Internship report in IRISA Team LogicA

Félix Ridoux

May-June 2020

Contents

1 Introduction

I have spent two months as an intern in IRISA Team LogicA. The initial topic of this internship was the exploration of *answer set programming* (ASP) for the benefit of Team LogicA's research. This led us to make a presentation of ASP to the team. Then we have studied the semantics of *attack trees* and implemented it by constructing automatons which recognize the semantics. This report presents the details of this work.

2 Answer set programming

Answer set programming (ASP) is a declarative and logic programming paradigm [6]. In ASP a program is a set of facts, rules and constraints which specify a problem. The ASP solver interprets this set of facts, rules and constraints as logical formulas, and calculates the *stable model* of these formulas. Informally, a *stable model* is the minimal set of facts that can be deduced from the specification, and this is considered as the answer to the problem. There can be several different stable models, corresponding to several possible answers.

Interestingly, the semantics of ASP is very close to propositional logic with *negation as failure*. This makes it easy to understand the ASP semantics when there are few negations, and rather subtle when there are a lot of intricate negations. In practice, we are using the CLINGO [5] implementation of ASP.

ASP and CLINGO can be used for a wide range of applications [3, 2], including robotics and job scheduling.

2.1 CLINGO syntax

The CLINGO syntax is composed of three families of constructions.

Definition 1 (Syntax of CLINGO)

- *Facts:* `head.` *A head asserts that something is true;*
- *Rules:* `head :- body.` *A rule asserts that the head is true if the body is true;*
- *Constraints:* `:- body.` *A constraint asserts that the body must not be proved true;*

where `body` *and* `head` *consists of a set of comma separated atoms and each atom can be either a predicate symbol applied to values or a comparison between two values. Note that the final dot is part of the notation. It is often called "full-stop". We will drop the full-stops in case of confusion with the meta-language punctuation.*

Here are some examples with their informal semantics.

Example 1 (Syntax of CLINGO)

- *Facts:* `cat(tom). mouse(jerry).` *It can be interpreted as Tom is a cat, and Jerry is a mouse. Here,* `tom` *and* `jerry` *are constant values. The CLINGO convention is that identifiers of constant values start with a lower-case letter.*

  Several facts with the same predicate can be factorized as follows: `mouse(jerry ; mickey)` *is a short-hand for* `mouse(jerry)` *and* `mouse(mickey)`.

  Similarly, `digit(0..9)` *is a short-hand for* `digit(0)`, ..., *and* `digit(9)`.

- *Rules:* `eat(X,Y) :- cat(X), mouse(Y).` *It can be interpreted as follows: for all* `X` *and* `Y` *if* `X` *is a cat and* `Y` *is a mouse, then* `X` *eats* `Y`. *Here,* `X` *and* `Y` *are variables. The CLINGO convention is that identifiers of variables start with an upper-case letter. There can be no variables in facts. Similarly, there can be no variable in a rule head that does not occur in the same rule body.*

- *Constraints:* `:- cat(X), mouse(X).` *It can be interpreted as for all* `X`, `X` *cannot be simultaneously a cat and a mouse.*

Finally, CLINGO has two negations:

Definition 2 (Syntax of negations in CLINGO)

- *Negation as failure:* `not p`. *Negation as failure forms atoms that can occur only in rule and constraint bodies.* `not p` *is true iff* `p` *is not proved true by the rest of the program.*
- *Classical negation:* `-p`. *Classical negation forms atoms that can occur either in fact and clause heads or in rule and constraint bodies.* `-p` *is true iff* `p` *is proved false by the rest of the program.*

This difference between being proved false and not being proved true is akin to the same difference in intuitionistic logic [7].

2.2 CLINGO semantics

2.2.1 Propositional image of a program

A fundamental aspect of CLINGO is that all programs can be translated in a set of propositional formulas, which is called the *propositional image* of the program. It permits to calculate stable models, which are the formal semantics of a CLINGO program. The propositional image of a program is computed using the following rules.

Definition 3 (Program $\longrightarrow$ propositional image)

- *Rules:* `c1,...,cn :- h1,...,hm` *denotes* $c_1 \vee \ldots \vee c_n \longleftarrow h_1 \wedge \ldots \wedge h_m$*;*
- *Facts:* `c1,...,cn` *denotes* $c_1 \vee \ldots \vee c_n \longleftarrow \top$ *which is equivalent to* $c_1 \vee \ldots \vee c_n$*;*
- *Constraints:* `:- h1,...,hn` *denotes* $\bot \longleftarrow h_1 \wedge \ldots \wedge h_n$ *which is equivalent to* $\neg h_1 \vee \ldots \vee \neg h_n$*;*
- *Rules and constraints are fully instantiated (ASP jargon says "grounded") by replacing all their variables by all possible combinations of values;*

Symbol $\longleftarrow$ *stands for right to left implication, symbols* $\top$ *and* $\bot$ *stand for logical "true" and "false", and symbols* $\vee$ *and* $\wedge$ *stand for logical "or" and "and".*

The restriction on variable occurrences insures that grounding always ends in a propositional formula.

Example 2 (Program $\longrightarrow$ propositional image) *For instance, the propositional image of:*

```
parent(c, e ; c, p).
parent(w,d ; w,c ; h,d ; h,c).
descendant(X,Y) :- parent(X,Y).
descendant(X,Y) :- descendant(X,Z), descendant(Z,Y).
frereOuSoeur(X,Y) :- parent(X,A), parent(Y,A), X>Y.
```

is built as follows:

- *Facts:*

$$\begin{array}{rl} & parent(c,e) \wedge parent(c,p) \\ \wedge & parent(w,d) \wedge parent(w,c) \wedge parent(h,d) \wedge parent(h,c); \end{array}$$

- *Rules:*

  all possible groundings of formula:

$$\begin{array}{rll} descendant(x,y) & \longleftarrow & parent(x,y) \\ & & \vee\ (descendant(x,z) \wedge descendant(z,y)) \end{array}$$

  all possible groundings of formula:

$$\begin{array}{rll} descendant(x,y) & \longleftarrow & parent(x,y) \\ & & \vee\ (descendant(x,z) \wedge descendant(z,y)) \end{array}$$

  such that $x > y$*. Remark that a total order is maintained by the system among the set of all possible values. It is used here to avoid redundant symmetry and spurious reflexivity in predicate* `frereOuSoeur`*.*

It is important to insist that a propositional image is a *set* of propositional formulas. Indeed, the order of facts, rules and constraints in the program doesn't impact its semantics. Neither does the order of atoms in bodies. However, order may affect performance.

2.2.2 Negation

As we have written above, CLINGO proposes two negations.

Definition 4 (Informal semantics of negations in CLINGO)

- *Negation as failure (written* `not p` *in CLINGO syntax and* $\neg p$ *in propositional images).*

  The informal semantics of negation as failure is:

- *$\neg p$ is true if nothing proves p, that is to say p is not a consequence of the facts and rules and constraints;*
- *$\neg p$ is false otherwise.*

- *Classical negation (written `-p` in CLINGO syntax and $-p$ in propositional images).*

  The informal semantic of classical negation is:

  - *$-p$ is true if there is evidence of $-p$ and nothing proves p, that is to say $-p$ is consequence of fact, rules and constraints, and p is not;*
  - *$-p$ is false otherwise.*

It is important to note that classical negation is more restrictive than negation as failure; if $-p$ then $\neg p$, but not the converse. This fact is very powerful for modelling real-life problems, like those modelled by rules with exceptions. For instance,

```
love(P, C) :- parent(P, C), -wicked_parent(P).
```

expresses the fact that parents love their children only if it is proved that they are not wicked parents, but

```
love(P, C) :- parent(P, C), not wicked_parent(P).
```

expresses the fact that parents love their children except if it appears that they are wicked parents, and

```
love(P, C) :- parent(P, C), not -love(P, C).
```

expresses the fact that parents love their children except if it appears that they do not.

2.2.3 Stable models

As we have said in the beginning of this section, an ASP solver calculates stable models. But what is precisely a stable model? To answer this question we need new definitions.

Let us use the set of propositional formula $F_{example} = \{p \vee q\}$ for all examples in this section.

Definition 5 (Vocabulary) *The vocabulary of a set F of formulas is the set of propositional variables present in F.*

Example 3 (Vocabulary) *The vocabulary of $F_{example}$ is $\{p, q\}$.*

Definition 6 (Interpretation) *An interpretation I of a set F of propositional formulas is a subset of the vocabulary of F.*

An interpretation is the extensional definition of an assignment of value true to propositional variables.

Example 4 (Interpretation) *All interpretations of $F_{example}$ are $\emptyset$, $\{p\}$, $\{q\}$, and $\{p, q\}$.*

Definition 7 (Satisfiability) *A propositional variable is true in an interpretation I iff it belongs to I. A propositional formula is satisfied in an interpretation I iff the application of the truth tables of propositional calculus to its variables produces true.*

Definition 8 (Model) *A model of a set F of propositional formulas without negation is an interpretation of F which satisfies all propositional formulas of F.*

By extension, a model of a program is an interpretation of its propositional image that makes all its formulas true.

Example 5 (Model) *The models of $F_{exemple}$ are $\{p\}$, $\{q\}$, and $\{p, q\}$.*

Definition 9 (Minimal model) *A model* M *of a set F of propositional formulas without negation is minimal iff F has no other model which is included in* M.

Example 6 (Minimal model) *The minimal models of $F_{exemple}$ are $\{p\}$, $\{q\}$.*

When the program obey specific restrictions the set of minimal models benefits from additional properties. For instance, for all programs whose facts and clause heads are singletons, there is a unique minimal model. These programs are called *definite programs*. They are similar to PROLOG programs.

Definition 10 (Critical part) *A critical part of a propositional formula F is a sub-formula of F with the form $\neg g$ which is not included in another sub-formula of this form.*

Example 7 (Critical part) *Let p_1 and p_2 be two propositional variables. The critical parts of the propositional formula $\neg p_1 \wedge \neg\neg p_2$ are $\neg p_1$ and $\neg\neg p_2$. Formula $\neg p_2$ is not a critical part because it is part of $\neg\neg p_2$.*

Definition 11 (Reduction) *The reduction of a set F of propositional formulas with respect to an interpretation I is a the set $reduc_I(F)$ of propositional formulas obtained by replacing by $\top$ each critical part of formulas of F satisfied by I, and by $\bot$ the other.*

Example 8 (Reduction) *Let F= $\{p \longleftarrow \neg q, s \longleftrightarrow \neg\neg q\}$ and I= $\{p\}$, $reduc_I(F) = \{p \longleftarrow \top, s \longleftarrow \bot\}$*

Note that a reduction always results in a negation-free formula. Reduction is a way to stratify a program with respect to negation as failure.

Definition 12 (Stable model) *Let F be a set of propositional formulas and I an interpretation of F. If I is a minimal model of $reduc_I(F)$, then I is a stable model of F. So, stable models of F are fixpoints of*

$$x \mapsto \textit{minimal-model}(reduc_x(F)).$$

Example 9 (Stable model) *Let F= $\{p \vee q \longleftarrow \neg q\}$ and $I_1 = \emptyset$, $I_2 = \{p\}$, $I_3 = \{q\}$, and $I_4 = \{p, q\}$ the four interpretations of F. Then,*

- *$reduc_{I_1}(F) = \{p \vee q \longleftarrow \top\}$ which has two minimal models: $\{p\}$ and $\{q\}$.*
  $I_1 = \emptyset$ is not a minimal model of $reduc_{I_1}(F)$, so I_1 is not a stable model of F.
- *$reduc_{I_2}(F) = \{p \vee q \longleftarrow \top\}$ which has two minimal models: $\{p\}$ and $\{q\}$.*
  $I_2 = \{p\}$ is a minimal model of $reduc_{I_2}(F)$, so I_2 is a stable model of F.
- *$reduc_{I_3}(F) = \{p \vee q \longleftarrow \bot\}$ which has one minimal models: $\emptyset$.*
  $I_3 = \{q\}$ is not a minimal model of $reduc_{I_3}(F)$, so I_3 is not a stable model of F.
- *$reduc_{I_4}(F) = \{p \vee q \longleftarrow \bot\}$ which has one minimal models: $\emptyset$.*
  $I_4 = \{p, q\}$ is not a minimal model of $reduc_{I_4}(F)$, so I_4 is not a stable model of F.

Conclusion: F admits only $I_2 = \{p\}$ as a stable model.

Consequence 1 (Stable model) *Let F be a set of propositional formulas without negation, then a model of F is stable if and only if it is minimal.*

Convention 1 (Classical negation) *Let F be a set of propositional formulas, if F contains a classical negation of a propositional variable f then the vocabulary contains f and $-f$.*

Classical negation of f is treated as a new propositional variable $-f$.

Example 10 (Classical negation) *Let F= $\{p, -p \vee q\}$ then the vocabulary of F is $\{p, -p, q\}$*

Definition 13 (Consistency) *A model is inconsistent if it contains both propositional variables f and $-f$. It is consistent otherwise.*

Example 11 (Consistency)

- $\{p, -p, q\}$ *is inconsistent;*
- $\{p, q\}$ *and* $\{-p, q\}$ *are consistent.*

Algorithm 1 (Computing stable models with classical negation) *Let F be a set of propositional formulas with classical negation, then the stable models of F can be computed in two step:*

- *First, calculate the stable model of F according to Convention 1;*
- *Secondly, eliminate all the inconsistent stable models obtained from previous step.*

Example 12 (Computing stable models with classical negation) *Let F= $\{p, -p \vee q\}$:*

- *The vocabulary of F is $\{p, -p, q\}$;*
- *There are $2^3 = 8$ interpretations of F ($\emptyset, \ldots, \{p, -p, q\}$);*
- *Among these interpretations, $\{p, q\}$, $\{p, -p\}$ and $\{p, -p, q\}$ are models of F;*
- *Among these models, $\{p, q\}$ and $\{p, -p\}$ are minimal;*
- *Let us calculate the stable models of F according to Convention 1. F does not contains negation as failure, so minimal models are stable model, so $\{p, q\}$ and $\{p, -p\}$ are stable models of F;*
- *F contains classical negation so the consistency of the stable models must be checked:*
  - *$\{p, q\}$ is consistent;*
  - *$\{p, -p\}$ is inconsistent.*

  Then F has only one stable model: $\{p, q\}$.

Several heuristics are used to accelerate the computation of stable models.

Algorithm 2 (The fact heuristics) *Stable models of a program P are the same as stable models of P where propositional variables that occur as facts are replaced by $\top$ in all rule bodies of P.*

Note that applying this heuristics may create new facts. So, it can be applied iteratively.

Algorithm 3 (The anti-fact heuristics) *Stable models of a program P are the same as stable models of P where propositional variables that do not occur neither as facts nor as rule heads are replaced by $\bot$ in all rule bodies of P. Note that the concrete effect of this heuristics is to suppress rules, because a rule that contains a $\bot$ in its body can produce no conclusion in a minimal model.*

Again, as this heuristics tends to suppress rules, it also suppresses rule heads, and can thus create new locations for its application.

Algorithm 4 (The constraint heuristics) *Stable models of a program P with constraints (headless rules) are the same as stable models of P without the constraints, intersected with the models of the constraints.*

This heuristics can help modularizing the computation of stable models.

In summary, CLINGO programs can be translated in a propositional image which consists in a set of propositional formulas. This is called *grounding*. Then, stables models are computed. This is called *solving*. In the CLINGO system, grounding and solving are done by two independent sub-systems, where the output of the grounder is piped into the input of the solver [6].

2.3 Other useful constructs of CLINGO

Besides its logical core, CLINGO offers several constructs that help modelling problems.

2.3.1 Choice rules

Choice rules permit to generate several models in one instruction. It is very useful for combinatorial applications. Indeed, choice rules permit, for instance, to generate very concisely complete families of some kind of models, which is very tricky to do using the logical core of CLINGO.

Definition 14 (Choice rules)

- `{`h_1`;...;`h_n`}`. *This instruction generates a set of models equal to the set of parts of* $\{h_1, \ldots, h_n\}$*;*
- i `{`h_1`;...;`h_n`}` j. *This instruction generates a set of models equal to the set of parts of* $\{h_1, \ldots, h_n\}$ *which have more than* i *elements and less than* j *elements;*
- `{`h_1`;...;`h_n`} =` j. *This instruction generates a set of models equal to the set of parts of* $\{h_1, \ldots, h_n\}$ *which have exactly* j *elements;*
- `{p(X):q(X)}`. *Let* $x_1, ..., x_n$ *be the values satisfying predicate* q*. This instruction generates a set of models equal to the set of parts of* $\{p(x_1), \ldots, p(x_n)\}$*;*
- i `{p(X):q(X)}` j. *Let* $x_1, ..., x_n$ *be the values satisfying predicate* q*. This instruction generates a set of models equal to the set of parts of* $\{p(x_1), \ldots, p(x_n)\}$ *which have more than* i *elements and less than* j *elements;*
- `{p(X):q(X)}=`j. *Let* $x_1, ..., x_n$ *be the values satisfying predicate* q*. This instruction generates a set of models equal to the set of parts of* $\{p(x_1), \ldots, p(x_n)\}$ *which have* j *elements.*

All these forms can be augmented with a body which expresses constraints on the choice rules.

Choice rules with bodies demand to distinguish between local and global variables.

Definition 15 (Local/global variables in choice rules) *A variable in a choice rule with a body is local iff its scope is limited to the head of the choice rule. It is global otherwise, i.e. if its scope contains the body.*

When grounding a choice rule, its global variables are replaced by constant values following the procedure described in Definition 3. The result is a choice rule where only local variables remain, and that is interpreted as in Definition 14.

Next example shows how subtle the role of local/global variable can be. In fact, it is very similar to the distinction between bound and free variables in other domains: quantifiers in logic, or integrals in calculus.

Example 13 (Local/global variables in choice rules) *The following example solves a Sudoku problem.*

```
val(1..9).
% values are digits between 1 and 9

border(1 ; 4 ; 7).
% subsquares start at line/column 1, 4 and 7

s(r1, c1, v1 ; ... ; ... ...).
% cell r1xc1 contains v1, ...
% the initial values written in cells of the problem instance

{s(R,C,V): val(V) } = 1 :- val(R) ; val(C).
% for all positions RxC, there is only one value
% R and C are global, V is local

{s(R,C,V): val(R) } = 1 :- val(C) ; val(V).
```

```
% for all column C and value V, only one row R contains it
% C and V are global, R is local

{s(R,C,V): val(C) } = 1 :- val(R) ; val(V).
% for all row R and value V, only one column C contains it
% R and V are global, C is local

{s(R,C,V):
    val(R), val(C), R1<=R,
    R<=(R1+2), C1<=C, C<=(C1+2)} = 1
  :- val(V), border(R1), border(C1).
% for all values V and subsquares of upper-left corner R1xC1,
% there is only one position RxC that contains V
% V, C1 and R1 are global, C and R are local
```

In short, values of global variables are produced by grounding, while values of local variables are produced by combinatorial choices.

A frequent idiom is to generate candidate models using choice rules, and then to filter the solution models using constraints (see Example 14 which solves the graph coloring optimization problem).

2.3.2 Optimization and aggregation

CLINGO permits to express optimization and aggregation. Both are very useful in practice. It permits, for example, to express interesting constraints on the number of values which satisfy a predicate.

- `#count{X, Y : p(X, Y)}`. This instruction returns the number of value tuples which satisfy p;
- `#count{X : p(X, Y)}`. This instruction returns the number of different `X` values in value tuples which satisfy p;
- `#sum{X : p(X)}`. This instruction returns the sum of integers which satisfy p;
- `#sum{X : p(X, Y)}`. This instruction returns the sum of different integers `X` that occurs in a pair `(X, Y)` which satisfies p;
- `#sum{X, Y : p(X, Y)}`. This instruction returns the sum of all integers `X` (not necessarily different) that occurs in a in a pair `(X, Y)` which satisfies p (if `(1, a)` and `(1, b)` are two such pairs, the sum is 2);
- `#max{X : p(X)}`. This instruction returns the greatest value which satisfies p. By convention, character sequences are greater than integers and sorted in lexicographical order;
- `#min{X : p(X)}`. This instruction returns the smallest value which satisfies p.
- `#maximize{X : p(X)}`. This instruction returns the stable model which maximizes the `#sum{X : p(X)}` instruction;
- `#minimize{X : p(X)}`. This instruction returns the stable model which minimizes the `#sum{X : p(X)}` instruction;
- `#sum{X*X,C : p(X, C)}`. This instruction returns the sum of square of all `X` such as `(X, C)` satisfies p;
- `#count{X : p(X)}` is equivalent to `#sum{1,X : p(X)}`. This is very useful for computing a set cardinality.

2.3.3 Interface with other programming languages

The details of an ASP program may be very tedious, for instance, declaring all vertices and edges of a graph. Conversely, the result can consist in many models that contain many variables. It would be impracticable to build the program by hand, and to look at the results with the eyes. Instead, CLINGO permits to call a CLINGO program and catch its results from a program written in another language.

The idea is to prepare a CLINGO program from external data, to call the CLINGO program, catch its results, and treat them automatically. Python is one of the programming languages that can be used for preparing and calling a CLINGO program.

Conversely, CLINGO can call procedures written in another programming language, for instance, in order to define more constraints.

As a consequence, writing CLINGO programs by hand, and reading results on a screen is limited to a development phase. When exploiting a CLINGO program on real data, it will often be packaged in a main program that will encode the data of the problem into CLINGO statements, launch the resolution of the problem, and analyse the results.

2.4 Example: graph coloring with CLINGO

To give an example of programming in CLINGO, we present a solution to the graph coloring problem. Graph coloring is the problem of coloring the vertices of a graph, without coloring two neighbouring vertices with the same color. In this solution, we solve the optimization variant of this problem: to find the minimum number of colors needed to color the vertices of a graph without coloring two neighbouring vertices with the same color.

Firstly, we introduce three predicate:

- `vertex/1` expresses that its argument is a vertex of the graph;
- `edge/2` expresses that its two arguments are linked by an edge, they are neighbours;
- `color/2` expresses that its first argument has the color noted in its second argument. We assume that colors are coded as integers.

The following program solves the optimal graph coloring problem:

Example 14 (Solving an optimization problem)

```
vertex(a ; b ; c ; d ; e).
% Vertices of the graph are created.

edge(a, b ; b, c ; c, d ; d, e ; e, a).
% Edges of the graph are created.

{color(X,1..N)} = 1 :- vertex(X), N = #count{Y : vertex(Y)}.
% All the models where a vertex has exactly one color are created.
% The choice rule creates one color(X,C) for every X

:-edge(X,Y),color(X,C),color(Y,C).
% All models where two linked vertex have the same color are deleted.
% The constraint filters out invalid models

#minimize{1,C:color(X,C)}.
% The number of color of the graph is minimized.
```

Note that all these facts, rules and constraints can be presented in any order. The result will always be the same, even though the computation times may be very different.

A possible approach for developing this kind of program is to introduce statements progressively, and test them incrementally to check that it behaves as expected. A possible test is to check that there is as many models as a theoretical enumeration predicts. For instance, with the previous example this approach amounts to testing the `vertex` statement alone (1 model with 5 elements expected), then the `vertex` and `edge` statements (1 model with 10 elements expected), then the `vertex` and `edge` and `color` statements ($5^5 = 3125$ models expected), and so on.

This works fine, but only for small instances because it is often the case (see the example above) that the first statements produce an enormous amount of models that are filtered out by the other statements. ASP solvers have a more global approach and do not follow this generate and test approach. So, it is often the case that a program works fine, but some part of it is intractable.

2.5 Conclusion on answer set programming

ASP is a very interesting way of programming, especially for combinatorial applications. Furthermore, as opposed to PROLOG-like languages the concrete implementations of ASP languages are very close to its semantics, which is a great advantage of ASP.

We have presented ASP and CLINGO to the LogicA team in its internal seminar, and we have used it in relation to the *attack* tree research of the team.

3 The semantics of attack trees

The second part of our internship was to design concrete operations on attack trees.

Attack tree is a conceptual diagram for describing security of system. It can be used by an attacker to plan an attack or by a defender to describe the potential threats on a system. Attack trees have a tree structure. The goal of the attack is stored in the root and the different way of reaching the goal are describes by some paths in the tree. Attack tree contains three kind of nodes.

- OR nodes which express several alternative paths to achieve a goal (do this or that) ;
- SAND nodes which express several paths that must be followed in sequence (do this then that);
- AND nodes which express several paths that can be intermingled (do this and that).

In summary, an attack tree is roughly similar to a regular expression, except that the variant we have worked with only describes finite languages. Starting from this, it is interesting to compute the language of all paths to the goal as a set of words, or to compute an automaton that recognizes the language. Both have a finite representation, and can be converted into each other. However the sizes of these two representations may differ greatly.

It is also interesting to consider only *minimal* words (that is words that have no subwords in the language; this will be formally specified below) and compute an automaton that accepts them and only them. The minimality property is based on a subword relation which it is tempting to express in ASP, but appears to be more efficient to implement in an imperative programming language. This is the subject of Section 3.1.

The semantics of attack trees is presented in Section 3.2. It combines three operations, union, word product, and shuffle [9], that can be implemented either extensionally by computing word sets, or intensionally by computing automatons. Furthermore, one may wish to compute all words or only minimal words. One may also wish to start from a language and compute a possible shuffle decomposition. Section 3.2 presents these ideas.

3.1 The subword relation

The subword relation is defined as follows:

Definition 16 (Subword, by induction) *Let Σ be an alphabet, $\bullet$ be the concatenation operator, and ϵ be the empty word.*

Predicate $sw \subseteq \Sigma^* \times \Sigma^*$ (has subword) is defined as follows:

- *$sw(\epsilon, \epsilon)$;*
- *for all x and y, if $sw(x, y)$ then for all $a \in \Sigma$, $sw(a \bullet x, y)$ and $sw(a \bullet x, a \bullet y)$.*

Remark 1 (A few elementary results on $sw(.,.)$)

1. *The sw relation is transitive.*
2. *For every word $w = u \bullet v$, we have $sw(w, \epsilon)$, and $sw(w, u' \bullet v')$ if $sw(u, u')$ and $sw(v, v')$.*

The sw predicate expresses a relation between two words. This relation is an order relation. The rest of this section analyses how to implement this relation.

3.1.1 A naive computation of predicate sw

A way to compute sw with CLINGO is simply to follow Definition 16. This definition permits to deduce that the empty word is a subword of all words. So we start from the fact that the empty word is a subword of x and then we build the other subwords of x following the definition. Finally, we test if `y` as been built. The tricky part of the implementation is that CLINGO does not have the usual operators for manipulating character sequences, like for example the concatenation operator. A solution is to represent the words by a cascade of nested constructors. For instance:

- the word *abcd* becomes $f(a, f(b, f(c, f(d, end))))$;
- by convention the empty word will be represented by *end*.

Then, we use the fact that we can include CLINGO program in a Python program for generating this constructor cascade. Finally, we reuse Python for exploiting sw from the stable model generated by the CLINGO program. The global resolution scheme is:

- Generate CLINGO input with Python;
- Calculate a stable model of the CLINGO program;
- Decide sw from the stable model with Python.

The CLINGO program will be as follows:

```
#const c0=x.
%the value of c0 is x.

#const c1=y.
% the value of c1 is y.

word(c0).
% c0 is a word.

word(Y) :- word(f(X,Y)).
% if x+y is a word then y is a word
```

```
sw(X,end) :- word(X).
% if X is a word then the empty word is a subword of X.

sw(f(X,Y),f(X,Z)) :- sw(Y,Z), word(f(X,Y)).
% if z is a subword of y and x+y is a word,
% then x+z is a subword of x+y.

sw(f(W,X),f(Y,Z)) :- sw(X,f(Y,Z)), word(f(W,X)).
% if y+z is a subword of y and w+x is a word,
% then y+z is a subword of w+x.

:- not sw(c0,c1).
% if c1 is not a subword of c0, then the current model is not a model of the program.
```

To decide if $sw(x, y)$, the external program encodes x and y as f and end structures, and then it launches the CLINGO program. The result is obtained by testing whether the CLINGO program has a stable model:

- if it has a stable model the external program returns $true$;
- otherwise it returns $false$.

We have tested the empirical time complexity of this solution. To do so, we made the following experiment:

- For a given $size$ of word:
  - by choosing letters in the latin alphabet, construct a random word x with size $size$ and a random word y with size $size/2$, measure the run time in second of $sw(x, y)$, and repeat;
  - calculate the average run time in second of $sw(x, y)$ for size $size$.
- repeat for different values of $size$.

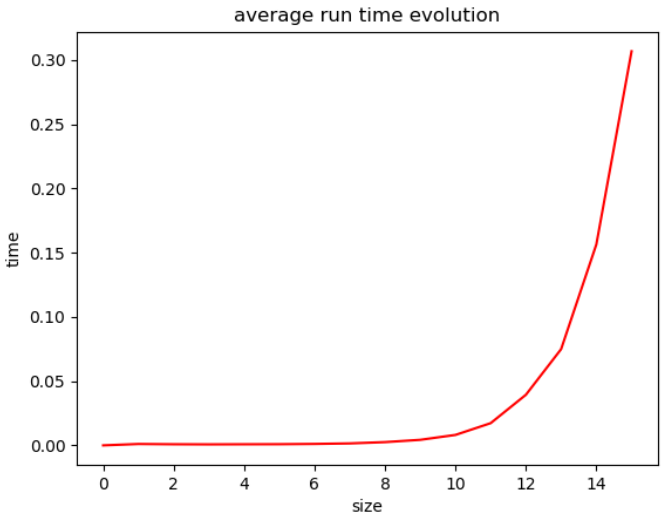

Figure 1: Empirical time complexity of sm, linear scales

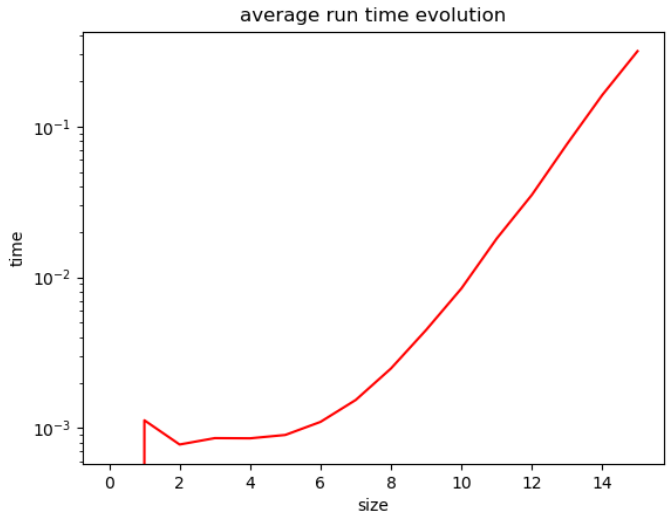

Figure 2: Empirical time complexity of sm, linlog scales

Figure 1 displays the graph of these measures. The graph shows that the empirical time complexity of the CLINGO program for testing sw seems to be exponential. This hypothesis can be confirmed by displaying the same data as a linlog scale graph as in Figure 2. On this figure we see that, asymptotically, the graph seems to be linear. This confirm that the empirical time complexity of the CLINGO program for testing sw is exponential. However, an algorithm exists for deciding sw in linear time. So, this approach is definitely not efficient. Experiments show that it cannot be really used for words longer than 15.

This result is not surprising. Indeed, the CLINGO program generates all the subwords of x, then tests if y is among them. The cost of this generation is huge because if the size of a word is n, it has 2^n subwords. This observation forces us to find another way for solving this program with CLINGO.

3.1.2 A more efficient computation of predicate sw

In this part, we describe a better way to decide $sw(x, y)$ with CLINGO. As we have seen previously, an efficient solution must avoid to generate all the subwords of a word. An idea is to read the candidate subword y letter by letter, from left to right. Then, for each letter, a *place* in the superword (x) is searched, and this procedure is iterated with the next letter and the rest of x.

A letter has a *place* in x iff:

- This letter is present in x;
- No previous letter has a place farther.

For implementing this, we use a new way to represent words: a word is now represented by its letters and positions, using predicates as follow:

- If x is a superword of size n and $x(i)$ denotes the i-th character of x, x will be represented in CLINGO by:
  - `word(1,`$x(1)$`,1)`$\ldots$`word(1,`$x(i)$`,i)`$\ldots$`word(1,`$x(n)$`,n).`
- If y is a subword of size n and $y(i)$ denotes the i-th character of y, y will be represented in CLINGO by:

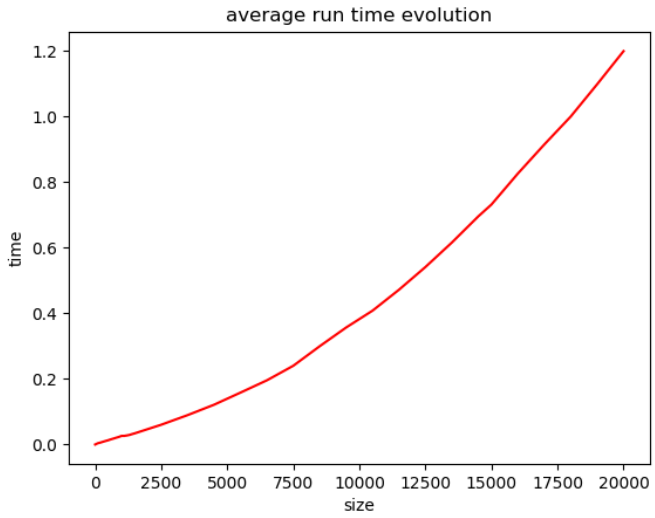

Figure 3: Empirical time complexity of sm, linear scales

– `word(2,`$y(1)$`,1)`$\ldots$`word(2,`$y(i)$`,i)`$\ldots$`word(2,`$y(n)$`,n).`

As for the naive solution, for deciding $sw(x, y)$, x and y are encoded in CLINGO by an external program written in Python. A stable model is then computed. It is then analyzed by the external program to compute the final answer.

The CLINGO program is as follows:

```
place(B, 1, N) :- mot(2, B, 1),
                  N = #min{A:mot(1, B, A)},
                  N != #sup.
% if the first character B of y is in x, at place B,
% then N is the minimal integer such that x(N)=B.

place(B, M+1, N1) :- place(A, M, N2), mot(2, B, M+1),
                     N1 = #min{Z:mot(1, B, Z), Z > N2},
                     N1 != #sup.
% if A the M-ith character of y is placed at N2 in x, and
% if B is the M+1-th character of y, and
% if N1 is the left-most place in x that contains B, such that N1>N2,
% then B the M+1-th character of y is placed at N1

:- mot(2, X, N), not place(X, N, _).
% it is not possible that a character of y (mot(2, X, N)) is not placed
```

We have measured the empirical time complexity of this solution, using the same method as above. Figure 3 shows the results. The graph shows that the empirical time complexity seems to be polynomial or exponential. Figure 4 displays the same data in a linlog graph. It shows that the graph definitely does form a convex line, so this empirical complexity is lower than exponential. Finally, Figure 5 displays the same data in a loglog graph. It forms a slightly concave line, close to a straight line with slope 1. So, this empirical complexity is a little bit more complex than linear. A simple analysis of the problem shows that it is of linear complexity. So, the ASP solution seems to be slightly sub-optimal.

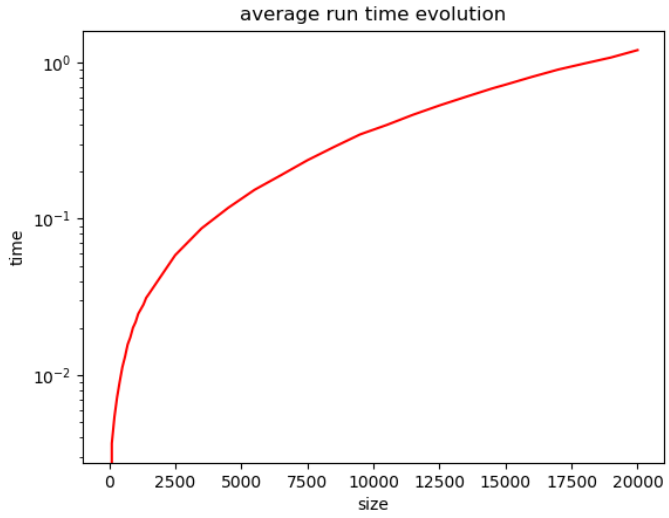

Figure 4: Empirical time complexity of *sm*, linlog scales

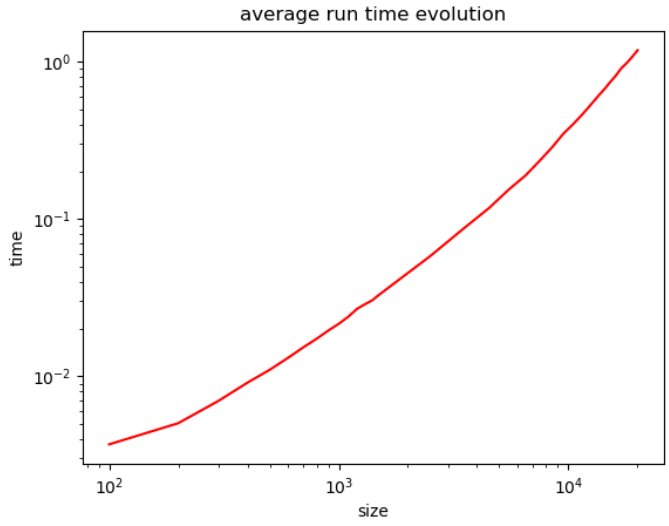

Figure 5: Empirical time complexity of *sm*, loglog scales

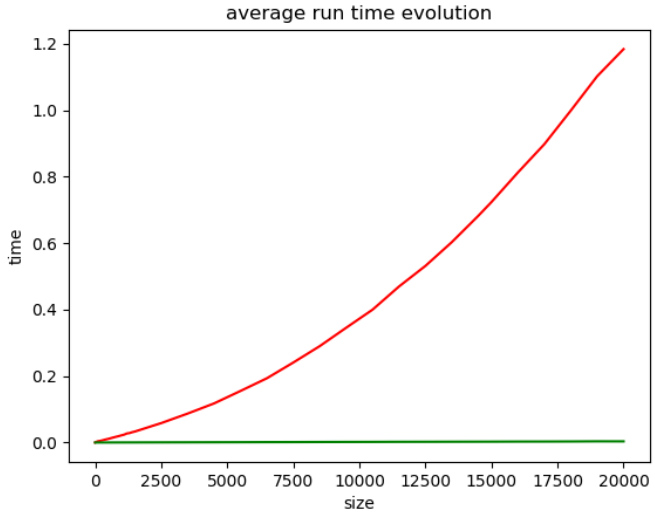

Figure 6: Comparisons of empirical time complexity, linear scales

3.1.3 An iterative computation of sw

The CLINGO solution above encodes an iterative procedure that could be programmed more effectively using a plain imperative programming language. So, we have designed a Python program for deciding $sw(x, y)$, using the same iterative solution.

The Python program is as follows:

```
def smIt(x,y):
   CLINGO"""
   lenx=len(x)
   leny=len(y)
   i=0
   j=0
   while i<lenx and j<leny :
      if x[i]==y[j]:
         i=i+1
         j=j+1
      else:
         i=i+1
   return leny==j
```

Figure 6 compares the performances of the two iterative solutions. The comparison is done with the same protocol as previously. On this figure, the run time of the CLINGO program is represented in red and the run time of the Python program is represented in green. As expected, the Python program is much more efficient than the CLINGO program.

These experiments do not say CLINGO is useless. Rather, they say that CLINGO should be reserved for problems for which no evident algorithmic solution exists.

3.1.4 Minimal elements for the subword order

We use the subword relation to order a set of words, and determine its minimal elements, i.e. the words of the set which have no proper subwords in the set.

Definition 17 (min(.)) *Let S be a set of word, then the set of all its elements that have no proper subwords in S is noted $min(S)$. These elements are called minimal elements.*

Remark 2 (A few elementary results on $min(.)$)

1. $\min(S) \subseteq S$ *for all S follows directly from the definition.*
2. $\min(\min(S)) = \min(S)$ *for all S follows directly from the definition because words in* $\min(S)$ *have no proper subwords in S, so neither in a subset of S.*
3. *However,* $\min(S) \subseteq \min(T)$ *does not follow at all from $S \subseteq T$. Consider for instance $S = T - min(T)$, where $-$ is set theoretic subtraction. Then, $min(T - min(T))$ is certainly not included in $min(T)$.*

3.2 Attack Trees

3.2.1 The syntax of an attack tree

Definition 18 (Attack tree) *An attack tree is built from three kinds of nodes:*

- *OR node*
- *AND node*
- *SAND node*

The leaves of an attack tree are words defined on a given alphabet.

3.2.2 The semantics of an attack tree

Before defining the semantics of an attack tree, let us define some operators on languages.

Definition 19 (Union operator) *The union operator, noted $\cup$, corresponds to the union of two sets of words.*

That $\cup$ is monotonic for set inclusion is a well known result of set theory.

Definition 20 (Product operator) *The product operator, noted $\odot$, is an operation between two sets of words. Its semantics is as follow:*

Let us note $\bullet$ the concatenation between to words. Let A and B be two sets of words. Then:

$A \odot B = \{x \bullet y \mid x \in A \wedge y \in B\}$

That $\odot$ is monotonic for set inclusion in its left and right operands follows directly from this definition.

Definition 21 (Shuffle operator) *The shuffle operator, noted $\sqcup\!\sqcup$, is an operation between two set of words. Its definition is as follow:*

Let A and B be two sets of words. Then:

$A \sqcup\!\sqcup B$ is the smallest set composed of all word x such as there exists y in A and z in B and $sw(x, y) \wedge sw(x, z)$ [8].

That $\sqcup\!\sqcup$ is monotonic for set inclusion in its left and right operands follows directly from this definition. It follows also that any word in A or B is a subword of words of $A \sqcup\!\sqcup B$.

Example 15 (Shuffle operator)

$\{ab, c\} \sqcup\!\sqcup \{d\} = \{abd, adb, dab, cd, dc\}$

Then, the semantics of an attack tree consists in a set of words defined by induction as follows:

Definition 22 (Semantics of an attack tree) *The semantics of an attack tree θ, written $[\![\theta]\!]$, is defined by induction as follows:*

- $[\![a]\!] \stackrel{\text{def}}{=} \{a\}$;
- $[\![OR(\theta_1, \ldots, \theta_n)]\!] \stackrel{\text{def}}{=} [\![\theta_1]\!] \cup \cdots \cup [\![\theta_n]\!]$;
- $[\![SAND(\theta_1, \ldots, \theta_n)]\!] \stackrel{\text{def}}{=} [\![\theta_1]\!] \odot \cdots \odot [\![\theta_n]\!]$;
- $[\![AND(\theta_1, \ldots, \theta_n)]\!] \stackrel{\text{def}}{=} [\![\theta_1]\!] \sqcup\!\sqcup \cdots \sqcup\!\sqcup [\![\theta_n]\!]$.

Since this semantics is a set of words, we call it the *language of the attack tree*. Furthermore, for the same reason, we can define the *minimal semantics* of an attack tree as the set of minimal elements of its semantics, using the definition of minimal element given in Definition 17.

Definition 23 (Minimal semantics of an attack tree) *The minimal semantics, $[\![\theta]\!]_{min}$, of an attack tree θ is the set of all minimal words of $[\![\theta]\!]$.*

When needed we call $[\![.]\!]$ the standard semantics of θ to distinguish it from the minimal semantics.

A naive way of computing the minimal semantics of an attack tree is to compute its standard semantics (a set of words) and then compute its minimal words. However, this solution may be inefficient, because it may compute a large number of words, of which very few are minimal. The naive algorithm is like computing with fractions without ever simplifying them before the end of the computation. Deciding when to apply simplification is largely a matter of heuristics.

Example 16 (Arbitrarily large benefit of the direct computation of the minimal semantics) *The set of minimal words of an attack tree $AND(OR(a, a^2, \ldots, a^n), OR(b, b^2, \ldots, b^m))$ is the singleton $\{ab\}$, though its standard semantics contains $\frac{(n+m)!}{n!m!}$ words.*

However, the benefit need not be as large as in Example 16.

Example 17 (Arbitrarily small benefit of the direct computation of the minimal semantics) *All words in the standard semantics of $AND(OR(a_1, a_2, \ldots, a_n), OR(b_1, b_2, \ldots, b_m))$ are minimal, but still, computing directly the minimal semantics may be a winner because it dispenses from checking the minimal words of a large set.*

The minimal semantics of an attack tree can be computed directly by induction on the attack tree. An automaton that recognizes the minimal semantics can also be computed directly from the attack tree. This is the subject of the rest of this section and of the following section.

The following theorems show that at each stage of the computation of the semantics of an attack tree, the minimal semantics of a node only depends on the minimal semantics of the children nodes. There is no need to compute the standard semantics. Moreover, the relationship between the minimal semantics of a node and the minimal semantics of its children is simple and easy to compute. In fact, it is the same operations as for the standard semantics, but applied to smaller sets of words, and sometimes completed with a minimization of the result.

The following theorems show that the minimal semantics of a leaf is its standard semantics, and that the minimal semantics of a node is the minimization of its standard operation applied to the minimal semantics of its children (OR nodes), and even sometimes simply the standard operation applied to the minimal semantics of its children (SAND nodes and AND nodes).

As a consequence, the direct computation of the minimal semantics always operates on smaller sets of words, but may incur the extra cost of minimizing the results for OR nodes only. Heuristics for not minimizing at all OR nodes could be useful, just as an heuristic for not reducing fractions at each step is useful, but it requires to know better the distribution of OR, SAND, and AND nodes in typical use cases of attack trees. This is left to further works.

Theorem 1 (Minimal semantics of a leaf) *Let a be a leaf, then $[\![a]\!]_{min} = \{a\}$*

Proof 1 ($[\![a]\!]_{min} = \{a\}$)
$[\![a]\!]_{min} = \min([\![a]\!]) = \min(\{a\})$, *by definition of* $[\![.]\!]_{min}$ *and* $[\![.]\!]$, *and because a is a leaf,*
$= \{a\}$, *because word a has no proper subword in* $\{a\}$.
QED

The minimal semantics of an OR node is the minimal elements of the union of the minimal semantics of its children.

Theorem 2 (Minimal semantics of an OR tree) *Let* $\theta_1, \ldots, \theta_n$ *be attack trees. Then:*
$[\![OR(\theta_1, \ldots, \theta_n)]\!]_{min} = \min([\![\theta_1]\!]_{min} \cup \ldots \cup [\![\theta_n]\!]_{min})$.

Proof 2 ($[\![OR(\theta_1, \ldots, \theta_n)]\!]_{min} = \min([\![\theta_1]\!]_{min} \cup \ldots \cup [\![\theta_n]\!]_{min})$)
$[\![OR(\theta_1, \ldots, \theta_n)]\!]_{min} = \min([\![OR(\theta_1, \ldots, \theta_n)]\!])$, *by definition of* $[\![.]\!]_{min}$,
$= \min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!])$, *by definition of* $[\![.]\!]$.
Prove that $\min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!]) = \min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!]))$ *by mutual inclusion.*

- *Prove that* $\min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!]) \subseteq \min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!]))$*:*
  Let x *be an element of* $\min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!])$ *then:*
  1. x *is (a) an element of some* $[\![\theta_i]\!]$ *with (b) no proper subwords in* $[\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!]$,
  2. x *has no proper subwords in* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$, *from 1.(b) and because* $\min(S) \subseteq S$ *(see remark 2),*
  3. x *is an element of some* $[\![\theta_i]\!]$ *with no proper subwords in* $\min([\![\theta_i]\!])$, *from 1.(a) and 2, hence* x *is in* $\min([\![\theta_i]\!])$, *hence in* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$,
  4. x *is an element of* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$ *with no proper subwords in* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$, *from 2. and 3.,*
  5. *hence,* x *is an element of* $\min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!]))$.
- *Prove that* $\min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])) \subseteq \min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!])$*:*
  Let x *be an element of* $\min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!]))$ *then:*
  1. x *is (a) an element of* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$ *with (b) no proper subwords in* $\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])$,
  2. x *is an element of some* $\min([\![\theta_i]\!])$, *from 1.(a), hence* x *is an element of some* $[\![\theta_i]\!]$, *because* $\min(S) \subseteq S$ *(see remark 2),*
  3. x *has no proper subwords in* $[\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!]$, *from 1.(b) and because sw relation is transitive (see remark 1),*
  4. x *is an element of* $\min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!])$

So, $\min([\![\theta_1]\!] \cup \ldots \cup [\![\theta_n]\!]) = \min(\min([\![\theta_1]\!]) \cup \ldots \cup \min([\![\theta_n]\!])) = \min([\![\theta_1]\!]_{min} \cup \ldots \cup [\![\theta_n]\!]_{min})$, *by definition of* $[\![.]\!]_{min}$.
QED

The minimal semantics of a SAND node is the sequential product of the minimal semantics of its children.

Theorem 3 (Minimal semantics of a SAND tree) *Let* $\theta_1, \ldots, \theta_n$ *be attack trees. Then:*
$[\![SAND(\theta_1, \ldots, \theta_n)]\!]_{min} = [\![\theta_1]\!]_{min} \odot \ldots \odot [\![\theta_n]\!]_{min}$.

Proof 3 ($[\![SAND(\theta_1, \ldots, \theta_n)]\!]_{min} = [\![\theta_1]\!]_{min} \odot \ldots \odot [\![\theta_n]\!]_{min}$)

$$\begin{array}{rcll} [\![SAND(\theta_1, \ldots, \theta_n)]\!]_{min} & = & \min([\![SAND(\theta_1, \ldots, \theta_n)]\!]), & \text{by definition of } [\![.]\!]_{min}, \\ & = & \min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]), & \text{by definition of } SAND(.). \end{array}$$

Prove that $\min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]) = \min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!])$ *by mutual inclusion.*
First note that (*) $\min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]) \subseteq [\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$, *from remark 2,*
and that (**) $\min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!]) \subseteq [\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$, *by monotonicity of* $\odot$.

- *Prove that* $\min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]) \subseteq \min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!])$ *by refutation:*

  If $\min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]) \not\subseteq \min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!])$ *then:*

  1. *there is a word* w *such that (a)* w *in* $[\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$ *and* w *has no proper subword in* $[\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$ *and (b)* w *has a proper subword* sw *in* $\min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!])$,
  2. *so* $sw \in [\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$, *by note (**),*
  3. *so* sw *is a proper subword of* w *in* $[\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$.
  4. *this contradicts 1.(a).*

- *Prove that* $\min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!]) \subseteq \min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!])$ *by refutation:*

  If $\min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!]) \not\subseteq \min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!])$, *then:*

  1. *there is a word* $w = w_1 \bullet \ldots \bullet w_n$ *where (a) for all position* i w_i *belongs to* $\min([\![\theta_i]\!])$, *hence belongs to* $[\![\theta_i]\!]$ *and has no proper subword in* $[\![\theta_i]\!]$, *such that (b)* w *belongs to* $[\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$ *and has a proper subword in* $[\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]$, *by note (**) and definition of* $\min(.)$,
  2. *there is a position* i *such that* w_i *has a proper subword* w_i' *in* $[\![\theta_i]\!]$, *from 1.(b) and remark 1.*
  3. *this contradicts 1.(a).*

So, $\min([\![\theta_1]\!] \odot \ldots \odot [\![\theta_n]\!]) = \min([\![\theta_1]\!]) \odot \ldots \odot \min([\![\theta_n]\!]) = [\![\theta_1]\!]_{min} \odot \ldots \odot [\![\theta_n]\!]_{min}$, *by definition of* $[\![.]\!]_{min}$. *QED*

The minimal semantics of an AND node is the shuffle of the minimal semantics of its children.

Theorem 4 (Minimal semantics of an AND tree) *Let* $\theta_1, \ldots, \theta_n$ *be attack trees. Then:* $[\![AND(\theta_1, \ldots, \theta_n)]\!]_{min} = [\![\theta_1]\!]_{min} \sqcup \ldots \sqcup [\![\theta_n]\!]_{min}$.

Proof 4 $[\![AND(\theta_1, \ldots, \theta_n)]\!]_{min} = [\![\theta_1]\!]_{min} \sqcup \ldots \sqcup [\![\theta_n]\!]_{min}$

$$\begin{array}{rcll} [\![AND(\theta_1, \ldots, \theta_n)]\!]_{min} & = & \min([\![AND(\theta_1, \ldots, \theta_n)]\!]), & \textit{by definition of } [\![.]\!]_{min}, \\ & = & \min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]), & \textit{by definition of } AND(\ldots). \end{array}$$

Prove that $\min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]) = \min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!])$ *by mutual inclusion.*
First note that (*) $\min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]) \subseteq [\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$, *from remark 2,*
and that (**) $\min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!]) \subseteq [\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$, *by monotonicity of* $\sqcup$.

- *Prove that* $\min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]) \subseteq \min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!])$ *by refutation (similar to first part of previous proof):*

  If $\min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]) \not\subseteq \min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!])$ *then:*

  1. *there is a word* w *such that (a)* w *in* $[\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$ *and* w *has no proper subword in* $[\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$ *and (b)* w *has a proper subword* sw *in* $\min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!])$,
  2. *so* $sw \in [\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$, *by note (**),*
  3. *so* sw *is a proper subword of* w *in* $[\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$.
  4. *this contradicts 1.(a).*

- *Prove that* $\min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!]) \subseteq \min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!])$ *by refutation:*

  If $\min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!]) \not\subseteq \min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!])$, *then:*

  1. *there is a word* $w \in \{w_1\} \sqcup \ldots \sqcup \{w_n\}$ *where (a) for all* i w_i *belongs to* $\min([\![\theta_i]\!])$, *hence belongs to* $[\![\theta_i]\!]$ *and has no proper subword in* $[\![\theta_i]\!]$, *such that (b)* w *belongs to* $[\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$ *and has a proper subword in* $[\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]$, *by note (**) and definition of* $\min(.)$,

2. *then* $w \in \{u_1\} \sqcup \ldots \sqcup \{u_n\}$ *where for every* i u_i *belongs to* $[\![\theta_i]\!]$*, by (1).b,*

3. *there is an* i *such that* u_i *is a proper subword of* w_i *in* $[\![\theta_i]\!]$*, and from 1.(b).*

4. *this contradicts 1.(a).*

So, $\min([\![\theta_1]\!] \sqcup \ldots \sqcup [\![\theta_n]\!]) = \min([\![\theta_1]\!]) \sqcup \ldots \sqcup \min([\![\theta_n]\!]) = [\![\theta_1]\!]_{min} \sqcup \ldots \sqcup [\![\theta_n]\!]_{min}$*, by definition of* $[\![.]\!]_{min}$*.*

QED

Consequence 2 (Minimal semantics of an attack tree) *The minimal semantics of an attack tree* θ*, written* $[\![\theta]\!]_{min}$*, can be defined by induction as follows:*

- $[\![a]\!]_{min} \stackrel{\text{def}}{=} \{a\}$*;*
- $[\![OR(\theta_1, \ldots, \theta_n)]\!]_{min} \stackrel{\text{def}}{=} \min([\![\theta_1]\!]_{min} \cup \cdots \cup [\![\theta_n]\!]_{min})$*;*
- $[\![SAND(\theta_1, \ldots, \theta_n)]\!]_{min} \stackrel{\text{def}}{=} [\![\theta_1]\!]_{min} \odot \cdots \odot [\![\theta_n]\!]_{min}$*;*
- $[\![AND(\theta_1, \ldots, \theta_n)]\!]_{min} \stackrel{\text{def}}{=} [\![\theta_1]\!]_{min} \sqcup \cdots \sqcup [\![\theta_n]\!]_{min}$*.*

Consequence 3 (Complexity of the direct computation of the minimal semantics) *In the case of an attack tree with only* $SAND$ *and* AND *nodes.*

The cost of computing directly the minimal semantics is bounded by the cost of computing the standard semantics, because it incurs exactly the same operations for every nodes, but applied to possibly smaller sets of words.

The ratio between the cost of the direct computation to the cost of the standard one can be made arbitrarily close to 0 (see Example 16). It can also be made arbitrarily close to 1 (see Example 17).

The cost of computing directly the minimal semantics of an attack tree is strictly bounded by the cost of computing the standard semantics, and then minimizing the result, because the latter incurs the extra cost of minimization.

When an attack tree also contains OR nodes, things are more complicated, as examples 16 and 17 show. In this case, everything depends on the relative positions and numbers of the different types of nodes.

3.2.3 The automaton of an attack tree

The previous section defined the (minimal) semantics of an attack tree extensionally as sets of words. One can also define it intensionally as automatons that recognize the sets of words.

Recall 1 (Finite automaton) *A finite automaton is a 5-tuple* $A = (\Sigma, Q, \delta, i, F)$ *where:*

- Σ *is a finite set of symbols called the* alphabet*;*
- Q *is a finite set of* states*;*
- δ *is a relation from* $Q \times \Sigma \times Q$ *called* transitions*;*
- i *is a state of* Q *called* initial state*;*
- F *is a subset of* Q *which consists of elements called* final states*.*

Recall 2 (Finite automaton with ϵ-transition) *Same definition as above with*

- δ *is a relation from* $Q \times (\Sigma \cup \{\epsilon\}) \times Q$*.*

If a finite automaton has no ϵ*-transition it is called* ϵ*-free.*

Recall 3 (Finite automaton with ϵ-transition $\longrightarrow$ finite automaton) *Every finite automaton with ϵ-transitions can be transformed in an ϵ-free finite automaton that recognizes the same language. The principle of the transformation is the same as for determinization.*

We assign to each attack tree θ an automaton as follows:

Definition 24 (Attack Tree Semantics $\longrightarrow$ Automaton)

- *If θ is a leaf which consists of a word $S = \sigma_1 \bullet \ldots \bullet \sigma_n$ (let $\bullet$ denotes concatenation), then its automaton is $A = (\Sigma, Q, \delta, i, F)$ where :*
  - $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$;
  - $Q = \{q_1, \ldots, q_{n+1}\}$;
  - $\delta = \{(q_1, \sigma_1, q_2), \ldots, (q_n, \sigma_n, q_{n+1})\}$;
  - $i = \sigma_1$;
  - $F = \{q_{n+1}\}$.
- *If θ is an OR tree, $\theta = OR(\theta_1, \theta_2)$ where θ_1 and θ_2 are attack trees associated with finite automaton $A_1 = (\Sigma_1, Q_1, \delta_1, i_1, F_1)$ and $A_2 = (\Sigma_2, Q_2, \delta_2, i_2, F_2)$, then its automaton is $A = (\Sigma, Q, \delta, i, F)$ where :*
  - $\Sigma = \Sigma_1 \cup \Sigma_2$;
  - $Q = \{i\} \cup Q_1 \cup Q_2$ *(we assume $\{i\}$, Q_1 and Q_2 to be disjoint sets; if it is not the case, they can be renamed);*
  - $\delta = \{(i, \epsilon, i_1), (i, \epsilon, i_2)\} \cup \delta_1 \cup \delta_2$;
  - $F = F_1 \cup F_2$.
- *If θ is a SAND tree, $\theta = SAND(\theta_1, \theta_2)$, where θ_1 and θ_2 are attack trees associated with finite automaton $A_1 = (\Sigma_1, Q_1, \delta_1, i_1, F_1)$ and $A_2 = (\Sigma_2, Q_2, \delta_2, i_2, F_2)$, then its automaton is $A = (\Sigma, Q, \delta, i, F)$ where:*
  - $\Sigma = \Sigma_1 \cup \Sigma_2$;
  - $Q = Q_1 \cup Q_2$ *(we assume Q_1 and Q_2 to be disjoint sets; if it is not the case, they can be renamed);*
  - $\delta = \{(f, \epsilon, i_2) \mid f \in F_1\} \cup \delta_1 \cup \delta_2$;
  - $i = i_1$;
  - $F = F_2$.
- *If θ is an AND tree, $\theta = AND(\theta_1, \theta_2)$, where θ_1 and θ_2 are attack trees associated with finite automaton $A_1 = (\Sigma_1, Q_1, \delta_1, i_1, F_1)$ and $A2 = (\Sigma_2, Q_2, \delta_2, i_2, F_2)$, then its automaton is $A = (\Sigma, Q, \delta, i, F)$ where:*
  - $\Sigma = \Sigma_1 \cup \Sigma_2$;
  - $Q = Q_1 \times Q_2$;
  - $\delta = \{((q_1, q_2), \sigma, (q, q_2)) \mid ((q_1, q_2), (q, q_2)) \in Q \times Q \wedge (q_1, \sigma, q) \in \delta_1\}$
    $\cup \ \{((q_1, q_2), \sigma, (q_1, q)) \mid ((q_1, q_2), (q_1, q)) \in Q \times Q \wedge (q_2, \sigma, q) \in \delta_2\}$
  - $i = (i_1, i_2)$;
  - $F = F_1 \times F_2$.

We note $A(\theta)$ the automaton of θ.

In this definition we have used automaton with ϵ-transitions because they are easier to combine.

We extend the previous definition for n-ary nodes using associativity of $\cup$, $\odot$, $\sqcup\!\sqcup$ operations as follows:

- $A(OR(\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n))$
  $= A(OR(\theta_1, OR(\theta_2, OR(\ldots, OR(\theta_{n-1}, \theta_n)\ldots))))$
- $A(SAND(\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n))$
  $= A(SAND(\theta_1, SAND(\theta_2, SAND(\ldots, SAND(\theta_{n-1}, \theta_n)\ldots))))$
- $A(AND(\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n))$
  $= A(AND(\theta_1, AND(\theta_2, AND(\ldots, AND(\theta_{n-1}, \theta_n)\ldots))))$

Example 18 *$A(abc) = (\Sigma, Q, \delta, i, F)$ where:*

- $\Sigma = \{a, b, c\}$;
- $Q = \{q_1, q_2, q_3, q_4\}$;
- $\delta = \{(q_1, a, q_2), (q_2, b, q_3), (q_3, c, q_4)\}$;
- $i = q_1$;
- $F = \{q_4\}$.

A picture of this automaton is display Figure 7.

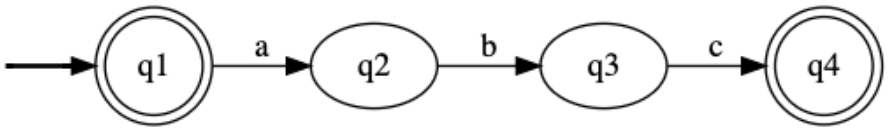

Figure 7: Picture of $A(abc)$

Example 19 *$A(OR(abc, def))$ after minimization is visible Figure 8*

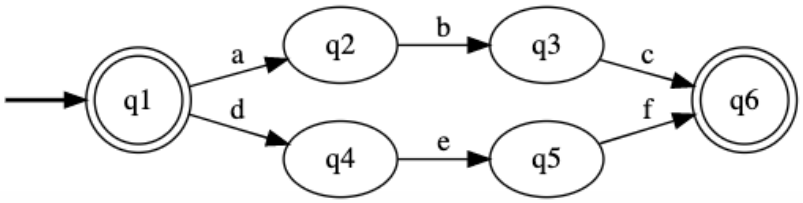

Figure 8: Picture of $A(OR(abc, def))$

Similarly, to build an automaton which recognizes the minimal semantics of an attack tree, we have to define a way of computing the automaton which accepts just the minimal words accepted by the other automaton. For doing that we present some new definitions.

Definition 25 (Complementary automaton) *The complementary of an automaton A with respect to an alphabet Σ is an automaton noted A^c which recognizes the set of all words in Σ^* that are not accepted by A.*

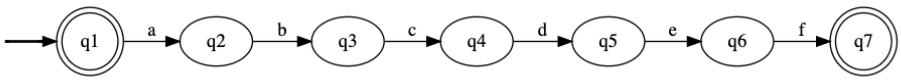

Figure 9: Picture of $A(SAND(abc, def))$

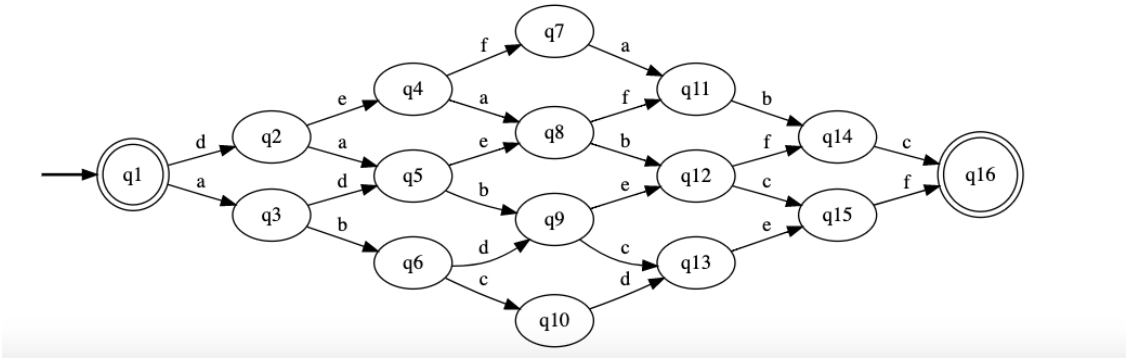

Figure 10: Picture of $A(AND(abc, def))$

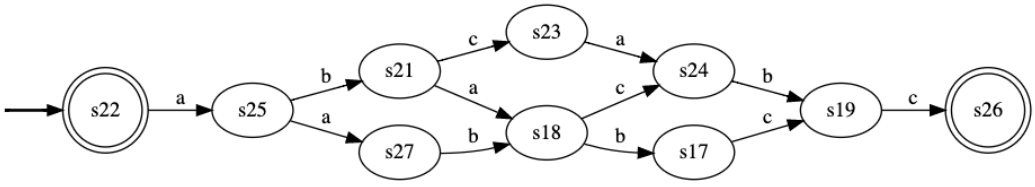

Figure 11: Picture of $A(AND(abc, abc))$

- *If* $A = (\Sigma, Q, \delta, i, F)$ *then* $A^c = (\Sigma^c, Q^c, \delta^c, i^c, F^c)$ *where:*
  - $\Sigma^c = \Sigma$;
  - $Q^c = Q$;
  - $\delta^c = \delta$;
  - $i^c = i$;
  - $F^c = Q - F$.

Definition 26 (Intersection of two automatons) *The intersection of two automatons* A_1 *and* A_1 *is an automaton noted* $A_1 \cap A_2$ *which recognizes the set of all words which are accepted both by* A_1 *and by* A_2.

- *If* $A_1 = (\Sigma_1, Q_1, \delta_1, i_1, F_1)$ *and* $A_2 = (\Sigma_2, Q_2, \delta_2, i_2, F_2)$, *then* $A_1 \cap A_2 = (\Sigma, Q, \delta, i, F)$ *where:*
  - $\Sigma = \Sigma_1 \cup \Sigma_2$;
  - $Q = Q_1 \times Q_2$;
  - $\delta = \{((q_{1a}, q_{2a}), \sigma, (q_{1b}, q_{2b})) \mid ((q_{1a}, q_{2a}), (q_{1b}, q_{2b})) \in Q \times Q \wedge (q_{1a}, \sigma, q_{1b}) \in \delta_1 \wedge (q_{2a}, \sigma, q_{2b}) \in \delta_2\}$;
  - $i = (i_1, i_2)$;
  - $F = F_1 \times F_2$.

Definition 27 (Superword automaton) *The superword automaton of an automaton* A *is an automaton noted* A^s *which recognizes the set of all words of* Σ^* *which have a subword accepted by* A.

- *If* $A = (\Sigma, Q, \delta, i, F)$ *then* $A^s = (\Sigma^s, Q^s, \delta^s, i^s, F^s)$ *where:*
  - $\Sigma^s = \Sigma$;
  - $Q^s = Q$;
  - $\delta^s = \delta \cup \{(q, \sigma, q) \mid \sigma \in \Sigma \wedge q \in Q\}$;
  - $i^s = i$;
  - $F^s = F$.

Note that the previous automaton does not recognize exactly Σ^*, because not every element of Σ^* is a superword of a word accepted by A.

Definition 28 (Difference of two automatons) *The difference of two automatons* A_1 *and* A_2 *is an automaton noted* $A_1 - A_2$ *which recognizes the set of all words which are accepted by* A_1 *and not by* A_2.

- $A_1 - A_2 = A_1 \cap A_2^c$

Definition 29 (Attack Tree Minimal Semantics $\longrightarrow$ Automaton)

- *If* θ *is a leaf which consists of a word* $S = \sigma_1 \bullet \ldots \bullet \sigma_n$ *(let* $\bullet$ *denotes concatenation), then its minimal semantics is recognized by the automaton* $A = (\Sigma, Q, \delta, i, F)$ *where :*
  - $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$;
  - $Q = \{q_1, \ldots, q_{n+1}\}$;
  - $\delta = \{(q_1, \sigma_1, q_2), \ldots, (q_n, \sigma_n, q_{n+1})\}$;
  - $i = \sigma_1$;
  - $F = \{q_{n+1}\}$.
- *If* θ *is an OR tree,* $\theta = OR(\theta_1, \theta_2)$, *where* θ_1 *and* θ_2 *are attack trees with minimal semantics recognized by finite automatons* A_1 *and* A_2, *then its minimal semantics is recognized by the automaton* $A = (A_1 - (A_2^s - A_2)) \cup (A_2 - (A_1^s - A_1))$.

- *If θ is a SAND tree, $\theta = SAND(\theta_1, \theta_2)$,*
  where θ_1 and θ_2 are attack trees with minimal semantics recognized by finite automatons A_1 and A_2, then its minimal semantics is recognized by the same automaton as for the standard semantics (see definition 24 and theorem 3).

- *If θ is an AND tree, $\theta = AND(\theta_1, \theta_2)$,*
  where θ_1 and θ_2 are attack trees with minimal semantics recognized by finite automatons A_1 and A_2, then its minimal semantics is recognized by the same automaton as for the standard semantics (see definition 24 and theorem 4).

We note $A_{min}(\theta)$ the automaton that recognizes the minimal semantics of θ.

The previous definition involves several operations, like complementation, that can be very costly. It can be shown that the fact that the (minimal) semantics of an attack tree θ is always a finite language guaranties better upper bounds for the construction of $A_{min}(\theta)$.

Theorem 5 (Complexity of Attack tree $\longrightarrow$ ϵ-free Non-deterministic automaton) *Let us calculate the complexity of each automaton construction with respect to tree constructors. To do that we associate to each automaton construction a vector which contains three elements: time complexity for constructing the automaton, state complexity of the constructed automaton and number of words accepted by the constructed automaton (i.e. size of the language). This vector is noted as (t, s, w). Let θ_1 and θ_2 be two trees with complexity vectors (t_1, s_1, w_1) and (t_2, s_2, w_2), then:*

- *If θ is a leaf which consists in a word of length n then its complexity vector is $(\Theta(n), \Theta(n), \Theta(1))$.*

  Indeed, its ϵ-free automaton (in fact, deterministic) is made of n states for reading the n letters of 1 accepted word.

- *If θ is an OR tree with children trees θ_1 and θ_2 then its complexity vector is $(\Theta(1), \Theta(s_1 + s_2), \Theta(w_1 + w_2))$.*

  Indeed, its ϵ-free automaton is made of $s_1 + s_2 - 1$ states, only 1 of which needs to be actually created, and the number of accepted words is $w_1 + w_2$.

- *If θ is a SAND tree with children θ_1 and θ_2 then its complexity vector is $(O(s_1), \Theta(s_1 + s_2), \Theta(w_1 * w_2))$.*

  Indeed, its ϵ-free automaton is made of $s_1 + s_2 - 1$ states, at most s_1 of which need to be actually created, and the number of accepted words is $w_1 * w_2$.

- *If θ is a AND tree with children θ_1 and θ_2 then its complexity vector is $(\Theta(s_1 * s_2), \Theta(s_1 * s_2), \Theta(\frac{(w_1+w_2)!}{w_1!w_2!}))$.*

  Indeed, its ϵ-free automaton is made of $s_1 * s_2$ states, which all need be actually created, and the number of accepted words is $\frac{(w_1+w_2)!}{w_1!w_2!}$.

  The insistence on ϵ-free automaton is crucial here because otherwise the time complexity of computing the automaton would not be so simply related with the state complexity of the result.

3.3 Shuffle decomposition

An interesting question is to decide whether a language is the shuffle of two other languages. Such a decomposition is not always possible, and it is not always possible to decide whether it is possible. However, they are some results which can be described.

Definition 30 (Shuffle decomposition) *A shuffle decomposition of a language L consists in two languages L_1 and L_2 such that $L = L_1 \sqcup\!\sqcup L_2$*

Definition 31 (Non-trivial shuffle decomposition) *A shuffle decomposition of a language L is non-trivial if none of the decomposed languages is $\{\lambda\}$ (λ is the empty word).*

Definition 32 (Shuffle decomposable) *A language L is shuffle decomposable if it exists a non-trivial shuffle decomposition of L.*

It is important that it is a non-trivial shuffle decomposition because otherwise there always exits a shuffle decomposition of L, indeed $L = L \sqcup\!\sqcup \{\lambda\}$

Theorem 6 (Shuffle decomposability of an infinite language) *If L is an infinite language, then the shuffle decomposability of L, i.e. the problem of deciding whether L is shuffle decomposable, is undecidable.*

Theorem 7 (Shuffle decomposability of a finite language) *If L is a finite language, then the shuffle decomposability of L is decidable.*

Definition 33 (Shuffle quotient) *Let L and L_1 be some language over an alphabet Σ. Then the shuffle quotient of L by L_1 is noted $L \bowtie L_1$ such that $L \bowtie L_1 = \{w \in \sigma^* \mid L_1 \sqcup\!\sqcup \{w\} \subseteq L\}$.*

Then we can described a way to decide a restricted version of the shuffle decomposability problem. This version is as follows:

- Let L be a language over alphabet Σ, let Σ_1 and Σ_2 form a partition of Σ. Can we find a language L_1 over Σ_1 and a language L_2 over Σ_2 such that $L = L_1 \sqcup\!\sqcup L_2$?

This problem is decidable and we are going to described how.

Theorem 8 (Restricted shuffle decomposition solution) *If $A = (\Sigma, Q, \delta, i, F)$ is a NFA which recognizes a language L then if it exists L_1 over Σ_1 and a language L_2 over Σ_2 such as $L = L_1 \sqcup\!\sqcup L_2$ and $\{\Sigma_1, \Sigma_1\}$ is a partition of Σ then:*

- *$A_1 = (\Sigma_1, Q_1, \delta_1, i_1, F_1)$ recognizes L_1 with :*
  - $\Sigma_1 = \Sigma$;
  - $Q_1 = Q$;
  - $\delta_1 = \{(q_1, \sigma, q_2) \mid (q_1, \sigma, q_2) \in \delta \wedge \sigma \notin \Sigma_2\} \cup \{(q_3, \epsilon, q_4) \mid (q_3, \sigma, q_4) \in \delta \wedge \sigma \in \Sigma_2\}$;
  - $i_1 = i$;
  - $F_1 = F$.
- *$A_2 = (\Sigma_2, Q_2, \delta_2, i_2, F_2)$ recognizes L_2 with :*
  - $\Sigma_2 = \Sigma$;
  - $Q_2 = Q$;
  - $\delta_2 = \{(q_1, \sigma, q_2) \mid (q_1, \sigma, q_2) \in \delta \wedge \sigma \notin \Sigma_1\} \cup \{(q_3, \epsilon, q_4) \mid (q_3, \sigma, q_4) \in \delta \wedge \sigma \in \Sigma_1\}$;
  - $i_2 = i$;
  - $F_2 = F$.

Then, a way for testing if L is shuffle decomposable on the partition $\{\Sigma_1, \Sigma_1\}$ is to generate A_1 and A_2 and test whether A and $A_1 \sqcup\!\sqcup A_2$ recognize the same language.

3.4 Conclusion on attack trees

We have explored both the extensional (a set of words) and the intensional (an automaton that recognizes the set of words) computation of the (minimal) semantics of attack trees. As the resulting semantics is always a finite set, a naive approach is to first compute extensionally the standard semantics, then optionally compute its minimal words, and then compute an automaton. However, examples show that short-circuiting the standard extensional semantics may be much more efficient.

First, we have shown how to compute the minimal semantics directly. We have shown that in the case of trees with only $SAND$ and AND nodes, it is always better than computing the standard semantics and minimizing it. The benefit can be arbitrarily large. However, we have also shown that the benefit can be arbitrarily small when there are OR nodes.

Second, we have shown how to directly compute the intensional (minimal) semantics as an automaton. Again, the computation of automatons for the minimal semantics is always better than the computation of automaton for the standard semantics in the case of trees with only $SAND$ and AND nodes. However, we have nearly no idea whether computing the extensional semantics is better or worse than computing the intensional semantics in general. It seems to us that answering this requires an empirical study on a benchmark of attack trees that are representative of real-life situations. We have found no empirical data to start such experiments.

All the proposed operations have been programmed and tested. Technically, we have used the `automata-lib` Python library for manipulating automatons, and we have interfaced it with a graph visualizer for displaying resulting automatons, using the DOT [4] language as an intermediate format.

4 Conclusion

We have studied the theory and usage of the answer set programming framework in its CLINGO incarnation. We appreciated its truly declarative semantics (truly declarative because its operational semantics matches it formal semantics). We observed that being declarative is not the last word because several equivalent declarative expressions may hide very different operational behaviours with huge complexity differences. Our conclusion after two months, is that if an evident algorithm solves a problem it is probably better to program it in a standard imperative way, as in [1], rather than trying to simulate a sequential behaviour using logical formulas. On the contrary, if no evident algorithms come to mind ASP helps formulate a solution very concisely. Fortunately, CLINGO is designed to be inter-operable with imperative languages, which permits to get the best of both worlds.

Concerning the (minimal) semantics of attack trees, it was a totally new subject for us, and we realize that we have an ever-growing number of questions about them, but also about the plan of the team regarding their usage. The different results we present in this report where discoveries for us, and we where quite happy to find them and prove them, like the direct computation of the minimal semantics or of its automaton, but in retrospect we do not know whether these results are really original or not, neither whether these results are really useful for the team or not.

We have adopted a constructive approach in the sense that all definitions have been implemented and tested. We where really happy with the connection of library `automata-lib` with a graph visualizer, via the DOT format. We even believe that an unexpected benefit of this study could be to design an automaton playground using similar technologies to teach automaton operations. Indeed, the effects of minimization, determinization, elimination of ϵ-transitions, or various kinds of binary operations, become readily visible, as well as limit cases, and subtle conditions in algorithms.

We wish to thank the LogicA team for its kind reception, especially considering the difficulties related to the sanitary situation. We specially thank the team leader, Sophie Pinchinat, for offering us this intership, and for her precious advice during this work. We also wish to thank Sophie for all her efforts to maintain a team work while everybody was working from home.

This intership was a first step of our project to enter a research career in computer science, and it confirmed us in this project. We discovered domains about which we did not know anything before this internship, like attack trees, we also discovered a more demanding attitude towards knowledge that we did not experienced so strongly, even in maths courses.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[2] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016.

[3] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018.

[4] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 19(3):214–230, 1993.

[5] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[6] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[7] David Pearce. A new logical characterisation of stable models and answer sets. In Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming, NMELP '96, Bad Honnef, Germany, September 5-6, 1996, Selected Papers*, volume 1216 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 1996.

[8] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[9] Wojciech Widel, Maxime Audinot, Barbara Fila, and Sophie Pinchinat. Beyond 2014: Formal Methods for Attack Tree-based Security Modeling. *ACM Computing Surveys*, 52(4):1–36, September 2019.