Alloy* SyGUS algorithm;
- We considered Genetic Algorithm to aid the user on the user's inputs needed by the APS. But another way of attacking this problem is trying to find the minimum scope as reported in the study [43]. Maybe this study cannot help in the soft sketches part but it can alleviate the iterations of the Genetic Algorithm counterpart;
- We intend to perform an exhaustive analysis on the potential benefits and drawbacks when using reuse in the general case in addition to implementing some form of ranking of the best potential functions to be called for a given problem;
- Another interesting trend is to synthesise refactorings. We will explore how to synthesise the left/right-hand side of a refactoring as well as the provisos to assure to correctness of the refactoring;
- We also intend to generate soft sketches automatically by extending our genetic algorithm. This can allow our synthesiser to be used in a large variety of problems;
- We will explore the intermediate states of candidate programs (as reported by Krawiec [39]). This may help to improve the way APS-GA finds out a solution;
- Although nested loops can be produced by our synthesiser, we have not yet explored deeper this feature. Such an exploration also remains as future work;
- Finally, we would like to evolve our synthesiser to deal with multiple arrays, recursion, other types of variables, etc. We plan to do it incrementally by extending the syntax and the semantics of our language and, occasionally, combine different tools.

CRediT authorship contribution statement

Alexandre Correia: Investigation, Software, Validation, Writing. **Alexandre Mota:** Conceptualization, Methodology, Software, Validation, Investigation, Writing. **Juliano Iyoda:** Conceptualization, Methodology, Validation, Investigation, Writing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Alexandre Mota would like to thank the Brazilian National Council for Scientific and Technological Development (CNPq) to support his work under grant number 305729/2018-7.

References

[1] A. Solar-Lezama, Program synthesis by sketching, Ph.D. thesis, EECS Department, University of California, Berkeley, 2008.
[2] S. Gulwani, O. Polozov, R. Singh, Program synthesis, Found. Trends Program. Lang. 4 (2017) 1–119, https://doi.org/10.1561/2500000010.
[3] S. Srivastava, Satisfiability-based program reasoning and program synthesis, Ph.D. thesis, University of Maryland, 2010.
[4] S. Srivastava, S. Gulwani, J.S. Foster, From program verification to program synthesis, in: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, ACM, New York, NY, USA, 2010, pp. 313–326.
[5] N. Polikarpova, I. Kuraj, A. Solar-Lezama, Program synthesis from polymorphic refinement types, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 522–538.
[6] N. Polikarpova, I. Sergey, Structuring the synthesis of heap-manipulating programs, Proc. ACM Program. Lang. 3 (2019), https://doi.org/10.1145/3290385.
[7] O. Polozov, S. Gulwani, FlashMeta: a framework for inductive program synthesis, in: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, 2015, pp. 107–126, https://www.microsoft.com/en-us/research/publication/flashmeta-framework-inductive-program-synthesis/.
[8] V. Le, S. Gulwani, Flashextract: a framework for data extraction by examples, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 542–553.
[9] S. Gulwani, Automating string processing in spreadsheets using input-output examples, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 317–330.
[10] E. Kitzelmann, Inductive programming: a survey of program synthesis techniques, in: International Workshop on Approaches and Applications of Inductive Programming, Springer, 2009, pp. 50–73.
[11] W.A. Ibrahim, M.M. Morcos, Artificial intelligence and advanced mathematical tools for power quality applications: a survey, IEEE Trans. Power Deliv. 17 (2002) 668–673.
[12] S.K. Murthy, Automatic construction of decision trees from data: a multi-disciplinary survey, Data Min. Knowl. Discov. 2 (1998) 345–389.
[13] F.K. Došilović, M. Brčić, N. Hlupić, Explainable artificial intelligence: a survey, in: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO, 2018, pp. 0210–0215.
[14] A. Mota, J. Iyoda, H. Maranhão, Program synthesis by model finding, Inf. Process. Lett. 116 (2016) 701–705, https://doi.org/10.1016/j.ipl.2016.06.003.
[15] A. Milicevic, J.P. Near, E. Kang, D. Jackson, Alloy*: a general-purpose higher-order relational constraint solver, Form. Methods Syst. Des. (2017), https://doi.org/10.1007/s10703-016-0267-2, https://aleksandarmilicevic.github.io/hola.
[16] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, Cambridge, MA, USA, 1993.
[17] D. Jackson, Software Abstractions: Logic, Language, and Analysis, revised ed., MIT Press, 2012.

[18] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, SIGOPS Oper. Syst. Rev. 40 (2006) 404–415, https://doi.org/10.1145/1168917.1168907.
[19] M. Hofmann, Igor2 – an analytical inductive functional programming system: tool demo, in: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ACM, 2010, pp. 29–32.
[20] K. Becker, J. Gottschlich, AI programmer: autonomously creating software programs using genetic algorithms, preprint, arXiv:1709.05703, 2017.
[21] S. Gulwani, Automating string processing in spreadsheets using input-output examples, ACM SIGPLAN Not. 46 (2011) 317–330.
[22] V. Le, S. Gulwani, Flashextract: a framework for data extraction by examples, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 542–553.
[23] R. Singh, S. Gulwani, Learning semantic string transformations from examples, Proc. VLDB Endow. 5 (2012) 740–751.
[24] R. Alur, D. Fisman, R. Singh, A. Solar-Lezama, Sygus-comp 2017: results and analysis, in: Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017, 2017, pp. 97–115.
[25] A. Correia, J. Iyoda, A. Mota, Combining model finder and genetic programming into a general purpose automatic program synthesizer, Inf. Process. Lett. 154 (2020).
[26] R. Alur, R. Bodík, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: FMCAD, IEEE, 2013, pp. 1–8, http://dblp.uni-trier.de/db/conf/fmcad/fmcad2013.html#AlurBJMRSSSTU13.
[27] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, Cambridge, MA, USA, 1993.
[28] C.A.R. Hoare, I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, Laws of programming, Commun. ACM 30 (1987) 672–686, https://doi.org/10.1145/27651.27653.
[29] H.P. Maranhão, Program synthesis from denotational semantics, Master's thesis, Universidade Federal de Pernambuco, 2016.
[30] A. Milicevic, J.P. Near, E. Kang, D. Jackson, Alloy*: a higher-order relational constraint solver, Technical Report, MIT, 2014.
[31] C. Morgan, Programming from Specifications, 2nd edition, Prentice Hall International series in Computer Science, Prentice Hall, 1994.
[32] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, X. Li, UML activity diagram-based automatic test case generation for Java programs, Comput. J. 52 (2009) 545–556.
[33] S.-H. Cha, Comprehensive survey on distance/similarity measures between probability density functions, Int. J. Math. Models Methods Appl. Sci. 1 (2007) 300–307, http://www.gly.fsu.edu/~parker/geostats/Cha.pdf.
[34] E. Torlak, R. Bodik, Growing solver-aided languages with Rosette, in: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013, ACM, New York, NY, USA, 2013, pp. 135–152.
[35] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, ACM, New York, NY, USA, 2006, pp. 404–415.
[36] D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals, Genet. Program. Evol. Mach. 14 (2013) 3–29, https://doi.org/10.1007/s10710-012-9177-2.
[37] Z. Manna, R.J. Waldinger, Toward automatic program synthesis, Commun. ACM 14 (1971) 151–165, https://doi.org/10.1145/362566.362568.
[38] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, P. Kohli, Neuro-symbolic program synthesis, CoRR, arXiv:1611.01855, 2016.
[39] K. Krawiec, Behavioral Program Synthesis with Genetic Programming, Studies in Computational Intelligence, vol. 618, Springer International Publishing, 2016.
[40] X. Qiu, A. Solar-Lezama, Natural synthesis of provably-correct data-structure manipulations, Proc. ACM Program. Lang. 1 (2017) 1–28.
[41] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, SIGOPS Oper. Syst. Rev. 40 (2006) 404–415.
[42] A. Milicevic, Advancing declarative programming, Ph.D. thesis, MIT, 2015.
[43] T. Nelson, S. Saghafi, D.J. Dougherty, K. Fisler, S. Krishnamurthi, Aluminum: principled scenario exploration through minimality, in: 2013 35th International Conference on Software Engineering, ICSE, 2013, pp. 232–241.