DOI: 10.36558/rsc.v13i2.8160

Using a First-Order-Based Language to Automatically Synthesize Valid Arguments of Propositional Calculus

Elthon Oliveira
Núcleo de Ciências Exatas
Campus Arapiraca
Universidade Federal de Alagoas
elthon@arapiraca.ufalbr

Filipe Oliveira
Núcleo de Ciências Exatas
Campus Arapiraca
Universidade Federal de Alagoas
filipe.oliveira@arapiraca.ufal.br
Bolsista Pibic/CNPq e Fapeal

Rafaella Rosendo
Núcleo de Ciências Exatas
Campus Arapiraca
Universidade Federal de Alagoas
rafaella.rosendo@arapiraca.ufal.br
Bolsista Pibic/CNPq

ABSTRACT

The elaboration of problems with common specific characteristics is considered a tedious task on the part of the teacher. In the context of Logic discipline, this article presents an approach designed for generating propositional calculus formulas and valid arguments. Such a process occurs according to parameters customized by the user. This work adapts the technique of Sketch Generation from Program Synthesis, which is used in conjunction with constraint programming in the Alloy modeling language to make formulas and arguments generation possible. Resources capable of generating elements based on parameters supplied by the user were developed in the form of a mobile app. Such an app hides from the user all the complexity of the process.

CCS Concepts

•Theory of computation → Logic; •Applied computing → Education;

Keywords

Propositional Calculus; Sketch; Valid Arguments

1. INTRODUCTION

Program Synthesis [7] is the act of automatically finding a program in a specific programming language. This search must satisfy the userÂ intention which is described as a specification. This problem has been considered the *Holy Grail* of Computer Science. Some researchers considered Program Synthesis one of the main problems in programming theory [13]. In this paper, we use and adapt the *Program Sketching* approach, along with constraint programming, to synthesize valid *Propositional Calculus* formulas and arguments. These techniques are arranged through the written specifications, which have the purpose of being used in the Alloy modeling language.

In the context of subject formal teaching, creating new problems to be solved by the students is a habitual task of the professor. Considering that these problems need specific solving characteristics, as a specific level of difficulty or that these problems need to involve certain concepts, the task can be deeply tedious. A solution for that would be to automatize the problem generation.

The approach presented here seeks to eliminate manually generating valid arguments. Moreover, this approach provides a tool for the students to practice the subject concepts and for the professors, which will have an inexhaustible source of practice material for their students. The tool developed with this approach will be capable of preventing fraud [11] in the classroom or MOOCs (*Massive Open Online Courses*), since each student can receive a different set of problems but with the same difficulty level.

The resources developed in this work aim for the automatic generation of arguments and formulas of *Propositional Calculus*. Such a generation occurs based on the parameters provided by the user. These parameters are moved through requisitions made in an API, developed to establish communication between the mobile tool and the server where Alloy specifications and the synthesis engine are located. The server processes the given parameters and returns the elements (formulas or arguments) to the user.

This paper is organized as follows. First are quoted related works on Section 2. Some essential concepts for understanding this paper approach are presented in Section 3. In Section 4 the solution architecture is exhibited. The resources developed to validate the approach are presented in Section 5. Conclusions about the paper and future works are mentioned in Section 6.

2. RELATED WORK

The work [1] presents two main components aimed at *natural deduction*: the generation of these components is intended to allow computer-aided education for this great domain. The key technology used here is called *Universal Graph of Proofs (UPG)*, the technology encodes all the possible inference rules' applications overall small abstracted propositions using its bit vector-based truth table representation. This technology allows the generation of solutions and, from *backward research* made in the *UPG* it is possible to generate problems. The main algorithmic understanding for problem generation is performing the modeling as a reversed generation of the solution, and the needed back search is abled because of *UPG*. Therefore, the authors consider two ends in this segment, producing similar problems and parameterized problems.

In [16] the authors describe a method for automatic feedback generation to initial programming problems, using an extension of the *Program Sketching* approach. The solving of ruler and compass geometry problems is described in [6]. The tool proposed by the authors can successfully synthesize constructions for a variety of geometry problems. For algebra studies, the solution presented in [15] generates new algebra problems similar to one that is given by the user. Another proposal algebra is described by [2], in which the authors synthesize math problems for middle schoolers and high schoolers.

The *Program Sketching* approach has many applications in the field and specifically in *Program Synthesis*. We can mention [23], in which the authors propose an approach to automatically synthesize a new version of a database program based on its original version and the source and destination schemas. [24] automatically synthesizes *SQL (Structured Query Language)* searches from Natural Language, using standard analysis technicians in the program outline. The authors from [12] show how to synthesize probabilistic programs based on real-world datasets. Those authors use *Program Sketching* to specify the program skeletons containing *holes* and the Markov Chain Monte Carlo (MCMC) to efficiently instantiate the *holes*.

The works above mentioned concern the generation of any instance belonging to an application domain, whether it is aimed at education or a problem that is being solved with the *Program Sketching* approach. The approach presented in this paper is similar to the one described in [1], mainly regarding the problem generation in a parameterizable way for the *Natural Deduction* study. However, the authors of the work use a technique based on *UPG*. Here, we adapt the *Program Sketching* approach to assemble specifications inserted in the *Alloy* modeling language. From these specifications, it is possible to generate several instances of formulas and arguments of *Propositional Calculus* in a parameterizable way. The other mentioned works are similar to the presented approach when it comes to the purpose, that is, in the automatic generation of problems. In a more specific context, this paper addresses a different problem from the ones mentioned in those works, which is *Propositional Calculus*.

3. BACKGROUD

3.1 Program Synthesis

Program Synthesis is the task of automatically finding a program that satisfies a certain specification in a specific programming language [7]. Since the beginning of Artificial Intelligence (AI) in 1950, this problem became the *Holy Grail* of Computer Science. This is due to the main idea of the theme being distant from the reality of a common computer program that is applied on the daily basis. Whereas, unlike typical compilers that translate a code completely specified in high level to its low-level machine representation, *Program Synthesis* performs some form of research through the program scope for generating a program that is consistent with a variety of restrictions. [7].

Program Synthesis is notoriously a challenging problem. Its two main challenges are:

- **Program Scope** - the number of programs in any trivial programming language grows exponentially with the program size. This large amount of possible candidates to a program, for a long time, has rendered an intractable task.

- **User Intent** - Many real-life applications domain for *Program Synthesis* are too complex to be completely described as a formal or informal specification. The methods for expressing the user intention range from logical specifications for informal descriptions in natural language to input and output examples.

In the *Program Synthesis* literature, there are some techniques for synthesizing programs. The main approaches are:

- Program Sketching - the programmer gives a basic structure of the program, a sketch which has gaps in its structure that need to be filled with the synthesis process [7].

- Enumerative Research - an obvious brute force approach with an elegant trick, and despite its apparent naivete, has been used with great effect [3].

- Stochastic Research - this approach learns a distribution about the program scope in the scope of hypotheses that are conditioned by the specification and then tests programs from the distribution to learn a consistent program [7].

- Programming by Examples - this is a subfield of *Program Synthesis* where the specification of a certain program is given in the form of input and output examples [5].

As mentioned before, we developed an approach based on *Program Sketching*. This technique allows the programmer to express the high-level details of a problem by writing a sketch. A sketch is a partial program that encodes a solution structure and leaves its low-level details unspecified [17]. In *Program Sketching* the programmer also needs to provide a reference implementation or a set of test routines that the synthesizer code must pass.

When entering a partial program for a specific synthesizer, the user must remember to do it so leaving the *holes* that must be filled with the synthesis. The partial program *holes* complete the synthesized program that must be tested with the set of routines provided by the user [18]. In this kind of synthesis, the main purpose is to fill that *holes*, which are parts that need to be synthesized in a partial program.

This paper approach formally defines the structure of *Propositional Calculus* elements, which characterizes the sketches of the elements. From these sketches, an analyzer automatically finds instances of the elements with customized characteristics along with the constraint insertion for each type. Therefore, the *Program Sketching* technique is applied in the synthesis of these objects.

There are some tools available to work with *Program Synthesis*. The main ones are listed below:

- **SKETCH**: a tool in which the programmer needs to give a partial program, along with the purpose specification of the program. The partial program must contain the *holes* that will be filled with the synthesis and the sketches are written in their language (Sketch), similar to C language [4]. SKETCH [20] has a fair amount of limitations, some of them are: it only synthesizes constants, it is necessary to specify the program behavior, it must provide a partial program, and finally, the tool synthesizes imperative programs belonging only to a single language, which is Sketch.

- **Rosette**: a programming language that extends the Racket language. Racket is a multi-paradigm language belonging to the language family LISP with the ends of serving as a platform for the design, creation, and implementation of other programming languages [4]. It allows verifying and synthesizing programs, the latter working with partial programs (sketches) definition. The holes in the given sketches can be fulfilled by exact or symbolic values, such as numbers or expressions. With Rosette [19], it is possible to generate programs that belong to any programming language as long as it is defined using the Racket creation functionalities and to generate expressions of that language using Rosette.
- **PROSE**: *Microsoft Program Synthesis using Examples SDK* also known as Microsoft PROSE SDK, is a framework for synthesizing programs using input and output examples. Given a DSL - Domain-Specific Language and a set of examples, PROSE manages to synthesize a program consistent with the given examples [4]. With PROSE SDK [10], a user who does not know much about the field can provide examples. Then, the tool can be used to generate programs without the user knowing how the programming language works. However, building the DSL necessary for the synthesis process requires semantics knowledge of the language to be synthesized.
- **Alloy**: based on first-order logic, Alloy language is capable of describing structures and relationships [8]. The Alloy language has been used in a great range of applications, from finding loopholes in security mechanisms to designing telephone communication networks. Given its ease of specifying models and automatically generating instances for those models, the Alloy is also a greatly used tool in Program Synthesis. There is also tooling support named *Alloy Analyzer*, which is a solver that analyzes structures, relationships, properties, and constraints in search of models that satisfy them.

When working with *Program Synthesis*, it is important to highlight that synthesizers are developed to generate different types of programs. These 'types' of programs are defined through *grammars*, which generate objects through synthesizers, such as a simple algorithm, a number, a mathematical equation, or even a formula from *Propositional Calculus*. In the context of *Computer Theory*, a *grammar* is a model used to generate a language.

3.2 Propositional Language

When defining a *Formal Language* it is needed to define its two basic components: an *alphabet* and the *production rules*. *Propositional Calculus alphabet* is defined by:

- a set of *propositional symbols*, which are also called *atoms*, or *propositional variables*: $Q = \{p_0, p_1, ...\}$;
- the *unary connective* $\neg$ (negation, reads **'no'**).
- the *binary connectives* $\wedge$ (conjunction, reads **'and'**), $\vee$ (disjunction, reads **'or'**), $\rightarrow$ (implication, reads **'if..then..'**), and $\leftrightarrow$ (biconditional, reads **'..if and only if..'**); and
- punctuation elements, which are exclusively parentheses: '(' and ')'.

The L_{LP} elements of *Propositional Calculus* are called *formulas*, more specifically *well-formed formulas - wff's*. The following is a formal definition of *wff* according to [14].

The set of *Propositional Calculus* formulas is defined by *induction* and has three cases: a basic case and two inductive cases. Therefore, the L_{LP} set of propositional formulas is inductively defined as the smallest set satisfying the following formation rules.

1. **Basic case:** all propositional symbols are in L_{LP}, that is, $Q \subseteq L_{LP}$. Propositional symbols are called atomic formulas or atoms. Thus, propositional symbols are also *wff's*.
2. **Inductive case 1:** If $\alpha \in L_{LP}$, then $\neg\ \alpha \in L_{LP}$.
3. **Inductive case 2:** If α, $\beta \in L_{LP}$, then $(\alpha \wedge \beta) \in L_{LP}$, $(\alpha \vee \beta) \in L_{LP}$, $(\alpha \rightarrow \beta) \in L_{LP}$, $(\alpha \leftrightarrow \beta) \in L_{LP}$.

In addition to *wff's* there are also *Propositional Calculus* arguments. These elements are formed by a set of premises, and a conclusion, which are *wff's*. Then, the argument generation happens partially based on the *wff's* specifications.

The definitions presented in this Section are used in the automatic generation of formulas and arguments of *Propositional Calculus*. Furthermore, a *context-free grammar* that describes the words of the *Propositional Language* was used in the developed synthesizer.

3.3 Alloy

Alloy is a declarative model specification language in which it is possible to use logical formalisms to generate sufficient constraints for structuring a given problem [8]. This tool is widely used in the creation of software systems models which are greatly used by Software Engineering. For this, Alloy uses set theory and first-order logic to produce enough (and even complex) constraints to correctly specify the behavior of a certain system.

Below are listed some concepts used by Alloy:

- **Set Theory**: an area of mathematics studies that introduces the concept of sets. Sets are collections that store single elements in an unordered way. In Alloy, sets are represented by model entities and elements are instances of those entities. A relationship between two entities also represents a set in which elements can be expressed by tuples of element pairs of the entities involved in the relationship. Sets have operations that serve the purpose of manipulating their elements to generate other sets or obtain logical results. Such operations can be classified as a union, intersection, difference, belonging, equality, etc.
- **Object-Orientation (OO)**: paradigm widely used in the computer science field that serves to represent entities as objects. In Alloy, each object contains relationships. That is a concept based on the set theory itself. In OO,

new features are presented, such as creating abstract entities, supporting inheritance between entities (objects), etc.

- **First-Order Logic**: a field of mathematics in which logical formalisms are studied, which are formed by logical operations. In Alloy, first-order logic is widely used to express constraints on relationships between entities. In addition to set theory operations, it is also possible to express the cardinality/number of entities (sets) in each relationship. This cardinality is represented by quantifiers. Two quantifiers widely used in first-order logic are the so-called universal and existential quantifiers.

The Alloy language has some elements that need to be defined to understand the approach presented in this paper. Some of these commands are defined below:

- **Signature**: signatures are represented by the prefix *sig*, those represent the entities that are being modeled, followed by the entity name. Such entities can be related to each other by the use of logical quantifiers used in first-order logic, to limit the number of entities involved in the relationship.

- **Facts**: represented by the reserved word *fact*, these are explicit restrictions applied to entity relationships, those are made with first-order logic.

- **Predicates**: represented by the prefix *pred*, these are expressions that can return true or false values, which are used together with the run command can generate model instances.

Code 1 presents an example specification written in Alloy language. Lines 1 to 4 define some entities that represent a *Student* who has a contact list. In line 5 a fact is defined that says that every phone has an associated number. Line 6 is defining a *show* predicate that says that the *Student* has 4 phones in their list. In line 7, the Alloy Analyser is told to execute the *show* predicate and find a student model with 4 phones. The '4' in line 7 indicates that the model contains up to 4 different objects.

```
1   sig Number{}
2   abstract sig Phone{number: one Number}
3   abstract sig Person {phones: set Phone}
4   one sig Student extends Person {}
5   fact {all n: Number| one t: Phone| t.number = n}
6   pred show(){#Person.numbers = 4}
7   run show for 4
```

Code 1: Example using Alloy language.

4. SOLUTION ARCHITECTURE

In this section, the architecture of the developed solution is presented. The solution process is broken down into some steps that when added together, generate the Propositional Calculus elements. Figure 1 shows the walkthrough of the architecture.

In Figure 1, in step (1) the user accesses the mobile application interface. In step (2), the user chooses if he/she wants to generate formulas (2a) or arguments (2b) and gives the necessary parameters for generating the chosen object. In (3), the app puts together the information provided by the user and sends it to the *Web Service* (4) through a request.

In (5), the *Web Service* receives the information, verifies if the user asked for generating formulas or arguments (6), and assembles an Alloy code with all the information. The *Web Service* puts together the Alloy code with a basic Alloy specification for each element of *Propositional Calculus*, thus generating a complete file with Alloy specifications (7). The file is executed through the Alloy API (8). The Alloy API accesses an Alloy interpreter.

By default, the Alloy API returns an Alloy code, so it is necessary to transform this code into values belonging to the formal definition of the *Propositional Calculus*. This transformation takes place after the models are generated and returned by the Alloy API (9). The values to be achieved in the transformation, which are formally defined, range from the structure of objects to the insertion of real logical operators. This transformation is partial, so it is necessary to fill the spaces of the atoms with the letters of the alphabet (10). This filling of atoms is according to the order and quantity instanced in parameterization (2). After generating the objects, the *Web Service* returns them to the mobile application in (11). In the user app, a PDF file is generated and put available to the user. There is also a *preview* (12a) of the lists of generated elements and it is possible to view, share or save the PDF file (12b).

The architecture presented in this section has some essential components for the solution to work. These components are detailed in Section 5.

5. SOLUTION COMPONENTS

The presented approach has three main components to be detailed: the Alloy specifications, the API, and the mobile software tool.

5.1 Alloy Specifications

In this section, Alloy specifications for well-formed formulas and valid arguments are presented. The logic validity of the generated arguments is guaranteed by construction.

Combining the technique of programming by constraints technique using the Alloy language with the *Program Skecthing* technique, it is possible to generate formulas and valid arguments of the Propositional Calculus.

Two software tools were developed, a mobile application and an API (*Application Programming Interface*) that is available in web service format. To summarize, the developed synthesis process consists of three steps: (i) user intention about the characteristics of the formula (or argument) provided through the mobile application; (ii) parameterization based on Alloy specifications on the service web; and (iii) generation of models using the Alloy Analyzer API.

The specification referring to *wff* can be seen in Code 2. In line 1, the abstract entity *Formula* is defined. Such an entity can be *Unary* ou *Binary* (lines 2 and 3). Lines 4 to 6 define the entities that can be instantiated: atom, negation, conjunction, disjunction, implication or bi-implication.

```
1    abstract sig Formula{}
2    abstract sig Unary extends Formula{
3     child: Formula
4   }
5    abstract sig Binary extends Formula{
6     left, right: Formula
7   }
8    sig Atom extends Formula {}
9    sig Not extends Unary{}
10   sig And, Or, Imply, BiImply extends Binary {}
```

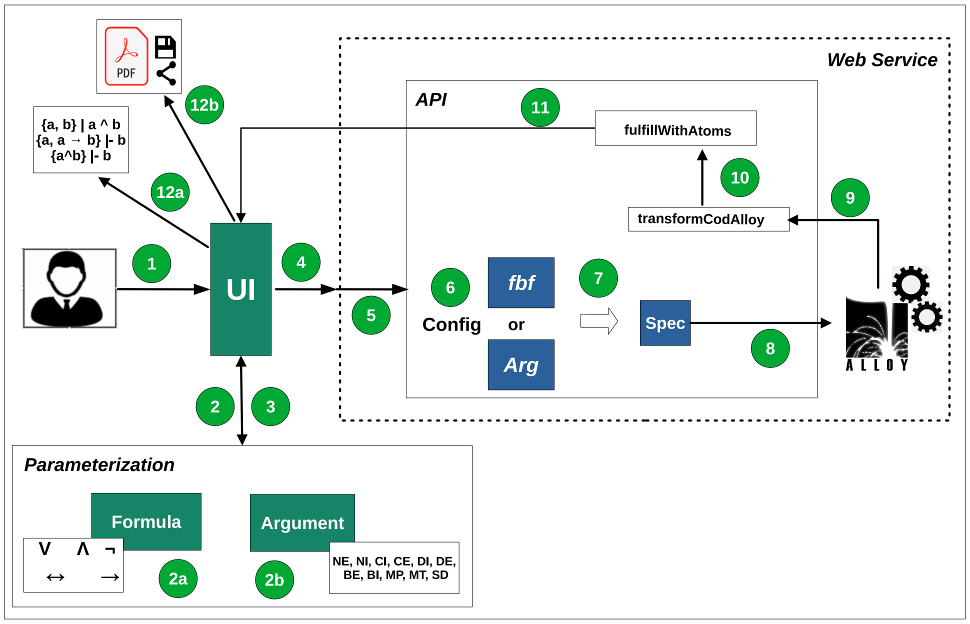


Figure 1: Solution architecture.

```
11   one sig FBF{ mainOperator: one Formula }
```

Code 2: Basic entities for *wff's*.

The Alloy Analyzer searches for models that meet the specifications. Line 7 tells the analyzer that, in the search process, it must find models that have only one instance of a wff, which is related to only one object defined as a $Formula$. The single instance wff is the formula root, its main operator. However, just with this specification, the analyzer can find models in which there will present wrong and unwanted features. For instance, disjointed formulas or binary formulas where some part is the formula itself, a self-reference producing an infinite *loop*, etc.

In Code 3, line 1 presents a fact that prevents formulas from having cycles in the tree generated by the Alloy Analyzer. This prevents the infinite *loop*. The fact in line 2, on the other hand, defines that every $Formula$ object must be in the same tree generated from the only WFF object. Thus, avoiding unlinked $Formula$ objects.

```
1   fact NoCycle{no n,n':Formula | n in n'.^(child+
      ↪ left+right) and n' in n.^(child+left+
      ↪ right)}
2   fact EveryNodeAtAFBF{all n:Formula | one t:FBF |
      ↪ n in t.mainOperator.*(child+left+right)}
3   pred Config(){ #And>0 #Or>0 #Not>0 #Imply=0Â #
      ↪ BiImply=0 (#Atom≥3 ∧ #Atom≤6) }
```

Code 3: Facts about *wff's* and parameterizable predicate.

Line 3 shows a configuration for formula synthesis. While the specification already presented always remains the same, this part is generated from the customization made by the user through the mobile application developed. In this presented case, user defines that formulas with at least one conjunction, at least one disjunction, at least one negation, no implication, and no bi-implication must be generated. It is also defined that all *wff's* must have between three and six distinct atoms.

In Code 4 the specification regarding the structures of the inference rules is presented. Line 1 is the abstract entity *Rule*. Each signature corresponds to a rule, namely: NE and NI - negation exclusion and inclusion; CE and CI - conjunction exclusion and inclusion; DE and DI - disjunction exclusion and inclusion; BE and BI - bi-implication exclusion and inclusion; MP - *modus ponens*; MT - *modus tollens*; and SD - disjunctive syllogism.

```
1   abstract sig Rule { }
2   sig NE extends Rule {p1:Not, r:Formula}
3   sig NI extends Rule {p1:Formula, r:Not}
4   sig CI extends Rule {p1:Formula, p2:Formula, r:
      ↪ And}
5   sig CE extends Rule {p1:And, r:Formula}
6   sig DI extends Rule {p1:Formula, r:Or}
7   sig DE extends Rule {p1:Imply, p2:Imply, p3:Or, r
      ↪ :Formula}
8   sig BI extends Rule {p1:Imply, p2:Imply, r:
      ↪ BiImply}
9   sig BE extends Rule {p1:BiImply, r:Imply}
10  sig MP extends Rule {p1:Formula, p2:Imply, r:
      ↪ Formula}
11  sig MT extends Rule {p1:Formula, p2:Imply, r:
      ↪ Formula}
12  sig SD extends Rule {p1:Formula, p2:Or, r:Formula
      ↪ }
```

Code 4: Inference rules structures.

To exemplify one of the rules, we have in line 4 the conjunction inclusion. This rule structure is composed of three parts: any two *wff's*, and a resulting *wff* which must be conjunction. However, it is not yet specified that the latter must be a conjunction between the former ones.

Code 5 presents the facts that define how each inference rule must work. In line 4, for the conjunction inclusion, it is defined that for any CI instance, its result (an And instance) must have the left and right parts equal to the first parts of CI, regardless of order.

```
1   fact rules{
2   all ne: NE | ne.p1.child.child=ne.r
3   all ni: NI | ni.p1=ni.r.child.child
4   all ci: CI | (ci.r.left=ci.p1 and ci.r.right=ci.p2
        ↪ ) or (ci.r.left=ci.p2 and ci.r.right=ci.
        ↪ p1)
5   all ce: CE | ce.r = ce.p1.left or ce.r = ce.p1.
        ↪ right
6   all di: DI | di.p1 in di.r.(right+left)
7   all de: DE | ((de.p1.left=de.p3.left and de.p2.
        ↪ left=de.p3.right) or (de.p1.left=de.p3.
        ↪ right and de.p2.left=de.p3.left))
8    and de.p1.right=de.p2.right and de.r=de.p2.right
9   all bi: BI | bi.p1.right=bi.p2.left and bi.p2.
        ↪ right=bi.p1.left
10   and ((bi.r.right=bi.p2.right and bi.r.left=bi.p2
        ↪ .left) or (bi.r.right=bi.p2.left and bi.
        ↪ r.left=bi.p2.right))
11  all be: BE | (be.r.left=be.p1.left and be.r.right=
        ↪ be.p1.right) or (be.r.left=be.p1.right
        ↪ and be.r.right=be.p1.left)
12  all mp: MP | mp.p1 = mp.p2.left and mp.r = mp.p2.
        ↪ right
13  all mt: MT | (mt.p1.child = mt.p2.right and mt.r.
        ↪ child = mt.p2.left) or (mt.p1 = mt.p2.
        ↪ right.child and mt.r = mt.p2.left.child)
14  all sd: SD | (sd.p1.child = sd.p2.left and sd.r =
        ↪ sd.p2.right) or (sd.p1.child = sd.p2.
        ↪ right and sd.r = sd.p2.left)
15  }
```

Code 5: Inference rules behavior.

In Code 6, facts considered important to argument generation are presented. In lines 1 to 3, predicates used in these facts are defined, namely: line 1 - one formula is not the same as another; line 2 - right side is different from the left side of the formula; and line 3 - the right side of the formula is not the negation of the left side, and vice versa.

```
1   pred isNotEqualTo[a: Formula,a': Formula]{ (a.right
        ↪ ≠a'.right or a.left≠a'.left) and (a.right
        ↪ ≠a'.left or a.left≠a'.right)}
2   pred avoidA_A[a: Formula]{ a.right≠a.left }
3   pred avoidA_noA[a: Formula]{ (a.right.child≠a.left
        ↪ ) and (a.right≠a.left.child) }
4   fact {
5   all a,a': Not | a.child=a'.child implies a=a'
6   all a,a': And | a.isNotEqualTo[a']
7   all a,a': Or | a.isNotEqualTo[a']
8   all a,a': BiImply | a.isNotEqualTo[a']
9   all a,a': Imply | (a.right=a'.right and a.left=a'.
        ↪ left) implies a=a'
10  all x: And | x.avoidA_A
11  all x: Or | x.avoidA_A
12  all x: Imply | x.avoidA_A
13  all x: BiImply | x.avoidA_A
14  all x: And | x.avoidA_noA
15  all x: Or | x.avoidA_noA
16  all x: Imply | x.avoidA_noA
17  all x: BiImply | x.avoidA_noA
18  }
```

Code 6: Important facts to the argument generation.

The facts presented in Code 6 prevent two or more instances of an implication from generating the same formula more than once, for example. This makes formula generation more efficient and more effective. For instance, the implication may occur more than once among premises and conclusion. If this is the case, it will be generated only one instance in the model, found by the Alloy Analyzer, but with more than one relationship.

The specifications presented in Code 7 can be divided into five parts. Between lines 1 and 13, the facts that prevent the same application of a rule from occurring more than once are presented. For instance, the negation exclusion can occur more than once, but not generate the same result, for example. In lines 14 to 16, variables are defined for the sake of simplifying the specification of the following fact.

```
1   fact {
2   all a,a': NE | (a.r=a'.r) implies a=a'
3   all a,a': NI | (a.r=a'.r) implies a=a'
4   all a,a': CE | (a.p1=a'.p1 and a.r=a'.r) implies a
        ↪ =a'
5   all a,a': CI | (a.r=a'.r) implies a=a'
6   all a,a': DI | (a.p1=a'.p1 and a.r=a'.r) implies a
        ↪ =a'
7   all a,a': DE | ((a.p1.isNotEqualTo[a'.p1] and a.p2
        ↪ .isNotEqualTo[a'.p2]) or (a.p1.
        ↪ isNotEqualTo[a'.p2] and a.p2.isNotEqualTo
        ↪ [a'.p1]))
8   and a.p3.isNotEqualTo[a'.p3] implies a=a'
9   all a,a': BE | (a.p1=a'.p1 and a.r=a'.r) implies a
        ↪ =a'
10  all a,a': SD | (a.p1=a'.p1 and a.p2=a'.p2) implies
        ↪  a=a'
11  all a,a': MP | (a.p1=a'.p1 and a.p2=a'.p2) implies
        ↪  a=a'
12  all a,a': MT | (a.p1=a'.p1 and a.p2=a'.p2) implies
        ↪  a=a'
13  }
14  let P1 = NE<: p1+NI<: p1+CI<: p1+CE<: p1+DI<: p1+DE<:
        ↪ p1+BI<: p1+BE<: p1+MP<: p1+MT<: p1+SD<: p1
15  let P2 = CI<: p2+DE<: p2+BI<: p2+MP<: p2+MT<: p2+SD<:
        ↪ p2
16  let R = NE<: r+NI<: r+CI<: r+CE<: r+DI<: r+DE<: r+BI<: r
        ↪ +BE<: r+MP<: r+MT<: r+SD<: r
17  fact OneOrigin{
18  one rule: Rule | all f: Formula | f in rule.(P1+
        ↪ P2+p3+R).*(child +Binary<: left + Binary<:
        ↪ right) or f=rule.P1 or f=rule.P2 or f=
        ↪ rule.p3 or f=rule.R
19  }
20  one sig Argument{ premisse: set Formula,
        ↪ conclusion: one Formula }{
21  #premisse=3 and not (conclusion in premisse)
22  }
23  run Config for 4
```

Code 7: Argument generation restrictions and signature.

From line 17 to line 19, it is specified the fact that there is a single rule which is applied to the formulas (premises) and/or from which the other formulas (non-basic premises) can be reached, even with the application of other rules in intermediate steps. This fact provides two guarantees. Firstly, there will be a single origin in the proof process. And secondly, the conclusion can be inferred from the set of premises, that is, that the argument is valid.

Line 20 defines that there must be only one argument with premises and a conclusion. On the other hand, in line 21 is defined that there are three premises and none of them must be the conclusion. Finally, on line 23 an empty configuration (*Config* predicate) is defined without further restrictions, telling the analyzer to generate up to four instances of each entity.

5.2 API

An API (*Application Programming Interface*) was developed to establish a connection between the mobile app and the basic specifications written in Alloy language. Communication between the app and this API occurs through HTTP requests, with the exchange of files in *JavaScript Object Notation* (JSON) format. Figure 2 shows the layered architecture of the approach presented in this paper.

In Figure 2, in (1) there is the mobile application whose details are presented in Section 5.3, which accesses the API

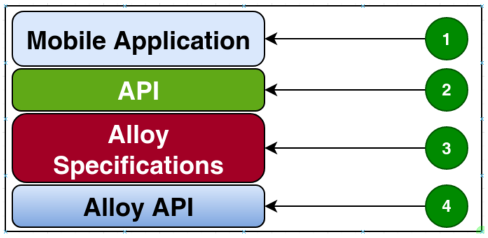


Figure 2: Layered architecture.

developed in (2). The API assembles the Alloy specifications (3) according to the base specifications presented in Section 5.1, and finally, it makes a call to the Alloy (4) API to access the language interpreter. The specifications are processed and the mobile app gets a return based on this process, and the parameters provided in the initial request.

The API was developed in programming language *Java*, using Spring Framework [22], and Spring Boot [21] technologies, mainly used in the development of corporate software. The decision of using the *Java* language in the API development was made because there is support for the Alloy language (Alloy API). This allows the passage of specifications and the extraction of generated models. Figures 3 and 4 show real examples returned from the API.

Figure 3 presents a set of returned formulas and seven parts are highlighted. In (1), the API is informed of a number range for Alloy to choose which number of different atoms the formulas will contain, in this case from 3 to 6 atoms. In (2), it indicates that the number of atoms does not have a last value fixed. In (3), the API is informed that one formula will be generated for each list, with the number of lists fixed at five, informed in (6). In (4) it is given that the generated formulas will contain at least the operators given in (5).

Figure 4 presents a set of returned arguments. In (1), it is specified which inference rules are used to validate the argument and which can be used to prove it. In (2), the API is informed of the number of different atoms the formulas present in the arguments will contain, it would be up to 4 different atoms. In (3), as an example, the API is informed that an argument is returned in each list, with a number of lists fixed at five, informed in (4). The parameters given in both Figure 3 and Figure 4 examples are received by the API, which assembles an Alloy code with these parameters. This Alloy code is merged with the base specification of each element of the Propositional Calculus that the API contains, thus generating a complete Alloy code. The complete Alloy Code is passed for the Alloy API to process and return the models.

5.3 Mobile application

The main purpose of the mobile app is to receive the user parameters referring to the automatic generation of formulas or arguments and to be able to show what the API returns. The application was developed in Flutter [9]. The choice of *Flutter* for app development was because it is an open-source SDK created by *Google*, which has been growing in the context of cross-platform apps for Android and IOS since its launch in May 2017.

Flutter uses the programming language *Dart* and provides several benefits for the developer, such as: creating applications with attractive interfaces, productivity, and speed. Moreover, it is open-source, making it possible for developers to create their code packages and offer them to the entire technology community, thus increasing the component alternatives in creating applications.

Figures 5 and 6 show screen sequences for generating formulas and arguments, respectively. Due to space restrictions, the screens are presented in a reduced form, but in the [Github] repository there is a complete tutorial for using the mobile application functionalities.

In Figure 5, in (1) the user performs the parameterization to be able to generate formulas. There are four sections on this screen: **amount of atoms** (*quantidade de Ã¡tomos* in portuguese) - user enters a settled range or number of different atoms for each formula; **amount of wff's** (*quantidade de fbf's* in portuguese) - user informs how many wff's he wants in each exercise list; **operator selection** (*seleÃ§Ã£o de operadores* in portuguese) - user selects the operators that the generated formulas will contain and whether they want only those operators or at least these ones; and **different exercise lists** (*lista de exercÃcios diferentes* in portuguese) - user enters how many exercises lists he/she wants.

After collecting the parameters, the app sends an HTTP request to the API, which generates the formulas instances based on the given parameters, and returns them to the application in (2) (Figure 5). After the formulas are received by the app, the user has a preview of the formulas and can download a PDF with the lists on the cellphone, take a look at it or share it with third-party apps.

In Figure 6 the functioning is similar to that shown in Figure 5. In (1) are also presented four sections on the screen: **select inference rules** - user selects inference rules used to validate arguments; **amount of different atoms** - user informs how many different atoms the formulas present in the generated arguments will contain; **amount of arguments** - user informs the number of arguments in each list; and **lists of different exercises** - user informs the number of lists he/she wants.

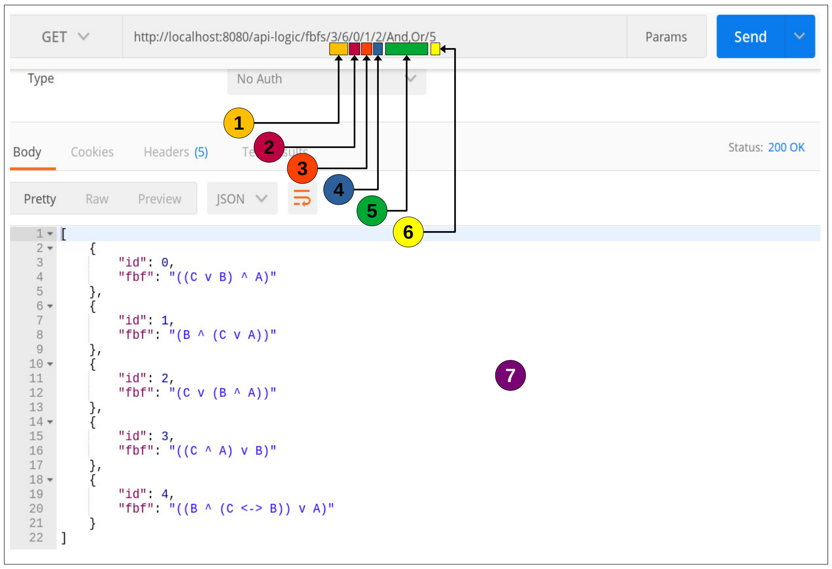
The request made by the mobile app in the generation of arguments is similar to the generation of formulas (shown in Figure 5). So are the functionalities that can be done from the return to the app in (2).

6. FINAL REMARKS AND FUTURE WORK

This paper presented an approach for automatic generating *Propositional Calculus* formulas and arguments. For this, a first-order-based language called Alloy modeling language is used with its analyzer. The approach is quite promising, as it seeks to meet a real demand for Logic students and professors.

As means to make it happen, a tool was developed implementing communication and process algorithms in the form of an API, and embedding the Alloy API. With such a tool, students have access to an arsenal of problems to practice the curricular contents. Therefore, the professors will not need to invest time in the elaboration of exercises for the students. It is enough for the teacher to simply specify and share characteristics of the questions or formulas to be generated.

Regarding the related works, there is a considerable similarity with Ahmed et al. [1]. The authors use the *UPG* and the bit vector to be able to generate problems and solutions for natural deduction. We focus only on the problem


GET http://localhost:8080/api-logic/fbfs/3/6/0/1/2/And,Or/5

```
[
    {
        "id": 0,
        "fbf": "((C v B) ^ A)"
    },
    {
        "id": 1,
        "fbf": "(B ^ (C v A))"
    },
    {
        "id": 2,
        "fbf": "(C v (B ^ A))"
    },
    {
        "id": 3,
        "fbf": "((C ^ A) v B)"
    },
    {
        "id": 4,
        "fbf": "((B ^ (C <-> B)) v A)"
    }
]
```

Figure 3: Example of API response for formula generation.

generation part by adapting a *Program Synthesis* technique, *Program Sketching*, and using programming by constraints. Thus, it is possible to specify the basic structure of the Logic elements through sketches and define the formation rules of elements through constraints. Formulas are structurally validated according to the formal definition of *wff*. Arguments are validated according to the use of inference rules within generation time.

This work successfully achieved the generation of valid arguments from *Propositional Calculus*. Such a result was obtained through the use of the Alloy modeling language and its Analyser. In addition, a mobile app was developed so students and teachers can generate formulas and arguments with a user-friendly interface. Another important result is the developed API that establishes communication between the application and Alloy specifications.

For future works, to enrich the basic Alloy specifications is intended so it would be possible to deal with hypothetical rules in arguments, such as *conditional proof* and *reductio ad absurdum*. Another improvement will be the generation of hints to students to solve argument proofs. The API will be improved so that it will be possible for other applications to be created based on it, implementing features such as security and authentication. Finally, improving the mobile application is also intended to generate better elaborated and more complex valid arguments.

7. REFERENCES

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *IJCAI*, pages 1968–1975. Citeseer, 2013.

[2] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 773–782, 2013.

[3] J. Bornholt. Program synthesis explained. `https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html`. Accessed in 03/31/2023., 2015.

[4] J. P. M. Cordeiro. Síntese de programas utilizando a linguagem alloy, 2017.

[5] S. Gulwani. Programming by examples (and its applications in data wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press, January 2016.

[6] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. *ACM SIGPLAN Notices*, 46(6):50–61, 2011.

[7] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[8] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[9] G. D. C. C. A. . I. License). Flutter. `https://flutter.dev`. Accessed in 03/31/2023., 2017.

[10] Microsoft. Microsoft program synthesis using examples (prose). `https://www.microsoft.com/en-us/research/project/prose-framework/`. Accessed in 03/31/2023., 2021.

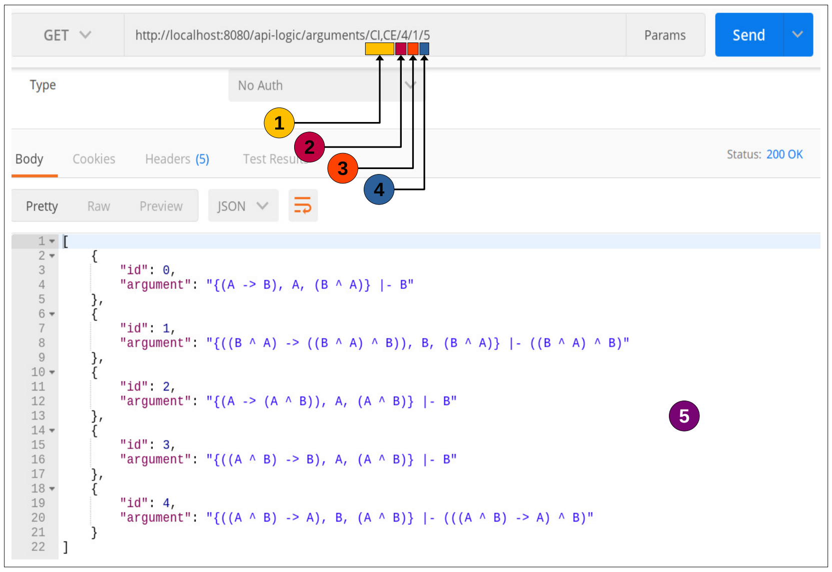
[11] M. Mozgovoy, T. Kakkonen, and G. Cosma. Automatic student plagiarism detection: future perspectives. *Journal of Educational Computing Research*, 43(4):511–531, 2010.

[12] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices*, 50(6):208–217, 2015.

[13] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

[14] F. S. C. d. Silva, M. Finger, and A. C. V. d. Melo. *Lógica para computaÃ§Ã£o*. Cengage Learning, 2006.

[15] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.



```
GET http://localhost:8080/api-logic/arguments/CI,CE/4/1/5
```

```
[
    {
        "id": 0,
        "argument": "{(A -> B), A, (B ^ A)} |- B"
    },
    {
        "id": 1,
        "argument": "{((B ^ A) -> ((B ^ A) ^ B)), B, (B ^ A)} |- ((B ^ A) ^ B)"
    },
    {
        "id": 2,
        "argument": "{(A -> (A ^ B)), A, (A ^ B)} |- B"
    },
    {
        "id": 3,
        "argument": "{((A ^ B) -> B), A, (A ^ B)} |- B"
    },
    {
        "id": 4,
        "argument": "{((A ^ B) -> A), B, (A ^ B)} |- (((A ^ B) -> A) ^ B)"
    }
]
```

Figure 4: Example of API response for argument generation.

[16] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013.

[17] A. Solar-Lezama. The sketching approach to program synthesis. In Z. Hu, editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] A. Solar-Lezama and R. Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[19] E. Torlak. The rosette guide. `https://docs.racket-lang.org/rosette-guide/index.html`. Accessed in 03/31/2023., 2020.

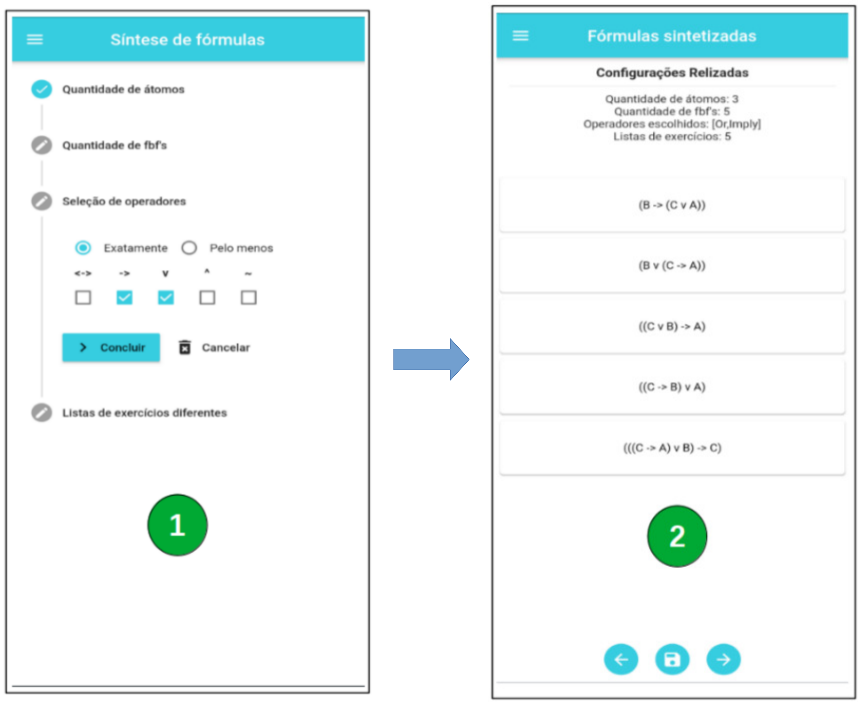
[20] N. Tung. Sketch-wrapper. `https://bitbucket.org/gatoatigrado/sketch-wrapper/src/master/`. Accessed in 03/31/2023., 2012.

[21] I. VMware. Spring boot. `https://spring.io/projects/spring-boot`. Accessed in 03/31/2023., 2023.

[22] I. VMware. Spring framework. `https://spring.io/projects/spring-framework`. Accessed in 03/31/2023., 2023.

[23] Y. Wang, J. Dong, R. Shah, and I. Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2019.

[24] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.



[✓ Exatamente] Pelo menos

<->	->	v	^	~
☐	✓	✓	☐	☐

Configurações Relizadas

Quantidade de átomos: 3
Quantidade de fbf's: 5
Operadores escolhidos: [Or,Imply]
Listas de exercícios: 5

(B -> (C v A))

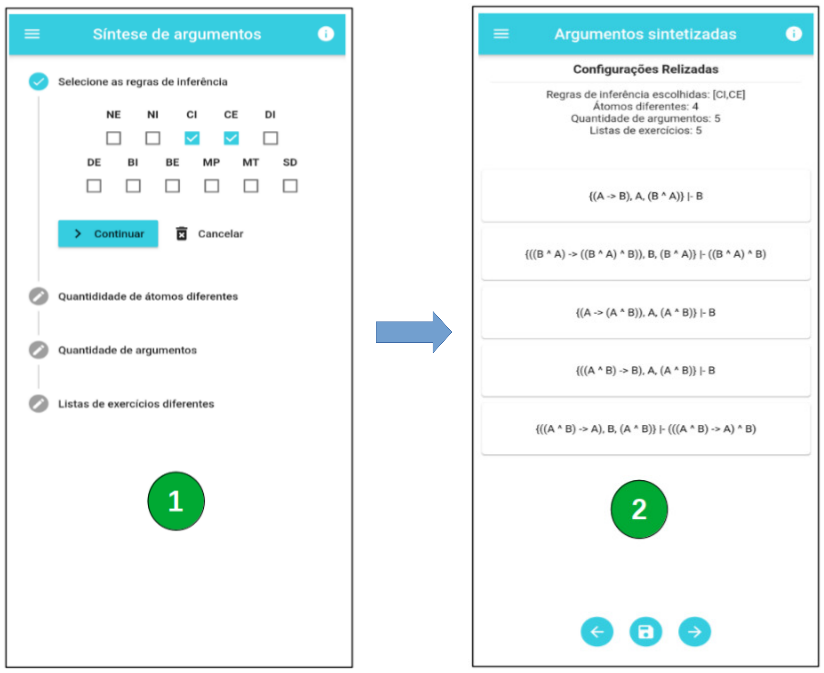
(B v (C -> A))

((C v B) -> A)

((C -> B) v A)

(((C -> A) v B) -> C)

Figure 5: Sequence for formula generation.


Síntese de argumentos
Selecione as regras de inferência
Continuar
Cancelar
Quantididade de átomos diferentes
Quantidade de argumentos
Listas de exercícios diferentes
1
Argumentos sintetizadas
2

NE	NI	CI	CE	DI	
☐	☐	✓	✓	☐	
DE	BI	BE	MP	MT	SD
☐	☐	☐	☐	☐	☐

Configurações Relizadas

Regras de inferência escolhidas: [CI,CE]
Átomos diferentes: 4
Quantidade de argumentos: 5
Listas de exercícios: 5

{(A -> B), A, (B ^ A)} |- B

{((B ^ A) -> ((B ^ A) ^ B)), B, (B ^ A)} |- ((B ^ A) ^ B)

{(A -> (A ^ B)), A, (A ^ B)} |- B

{((A ^ B) -> B), A, (A ^ B)} |- B

{((A ^ B) -> A), B, (A ^ B)} |- (((A ^ B) -> A) ^ B)

Figure 6: Sequence for argument generation.