Scientific Annals of Computer Science vol. 34 (1), 2024, pp. 39–66
doi: 10.47743/SACS.2024.1.39

Alk: A Formal-Methods-based Educational Platform for Enhancing Algorithmic Thinking[1]

Alexandru-Ioan Lungu[2] , Vlad Teodorescu[3] ,
Andrei Zaborilă[4] , Oana Andrei[5] and Dorel Lucanu[6]

Abstract

Algorithm design courses are fundamental to computer science curricula, but fostering algorithmic thinking in students is challenging due to the diverse skills and creativity required. Dedicated teaching support tools can help both course instructors and students in this effort. We have developed the Alk platform to promote algorithmic thinking, leveraging the theoretical foundations of Matching Logic. Alk features an intuitive algorithm language that provides a flexible computational model suitable for analysis, symbolic execution, and checking properties of algorithms. In this paper, we present an overview of the Alk platform tool and demonstrate, through use cases, how it fosters various algorithmic thinking skills. We conclude that the Alk platform is a valuable tool for learning and teaching algorithms, effectively enhancing students' understanding and skills. Future work will extend its capabilities of supporting symbolic execution, probabilistic algorithms, as well as estimation of execution time, further broadening its impact on computer science education.

Keywords: Algorithm design, Algorithmic thinking, Algorithm analysis, Algorithmic language

[1]Oana Andrei's contribution is based upon work from COST Action CA20111, supported by COST (European Cooperation in Science and Technology)

[2]Alexandru Ioan Cuza University of Iași, Romania, Email: lungualex00@gmail.com

[3]Alexandru Ioan Cuza University of Iași, Romania & BitDefender, Iași, Romania, Email: teodorescu.vlad@yahoo.com

[4]Alexandru Ioan Cuza University of Iași, Romania & BitDefender, Iași, Romania, Email: zabo.zabo20@gmail.com

[5]University of Glasgow, Email: oana.andrei@glasgow.ac.uk

[6]Alexandru Ioan Cuza University of Iași, Romania, Email: dorel.lucanu@gmail.com

1 Introduction

Algorithmic thinking is a central component of computational thinking [36], essential for designing correct and efficient algorithms. It involves a range of cognitive skills, including: decomposition and abstraction (describe, abstract, and decompose problem); designing algorithms; test solution with debugging (detect and fix errors), iteration (repeat algorithm design process), and efficiency optimisation [19]. Mastering these skills is essential for computer science students, while teaching them algorithmic thinking skills is challenging due to the diverse skills and creativity involved. Fostering these skills in students requires effective teaching tools and methodologies.

The conventional method for learning how to design and write algorithms typically involves textbooks, most of which using pseudocode to describe algorithms. Pseudocode serves as an intermediary between a language with a formal semantics (code) and an informal language with free syntax (pseudo). The implicit requirement for pseudocode is that the informal part must be translatable into a formal one to achieve a complete formal description of an algorithm written in pseudocode; however, this task can often be challenging. Consequently, algorithms written in pseudocode cannot be directly executed or tested.

Several textbooks employ programming languages, such as C/C++, Java, Python, to describe algorithms. The clear advantage of this approach is that the algorithms can be executed and tested using an implementation of the programming language. However, the description of the algorithm becomes more laborious, potentially detracting from the focus on algorithmic thinking. Additionally, novices are required to simultaneously learn algorithm design techniques and the intricacies of the programming language, which can be a significant challenge.

This paper presents Alk [25], an open-source educational platform designed to facilitate the learning process of algorithm design and analysis. The Alk platform aims to assist learners in developing a strong foundation in algorithmic thinking. The design and the implementation of the platform have been guided the following key principles:

P1: Incorporate an algorithmic language that is simple, expressive, intuitive, and closely aligned with the language used in textbooks.

P2: Support the execution and testing of both algorithm segments algorithms and complete algorithms.

P3: Support the comprehension of algorithm behaviours and the development of algorithmic thinking.

P4: Support various instructional approaches, ranging from light ones like test-driven design to more rigorous techniques like correct-by-design/construction.

P5: Support the analysis of various aspects of algorithms, including their correctness and efficiency.

P6: Support different types of algorithms such as deterministic, non-deterministic, and probabilistic.

P7: Ground the semantics and supporting tools in formal foundations.

Paper Structure. Section 2 presents an overview of the Alk platform components and features, and briefly describes the foundations of the Alk platform, including the syntax and the semantics for the algorithmic language, the computational model, and the analysis tools. Section 3 demonstrates how Alk can be used to support the acquisition of algorithmic thinking skills through several examples. In Section 4 we review related works, examine how Alk's design and implementation principles support fostering algorithmic thinking, and discussion a preliminary evaluation of the platform based on the feedback from students. Finally, Section 5 ends the paper with some concluding remarks and future work planned for the development of the Alk platform.

2 Alk Features and Foundations

2.1 Platform Components

The current version of the platform [25] includes the following features:

Algorithmic language [27] relying on a formal definition for both the syntax (similar to the imperative part of C or Java) using annotations for program verification (similar to Dafny [20]) and the semantics (inspired from the K Framework approach [6]).

Interpreter for testing algorithms, even when these are partially developed. Since the state of the algorithm is specified at an abstract level, it is easy to specify the initial state of an execution.

Symbolic execution engine [27] for testing algorithms with symbolic values and for checking their properties, such as invariants or simple contracts.

Debugger for understanding the execution of the algorithms and for finding errors in the algorithms design.

Abstract-interpretation-based dataflow analysis component for approximating execution time and for computing information needed for analysis, e.g, identifying the variables modified by a loop or the variable types (when it is possible). This component is currently in alpha version.

Visual Studio Code extension [34] for installing Alk and using all its components. Command line option is also available.

The theoretical foundation of Alk is provided by Matching Logic [31, 5, 4] - a formal logic mainly used for reasoning about the correctness of computer programs. The interpreter, the symbolic execution engine, and the property checker were uniformly developed using the formal definition of the language.

2.2 Syntax

The syntax of the Alk algorithmic language [24] is designed to be:

- *simple*, as it excludes implementation details for fundamental data structures;
- *expressive*, allowing the description of popular paradigms (e.g., deterministic, nondeterministic, probabilistic algorithms) can be described;
- *intuitive*, closely resembling the language used in textbooks.

The most specific part is given by the data structures defining the values for variables, which includes scalars (booleans, integers, floats, strings) and heterogeneous arrays, lists, structures, sets, and maps (see Figure 1). This allows great freedom in modelling the concepts from the problem domain used in algorithm descriptions. For each data structures, the most usual operations and functions are defined (see the reference manual [24]).

$$
\begin{array}{rl}
\mathit{Scalar} ::= & \texttt{int} \mid \texttt{bool} \\
& \mid \texttt{float} \mid \texttt{string} \\
\mathit{Field} :: & \mathit{Id} \\
\mathit{Val} ::= & \mathit{Scalar} \\
& \mid \texttt{array}\langle \mathit{Val} \rangle \\
& \mid \texttt{list}\langle \mathit{Val} \rangle \\
& \mid \texttt{set}\langle \mathit{Val} \rangle \\
& \mid \texttt{map}\langle \mathit{Key}, \mathit{Val} \rangle \\
& \mid \texttt{struct}\langle \mathit{Field} \rangle \\
& \mid \texttt{struct}\langle \mathit{Field}\,(, \mathit{Field})^* \rangle
\end{array}
\qquad
\begin{array}{rl}
\texttt{array}\langle \mathit{Val} \rangle & ::= [(\mathit{Val},)^*] \\
\texttt{list}\langle \mathit{Val} \rangle & ::= \langle (\mathit{Val},)^* \rangle \\
\texttt{set}\langle \mathit{Val} \rangle & ::= \{(\mathit{Val},)^*\} \\
\texttt{struct}\langle F_1 \ldots, F_k \rangle & ::= (F_1 \to \mathit{Val}, \ldots, F_k \to \mathit{Val}) \\
\mathit{Key} & ::= \mathit{Val} \\
\texttt{map}\langle \mathit{Key}, \mathit{Val} \rangle & ::= (\mathit{Key} \mapsto \mathit{Val})^*
\end{array}
$$

Figure 1: Alk values syntax

2.3 Semantics

The semantics of the Alk language is described in [26, 27] and it is formally defined using Matching Logic (ML) [31, 5, 4]. The semantics is given at an abstract level in order to help the understanding the execution of the algorithms and to supply a computation model (see Section 2.4), which is suitable for analysis and rigorous algorithm thinking.

The (operational) semantics of an algorithm can be thought as being an abstract state machine. The abstract states (configurations) are pairs $\langle \alpha \rangle \, \langle \sigma \rangle$, where α is the (piece of) the algorithm to be executed, written as a sequential list, and σ is the (memory) state consisting of pairs *variable-name* $\mapsto$ *value*. A value can be a scalar or a compound value, according to the definition in Section 2.2 The transitions are given by the semantics of the expressions and statements, which are specified as rewrite rules over configurations. Formally, these rewrite rules are particular ML patterns (see [26] for details). As an example, we include here the rules defining the semantics for `if` statement:

$$
\begin{array}{r}
\langle \texttt{if}\ (E)\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \Rightarrow \langle E \rightsquigarrow \texttt{if}\ \_\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \\
\langle V \rightsquigarrow \texttt{if}\ \_\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \Rightarrow \langle \texttt{if}\ V\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \\
\langle \texttt{if true}\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \Rightarrow \langle S_1 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \\
\langle \texttt{if false}\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle \Rightarrow \langle S_2 \rightsquigarrow \alpha \rangle \, \langle \sigma \rangle
\end{array}
$$

where E is an Alk non-value expression and V is a value (here it should be a boolean). The first rule picks up the condition E to be evaluated in the next steps. Assuming that E is evaluated to the value V, this

is put in the placeholder left by E using the second rule. This kind of rules define *structural transitions*, since they "prepare" a configuration for computations: the first rule prepares the configuration to compute the value of the expressions (therefore we call a heating rule), and the second one prepares the configuration for the next step (and therefore we call it a cooling rule). The last two rules choose one of the two branches, according to the value V (supposing that it is boolean); these are also structural rules as no computation is performed.

2.4 Computational Model

Algorithm analysis requires a model of the execution mechanism, a model of the resources used by that mechanism, and a cost model for these resources [8]. The computational model supplied by Alk consists of an abstract state machine defined by its semantics, along with a set of assumptions represented at the meta-level.

Each value in Alk has assigned a *size*: for scalars this is given as a meta-level assumption, and for compound values the size is given by the sum of the components' sizes. For instance, for integers the size could be uniform (all integers have the size $O(1)$), logarithmic (the size of n is $\log n$), or linear (the size of n is n). The size of a state is the sum of the sizes of the values stored withing it.

Each builtin operation over values has assigned a *time*, given also as a meta-level assumption. This is the time of a computation step of the form

$$\langle op_{\text{Alk}}(v_1, \ldots, v_k) \rightsquigarrow \alpha \rangle \langle \sigma \rangle \Rightarrow \langle op_{\text{asm}}(v_1, \ldots, v_k) \rightsquigarrow \alpha \rangle \langle \sigma \rangle$$

where op_{Alk} is the Alk representation of the operation and op_{asm} is the operation "internally" performed by the abstract state machine.

The evaluation of an expression is given by a sequence of transition steps

$$\langle E \rightsquigarrow \alpha \rangle \langle \sigma \rangle \Rightarrow^* \langle v \rightsquigarrow \alpha \rangle \langle \sigma' \rangle$$

and its evaluation time is the sum of times of the one-step computations.

The time of structural transitions is zero. For instance, the time of an execution

$$\langle \texttt{if}\ (E)\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle \langle \sigma \rangle \Rightarrow^* \langle S_i \rightsquigarrow \alpha \rangle \langle \sigma \rangle$$

is the time needed to evaluate E:

$$\langle E \rightsquigarrow \texttt{if}\ \_\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle\, \langle \sigma \rangle \Rightarrow^* \langle V \rightsquigarrow \texttt{if}\ \_\ S_1 \texttt{else}\ S_2 \rightsquigarrow \alpha \rangle\, \langle \sigma \rangle$$

Of course, the result depends on the model we used for value size and operation time. A single model should be used for an execution.

We believe that this approach to defining the computational model for algorithms in Alk is:

- *simple* enough to be easily understood,
- *abstract* enough to avoid the implementation details, and, at the same time,
- *detailed* enough to formally define the execution cost functions.

2.5 Analysis Tools

The soundness of the analysis tools is ensured by two formal mechanisms:

- *symbolic execution* assisted by a *theorem prover* (for instance, the SMT solver Z3), and
- *data flow analysis* combined with *abstract interpretation*.

For instance, the loop invariants and algorithm contracts are checked with symbolic execution as follows [27, 26]:

1. these properties are specified as ML patterns;
2. the ML patterns are translated into symbolic programs;
3. the successful executions of the symbolic programs imply the validity of the properties.

The properties regarding the efficiency and the type-checking are investigated and inferred using data flow analysis combined with abstract interpretation. This work is currently in progress, and the details will be provided elsewhere.

3 Algorithmic Thinking in Alk by Examples

In this section we illustrate the use of various Alk features in supporting the acquisition of algorithmic thinking skills through several examples. We assume that the primary features of the algorithmic language syntax and semantics [27] are straightforward to understand.

3.1 Describing and Testing an Algorithm

Figure 2 shows a randomized algorithm for searching a sorted compact list with all keys distinct. The left-hand side shows the pseudocode representation [8] while the right-hand side the Alk language representation. Note that the two representations are similar: the i-th row on the left-hand side corresponds (and is almost identical) to i-th row on the right-hand side. An exception is the data structure used for the list L: a linked list is used in the pseudocode, whereas an abstract list of structures (see Section 2.2) is used in Alk. While the relationship between L and keys on the left-hand side is implicit (we have to read the explanations from the textbook to relate the two variables), this relationship is explicitly specified in Alk. This serves as an example when pseudocode segments must be formalized for algorithm testing. With Alk, this translation is minimal, intuitive, and smooth.

The method `size()` returns the size of the list, and `at(`i`)` returns the i-th element of the list. Moreover, the Alk algorithm can be executed as it is, whereas for the left-hand side algorithm in pseudocode we need an implementation of the linked lists. Instead of RANDOM function, we used the probabilistic choice `uniform` statement, which uses a uniform distribution.

```
COMPACT-LIST-SEARCH(L,n,k)
 1  i = L
 2  while i ≠ NIL and key[i] < k
 3        j = RANDOM(1,n)
 4        if key[i] < key[j] and key[j] ≤ k
 5              i = j
 6              if key[i] == k
 7                    return i
 8        i = next[i]
 9  if i == NIL or key[i] > k
10        return NIL
11  else return i
```

```
COMPACT_LIST_SEARCH(L, n, k) {
  i = 0;                                  1
  while(i<L.size() && L.at(i).key<k)      2
  {
    uniform j from [1..n];                3
    if (L.at(i).key<L.at(j).key &&        4
        L.at(j).key<=k) {
      i = j;                              5
      if (L.at(i).key==k)                 6
        return i;                         7
    }
    i = i+1;                              8
  }
  if (i==L.size() || L.at(i).key>k)       9
    return -1;                            10
  else return i;                          11
}
```

compact-list-search.alk

Figure 2: A compact list search algorithm as described in textbooks [8] (left) and represented in Alk (right)

We can test the Alk algorithm in Figure 2 using the Alk Interpreter by including a calling statement in the same file:

```
x = COMPACT_LIST_SEARCH(L, n, k);
```

and specifying the initial state (i.e., the initial values for variables `L`, `n`, and `k`) in the command line[7] as the value of the option `-i`:

```
$ alki -a compact-list-search.alk -m -i "L |-> < {key -> 5}, {key -> 8},
 {key -> 12}, {key -> 15}> n |-> 3 k |-> 12"

x |-> 2
k |-> 12
L |-> < {key -> 5}, {key -> 8}, {key -> 12}, {key -> 15} >
n |-> 3
Note that the executed algorithm is probabilistic.
The probability for this execution is: 0.1111111111
```

The option `-m` is for displaying metadata, which includes the final state. In this way, the execution of the algorithm is seen as a transition process from the initial state to the final state. The final state also includes the computed value of the variable `x`. We see that the interpreter also displays the probability of the execution, given here by the execution of `uniform`.

We can also decompose the problem to execute just a fragment of the algorithm, such as testing the body of the `while` loop with the Alk Interpreter:

```
uniform j from [1..n];
if (L.at(i).key<L.at(j).key && L.at(j).key<k) {
     i = j;
     if (L.at(i).key == k)
       return i;
}
i = i+1;
```

fragment.alk

Supposing that this fragment of algorithm is included in the file `fragment.alk`, it can be directly executed by mentioning only the starting state:

```
$ alki -a fragment.alk -m -i "L |-> < {key -> 5}, {key -> 8},
  {key -> 12}, {key -> 15}> n |-> 3 k |-> 12 i |-> 1 "

i |-> 2     j |-> 3    k |-> 12    n |-> 3
L |-> < {key -> 5}, {key -> 8}, {key -> 12}, {key -> 15} >
Note that the executed algorithm is probabilistic.
The probability for this execution is: 0.3333333333
```

[7]The VS Code extension supplies a menu for various kinds of executions.

3.2 Incremental Algorithm Design and Verification

Acquiring rigorous algorithmic thinking skills is not a straightforward process. Most of the students initially rely on their intuition to design an algorithm and then conduct tests to check if the designed algorithm performs as intended. In this section, we demonstrate how the formal methods integrated into Alk can be used to systematically develop algorithms, ensuring that users are confident they have designed a correct solution.

We consider the following problem as a running example:

> *A box contains n matches. Two players take turns removing one or two matches from the box. The player whose turn it is and who has no legal move to make loses. Who has the winning strategy? What is that strategy?*

We demonstrate how the Alk platform can be used to develop an algorithmic solution for this problem, along with a proof of its correctness. The solution is obtained through a stepwise incremental process.

3.2.1 The Base Case

It is clear that for $n = 1$ or $n = 2$, the first player can win by taking all n matches. For $n = 3$, the second player has a winning strategy by taking the remaining matches after the first player's turn; this strategy can be described as a nondeterministic algorithm:

```
// the first turn
choose take from {1,2};
n = n - take;
// the next turn
take = n;
n = n - take;
```

and tested using the Alk interpreter:

```
$alki -a step1.alk -m -i "n |-> 3"

take |-> 2
n |-> 0

Note that the executed algorithm is nondeterministic.
```

Note the use of the nondeterministic choice statement `choose`, that means that the value for `take` is arbitrarily chosen, i.e., no probability distribution

is assumed. The above run is for $n = 3$ and only one branch of the nondeterministic executions tree is executed: we see that the first player took one match while the second player took the remaining two matches. We can execute the algorithm exhaustively by adding the option `-e` to the command line:

```
$ alki -a step1.alk -m -i "n |-> 3" -e

take |-> 2
n |-> 0

take |-> 1
n |-> 0
```

The interpreter displays now the two possible nondeterministic executions of the algorithm. The final configurations are displayed in an arbitrary order.

3.2.2 First Abstraction

A first question that comes to mind is whether the step described by step1.alk can be generalized to the case $n = 3k$, for some integer $k > 0$. We may try to transform this step into a winning strategy if the property $n = 3k$, for some k, is preserved by this step. Since n is greater than three, it follows that the next turn must take n mod 3 matches:

```
@havoc n: int;
@assume n % 3 == 0;
// the first turn
choose take from {1,2};
n = n - take;
// the next turn
take = n % 3;
n = n - take;
@assert n % 3 == 0;
```

n3k-symb.alk

The annotation statement `@havoc` initializes the variable `n` with an integer symbolic value, and `@assume` constraints this value to be a multiple of three. The statement `@assume` checks if the same property holds at the end of the step. We use symbolic execution for checking this algorithm, relying on the SMT solver Z3 for checking the feasibility. The symbolic execution validates our assumption as there are no errors reported:

```
$ alki -a n3k-symb.alk -s -smt="Z3"
Note that the execution was symbolic.
Note that the Z3 engine was used for verification.
Note that the executed algorithm is nondeterministic.
```

The last output message highlight that only one nondeterministic branch was verified. In order to check all branches, we may replace the statement

```
choose take from 1,2;
```

with the following two semantic equivalent statements:

```
@havoc take: int;
@assume 1 <= take && take <= 2;
```

3.2.3 A Winning Strategy for $n = 3k$

Since $n = 3k$ is an invariant of the base step, we can transform it into a winning strategy for the second player:

```
winingStrategy(n) {
  turn = 1;
  while (n > 0) {
    // the current turn
    choose take from {1,2};
    n = n - take;
    // the next turn
    turn = 3 - turn;
    take = n % 3;
    n = n - take;
    // prepare the next current turn (if any)
    if (n > 0) turn = 3 - turn;
  }
  return turn;
}
```

step2.alk

The refinement step consists of including the base step from n3k-symb.alk into in a `while` loop. We also consider a variable `turn` that maintains the player's number whose turn follows. In order to check our assumption, we test this strategy using the following statement, noting that no declaration is needed for the array `w`:

```
for (i = 1; i < 6; ++i)
  w[i-1] = winingStrategy(3*i);
```

The execution of this statement produces the following output:

```
w |-> [2, 2, 2, 2, 2]
i |-> 6
```

We see that for all five values, the winner is the second player, which corresponds to our expectations.

Now we can use the Alk platform to prove that the strategy given in step2.alk is indeed a winning strategy. However, using only the invariant $n \bmod 3 = 0 \wedge n \geq 0$ is not enough to conclude that the second player is the winner. We have to enrich the loop invariant with a relation between n and the player whose turn it is to play. Based on the previous experience, we first tried $n > 0 \implies turn = 1$. But it is still not enough, because at the end of the loop we have to conclude that $turn = 2$, so we have to add $n = 0 \implies turn = 2$. The annotated algorithm is as follows:

```
1  winingStrategy(n)
2  @requires n: int
3  @requires n > 0 && n % 3 == 0
4  @ensures \result  == 2
5  {
6    turn = 1;
7    while (n > 0)
8    @modifies n, turn;
9    @invariant n % 3 == 0 && n >= 0
10   @invariant n > 0 ==> turn == 1
11   @invariant n == 0 ==> turn == 2
12   {
13     // the same as in step2.alk
14   }
15   return turn;
16 }
```

step2-with-spec.alk

Since Alk language does not include type declarations for variables and the symbolic execution needs such information, we may add type assertion as pre-conditions for parameters. For the other variables, the type is automatically inferred. The variable \result is used for specifying the returned value.

The properties specified as annotations can be checked using the symbolic execution engine and Z3 SMT solver [27, 26]:

```
$alki -a step2-with-spec.alk  -s -smt="Z3"
[9:15] Loop invariant was verified!
...
Successfully verified: winingStrategy
```

3.2.4 A General Winning Strategy

The general strategy is obtained by considering the two cases: $n \bmod 3 = 0$ and $n \bmod 3 \neq 0$. We demonstrated how for $n \bmod 3 = 0$ the second player has a winning strategy. If $n \bmod 3 \neq 0$, then the first player in her first turn can establish $n \bmod 3 = 0$ and, since in the loop she has the second turn, it follows that she has a wining strategy. The algorithmic description of the resulting strategy, together with its proof, is as follows:

```
1  winingStrategy(n)
2  @requires n: int;
3  @requires n > 0;
4  @ensures \result: int
5  @ensures (\old(n) % 3 == 0 ==> \result  == 2) &&
6           (\old(n) % 3 != 0 ==> \result  == 1)
7  {
8   turn = 1;
9   if (n % 3 != 0) {
10    take = n % 3;
11    n = n - take;
12    if (n > 0) turn = 2;
13  }
14  while (n > 0)
15  @modifies n, turn
16  @invariant n % 3 == 0 && n >= 0
17  @invariant n>0  ==> (\old(n) % 3 == 0 ==> turn == 1) &&
18                      (\old(n) % 3 != 0 ==> turn == 2)
19  @invariant n==0 ==> (\old(n) % 3 == 0 ==> turn == 2) &&
20                      (\old(n) % 3 != 0 ==> turn == 1)
21  {
22    // the same as in step2.alk
23  }
24  return turn;
25 }
```

winstrat.alk

The invariant relating n and *turn* are updated according to the new hypothesis under which a loop starts.

3.3 Step-by-step Executions and Debugging

A debugger, in a platform devoted to algorithms design and analysis, is useful at least for two purposes: 1) to understand the step-by-step execution of an algorithm, especially for someone less familiar with programming, and 2) to find the source and the cause of a bug. In this section we show how the Alk debugger can be used for finding and fixing a bug.

Figure 3 shows an implementation of the general winning strategy with a small, but subtle, bug.

```
1  winingStrategy(n)  {
2      turn = 1;
3      if (n % 3 != 0)  {
4          take = n % 3;
5          n = n - take;
6          turn = 2;
7      }
8      while (n > 0)  {
9          // the current turn
10         choose take from {1,2};
11         n = n - take;
12         // the next turn
13         turn = 3 - turn;
14         take = n % 3;
15         n = n - take;
16         // prepare the next current turn (if any)
17         if (n > 0) turn = 3 - turn;
18     }
19     return turn;
20 }
```

winstrat-with-bug.alk

Figure 3: Implementation of the general winning strategy with a bug

Testing the algorithm with

```
for (i = 1; i < 10; ++i) {
    w[i-1] = winingStrategy(i);
}
```

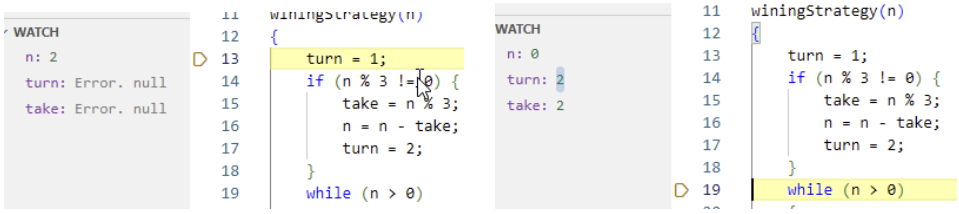
we obtain

```
w |-> [2, 2, 2, 1, 1, 2, 1, 1, 2]
i |-> 10
```

We see that the algorithm returns the second player as the winner for $n = 1, 2$, whereas the first player is expected to win. If we annotate the algorithm with invariants and execute it symbolically, we get a loop invariant violation message in the Visual Studio Code extension[8]:

```
@invariant n > 0 ==> (\old(n) % 3 == 0 ==> turn == 1) &&
                     (\old(n) % 3 != 0 ==> turn == 2);
@invariant n == 0 ==> ((\old(n) % 3 == 0 ==> turn == 2) &&   <--Loop invariant violation!
                       (\old(n) % 3 != 0 ==> turn == 1));
```

This message is not sufficiently informative to point to the exact error. Therefore we start the debugger for $n = 2$. We watch all three variables handled by the algorithm: `n`, `take`, and `turn`. After stepping through the first 2 lines of the `winningStrategy` function (see Figure 4, the left-hand side), the execution enters the `then` branch of the first `if`. When the `if` statement is completely executed (see Figure 4, the right-hand side), we notice that the values of `n` and `turn` become inconsistent: the value of `n`



Before the first step After five steps

Figure 4: Using the debugger for step-by-step execution

is zero, which means that the game is over after the first move; it follows that the value of `turn` was wrongly updated, it should have been unchanged. The bug can now be solved by checking that the game is not over ($n > 0$) before changing the turns (see the line 12 in winstrat.alk, page 52).

3.4 Using Alk together with Generative AI Tools

The AI tools, like ChatGPT[9], are versatile enough to supply algorithmic solutions to many problems. However, these answers are not always accurate and they must be validated by a trusted tool. Here we describe a scenario

[8] Visual Studio Code extension for Alk [34]
[9] OpenAI's ChatGPT https://openai.com/chatgpt

```
function computeSum(n, m):
    list = [2, 3, ..., n]  // Initialize the list with numbers 2 to n
    for round = 1 to m:
        minIndex = 0
        for i = 1 to length(list):
            if list[i] < list[minIndex]:
                minIndex = i
        list[minIndex] = list[minIndex] * list[minIndex] //Replace ...
    sum = 0
    for i = 1 to length(list):
        sum = sum + list[i] //Compute the sum of ...
    return sum

// Example usage
n = 5
m = 3
result = computeSum(n, m)
print(result)  // Output: 66
```

Figure 5: ChatGPT answer

showing how the Alk platform can be used to analyse, invalidate, and fix such an inaccurate answer.

ChatGPT 3.5 was asked to supply an algorithm for the following problem, inspired from the Euler Project site[10]:

> *A list initially contains the numbers* $2, 3, ..., n$*. At each round, the smallest number in the list is replaced by its square. If there is more than one such number, then only one of them is replaced. Write an algorithm, in pseudo-code, that computes the sum of all numbers in the list after* m *rounds.*

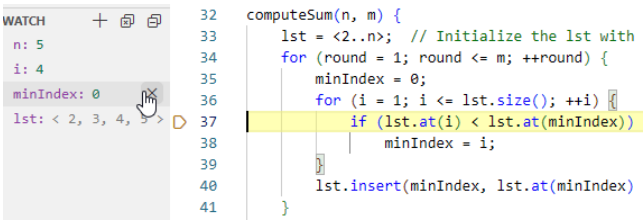
The answer given by ChatGPT 3.5 is displayed in Figure 5. It is interesting to see that ChatGPT 3.5 suggest the same test example as that from Euler project. We wrote a faithful translation of this algorithm in Alk and then tried to test it on the suggested example, and we got an error message (see Figure 6.a)). Running the debugger (see Figure 6.b)), we immediately see that it should be executed until `lst.size()-1` (= `length(list)-1`). This suggests that the last `for` is also wrong: changing the loop body to `sum = sum + lst.at(i-1);` fixes the error. Running the fixed version of the algorithm on the given example, we see that the sum should be 34 and not 66, as ChatGPT 3.5 suggests.

This simple but suggestive scenario shows that tools like the Alk platform can be seamlessly employed to understand and rectify algorithmic

[10]Euler Project - Problem 822 https://projecteuler.net/problem=822

```
computeSum(n, m) {
    lst = <2..n>;  // Initialize the lst with numbers 2 to n
    for (round = 1; round <= m; ++round) {
        minIndex = 0;
        for (i = 1; i <= lst.size(); ++i) {
            if (lst.at(i) < lst.at(minIndex))   <--The position in the list is out of bounds.
                minIndex = i;
        }
        lst.insert(minIndex, lst.at(minIndex) * lst.at(minIndex));  // Replace the smallest number with its square
    }
    sum = 0;
    for (i = 1; i <= lst.size(); ++i)
        sum = sum + lst.at(i);  // Compute the sum of all numbers in the lst
    return sum;
}
```

a) The Alk interpreter detects a first error



b) The Alk debugger finds the place of the error

Figure 6: The Alk translation invalidates the solution suggested by ChatGPT

solutions proposed by generative AI systems. Although more recent versions of ChatGPT and other language models have demonstrated improved performances and continue to advance, they are still expected to generate solutions with subtle bugs. These bugs can only be identified and resolved with the use of advanced verification-aware languages.

4 Discussions

4.1 Related Work

There are notable similarities between the Alk language and Python, particularly in their classification as dynamic languages. However, Alk differs in its data structures and syntax for control flow, which are more closely aligned with those of C/C++. While Python is a viable option for writing and testing algorithms, it necessitates the use of external tools for algorithm analysis. For instance, Nagini [13], which is operates on the Viper verification infrastructure [28], is required for such tasks.

The development of the Alk platform conceptually adheres to the methodology and logical foundations promoted by the K Framework[11] [6, 7], encompassing the formal definition of the language and its analysis components. Initial versions of the Alk language were written in K, serving as specifications for the subsequent Java-based implementation. The logical foundation of K is Matching Logic (ML), which we utilised to define the semantics and prove the soundness of the analysis tools [27, 26]. The development of the symbolic execution engine follows the methodology described in [23].

The annotation language used for the analysis is the usual one used by program verifiers such as Dafny [20], Why3 [3], Frama-C [17], JML [15], Key [1], Verifast [35]. These annotations are translated to reachability properties expressed as matching logic patterns [22]. The verification of these reachability properties using symbolic execution follows the methodology described in [33, 23]. The data-flow analysis and abstract interpretation components are implementing following the classical approach [29, 30, 9]. Although the Alk platform includes annotations inspired by program verification languages such as those mentioned above, these are pragmatically employed in algorithm analysis in a distinct manner. Typically, program specifications are complex as they must consider implementation details and the problem domain, making them challenging to teach [10, 12, 2]. In contrast, the annotations used in algorithm analysis within Alk are simpler, directly expressed in the terms of the abstract data structures included in the platform, and are oriented towards understanding the behavior of the algorithm.

In Section 3.2, we illustrated a technique inspired by Correct-by-Construction (CbC) [18, 32], which involves the incremental construction of functionally correct programs. Given that Alk is intended for educational purposes, we propose a lighter version of CbC based on Alk to better facilitate the learning process.

4.2 Realisation of Alk's Design Principles and Alignment with Algorithmic Thinking

In the following, we discuss the realisation of the seven design and implementation principles of the Alk platform. We also explore the alignment between these principles and the algorithmic skills outlined in listed in Section 1, as well as how these principles and skills are exemplified in the use cases

[11] K Framework https://kframework.org/

presented in Section 3.

Describing and Testing an Algorithm. The example in Section 3.1 effectively demonstrates design principles P1 (regarding simplicity and expressiveness). The algorithmic language in Alk, designed to be simple and similar to textbook languages, helps students break down and abstract problems, thereby supporting decomposition and abstraction skills.

Incremental Algorithm Design and Verification. The example in Section 3.2 illustrates principles P4 (supporting various instructional approaches), P6 (supporting different types of algorithms), and P7 (grounding in formal foundations). This example highlights the iterative nature of algorithm design, aiming to encourage students to repeat and refine their approaches, which aligns with the algorithmic thinking skill of iterative design. Similar to the previous example, it also showcases the platform's capability to handle diverse types of algorithm, including deterministic and non-deterministic, hence expanding students' design capabilities. The formal methods integrated into the platform, for instance reasoning about loop invariants, ensure that students not only design but also verify algorithms rigorously.

Step-by-step Executions and Debugging The example in Section 3.3 demonstrates principle P2 (supporting execution and testing) and P5 (supporting analysis of correctness). This feature of Alk enables students to execute algorithms step-by-step, providing immediate feedback and facilitating the debugging process. This aligns with the algorithmic thinking skills of testing solutions and debugging, where students learn to detect and fix errors iteratively. The interactive nature of this example helps students understand algorithm behaviour, making debugging a more intuitive and effective learning experience.

Understanding and Improving Algorithmic Solutions Proposed by Generative AI Systems. The example in Section 3.4 highlights principle P3 (supporting comprehension and development of algorithmic thinking) and and P5 (supporting analysis of correctness). The integration of AI tools such as ChatGPT and Copilot for generating solutions as code to algorithmic problems requires strong code literacy skills to ensure correctness [11]. We argue the same rationale applies to pseudocode solutions. While generative

AI tools can rapidly produce initial solutions, learners must critically analyse and verify these outputs to avoid becoming passive consumers of potentially flawed or incorrect (pseudo)code. Our example in Section 3.4 demonstrates how the platform can help students understand and improve AI-generated solutions, while fostering design, decomposition, testing, and debugging skills in algorithmic thinking.

Thereby the examples in Section 3 demonstrate how Alk's principles support the development of algorithmic thinking in students. However, while the Alk platform supports the analysis of algorithm correctness, its current implementation of principle P5 (supporting analysis of correctness and efficiency) is limited in terms of efficiency analysis. The evaluation of efficiency properties, particularly the estimation of worst-case execution time, as well as type-checking, are being explored through data flow analysis combined with abstract interpretation. This work is ongoing. Enhancing these areas will significantly improve the platform's ability to support the algorithmic thinking skill of efficiency optimisation.

4.3 Preliminary Student Evaluation

The Alk platform is used as a support tool for the Algorithm Design mandatory course for first year students at the Faculty of Computer Science of the Alexandru Ioan Cuza University of Iaşi. Its use is not compulsory but recommended. We gathered informal feedback from over 30 students on topics relating to the ease of writing and testing algorithms in Alk, installing the Visual Studio Code extension, and the overall helpfulness of the platform for learning and understanding algorithm design.

Most of the students did not encounter significant problems with installing the VS Code extension; some experienced minor issues, while very few faced serious difficulties. Some students reported challenges in using the algorithmic language to write algorithms, citing a lack of documentation and examples as primary reason. Although the platform includes some examples, it would benefit from more comprehensive documentation, including methodologies for designing and writing algorithms in Alk (similar to those from Section 3.2 and Section 3.4).

Approximately half of the students initially had problems in testing their algorithms. This difficulty arose because they were not accustomed to specifying the initial configuration as input, expecting instead to enter input data via the keyboard. In the current VS Code extension, the initial

configuration is specified in the Settings window, which is somewhat non-standard. We are exploring better solutions for this issue.

Many students indicated that the platform helped them learn to write algorithms, although a small group, primarily those who had issues with installation, did not find it helpful. The analysis tools were not widely used, as their implementation is still at the prototype level and the documentation is incomplete.

It is worth noting that the Alk platform proved highly useful during the COVID-19 pandemic lockdown by facilitating "study from home." Many students engaged with Alk during the "online" teaching sessions, running and analysing algorithms from home, reporting bugs, sharing algorithms, and building libraries.

It is important to note that this student feedback was collected as part of routine educational activities and not through a formal research study. Consequently, these findings are preliminary and should be interpreted with caution. Future evaluations will be based on rigorous research methodologies, adhering to formal ethical guidelines, including obtaining Institutional Review Board (IRB) approval. This will ensure our findings are robust, reliable, and ethically sound.

5 Conclusions and Future Work

We developed the Alk platform to enhance algorithmic thinking by leveraging the theoretical foundations of Matching Logic. Featuring an intuitive algorithm language, Alk provides a flexible computational model suitable for analysis, symbolic execution, and automated checking of properties of algorithms. This paper has presented an overview of the Alk platform's technical capabilities and demonstrated, through various use cases, its effectiveness in fostering algorithmic thinking skills. To maximise the platform's educational potential, we will provide in the future a comprehensive didactic guide, detailing situations and issues specific to teaching and learning algorithms [16]. The scenarios described in this paper represent the initial step towards developing such a guide.

We believe that the Alk platform is a valuable tool for learning and teaching algorithms, while also fostering algorithmic thinking skills, but it should not be used in isolation. To increase its effectiveness, it must be supplemented with appropriate methodologies, such as test-driven algorithms design and incremental correct-by-design/construction approaches.

Additionally, it is crucial to provide illustrative and guiding examples to help students better understand and apply these methodologies.

Utilising the Alk platform does not require special technical skills; however, some familiarity with Visual Studio Code is helpful. Our observations indicated that students tend to prefer using dedicated integrated development environments (IDEs) over the command line option.

The symbolic execution engine is derived directly from the operational semantics of the algorithmic language and it is assisted by the Z3 SMT solver in order to check the feasibility of the path conditions and the validity of assertions. A Boogie [21, 14] backend is under development. This backend will allow to enrich the annotation language with means allowing to specify properties from the problem domain to be used in the algorithm analysis.

The analysis of algorithms also encompasses the evaluation of their efficiency, particularly the estimation of the worst-case execution time. A component based on data flow analysis and abstract interpretation dedicated to this aspect is currently under development. The main idea involves annotating operations with the necessary information to derive, for example, recurrence relations that specify an upper bound.

Another challenge we aim to address in the future is the extension of the analysis capabilities to probabilistic algorithms.Additionally, the current version of the debugger is limited to concrete executions. We are exploring the potential benefits of extending it to support symbolic execution and investigating the design and implementation strategies required for such extension.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.

[2] Sandrine Blazy. Teaching deductive verification in Why3 to undergraduate students. In Brijesh Dongol, Luigia Petre, and Graeme Smith, editors, *3rd International Workshop and Tutorial on Formal Methods Teaching, FMTea 2019*, volume 11758, pages 52–66. Springer, 2019. doi:10.1007/978-3-030-32441-4_4.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer*, 17(6):709–727, 2015. doi:10.1007/s10009-014-0314-5.

[4] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:100638, 2021. doi:10.1016/j.jlamp.2021.100638.

[5] Xiaohong Chen and Grigore Roşu. Matching μ-logic. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785675.

[6] Xiaohong Chen and Grigore Roşu. Matching mu-logic: Foundation of K framework. In *8th Conference on Algebra and Coalgebra in Computer Science (CALCO'19)*, volume 139 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.CALCO.2019.1.

[7] Xiaohong Chen and Grigore Roşu. **K** - A semantic framework for programming languages and formal analysis. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *5th International School on Engineering Trustworthy Software Systems*, volume 12154 of *Lecture Notes in Computer Science*, pages 122–158. Springer, 2019. doi:10.1007/978-3-030-55089-9_4.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[9] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[10] Léo Creuse, Claire Dross, Christophe Garion, Jérôme Hugues, and Joffrey Huguet. Teaching deductive verification through Frama-C and SPARK for non computer scientists. In Brijesh Dongol, Luigia Petre, and Graeme Smith, editors, *3rd International Workshop and Tutorial on Formal Methods Teaching, FMTea 2019*, volume 11758, pages 23–36. Springer, 2019. doi:10.1007/978-3-030-32441-4_2.

[11] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves,

Eddie Antonio Santos, and Sami Sarsa. Computing education in the era of generative AI. *Commununications of the ACM*, 67(2):56–67, 2024. doi:10.1145/3624720.

[12] Jose Divasón and Ana Romero. Using Krakatoa for teaching formal verification of Java programs. In Brijesh Dongol, Luigia Petre, and Graeme Smith, editors, *3rd International Workshop and Tutorial on Formal Methods Teaching, FMTea 2019*, volume 11758 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2019. doi:10.1007/978-3-030-32441-4_3.

[13] Marco Eilers and Peter Müller. Nagini: A static verifier for Python. In Hana Chockler and Georg Weissenbacher, editors, *30th International Conference on Computer Aided Verification, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018*, volume 10981 of *Lecture Notes in Computer Science*, pages 596–603. Springer, 2018. doi:10.1007/978-3-319-96145-3_33.

[14] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The Boogie verification debugger (tool paper). In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *9th International Conference on Software Engineering and Formal Methods, SEFM 2011*, volume 7041 of *Lecture Notes in Computer Science*, pages 407–414. Springer, 2011. doi:10.1007/978-3-642-24690-6_28.

[15] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. Formal specification with the Java modeling language. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, pages 193–241. Springer, 2016. doi:10.1007/978-3-319-49812-6_7.

[16] Norbert Hundeshagen and Martin Lange. A proposal for a framework to accompany formal methods learning tools - (short paper). In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *4th International Workshop and Tutorial on Formal Methods Teaching, FMTea 2021*, volume 13122 of *Lecture Notes in Computer Science*, pages 35–42. Springer, 2021. doi:10.1007/978-3-030-91550-6_3.

[17] Nikolai Kosmatov and Julien Signoles. Frama-C, A collaborative framework for C code verification: Tutorial synopsis. In Yliès Falcone and César Sánchez, editors, *16th International Conference on Runtime Verification, RV 2016*, volume 10012 of *Lecture Notes in Computer Science*, pages 92–115. Springer, 2016. doi:10.1007/978-3-319-46982-9_7.

[18] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012. doi:10.1007/978-3-642-27919-5.

[19] Timothy H. Lehmann. Using algorithmic thinking to design algorithms: The case of critical path analysis. *The Journal of Mathematical Behavior*, 71:101079, 2023. doi:10.1016/j.jmathb.2023.101079.

[20] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *16th International Conference Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-16, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

[21] K. Rustan M. Leino. Program proving using intermediate verification languages (IVLs) like Boogie and Why3. In Ben Brosgol, Jeff Boleng, and S. Tucker Taft, editors, *2012 ACM Conference on High Integrity Language Technology, HILT '12*, pages 25–26. ACM, 2012. doi:10.1145/2402676.2402689.

[22] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. Generating proof certificates for a language-agnostic deductive program verifier. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):56–84, 2023. doi:10.1145/3586029.

[23] Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. A generic framework for symbolic execution: A coinductive approach. *Journal of Symbolic Computation*, 80:125–163, 2017. doi:10.1016/j.jsc.2016.07.012.

[24] Alexandru-Ioan Lungu and Dorel Lucanu. Alk Language reference manual. https://github.com/alk-language/java-semantics/wiki/Reference-Manual. Accessed: October, 2022.

[25] Alexandru-Ioan Lungu and Dorel Lucanu. Alk platform. https://github.com/alk-language/java-semantics. Accessed: June, 2024.

[26] Alexandru-Ioan Lungu and Dorel Lucanu. A matching logic foundation for Alk. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *19th International Colloquium on Theoretical Aspects of Computing, ICTAC 2022*, volume 13572 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2022. doi:10.1007/978-3-031-17715-6_19.

[27] Alexandru-Ioan Lungu and Dorel Lucanu. Supporting algorithm analysis with symbolic execution in Alk. In Yamine Aït Ameur and Florin Craciun, editors, *16th International Symposium on Theoretical Aspects of Software Engineering, TASE 2022*, volume 13299 of *Lecture Notes in Computer Science*, pages 406–423. Springer, 2022. doi:10.1007/978-3-031-10363-6_27.

[28] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017. doi:10.3233/978-1-61499-810-5-104.

[29] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. doi:10.1007/978-3-662-03811-6.

[30] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. MIT Press, 2020.

[31] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4), 2017. doi:10.23638/LMCS-13(4:28)2017.

[32] Tobias Runge, Ina Schaefer, Loek Cleophas, Thomas Thüm, Derrick G. Kourie, and Bruce W. Watson. Tool support for correctness-by-construction. In Anne Koziolek, Ina Schaefer, and Christoph Seidl, editors, *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-310 of *Lecture Notes in Informatics*, pages 93–94. Gesellschaft für Informatik e.V., 2021. doi:10.18420/SE2021_34.

[33] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuctă, Brandon M. Moore, Traian-Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. *Logical Methods in Computer Science*, 15(2), 2019. doi:10.23638/LMCS-15(2:5)2019.

[34] Vlad Teodorescu, Andrei Zaborilă, and Dorel Lucanu. Alk VS Code extension. https://github.com/Andreizabo/alk-vscode-extension/tree/main. Accessed: October, 2022.

[35] Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11(3), 2015. doi:10.2168/LMCS-11(3:19)2015.

[36] Jeannette M. Wing. Computational thinking. *Commununications of the ACM*, 49(3):33–35, 2006. doi:10.1145/1118178.1118215.