MASTER IN COMPUTER SCIENCE

Computer-Aided Reasoning for Product-Line Model Checking

Dawagne, Bruno

Award date:
2014

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Computer-Aided Reasoning for Product-Line Model Checking

Bruno Dawagne

Maître de stage : Vijay Ganesh

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Patrick Heymans

Co-promoteur : Maxime Cordy

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Acknowledgements

I would like to thank Prof. Patrick Heymans for all the opportunities he gave me. I would also like to thank Maxime Cordy for all his support and the time he took helping me through the redaction of this master thesis. Finally, I would like to thank Prof. Vijay Ganesh for its guidance and valuable advice.

Contents

Introduction

A software system is considered correct if it meets its design requirements. In the case of a distributed software, made of several processes working together, it is not always easy, or even practically possible, to verify that the system matches its functional requirements. The most common approach to ensure its correctness is testing. Tests can be realized on several abstraction levels: Unit tests verify the correctness of individual modules, integration tests check their proper cooperation and acceptance tests validate the functionalities of the complete system with respect to the end user's expectations. However, even with an high test coverage, some bugs can survive. Errors like race conditions and deadlocks are sometime hard to detect. They might for instance be triggered due to the scheduling policy of the running operating system. In this case, they might not even be reproducible on another platform.

Model checking is a formal verification technique that can, in some way, prove the absence of mistakes. It requires two inputs: A behavioral specification of the software system and a property in temporal logic. In return, a model checker yields an example of execution of the system that violates the property, or the certainty that no such execution exists. As one may expect, modeling a distributed system is a critical step. The behavioral model should indeed include all the information needed to decide on its correctness, as the communication sequences between processes, and avoid computation details, as the complete processing of the exchanged messages.

Model-checking tools have been applied with success in thousands of projects [Hol04]. For instance, they can prove the validity of mutual exclusion algorithms, such as Peterson's Algorithm. Before the appearance of model checking, proving or disproving such algorithm could lead to long and ponderous hand-written proofs. Now, concrete models of these algorithms can be checked against several temporal properties. This set of properties can include the effectiveness of mutual exclusion, i.e. that only one process at a time can enter a critical section, the absence of deadlocks or the fairness between processes induced by the verified algorithm.

A limitation of model checking is that it only allows one to verify the correctness of one single product. A modern approach when building software systems is however to create a set of strongly correlated products called a product line. Variants of a product line share common pieces or behaviors, and are built from a common, managed set of assets. Reusing core assets from one product to another allows one to benefit from economies of scale when developing a large number of products. A software product line is a product line of software-intensive systems. The variability that can take place in a product line is usually expressed in terms of features. Simply said, a feature is an asset

that can be included in a product and be missing in another. It usually defines some set of additional functionalities that can be inserted to a specific product to enhance it, or retrieved from another for incompatibility, economical or even simplicity reasons. Developing a software product line is like developing a highly configurable software system in which features can be activated or deactivated depending on the target's needs or specificities. Relations between features, as dependencies or incompatibilities, are usually expressed in a feature diagram.

Verifying properties on a software product line requires to specify the behavior of every single product of the product line and to verify them individually. However, the number of variants in a product line tends to grow exponentially with the number of features. Therefore, verifying products individually may require a huge amount of time and be unfeasible in practice. To workaround this, researchers from the University of Namur have adapted model-checking algorithms to software product lines. The process requires three inputs: A behavioral specification of the software product line, the feature diagram representing the variability of this product line, and a temporal property. In return, it yields the set of products that violate the property or the certainty that no such product exists. It also delivers an example of execution violating the property for each concerned product. The proposed algorithms have been implemented in a tool, named ProVeLines.

The aforementioned algorithms work as follows. The state-space of the product line is explored starting an initial state and following a nested depth-first search. On the other hand, the set of products that are actually able to go through a particular path is represented using boolean formulas. In order to avoid the verification of non existing products, these boolean formulas are often checked for satisfiability.

The boolean satisfiability problem is arguably the most famous example of NP-complete problem. Different algorithms and techniques have been proposed to solve it. In ProVeLines, the retained approach is to represent boolean formulas using Binary Decision Diagrams (BDDs). However, ProVeLines has recently been extended to support, in addition to standard boolean features, two additional types that are intensively used in practice: multi-features, i.e. features that may appear several times in a given product, and numeric features. The former entails changes only to the semantics of feature diagrams. The latter, however, also raises the need for non boolean expressions. To this end, an SMT solver has been integrated into ProVeLines to replace the BDD implementation.

SMT solvers are somehow the result of centuries of history in automated reasoning. Philosophers like George Boole have for long tried to formalize the law of thought, some of them dreaming about a machine to think, able to compute the result of any problem, or to prove any theorem. Unfortunately, the first order logic has been proved to be undecidable. In this context, SMT solvers might be the most evolved tools that we can get. This kind of solvers work by considering some subsets of the first order logic, like linear arithmetics over the integers, called theories.

The integration of an SMT solver into ProVeLines revealed a large overhead in verification time compared to the BDD implementation. Verifying a temporal property, even on a fully boolean model, could indeed take until 96,144 % more time using the SMT solver [CSHL13]. However, the cause of this problem is unclear. The present thesis is an attempt to investigate and solve that problem.

In the Chapter 1, we expand on the boolean satisfiability problem. We in-

troduce several ways of representing boolean formulas and different algorithms that we can use to quickly solve this problem. In particular, we mention BDDs and SAT solvers, which are at the heart of SMT solvers. Chapter 2 introduces the notion of theories, gives some examples, and presents the core algorithms running SMT solvers. Chapter 3 briefly explain the theories and practices behind ProVeLines and the mentioned software product-line model-checking algorithms. Among others, the link between these algorithms and the satisfiability problem is clarified. Chapter 4 presents a theoretical approach to efficiently integrate a SAT solver into ProVeLines. This is seen as an intermediary step between a BDD-based and an SMT-based implementation. As we shall see, this allows us to define new model-checking algorithms aimed at better integrate themselves with SAT and SMT solvers. In Chapter 5, we undertake the implementation of this theoretical approach and a concrete empirical study.

Part I

Theoretical Background

Chapter 1

The Satisfiability Problem

The boolean satisfiability problem, often denoted SAT, was the first known example of NP-complete decision problem. Other NP-complete problems have been proved to be by direct or indirect reduction to this first one. Understanding the satisfiability problem is therefore of paramount importance in the field of computational complexity. It is also the starting point of this thesis.

Due to its theoretical hardness, people sometime considered pointless to look for an efficient algorithm to solve it. However, different breakthroughs have, in the late 90's, renewed a common interest in solving the SAT problem on large formulas. This chapter aims at introducing this problem and several techniques to solve it, including the core algorithm of these so-called modern SAT solvers.

1.1 Boolean Functions and Satisfiability Problem

This section introduces the basic definitions needed in this chapter. These definitions are adapted from [SS96] and [MMZ+01].

Let $X = \{x_i\}_{i=1}^{n}$ be a set of $n \in \mathbb{N}$ boolean variables. A *boolean function* is a function of the form $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{\bot, \top\}$ is the *boolean domain*. It can be represented by a *well-formed formula*. Any variable $x \in X$, as well as both constants $\bot$ and $\top$ are well-formed formulas. Moreover, if F and G are both well-formed formulas, their negations $\neg F$ and $\neg G$, their conjunction $F \wedge G$, their disjunction $F \vee G$, their implication $F \Rightarrow G$ and their equivalence $F \Leftrightarrow G$ are all well-formed formulas. Henceforth, we simply call *formula* a well-formed formula.

Let $Y = \{y_i\}_{i=1}^{m}$ be a set of $m \in \mathbb{N}$ boolean variables. A *truth assignment* A over Y is a mapping from the variables of Y to values from the boolean domain. It can be equally represented by a function $A : Y \to \mathbb{B}$ or by a set $A = \{y \to v_y\}_{y \in Y}$, where each variable $y \in Y$ is mapped to the value $v_y \in \mathbb{B}$. Let F be a formula over a set of variable X. The assignment A is *complete* with respect to F if $X \subseteq Y$. It is *partial* otherwise.

Under a complete truth assignment, any formula F evaluates to either $\bot$ or $\top$. The evaluation of F under A, denoted $[A]F$, is determined according to the

following rules:

$$
\begin{array}{lll}
[A]\top & = \top & \\
[A]\bot & = \bot & \\
[A]x & = \top & \text{if, and only if,} \quad A(x) = \top \\
[A]\neg F & = \top & \text{if, and only if,} \quad [A]F = \bot \\
[A](F \wedge G) & = \top & \text{if, and only if,} \quad [A]F = \top \text{ and } [A]G = \top \\
[A](F \vee G) & = \top & \text{if, and only if,} \quad [A]F = \top \text{ or } [A]G = \top \\
[A](F \Rightarrow G) & = \top & \text{if, and only if,} \quad [A]F = \top \text{ implies } [A]G = \top \\
[A](F \Leftrightarrow G) & = \top & \text{if, and only if,} \quad [A]F = [A]G
\end{array}
$$

If $[A]F = \top$, A is called a *satisfying assignment* for F, and F is said to be *satisfied* by A. If $[A]F = \bot$, A is called an *unsatisfying assignment* for F, and F is said to be *unsatisfied* by A. A formula F is said to be *satisfiable* if there exists a satisfying assignment for it. It is said to be *unsatisfiable* otherwise. F is also said to be *valid*, or *tautological*, if it is satisfied by any complete assignment. It follows that a valid formula is always satisfiable.

The *satisfiability problem*, denoted SAT, is concerned with finding a satisfying assignment for a formula, or with proving that this formula is unsatisfiable. This problem has been proved to be NP-complete by Cook [Coo71].

The *validity problem* is concerned with proving that a formula is valid, or with finding an unsatisfying assignment for it. The validity problem is dual with the satisfiability problem, as a formula F is satisfiable if, and only if, $\neg F$ is not valid. Moreover, a satisfying assignment for the former is an unsatisfying one for the latter.

1.2 Normal forms

A boolean function can be represented by an infinite number of formulas. Among those formulas, however, several subsets can be defined to enclose all those that share a common syntactical form. These formulas are said to be in a *normal form*. Keeping formulas in a normal form facilitates their manipulation and allows one to easily determine some of their properties, as satisfiability and validity. In this section, three kinds of normal forms are discussed: Negation Normal Form, Conjunctive Normal Form and Disjunctive Normal Form.

1.2.1 Negation Normal Form

A formula is in *Negation Normal Form*, or *NNF*, if any negation operator $\neg$ appears before a single variable. A *literal* is a boolean variable or the negation of one. Any literal is a formula in NNF. Moreover, if F and G are both formulas in NNF, their conjunction $F \wedge G$, their disjunction $F \vee G$, their implication $F \Rightarrow G$ and their equivalence $F \Leftrightarrow G$ are all formulas in NNF. Their negations $\neg F$ and $\neg G$, however, are not necessarily in NNF.

Any formula can easily be transformed in NNF by pushing the negation

symbols toward the variables, using the following set of transformation rules:

$$\neg(\bigwedge_{i=1}^{n} F_i) \longrightarrow \bigvee_{i=1}^{n} \neg F_i$$
$$\neg(\bigvee_{i=1}^{n} F_i) \longrightarrow \bigwedge_{i=1}^{n} \neg F_i$$
$$\neg\neg F \longrightarrow F$$

In these rules, the symbol $\longrightarrow$ should be read as "is transformed in". The equivalence between the left hand side and right hand side of each of these rules relies on the validity of either one of the De Morgan's laws or the double negation rule.

$$\neg(\bigwedge_{i=1}^{n} F_i) = \bigvee_{i=1}^{n} \neg F_i$$
$$\neg(\bigvee_{i=1}^{n} F_i) = \bigwedge_{i=1}^{n} \neg F_i \qquad \text{(De Morgan's laws)}$$
$$\neg\neg F = F \qquad \text{(Double negation rule)}$$

We will see that formulas in Disjunctive Normal Form or Conjunctive Normal Form are also, by definition, in Negation Normal Form. This form can consequently be seen as less restrictive than, or a generalization of, the two hereafter mentioned normal forms.

Example 1.2.1. The formula $F = \neg(\neg a \vee (a \wedge \neg b))$ is not in Negation Normal Form, as the first negation symbol applies to the whole remaining formula. The formula $G = a \wedge (\neg a \vee b)$, however, is in NNF. G is actually obtained from F by iterated application of the mentioned transformation rules. This process can be observed in the following series of steps:

$$\begin{aligned}
&\neg(\neg a \vee (a \wedge \neg b)) \\
&\longrightarrow (\neg\neg a \wedge \neg(a \wedge \neg b)) \\
&\longrightarrow (\neg\neg a \wedge (\neg a \vee \neg\neg b)) \\
&\longrightarrow (a \wedge (\neg a \vee b))
\end{aligned}$$

The two first transformation steps rely on the De Morgan's laws while the last step is obtained by application of the double negation rule.

1.2.2 Conjunctive Normal Form

A formula is in *Conjunctive Normal Form*, or *CNF*, if it consists of a conjunction of clauses. A *clause* is a disjunction of literals. Any formula F in CNF is therefore of the form

$$F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{i,j}$$

where $\forall i \in \{1, 2, \ldots n\} : \forall j \in \{1, 2, \ldots m_i\} : l_{i,j}$ is a literal. As stated before, a formula in CNF is also in NNF. The negation operator $\neg$ indeed appears only in literals.

A formula in CNF is *canonical* if each of its clauses contains at most one occurrence of a variable. Henceforth, we consider that any formula in CNF is actually canonical. This goes without loss of generality given that

1. If a literal l appears several times in a clause, only one of these appearances can be kept.

2. If, for a variable x, both literals x and $\neg x$ appear in a clause, then this clause is satisfied by any complete assignment and can be removed from the original CNF.

This hypothesis allows us to consider the following result. A formula in CNF is valid if, and only if, it is reduced to the empty conjunction of clauses. Indeed, let us consider a formula F in CNF with a clause C such that

$$C = \bigvee_{i=1}^{m} l_i$$

Then any unsatisfying assignment for C is also an unsatisfying assignment for F. Such assignment is easy to find. It only need to include the set $\{x_i \to v_i\}_{i=1}^{n}$, where for any $i \in \{1, \dots n\}$, x_i is the variable of the literal l_i and v_i the boolean value that, assigned to this variable, makes this literal being evaluated to $\bot$, i.e.

$$v_i = \begin{cases} \bot & \text{if} \quad l_i = x_i \\ \top & \text{if} \quad l_i = \neg x_i \end{cases}$$

A formula in CNF is often called, throughout this thesis, a *clause database*.

Example 1.2.2. Let us consider the formula $F = (\neg a \vee c) \wedge (\neg b \vee \neg c \vee d) \wedge (\neg d \vee f) \wedge (\neg d \vee g) \wedge (\neg f \vee \neg g)$. This formula is obviously in Conjunctive Normal Form. Moreover, it is canonical. As it is not reduced to the empty conjunction of clauses, i.e. it contains at least one clause, it is not valid. The first clause $\neg a \vee c$ is indeed unsatisfied by the assignment $\{a \to \top, c \to \bot\}$. Any complete superset of it is therefore an unsatisfying assignment for F.

1.2.3 Disjunctive Normal Form

A formula is in *Disjunctive Normal Form*, or *DNF*, if it consists of a disjunction of cubes. A *cube* is a conjunction of literals. Any formula F in DNF is therefore of the form

$$F = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{i,j}$$

where $\forall i \in \{1, 2, \dots n\} : \forall j \in \{1, 2, \dots m_i\} : l_{i,j}$ is a literal. As stated before, a formula in DNF is also in NNF.

A formula in DNF is *canonical* if each of its cubes contains at most one occurrence of a variable. Henceforth, we consider that any formula in CNF is actually canonical. As for formulas in CNF, this goes without loss of generality. Considering this hypothesis, a formula in CNF is unsatisfiable if, and only if, it is reduced to an empty disjunction of cubes.

Conjunctive Normal Form and Disjunctive Normal Form are dual. Indeed, negating a formula in CNF and transforming it in NNF by iterated application

of the rules mentioned in Section 1.2.1 yields a formula in DNF. Moreover, the validity problem is easy to solve on CNF while the satisfiability problem is easy on DNF. The duality between these two normals forms is therefore correlated to the duality between these two problems. The actual hard remaining problem is to solve the satisfiability problem over formulas in CNF, or in an equivalent way, the validity problem over formulas in DNF. This challenge will be tackled in the next section.

1.3 Satisfiability over CNF

The satisfiability problem over formulas in CNF is tackled by application programs often called *SAT solvers*. As we shall see, these tools have known major improvements during the last decades. This section first introduces the classical DPLL alogrithm before presenting the breakthroughs of the CDCL algorithm, which is the core algorithm of most modern SAT solvers.

1.3.1 The DPLL Algorithm

The satisfiability problem over a clause database can be solved using a backtracking search algorithm. This was first published by Davis and Putnam as part of a solver for the first order logic [DP60] [DLL62]. It is now known as the *Davis-Putnam-Logemann-Loveland*, or *DPLL*, algorithm.

This algorithm maintains a truth assignment and tries to expand it until reaching a complete and satisfying assignment for a clause database. During this process, conflicts can arise that will force it backtrack. The algorithm actually iterates through different sub-processes.

The *decision process* first extends the current truth assignment by making a *decision assignment*, i.e. by selecting a free variable and a value for it. At this stage, a *decision tree* is maintained. Each node in the decision tree specifies a decision assignment. The *decision level* of a decision assignment is defined as its depth in the decision tree. If no free variable remains, then the current assignment is complete and satisfies the clause database. An example of decision tree is shown in Figure 1.1.

Once a decision has been made, the *deduction process* extends the current truth assignment by inferring some logical implications of this decision with respect to the clause database. This inference mechanism relies on the presence of *asserting clauses*. A clause is said to be asserting if all of its literals are assigned to $\bot$, except one. During the deduction process, the last free literal of each asserting clause is assigned to $\top$. This principle is called *Basic Constraint Propagation*, or *BCP*. It is iterated until no asserting clause remains or until a conflict arises. A conflict occurs when a clause is unsatisfied by BCP, i.e. when all of its literals are propagated to $\bot$.

Example 1.3.1. Let us consider the clause database $F = a \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$. Under the empty truth assignment, the first clause a is asserting and the assignment $a \to \top$ can be derived by BCP. The two other clauses then become asserting. From the second clause, $\neg a \vee b$, the assignment $b \to \top$ is propagated. However, this leads to a conflict as all the literals of the third clause are afterward assigned to $\bot$.

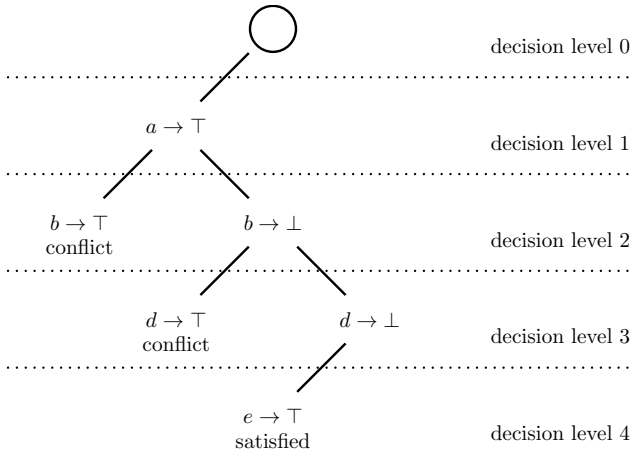


Figure 1.1: Decision tree created by the DPLL algorithm in the example 1.3.2

The additional variable assignments derived during the deduction process are called *deduction assignments*. The decision level associated with a deduction assignment is defined as the decision level of the decision assignment that has led to it.

If a conflict occurred during the deduction process, the *backtracking process* undoes all the assignments made at the current decision level and go back to the previous level in order for the opposite assignment to be tried. If it is not possible to backtrack any further, then the clause database is not satisfiable.

The overall process is formalized in Algorithm 1. This procedure determines whether a formula F in CNF is satisfiable or not. The procedure *propagate* first applies BCP to the partial assignment A, which is initially empty, with respect with the clause database F. If a conflict occurred during BCP, the process tries to backtrack. The procedure *backtrack* undoes all variable assignments made at the current decision level and returns the last decision assignment, so that it can be reverted. If it is not possible to backtrack any further, the process ends, returning false. If no conflict occurred, and if A is a complete and satisfying assignment for F, the process stops, returning true. Finally, if no conflict occurred and if A is not complete with respect to F, a decision is made. The function *decide* returns an arbitrary variable assignment $x \to v$ where x is a free variable of F and $v \in \mathbb{B}$. The process then restarts from the top of the algorithm.

Example 1.3.2. Let us suppose that a satisfying assignment need to be found for the clause database $F = (\neg a \vee c) \wedge (\neg b \vee \neg c \vee d) \wedge (\neg d \vee e) \wedge (\neg d \vee f) \wedge (\neg e \vee \neg f)$. Let us also suppose that the decision process picks the variables alphabetically and always tries to assign $\top$ first.

The decision tree produced by this scenario is shown in Figure 1.1. The algorithm first decides the assignment of a to $\top$ and derives, as the first clause $\neg a \vee c$ becomes asserting, $c \to \top$. It then decides the value of b as being $\top$, propagates the assignment $d \to \top$ due to the second clause, and runs into a conflict, as e and f are propagated to $\top$ by the third and fourth clauses, leading the last clause $\neg e \vee \neg f$ to be unsatisfied. This last decision is consequently reverted. The process decides next to assign d to $\top$, and runs into the exact

Algorithm 1 The DPLL Algorithm

```
boolean DPLL(cnf F)
  Integer decisionLevel ← 0
  Assignment A ← ∅
  loop
    propagate(A, F)
    if a conflict occurred then
      if decisionLevel = 0 then
        return false
      else
        (x → v) ← backtrack(A, decisionLevel)
        decisionLevel ← decisionLevel − 1
        A ← A ∪ {x → ¬v}
      end if
    else if A is complete then
      return true
    else
      (x → v) ← decide(A, F)
      decisionLevel ← decisionLevel + 1
      A ← A ∪ {x → v}
    end if
  end loop
```

same conflict, i.e. both e and f are propagated to $\top$. This decision is therefore also reverted. The last decision taken by the process, $e \to \top$, leads to a complete solution by propagating f to $\bot$. The satisfying assignment found is $\{a \to \top, b \to \bot, c \to \top, d \to \bot, e \to \top, f \to \bot\}$.

This algorithm actually falls twice into the same trap, as both conflicts occurring during this run have the same root cause: d cannot, and should not, be assigned to $\top$. As we shall see, part of the power of modern solvers relies on their ability to learn from that kind of mistakes.

1.3.2 The CDCL Algorithm

Over the past twenty years, several improvements have been made on top of the DPLL algorithm. The most noticeable are clause learning and backjumping [SS96], efficient propagation by use of lazy data structures and activity-based heuristics [MMZ+01]. They result in what is commonly named the *Conflict-Driven Clause Learning*, or *CDCL*, algorithm, which we present in this section.

Clause learning and backjumping

Clause learning and backjumping have first been introduced by Silva and Sakallah with the GRASP (Generic seaRch Algorithm for the Satisfiability Problem) solver [SS96]. Both mechanisms make use of a specific data structure maintained during the deduction process called the *implication graph*.

Let the deduction assignment $x \to v_x$ be the result of an asserting clause C. The antecedent assignments of x, denoted $A(x)$, are defined as the set of assignments over variables of C that have led C to become asserting. The

implication graph is a Directed Acyclic Graph (DAG) such that each vertex is either a variable assignment or a special conflict vertex, denoted κ. The set of predecessors of an assignment vertex $y \to v_y$ correspond to the antecedent assignments $A(y)$. An assignment vertex without any predecessor therefore corresponds to a decision assignment. A special conflict vertex κ is introduced when a conflict occurs. Its predecessors are all the assignments over variables of a clause which have led this clause to be unsatisfied. The implication graph created from the first conflict occurring in example 1.3.2 is presented in Figure 1.2.

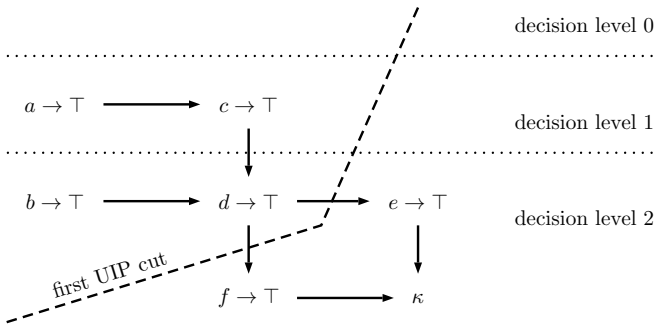


Figure 1.2: Implication graph produced by the first conflict occurring in example 1.3.2

When a conflict arises, a so-called *conflict clause* is generated by the *analysis process* and added to the clause database. The analysis process creates a partition of all the nodes in the implication graph by cutting it in half, putting all the decision assignments on one side, called the *reason side*, and the conflict vertex κ on the other side, called the *conflict side*. The conflict clause is then generated by taking all the assignments in the reason side that have at least one outgoing edge targeting an element of the conflict side. Let A_c be this set of assignments. The conflict clause C_c is then defined as

$$C_c = \bigvee_{(x \to v) \in A_c} l_x$$

where

$$l_x = \begin{cases} x & \text{if} \quad v = \bot \\ \neg x & \text{if} \quad v = \top \end{cases}$$

Several learning schemes have been introduced in the literature, each one characterized by its way of creating the aforementioned partition. In the *First Unique Implication Point*, or *FUIP*, scheme [SS96], the reason side is the largest set such that only one assignment made at the highest decision level in which a variable of the unsatisfied clause has been assigned has an outgoing edge targeting an element of the conflict side. This scheme is by far the most widely used and has empirically proved its efficiency [ZMMM01]. This clause can be further strengthened by a minimizing process [SB09], which we do not detail here.

Example 1.3.3. The implication graph created from the first conflict occurring in Example 1.3.2 is presented in Figure 1.2. It is partitioned according to the

FUIP scheme. One can see that the two edges crossing the partition line actually come from the same assignment $d \to \top$. The conflict clause generated from this graph by the mentioned scheme is therefore composed of the only literal $\neg d$. Such clause, containing only one literal, is called a *unit clause*. It has the particularity to be always asserting, even under the empty truth assignment.

A conflict clause is always a logical implication of the original clause database and, consequently, its addition does not modify the set of satisfying assignment of the original formula. As we shall see, however, conflict clauses facilitates the deduction process by improving the deductive power of the overall formula.

Let F_1 and F_2 be two equivalent set of clauses. We say that F_1 has a deductive power greater than or equal to F_2 relative to BCP if and only if [HS07]

1. For any variable assignment A that leads to a conflict in F_2, A also leads to a conflict in F_1.

2. For any variable assignment A that does not lead to a conflict in F_2 and any literal l such that l is derivable by BCP from A and F_2, l is also derivable from A and F_1.

Let suppose that a conflict occurred due to an unsatisfied clause C_u. Let us note L the highest decision level in which a variable of C_u has been assigned and C_c the conflict clause produced by the analysis process using the FUIP scheme. By construction, all the literals of C_c are assigned to $\bot$, exactly one of them at the decision level L, all the others at decision levels lower that L. The *backjumping process* therefore undoes all variable assignments made at decision levels higher or equal to L and the conflict clause is added to the clause database. At this stage, this new clause becomes asserting. Its addition to the clause database consequently increases the deductive power of the original formula by allowing BCP to produce an additional deduction assignment. For this reason, conflict clauses are also characterized as *1-empowering* [PD11]. It is noteworthy that L could not correspond to the highest decision level reached by the process. This case can only take place when the conflict in question is produced as a direct consequence of the addition of a clause due to a previous conflict. In this situation, the backjumping process actually undoes several decision levels, pruning an entire area of the decision tree.

Example 1.3.4. Let us go back to the conflict clause generated in Example 1.3.3. Once the clause $\neg d$ has been produced by the analysis process, the backjumping process undoes all the assignments made at decision level 2 and the conflict clause is added to the clause database. The deduction assignments $\neg d$ and $\neg b$ are consequently derived by BCP. From that state, any decision would lead the algorithm to a complete and satisfying assignment.

The resulting CDCL algorithm is presented in Algorithm 2. This procedure is very close to the one presented in Algorithm 1. We can however observe the appearance of the *analyze* function, producing a new clause when a conflict occurs. The *highestLevel* function then determines the highest decision level in which a variable of this clause has been assigned, and the *backjump* function undoes all the assignments made at decision levels higher or equal to this decision level. The process stops, returning $false$, when an empty clause is produced by the analysis process.

Algorithm 2 The CDCL Algorithm

```
boolean CDCL(cnf F)
  Integer decisionLevel ← 0
  Assignment A ← ∅
  ImplicationGraph I
  loop
    propagate(A, F, I)
    if a conflict occurred then
      Clause C ← analyze(I)
      if C is empty then
        return false
      end if
      decisionLevel ← highestLevel(C)
      backjump(decisionLevel, A, I)
    else if A is complete then
      return true
    else
      (x → v) ← decide(A, F)
      decisionLevel ← decisionLevel + 1
      A ← A ∪ {x → v}
    end if
  end loop
```

This algorithm omits a large number of details concerning the building blocks of modern SAT solvers. The following sections introduces some techniques and heuristics widely used in this area.

Clause Removal Policy

The clause learning mechanism can lead to an exponential growth of the clause database. Indeed, a modern SAT solver can produce several thousands of conflict clauses per second, and it may run for hours. That kind of situation can lead to a huge number of learned clauses that can actually slow down the propagation process. Hence, the implementers often face a trade-off between clause learning empowering and fastness of BCP. Several strategies can be considered, two of the most common being

1. Conflict clauses of length greater than a chosen integer k are only kept around and used for backjumping. Consequently, they are deleted as soon as they contain more than one free literal. Using this, the worst-case growth become polynomial in the number of variables in the clause database. This strategy was used by the GRASP solver [SS96].

2. An activity score is associated to each learned clause. The score of a clause is incremented when it is used during the analysis process. Inactive clause are then periodically removed. This strategy is used in the Minisat SAT solver [ES03a].

Activity Heuristic

The choice of assignments to make during the decision process is of paramount importance. Some decisions may indeed lead the algorithm to a straightforward solution while others lead to a long series of conflicts and backtracking. The decision process, however, can only rely on heuristics. A good decision heuristic should ideally find a good balance between the search of an optimal choice and a short execution time. In that matter, the *Variable State Independent Decaying Sum*, or *VSIDS*, heuristic appears as an efficient solution [MMZ$^+$01].

This mechanism can be described as follows. Each literal is associated with an activity score, initialized to 0. When a conflict clause is added to the clause database, the score associated with each literal of the clause is incremented. Periodically, all scores are divided by a constant. During the decision process, the unassigned literal with the highest score is assigned to $\top$. Ties are broken randomly.

The actual implementation of this strategy can widely vary from one solver to another. In [ES03a], for instance, track of a score is not kept for each literal, but only for each variable. The choice of the assigned value is then purely arbitrary. We have here described the heuristic as it was first published by Moskewicz et al. Moreover, even if this mechanism has empirically proved its efficiency, there is still no theoretical evidence of it. An intuitive explanation is that the VSIDS heuristic forces the solver to decide on conflicting variables, keeping hammering them until finding the appropriate assignments.

Watched Literals Scheme

During the deduction process, it is necessary to identify all the asserting clauses of a clause database. Giving the size that can be reached by such a formula, a simple traversal of it, scanning all clauses, could only deliver a poor efficiency. In that context, the *watched literals scheme* [MMZ$^+$01] has appeared as a quick mechanism to identify all new asserting clauses. The idea is to pick for every clause in the database an arbitrary couple of literals of it and to consider them as being *watched*. While none of these two literals are assigned to $\bot$, the associated clause is guaranteed not to become asserting. However, when one of the two watched literals is assigned to $\bot$, one of the three following scenarios must occur:

1. The clause is already satisfied, meaning that at least one of its literals is assigned to $\top$. The clause can therefore not become asserting.
2. The clause is neither satisfied nor asserting. Consequently, it still contains at least two unassigned literals, one of them being unwatched. This latter one can be used to replace the newly assigned literal as watched literal.
3. The clause is asserting and should be propagated.

The use of the watched literals scheme can greatly diminish the propagation time. Moreover, at the time of backtracking, there is no need for the search process to modify or replace the watched literals. Consequently, reassigning a variable that has recently been unassigned will tend to be fast. This can represent a significant gain of time when conflicts are likely to arise.

Example 1.3.5. Let C be a clause such that $C = a \vee \neg b \vee \neg c \vee d$. The first two literals are watched literals. When the variable c is assigned to $\top$, the clause is

not even scanned, as c is not watched. However, if a is additionally assigned to $\bot$, we fall in the second scenario, and a new watched literal has to be selected. The literal d is picked. When b is also assigned to $\top$, the third scenario occurs and d is propagated. This series of steps is illustrated in Figure 1.3.

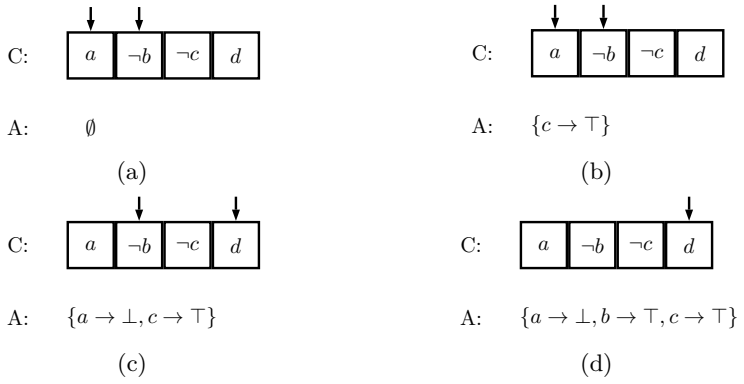


Figure 1.3: The watched literal scheme applied to the clause $a \vee \neg b \vee \neg c \vee d$

Restart Heuristic

Restarts are parts of another heuristic mechanism used with the CDCL algorithm. It was included in the GRASP [SS96] and Chaff [MMZ+01] solvers. The idea is to stop the current resolution process and restart it while keeping some information acquired during the previous run, such as learned clauses and activity scores. Consequently, the algorithm does not follow the same path, taking different decisions. Furthermore, one can add a certain amount of randomness in the decision process to help in the selection of distinct paths.

The insight behind this method is that the search process can be stuck in a unproductive area of the search space, where previous decisions inevitably lead to conflicts. Rebooting the overall process could then allow the solver to get out of this area. In order to maintain the completeness of the algorithm, the restart period is usually extended each time it occurs. The process could otherwise restart infinitely.

1.4 Binary Decision Diagrams

A *Binary Decision Diagram*, or *BDD*, is a representation for boolean functions first introduced by Bryant [Bry86] which can be used to solve the satisfiability problem on arbitrary formulas.

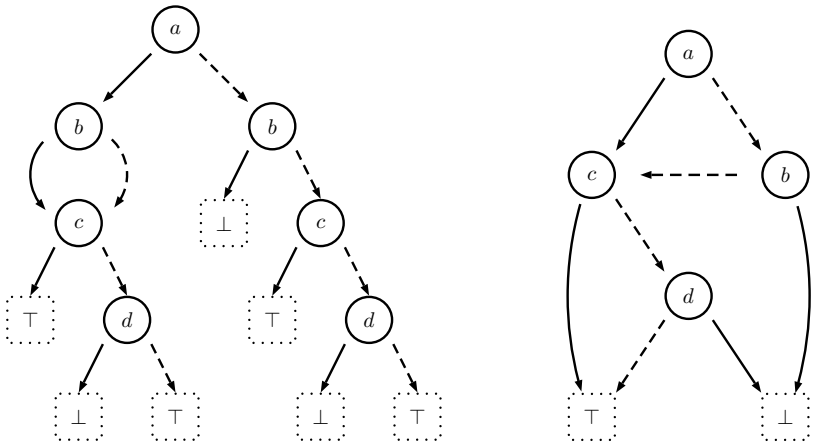
1.4.1 Principles and Canonicity

Let us first recall the definition of a strict total order. A *strict total order* $\prec$ over a set of elements X is a binary relation over this set which is

1. transitive: $\forall x, y, z \in X : (x \prec y) \wedge (y \prec z) \Rightarrow x \prec z$

2. trichotomous: $\forall x, y \in X$ exactly one of the three statements $x \prec y$, $y \prec x$ and $x = y$ is true

Lets $X = \{x_i\}_{i=1}^{n}$ be a set of boolean variables and $\prec$ a strict total order over that set. An *Ordered Binary Decision Diagram*, or *OBDD*, over $\langle X, \prec \rangle$ is defined as a Directed Acyclic Graph (DAG) with the following properties. Each terminal vertex is labeled with either $\top$ or $\bot$. Each non terminal vertex is labeled with a boolean variable $x \in X$ and has two outgoing edges. The target of the first edge is called the *high value* while the second target is called the *low value*. Moreover, if any of these targets is also a non terminal node, its label $y \in X$ must be such that $x \prec y$. Therefore, when following a path in the graph from the root to a leaf, variables must appear according to the order defined by $\prec$. Example of OBDDs are shown in Figure 1.4.



(a) OBDD (b) ROBDD

Figure 1.4: An unconstrained OBDD and an ROBDD, both representing the boolean formula $(\neg a \vee b) \wedge (\neg c \vee d)$

Each OBDD represents a boolean formula or, equivalently, a boolean function. The formula F encoded by an OBDD with root vertex v can be recursively described as follows:

1. If v is a terminal vertex, then
   (a) If v is labeled with $\bot$ then $F = \bot$
   (b) If v is labeled with $\top$ then $F = \top$
2. If v is a non terminal vertex labeled with the variable x then $F = (\neg x \wedge F_{low}) \vee (x \wedge F_{high})$, where F_{low} and F_{high} are respectively the formulas defined by the low and high values of v.

According to the Shanon's expansion law [Sha38], every boolean function can be represented by an OBDD. This law indeed states that every boolean formula F containing a boolean variable x can be rewritten as $(\neg x \wedge F_{\neg x}) \vee (x \wedge F_x)$ where $F_{\neg x}$ is the formula obtained by instanciating x to $\bot$ in F and F_x the formula obtained by instanciating x to $\top$ in F.

An OBDD can contain some redundancy, as several subgraphs could be *isomorphic*. Two OBDDs are isomorphic if there roots also are. Two vertices v_1 and v_2 are isomorphic if one of the following conditions is satisfied:

1. Both v_1 and v_2 are terminal vertices and are labeled with the same value, i.e. either $\bot$ or $\top$.

2. Both v_1 and v_2 are non terminal vertices and are labeled with the same variable. Their high values are isomorphic and so are their low values.

The isomorphism of non terminal nodes is thus a recursive property: it can only be determined on the basis of their children.

A *Reduced Ordered Binary Decision Diagrams*, or *ROBDD*, is an OBDD with the following properties:

1. Isomorphic but distinct sub-graphs are forbidden. This means that two isomorphic subgraphs must always have the same root and share the same data structure.

2. The high value of a non terminal vertex must be distinct from its low-value. This property forces to remove all irrelevant nodes from the diagram.

These two properties allow ROBDDs to be in a so-called *canonical form*. The canonicity property states that, given a strict total order on the set of variables, a boolean function can only be represented by one ROBDD. Seen another way, if a boolean function is represented by more than one ROBDD, all of them are isomorphic. This principle is of paramount importance as it allows one to rapidly identify several properties of the boolean function behind an ROBDD. In particular, it is noteworthy that

1. an unsatisfiyable boolean function can only be represented by an ROBDD consisting of only one terminal vertex labeled with the value $\bot$.

2. a valid boolean function can only be represented by an ROBDD consisting of only one terminal vertex labeled with the value $\top$.

3. all other ROBDD represent boolean function that are neither unsatisfiable nor valid.

In that last case, a satisfying assignment can easily be retrieved by following a path ending with a node labeled with $\top$. An ROBDD is commonly simply called a BDD. We also take this convention in the rest of this thesis.

Example 1.4.1. Let $\prec$ be a strict total order such that $a \prec b \prec c \prec d$. Figure 1.4 shows two representations of the boolean formula $(\neg a \vee b) \wedge (\neg c \vee d)$. In both sub-figures, the plain edges correspond to low values while dashed edges correspond to high values. Figure 1.4a represents an unconstrained OBDD over $\prec$, which contains several redundancies due to isomorphic sub-graphs. Figure 1.4b, on the other hand, does not. It actually depicts the only ROBDD over $\prec$ associated with this boolean function.

1.4.2 Operations on BDD's

When using BDDs, one must be able to manipulate them by performing boolean operations. The implementation of the negation is straightforward, as one only needs to negate the values labeling the terminal nodes of the original BDD. This simple manipulation is indeed sufficient to invert the sets of satisfying and unsatisfying assignments. We can show this by considering a proof by induction on the size of the BDD. For a BDD consisting of only one terminal node, negating the value labeling this node indeed results in negating the BDD. Let us now consider a BDD with a non terminal root vertex v. As mentioned before, the formula F represented by this BDD is defined as being $F = (\neg x \wedge F_{low}) \vee (x \wedge F_{high})$, where F_{low} and F_{high} are respectively the formulas defined by the low and high values of the vertex v. Let us suppose that negating the values labeling the terminal vertices of the low and high values of v is sufficient to negate the formulas that they encode. In this case, the formula represented by the BDD of root vertex v becomes

$$
\begin{aligned}
&(\neg x \wedge \neg F_{low}) \vee (x \wedge \neg F_{high}) \\
&= \neg(x \vee F_{low}) \vee \neg(\neg x \vee F_{high}) \\
&= \neg((x \vee F_{low}) \wedge (\neg x \vee F_{high})) \\
&= \neg((x \wedge F_{high}) \vee (\neg x \wedge F_{low}) \vee (F_{low} \wedge F_{high})) \\
&= \neg((x \wedge (F_{high} \vee (F_{high} \wedge F_{low})) \vee (\neg x \wedge (F_{low} \vee (F_{low} \wedge F_{high})))) \\
&= \neg((x \wedge F_{high}) \vee (\neg x \wedge F_{low})) \\
&= \neg F
\end{aligned}
$$

Binary operations are more difficult. Algorithm 3 returns the result of an arbitrary binary boolean operation $\circ$ on two BDDs of roots v_1 and v_2. Let x be a boolean variable. Then, for two formulas $F_1 = (\neg x \wedge F^1_{low}) \vee (x \wedge F^1_{high})$ and $F_2 = (\neg x \wedge F^2_{low}) \vee (x \wedge F^2_{high})$, the resulting formula $F_1 \circ F_2$ can be rewritten, by Shanon's expansion, as

$$F_1 \circ F_2 = (\neg x \wedge (F^1_{low} \circ F^2_{low}) \vee (x \wedge (F^1_{high} \circ F^2_{high}) \tag{1}$$

In the case of two formulas F_1 and F_2 such that $F_1 = (\neg x \wedge F^1_{low}) \vee (x \wedge F^1_{high})$ and that F_2 is free of the variable x, $F_1 \circ F_2$ can be be rewritten, again by Shanon's expansion, as

$$F_1 \circ F_2 = (\neg x \wedge (F^1_{low} \circ F_2)) \vee (x \wedge (F^1_{high} \circ F_2)) \tag{2}$$

In Algorithm 3, the *val* function returns the label of a terminal node, while the *var* function returns the variable labeling a non terminal vertex. The functions *low* and *high* respectively return the low and high values of a non terminal vertex. Finally, the *makeVertex(v)* function creates a vertex from a boolean value v while the *makeVertex(v, v_1, v_2)* builds a vertex labeled with the variable v and having v_1 and v_2 as high and low values. The idea of the algorithm is to recursively apply the two mentioned logical equivalences until reaching the terminal nodes of the BDDs. When v_1 and v_2 are both terminal vertices, the operation $\circ$ is simply applied on their labels. If v_1 and v_2 are both non terminal vertices labeled with the same variable, Equation 1 is applied. If v_1 and v_2 are such that $var(v_1) \prec var(v_2)$, or such that only v_2 is a terminal vertex, then the BDD of root v_2 is free of the variable $var(v_1)$, and Equation 2 is applied.

Algorithm 3 The Apply operation

vertex Apply(operation $\circ$, vertex v_1, vertex v_2)
  if v_1 and v_2 are both terminals **then**
    return makeVertex(val(v_1) $\circ$ val(v_2))
  else if v_2 is terminal **or** var(v_1) $\prec$ var(v_2) **then**
    return makeVertex(var(v_1), Apply($\circ$, high(v_1), v_2), Apply($\circ$, low(v_1), v_2))
  else if v_1 is terminal **or** var(v_2) $\prec$ var(v_1) **then**
    return makeVertex(var(v_2), Apply($\circ$, v_1, high(v_2)), Apply($\circ$, v_1, low(v_2)))
  else
    return makeVertex(var(v_1), Apply($\circ$, high(v_1), high(v_2)), Apply($\circ$, low(v_1), low(v_2)))
  end if

One should note that Algorithm 3 returns a BDD that should be reduced. However, we do not describe the reduction operation here.

Conclusion

In this chapter, we introduced the boolean satisfiability problem and several computing procedures that can be used to solve it. In particular, we expanded on SAT solvers and BDDs. These tools are at the heart of the challenge tackled by this master thesis. As we shall see, a deep understanding of their core algorithms is sometimes needed to use them efficiently.

Chapter 2

Satisfiability Modulo Theories

When dealing with complex constraints, the propositional logic can be limiting. For instance, expressing constraints over integer or rational numbers in propositional logic may require to use a binary encoding. Therefore, the use of a higher level language seems sometimes more natural.

Unfortunately, the first-order logic, i.e. the logic that includes predicate symbols, is undecidable. To workaround this limit, we can consider smaller sub-sets of it, called theories. The goal of this chapter is to introduce the basic concepts of the first-order logic, the notion of theory, and the common architectures of so-called SMT solvers.

2.1 First-Order Logic

This section introduces the syntax and semantics of the first-order logic, and expands on the satisfiability and validity problem. These definitions are adapted from [Gan13a].

2.1.1 Syntax And Semantics

A *first-order language* $\mathcal{L}$ is a triple $\langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ where $\mathcal{C}$ is a set of *constant symbols*, $\mathcal{F}$ a set of *function symbols* and $\mathcal{R}$ a set of *relation symbols*, also called *predicate symbols*. Each function symbol, as well as each predicate symbol, is associated with a natural number called its *arity*.

Let V be a set of variables. The set of *terms* $\mathcal{T}$ of a first-order language $\mathcal{L} = \langle \mathcal{C}, \mathcal{F}, \mathcal{R} \rangle$ is recursively defined as follows.

1. Any value from $\mathcal{C}$, as well as any variable in V, is a term.
2. If $\{t_i\}_{i=1}^n$ is a set of $n \in \mathbb{N}_0$ terms and if f is a function symbol from $\mathcal{F}$ with arity $n \in \mathbb{N}_0$, then $f(t_1, t_2, \dots t_n)$ is a term.

An *atomic formula*, or *atom*, over the first-order language $\mathcal{L}$ is an expression $p(t_1, t_2, \dots t_m)$ where p is a predicate symbol of arity $n \in \mathbb{N}_0$ and $\{t_i\}_{i=1}^n$ a set of terms. A *well-formed formula* of $\mathcal{L}$ is a boolean composition of atoms. Any

atom over $\mathcal{L}$ is a well-formed formula. Moreover, if F and G are both well-formed formulas, their negations $\neg F$ and $\neg G$, their conjunction $F \wedge G$, their disjunction $F \vee G$, their implication $F \Rightarrow G$ and their equivalence $F \Leftrightarrow G$ are also well-formed formulas. Finally, if F is a well-formed formula, its existential quantification over a variable $x \in V$, denoted $\exists x : F$, and its universal quantification over x, denoted $\forall x : F$, are both well-formed formulas. Henceforth, we simply call *formula* any well-formed formula.

In the formula $\exists x : F$, as well as in the formula $\forall x : F$, the sub-formula F is called the *scope* of the quantifier. An occurrence of a variable in a formula is *bound* if, and only if, it is in the scope of some quantifier over that variable. It is *free* otherwise. A formula with no free variable is said to be *closed*. A closed formula is also called a *sentence*. A formula with free variables is called *open*.

Example 2.1.1. Let p and q be two predicate symbols and x and y two variables. Then, the formula $\forall x : (p(x) \Rightarrow q(x))$ is a sentence. The variable x is always in the scope of the quantifier $\forall$. The formula $\forall x : (p(x) \Rightarrow q(y))$, on the other hand, is open, because the variable y is free.

The universe of discourse Ω is a set of objects of interest. It often refers to objects or concepts from the real world and is used to build an interpretation. An *interpretation* I for a first-order language $\mathcal{L} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ is a function defined over the set of symbols $\mathcal{C} \cup \mathcal{F} \cup \mathcal{R}$ such that

1. $\forall c \in \mathcal{C} : I(c) \in \Omega$
2. $\forall f \in \mathcal{F} : I(f) : \Omega^n \to \Omega$ where n is the arity of f
3. $\forall r \in \mathcal{R} : I(r) : \Omega^n \to \mathbb{B}$ where n is the arity of r and $\mathbb{B} = \{\bot, \top\}$ is the boolean domain.

A structure $S = \langle \Omega, I \rangle$ for a first-order language consists of a universe of discourse Ω paired with an interpretation I. An *variable assignment* A is a mapping from the variables of V to values from the universe of discourse Ω. It can equally be represented by a function $A : V \to \Omega$ or by a set $A = \{x \to v_x\}_{x \in V}$.

Under a structure $S = \langle \Omega, I \rangle$ and a variable assignment A, any term t of a first-order language $\mathcal{L} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ evaluates to a value from the universe of discourse Ω. The evaluation of t under S and A, denoted $\langle S, A \rangle(t)$, is determined according to the following rules:

1. $\forall c \in \mathcal{C} : \langle S, A \rangle(c) = I(c)$
2. $\forall x \in \mathcal{V} : \langle S, A \rangle(x) = A(x)$
3. $\forall f \in \mathcal{F}$ of arity $n \in \mathbb{N}_0$ and $\forall t_1, t_2, \dots t_n \in \mathcal{T}$:
   $\langle S, A \rangle(f(t_1, t_2, \dots t_n)) = I(f)(\langle S, A \rangle(t_1), \langle S, A \rangle(t_2), \dots \langle S, A \rangle(t_n))$

In a similar way, under a structure $S = \langle \Omega, I \rangle$ and a variable assignment A, any formula F evaluates to either $\bot$ or $\top$. The evaluation of F under S and A, denoted $\langle S, A \rangle(F)$, is performed according to the following rules:

$\forall r \in \mathcal{R}$ of arity $n \in \mathbb{N}_0$ and $\forall t_1, t_2, \dots t_n \in \mathcal{T}$:
$\langle S, A \rangle(r(t_1, t_2, \dots t_n)) = I(r)(\langle S, A \rangle(t_1), \langle S, A \rangle(t_2), \dots \langle S, A \rangle(t_n))$

$$
\begin{aligned}
\langle S, A\rangle(\neg F) &= \top \text{ if, and only if, } \langle S, A\rangle(F) = \bot \\
\langle S, A\rangle(F \wedge G) &= \top \text{ if, and only if, } \langle S, A\rangle(F) = \top \text{ and } \langle S, A\rangle(G) = \top \\
\langle S, A\rangle(F \vee G) &= \top \text{ if, and only if, } \langle S, A\rangle(F) = \top \text{ or } \langle S, A\rangle(G) = \top \\
\langle S, A\rangle(F \Rightarrow G) &= \top \text{ if, and only if, } \langle S, A\rangle(F) = \top \text{ implies } \langle S, A\rangle(G) = \top \\
\langle S, A\rangle(F \Leftrightarrow G) &= \top \text{ if, and only if, } \langle S, A\rangle(F) = \langle S, A\rangle(G) \\
\langle S, A\rangle(\exists x : F) &= \top \text{ if, and only if, there exists a value } v \in \Omega \text{ such that} \\
&\qquad \langle S, A \cup \{x \to v\}\rangle(F) = \top \\
\langle S, A\rangle(\forall x : F) &= \top \text{ if, and only if, all values } v \in \Omega \text{ are such that} \\
&\qquad \langle S, A \cup \{x \to v\}\rangle(F) = \top
\end{aligned}
$$

The fact $\langle S, A\rangle(F) = \top$ is also denoted by $S, A \models F$. A structure S is a *model* of a formula F, denoted $S \models F$, if for any variable assignment A, $S, A \models F$.

Example 2.1.2. Let us consider the formula $p(x) \vee q(x)$. Let also $S = \langle \Omega, I\rangle$ be a structure such that $\Omega = \{\square, \Diamond, \bigcirc\}$ and

1. $I(p)(\square) = \top$
2. $I(p)(\Diamond) = \top$
3. $I(p)(\bigcirc) = \bot$
4. $I(q)(\square) = \bot$
5. $I(q)(\Diamond) = \bot$
6. $I(q)(\bigcirc) = \top$

Under the assignment $A = \{x \to \square\}$, we have $\langle S, A\rangle(p(x)) = I(p)(A(x)) = I(p)(\square) = \top$ and $\langle S, A\rangle(q(x)) = I(q)(A(x)) = I(q)(\square) = \bot$. Consequently, we also have $\langle S, A\rangle(p(x) \vee q(x)) = \top$. As a matter of fact, the latter also holds for the two other possible variable assignments $A' = \{x \to \Diamond\}$ and $A'' = \{x \to \bigcirc\}$. The structure S is therefore a model of F.

2.1.2 The Satisfiability Problem

A formula F in a first-order language is *satisfiable* if there exists a structure S and a complete variable assignment A such that $S, A \models F$. It is *unsatisfiable* otherwise. F is also said to be *valid* if any structure S and any variable assignment A are such that $S, A \models F$. The *satisfiability problem* is concerned with finding a structure S and a variable assignment A for a formula F such that S, $A \models F$, or with proving that this formula is unsatisfiable. The *validity problem* is concerned with proving that a formula is valid, or with finding a structure S and a variable assignment A such that $\langle S, A\rangle(F) = \bot$. As for propositional logic, the satisfiability problem is dual with the validity problem: a formula F is satisfiable if, and only if, $\neg F$ is not valid. A valid formula is also called a *theorem*.

The satisfiability and validity problems of a first-order formula are undecidable. This was first published by both Church [Chu36] and Turing [Tur36]. It is however possible to build a computing procedure that will eventually confirm any theorem, but will seek forever in the case of a non valid formula. This was, for instance, the approach taken by Davis and Putnam [DP60] when publishing their work at the basis of the DPLL algorithm. The satisfiability and validity problems are thus *semi-decidable*.

2.2 Theories

Although the first-order logic is undecidable, some its subsets, called theory, are decidable. The definitions and results introduced in this section are adapted from [Gan13b]. A *theory* T is characterized by a signature Σ_T and a set of axioms A_T. The signature Σ_T is a set of constant, function and predicate symbols while the axioms of A_T are first-order sentences involving the symbols in Σ_T.

A structure $S = \langle \Omega, I \rangle$ is a model for a theory T, also called a *T-model*, if for all axiom A in A_T, $S \models A$. A T-model is thus a model for the axioms of the theory T. A formula F is *satisfiable modulo theory T* if there exists a T-model M and a variable assignment A such that $M, A \models F$. It is *unsatisfiable modulo theory T* otherwise. A formula F is *valid modulo theory T* if for all T-models M and for all variable assignments A, $M, A \models F$.

Example 2.2.1. Let us suppose a theory N with the signature $\Sigma_N = \{zero, next, plus\}$ where $zero$ is a constant symbol, $next$ a function symbol of arity 1 and $plus$ a predicate symbol of arity 3. The list of axioms of N, denoted A_N, is as following:

A_1: $\forall x : plus(x, zero, x)$

A_2: $\forall x, y, z : \big(plus(x, y, z) \Rightarrow plus(x, next(y), next(z))\big)$

Let also the structure $S = \langle \mathbb{N}, I \rangle$ be such that

1. $I(zero) = 0$
2. $\forall n \in \mathbb{N} : I(next)(n) = n + 1$
3. $\forall n, m, l \in \mathbb{N} : I(plus)(n, m, l) = \top$ if, and only if, $n + m = l$

Then, S is a N-model. Indeed, for all variable assignments $A = \{x \to n\}$, where $n \in \mathbb{N}$, we have

$$\begin{aligned} \langle S, A \rangle(plus(x, zero, x)) &= I(plus)(A(x), I(zero), A(x)) \\ &= I(plus)(n, 0, n) \\ &= \top \end{aligned}$$

as $n + 0 = n$. Consequently, $S \models A_1$. We can also prove that $S \models A_2$.

2.3 Common Theories

In order to better illustrate the previous section, this one introduces some common theories and related results. The definitions and results introduced in this section are adapted from [Gan13b] and [Gan07].

2.3.1 Theory of Equality with Uninterpreted Functions

The theory of equality with uninterpreted functions T_{EUF} is characterized by the signature Σ_{EUF}, including the only predicate symbol $=$ of arity 2. For any couple of terms t_1 and t_2, we note $t_1 = t_2$ the atomic formula $(=)(t_1, t_2)$. The set of axioms A_{EUF} is listed below:

1. Reflexivity: $\forall x : x = x$
2. Symmetry: $\forall x, y : ((x = y) \Rightarrow (y = x))$
3. Transitivity: $\forall x, y, z : (((x = y) \wedge (y = z)) \Rightarrow x = z)$

Additionally, an atom is added to A_{EUF} for any function symbol f of interest using the following pattern:

$$\forall x_1, \dots x_n, y_1, \dots y_n : \Big((\bigwedge_{i=1}^{n} x_i = y_i) \Rightarrow f(x_1, \dots x_n) = f(y_1, \dots y_n) \Big)$$

This property is called *function congruence*. Similarly, an atom is added to A_{EUF} for any predicate symbol p of interest, using the following pattern:

$$\forall x_1, \dots x_n, y_1, \dots y_n : \Big((\bigwedge_{i=1}^{n} x_i = y_i) \Rightarrow p(x_1, \dots x_n) \Leftrightarrow p(y_1, \dots y_n) \Big)$$

This property is called *predicate congruence*.

These two rules are not really atoms, as they must be instanciated and added to A_{EUF} for every function and predicate symbols. Such pattern is usually called an *axiom schema*.

Example 2.3.1. Let us consider the formula

$$F = p(f(x)) \wedge (x = y) \wedge (f(y) = f(z)) \wedge \neg p(f(z))$$

Due to $x = y$, and by function congruence, we know that $f(x) = f(y)$. Moreover, as $f(y) = f(z)$, and by transitivity, we have $f(x) = f(z)$. Therefore, as $p(f(x))$ and by predicate congruence, we have $p(f(z))$, which is obviously inconsistent with the last statement $\neg p(f(z))$ of F. This formula is consequently unsatisfiable modulo theory T_{EUF}.

2.3.2 Presburger Arithmetic

Presburger arithmetic is the common denomination of a theory $T_{Presburger}$ encoding addition and equality over natural integers. It was named after Mojżesz Presburger, who first introduced it. Its signature is $\Sigma_{Presburger} = \{0, 1, +, =\}$, where 0 and 1 are constant symbols, + a function symbol of arity 2 and = a predicate symbol of arity 2. For any couple of terms t_1 and t_2, we note $t_1 = t_2$ the atomic formula $(=)(t_1, t_2)$. We also note $t_1 + t_2$ the term $(+)(t_1, t_2)$. The set $A_{Presburger}$ of axioms of this theory includes reflexivity, symmetry and transitivity of =, as defined in Section 2.3.1. It also includes the following axioms:

1. $\forall x : \neg(x + 1 = 0)$
2. $\forall x : x + 0 = x$
3. $\forall x, y : (x + 1 = y + 1) \Rightarrow x = y$
4. $\forall x, y : x + (y + 1) = (x + y) + 1$

These axioms are completed by the axiom schema

$$\big(p(0) \wedge (p(x) \Rightarrow p(x+1))\big) \Rightarrow p(x+1)$$

where p must be replaced by any predicate symbol of interest.

Let $\mathcal{F}_{Presburger}$ be the subset of first-order formulas using only the constant, function and predicate symbols from $\Sigma_{Presburger}$. Then the validity problem modulo $T_{Presburger}$ over formulas of $\mathcal{F}_{Presburger}$ is decidable. Its complexity is however super exponential, i.e. in $\mathcal{O}(2^{2^n})$ where n is the size of the formula [FR74]. Common artifacts from linear arithmetic, such as subtraction and inequalities, can easily be expressed in the language $\mathcal{F}_{Presburger}$.

Example 2.3.2. Let us consider the formula in linear arithmetic $1 - 2 < 0$. This formula can be rewritten as $1 < 2$, or even further as $\exists x : 1 + x = 1 + 1$. The formula is therefore an atom from $\mathcal{F}_{Presburger}$.

2.3.3 Peano arithmetic

Peano arithmetic is the theory T_{Peano} with signature $\Sigma_{Peano} = \{0, 1, +, \times, =\}$, where 0 and 1 are constant symbols, + and × function symbols of arity 2 and = a predicate symbol of arity 2. It was named after Giuseppe Peano who axiomatized arithmetic over natural numbers. As for Presburger arithmetic, for any couple of terms t_1 and t_2, we note $t_1 = t_2$ the atomic formula $(=)(t_1, t_2)$. We also respectively note $t_1 + t_2$ and $t_1 \times t_2$ the terms $(+)(t_1, t_2)$ and $(\times)(t_1, t_2)$. The set of axioms A_{Peano} contains all the axioms of Presburger arithmetic in addition to the following two:

1. $\forall x : x = 0$
2. $\forall x, y : x \times (y + 1) = (x \times y) + x$

Note that the induction axiom schema of Presburger arithmetic also applies.

Let $\mathcal{F}_{Peano}$ be the subset of first-order formulas using only the constant, function and predicate symbols from Σ_{Peano}. Then the validity problem modulo T_{Peano} over items of $\mathcal{F}_{Peano}$ is undecidable.

2.4 SMT Solvers

The current techniques used to solve the satisfiability problem modulo a given theory T for a formula F can be divided in three categories [GHN+04] [NOT06]. All of them rely on an implementation of the CDCL algorithm.

2.4.1 The Eager Approach

In the eager approach, the formula F is translated, by a satisfiability-preserving transformation, into a boolean function F_{eager} in Conjunctive Normal Form. This operation is referred to as *encoding*, or *bit blasting*. The satisfiability of F_{eager} is then verified by a SAT solver. If there exists a satisfiying assignment for F_{eager}, a variable assignment satisfying F is built from it.

The strength of this approach is that it can always use of the best available off-the-shelf SAT solvers. Its weakness lies in the lack of flexibility. It is indeed

necessary to create a new ad-hoc translation procedure for each new included theory.

The alternative approaches explained below are commonly considered as more efficient [NOT06]. However, the eager approach is still used in modern SMT solvers, especially when dealing with the theory of bit-vectors, which we do not explain here. An example of modern SMT solver based on this approach is STP [GD07].

2.4.2 The Lazy Approach

In the lazy approach, each atom occurring in the formula F is initially replaced by a boolean variable, forgetting about the theory T. This transformation produces a boolean formula F_{lazy}, which is in turn translated into a boolean formula F_{cnf} in CNF. As for the eager approach, a SAT solver is called to decide the satisfiability of F_{cnf}. If this formula is unsatisfiable, then F is unsatisfiable modulo theory T. However, if the formula F_{cnf} is satisfiable, then the satisfying assignment A produced by the SAT solver is transformed into a conjunction of literals C_{lazy} by

$$C_{lazy} = \bigwedge_{(x \to v_x) \in A} l_x$$

where

$$l_x = \begin{cases} x & \text{if} \quad v_x = \top \\ \neg x & \text{if} \quad v_x = \bot \end{cases}$$

The literals of C_{lazy} are then replaced by the initial atomic formulas that they were abstracting in order to produce the conjunction of atoms C. This result is fed to a dedicated decision procedure called a *T-solver*, which is able to solve the satisfiability modulo theory T problem on a conjunction of atomic formulas. If C is satisfiable modulo T, the T-solver delivers a satisfying variable assignment for it, which is also a satisfying variable assignment for F. If C is unsatisfiable modulo T, the T-solver determines an inconsistent set of atoms of C, which we denote by A_{unsat}. Then a so-called *theory lemma* L is defined by

$$L = \bigvee_{a \in A_{unsat}} \neg a$$

Again, each atom occurring in L is replaced by its corresponding boolean variable to produce a clause, which is added to the clause database F_{cnf}. This process is repeated until either a satisfying variable assignment for F is found, or F is proved to be unsatisfiable.

Example 2.4.1. Let us suppose that we need to decide the satisfiability modulo the theory of linear arithmetic T_{LA} of the formula

$$F = (x > 0) \wedge ((x = 7) \vee (x < 5))$$

In the above presented method, the formula is first abstracted by the boolean formula $F_{lazy} = a \wedge (b \vee c)$, which is already in CNF. This formula is fed to a SAT solver, producing the satisfying truth assignment $\{a \to \top, b \to \top, c \to \top\}$. The conjunction of atomic formulas C is then created as $(x > 0) \wedge (x = 7) \wedge (x < 5)$ and given to a specific solver for linear arithmetic. As C is unsatisfiable modulo

T_{LA}, the inconsistent set of atoms $A_{unsat} = \{x = 7, x < 5\}$ is returned. The clause $\neg b \vee \neg c$ is therefore added to the clause database F_{lazy} and the SAT solver is restarted. The SAT solver delivers the second truth assignment $\{a \to \top, b \to \top, c \to \bot\}$ and the conjunction of atoms $x > 0 \wedge x = 7 \wedge x \geq 5$ is passed to the T_{LA}-solver. This solver finally yields $\{x \to 7\}$ as a satisfying variable assignment for F.

2.4.3 The DPLL(T) Architecture

The DPLL(T) architecture is a refinement of the lazy approach that leverage a finer integration between the T-solver and the SAT solver. In this approach, the T-solver provides an interface which is used by the SAT solver all along the resolution process. In a nutshell, as for the lazy approach, a boolean formula F_{cnf} is produced by abstracting the original formula F and the SAT solver is called. Each time the SAT solver makes a decision, the corresponding atom is asserted in the T-solver. When the set of assertions becomes unsatisfiable modulo theory T, the SAT solver backtracks. Following this strategy, if the SAT solver finds a satisfying assignment for the boolean formula, the original formula is satisfiable modulo theory T. The theory solver can also propagate theory-specific deduction assignments. However, we do not describe the complete DPLL(T) architecture here. Example of modern SMT solvers based on this approach are Z3 [DMB08] and CVC4 [BCD$^+$11].

Conclusion

In this chapter, we first introduced the first-order logic and its undecidability result. We then showed how theories allow one to workaround this result by considering smaller sets of structures. Finally, we have briefly presented the possible architectures behind SMT solvers. In particular, we demonstrated the importance of SAT solvers in these architectures.

Chapter 3

Software Product-Line Model Checking

Model checking is a formal verification technique that allows one to check the model of a system of interest against temporal properties. It requires two inputs: A behavioral specification of the system and a temporal property to verify on it. As we shall see, the actual formalisms used to express these inputs vary. In return, a model checker yields an example of execution of the system that violates the property, or the certainty that no such execution exists.

During the past few years, researchers from the University of Namur have proposed extensions of model-checking algorithms designed to deal with software product lines (SPL). A SPL is a highly configurable software system that can be adapted to the specific needs of several organizations. The variability in a SPL can be represented by boolean formulas, so that the satisfiability problem plays a major role in the mentioned algorithms.

This chapter first introduces the classical LTL model-checking algorithms, before generalizing them to software product lines.

3.1 LTL Model Checking

This section introduces the basic concepts of LTL model checking. A more detailed presentation of this technique can be found in [BK08], from which most of the definitions and examples of this section are inspired.

3.1.1 Program Graphs and Transition Systems

The behavior of a software system can be modeled by a *program graph (PG)*. Such graph is not meant to include all the computation details of a software system, but only those who are relevant for the properties that one wants to verify. Let V be a set of a variables. Let also $A(V)$ denote the set of all possible variable assignments over V and $F(V)$ the set of all possible formulas over the variables in V. A program graph PG over V is a graph in which each transition is labeled with both an action and a condition from $F(V)$, and where each action has an effect over the variables in V. Formally, PG is defined as a tuple $\langle Loc, Act, Effect, Trans, Loc_0, f_0 \rangle$ where

1. Loc is a set of *locations*
2. Act is a set of *actions*
3. $Trans \subseteq (Loc \times F(V) \times Act \times Loc)$ is the *conditional transition relation*
4. $Effect : Act \times A(V) \to A(V)$ is the *effect function*
5. $Loc_0 \subseteq Loc$ is the set of *inital locations*
6. $f_0 \in F(V)$ is the *initial condition*

This graph represents a process that can start in a location $l_0 \in Loc_0$, with an initial variable assignment satisfying f_0. This process then moves in the space of locations by taking transitions and by executing the actions labeling these transitions. Each executed action has an effect on the current variable assignment. For a transition to be executable, the current variable assignment must satisfies the condition labeling this transition.

Example 3.1.1. Let us consider the embedded software system of a very simple beverage vending machine. This machine delivers either tea or coffee and has a limited capacity of two cups of each. After reaching this limit, it must be refilled. The program graph describing the behavior of the system is presented in Figure 3.1. This representation makes use of a notation close to UML State Diagrams. The set of states is $S = \{Start, Select\}$. The only initial state is represented with a incoming edge with no source. Edges are labeled with actions from the set $Act = \{receive\_coin, return\_coin, serve\_coffee, serve\_tea, refill\}$. The guards between brackets are the conditions that must hold on the variables from the set $V = \{n_{coffee}, n_{tea}\}$ for the corresponding action to be available. The effect function is represented by sets of statements after forward slashes. We can observe that the machine automatically returns money when it cannot serve any drink. We here consider the initial condition as being $n_{coffee} = n_{tea} = 2$.

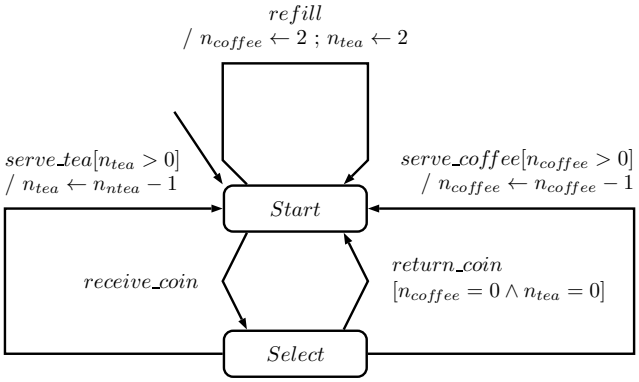


Figure 3.1: Program graph describing the embedded software system of a very simple beverage vending machine

The semantics of a program graph can be defined as being a transition system. A *transition system (TS)* is defined as a tuple $\langle S, Act, Trans, I, AP, L \rangle$ where

1. S is a set of *states*
2. Act is a set of *actions*
3. $Trans \subseteq (S \times Act \times S)$ is the *transition relation*
4. $I \subseteq S$ is the set of *initial states*
5. AP is a set of *atomic propositions*
6. $L : S \to \mathcal{P}(AP)$ is the *labeling function*, where $\mathcal{P}(AP)$ is the powerset of AP, i.e. the set of all subsets of AP.

Let $TS(PG)$ denote the transition system translating the program graph $PG = \langle Loc, Act, Effect, Trans_{PG}, Loc_0, f_0 \rangle$ over the set of variables V. It is defined as the tuple $\langle S, Act, Trans_{TS}, I, AP, L \rangle$ where

1. $S = Loc \times A(V)$
2. $Trans_{TS}$ is the smallest relation such that

$$\frac{(l, f, \alpha, l') \in Trans_{PG} \qquad a \models f}{((l, a), \alpha, (l', Effect(\alpha, a)) \in Trans_{TS}}$$

3. $I = \{(l, a) \mid l \in Loc_0 \wedge a \models f_0\}$
4. $AP = Loc \cup F(V)$
5. $L((l, a)) = \{l\} \cup \{f \in F(V) \mid a \models f\}$

The possible states of the software process defined by a program graph, in terms of locations and variable assignments, is therefore fully defined by the set of states of the transition system. The above definition includes a potentially infinite set of atomic propositions AP. However, a small subset of them is generally enough.

Example 3.1.2. Figure 3.2 shows the transition system generated from the program graph presented in Example 3.1.1. One can observe that, as a state from the transition system is made from an element of the cartesian product between Loc and $A(V)$, the size of this structure has grown exponentially with the number of variables in V.

A *terminal state* s of a transition system $TS = \langle S, Act, Trans, I, AP, L \rangle$ is a state s in S such that $\{s' \in S \mid \exists \alpha \in Act : (s, \alpha, s') \in Trans\} = \varnothing$. In this work, we make the hypothesis that no such state exists in a TS. An *execution* of TS is an infinite alternating sequence of states and actions $(s_i, \alpha_i)_{i>0}$ such that $s_1 \in I$ and $\forall i > 0 : (s_i, \alpha_i, s_{i+1}) \in Trans$. A *path* of TS is an infinite sequence of states $(s_i)_{i>0}$ such that $\exists \alpha_1, \alpha_2, \ldots \in Act : (s_i, \alpha_i)_{i>0}$ is an execution of TS. A *trace* of TS is an infinite sequence of sets of atomic propositions $(L(s_i))_{i>0}$ such that $(s_i)_{i>0}$ is a path of TS. We denote by $Traces(TS)$ the set of all traces of TS.

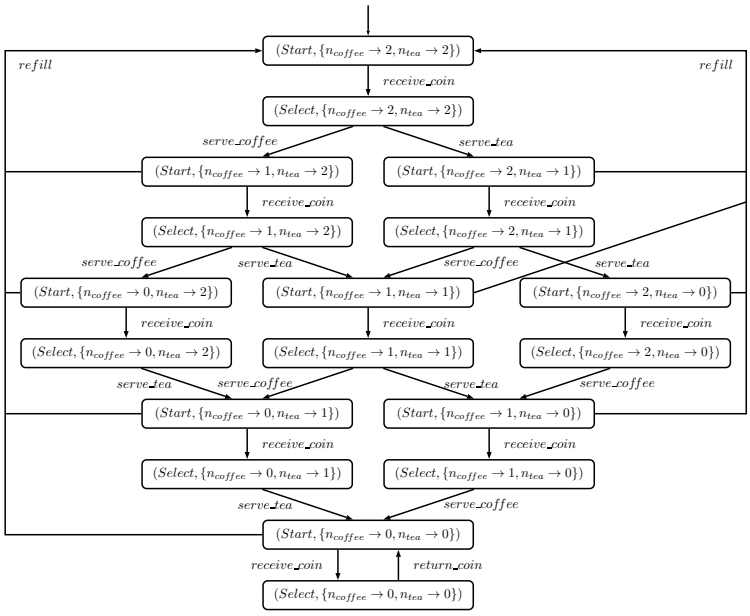


Figure 3.2: Transition system generated from the program graph presented in Example 3.1.1

3.1.2 Linear Time Properties and Temporal Logic

Let S^ω denote the set of all possible infinite sequence of elements from an arbitrary set S. This means that S^ω regroups all sequences of the form $(s_i)_{i>0}$ where $\forall i > 0 : s_i \in S$.

A *linear time property* over a set of atomic propositions AP defines a set of correct traces $P \subseteq \mathcal{P}(AP)^\omega$. A transition system TS over AP satisfies the property P if $Traces(TS) \subseteq P$. This is denoted by $TS \models P$.

Let AP be a set of atomic propositions. The set of all *linear temporal logic*, or *LTL*, formulas over AP is defined recursively as follows. Both constants $\bot$ and $\top$, as well as all elements $a \in AP$, are LTL fomulas. Moreover, if F and G are both LTL formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \cup G$, $\bigcirc F$, $\Diamond F$ and $\Box F$ are all LTL formulas. The semantics of an LTL formula F over AP is the set of infinite sequences $Words(F) = \{A \in \mathcal{P}(AP)^\omega \mid A \models F\}$ where the relation

$\models$ is defined by

$A \models \top$

$A \not\models \bot$

$A \models a$ if, and only if, $a \in A_1$

$A \models \neg F$ if, and only if, $A \not\models F$

$A \models F \wedge G$ if, and only if, $A \models F$ and $A \models G$

$A \models F \vee G$ if, and only if, $A \models F$ or $A \models G$

$A \models \bigcirc F$ if, and only if, $(A_i)_{i>1} \models F$

$A \models F \cup G$ if, and only if,

$$\exists k \geq 0 : \Big(\big(\forall j \in \{0, \ldots k-1\} : (A_i)_{i>j} \models F \big) \wedge \big((A_i)_{i>k} \models G \big) \Big)$$

$A \models \Diamond F$ if, and only if, $A \models \top \cup F$

$A \models \Box F$ if, and only if, $A \models \neg\Diamond\neg F$

where $A = (A_i)_{i>0} \in \mathcal{P}(AP)^\omega$, $a \in AP$ and F, G are LTL formulas. Let $a, b \in AP$. Figure 3.3 shows six LTL formulas with six infinite sequences in $\mathcal{P}(AP)^\omega$ satisfying them.

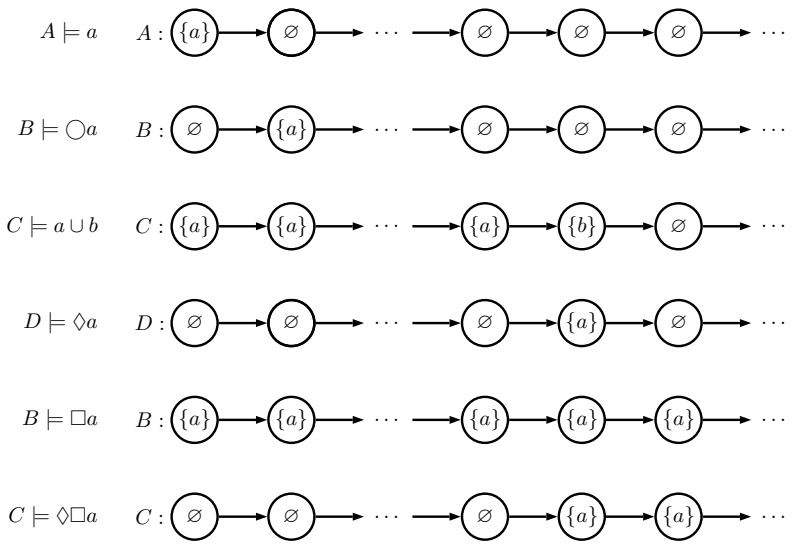


Figure 3.3: Semantics of several LTL formulas

A transition system TS verifies F, denoted $TS \models F$ if $TS \models Words(F)$ or, equivalently, if $Traces(TS) \subseteq Words(F)$. For any LTL formula F, the language of infinite words $Words(F)$ can be recognized by a Büchi automaton. Büchi automata are introduced in the next section.

3.1.3 Büchi Automata

A *Non-deterministic Büchi Automaton*, or *NBA*, is a tuple $A = \langle Q, \Sigma, \Delta, Q_0, F \rangle$ where

1. Q is a set of *states*
2. Σ is a set of symbols called the *alphabet*
3. $\Delta \subseteq (Q \times \Sigma \times Q)$ is the *transition relation*
4. $Q_0 \subseteq Q$ is the set of *initial states*
5. $F \subseteq Q$ is the set of *accepting states*

An *accepting run* of a an NBA $A = \langle Q, \Sigma, \Delta, Q_0, F \rangle$ is an infinite sequence $(q_i)_{i>0} \in Q^\omega$ such that $\forall i > 0 : \exists \alpha \in \Sigma : (q_i, \alpha, q_{i+1}) \in \Delta$ and such that $q_i \in F$ for infinitely many values $i > 0$. An infinite word $w = (\alpha_i)_{i>0}$ over the alphabet Σ is accepted by A if there exists an accepting run $(q_i)_{i>0}$ such that $\forall i > 0 : (q_i, \alpha_i, q_{i+1}) \in \Delta$. The language of an NBA A, denoted $\mathcal{L}(A)$, is composed from all words accepted by A. A language L is said to be *ω-regular* if, and only if, there exists a Büchi automaton A such that $\mathcal{L}(A) = L$.

Example 3.1.3. Let us consider the Büchi automaton presented in Figure 3.4. The initial state is represented with an incoming edge with no source. The accepting state is shown with a double line border. We can see that this automaton accepts all infinite words over the alphabet $\{a, b\}$ containing an infinite number of occurrences of b, i.e. the language $(a^*b)^\omega$.

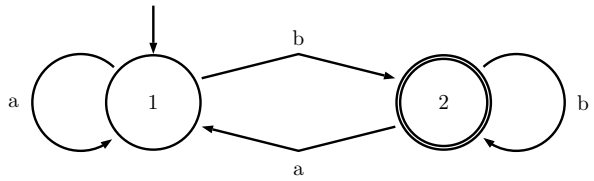


Figure 3.4: Büchi automaton recognizing the language $(a^*b)^\omega$

There are important differences between finite automata and NBAs. First, a finite automaton is meant to work on finite words while a Büchi automaton runs on infinite ones. Second, an execution of a finite automaton is accepting if ending in a terminal state. A word is accepted by a Büchi automaton when going through accepting states infinitely many times. Third, contrary to finite automata, deterministic Büchi automata are generally not as expressive as non deterministic ones.

The notion of Bûchi automaton can be generalized. A *generalized non deterministic Büchi automaton*, or *GNBA*, is basically a Büchi automaton with several sets of accepting states $\{F_i\}_{i=1}^n$. An *accepting run* of a GNBA $G = \langle Q, \Sigma, \Delta, Q_0, \{F_i\}_{i=1}^n \rangle$ is an infinite sequence $(q_i)_{i>0} \in Q^\omega$ such that $\forall i > 0 : \exists \alpha \in \Sigma : (q_i, \alpha, q_{i+1}) \in \Delta$ and such that $\forall j \in \{1, \dots n\} : q_i \in F_j$ for infinitely many values $i > 0$. For any GNBA G, there exists a NBA A such that A and G accept the same language. The set of languages accepted by GNBAs is therefore the set of ω-regular languages. GNBAs reveal to be practical when dealing with temporal logic. For any LTL formula F, we can indeed define an GNBA recognizing the language $Words(F)$.

Example 3.1.4. The GNBA recognizing the LTL formula $F = \bigcirc a$ is presented in Figure 3.5. One should note that their is no sets of accepting states. According to the definition of a GNBA, this means that every run is accepting. Consequently, any infinite word associated with a valid run of the GNBA is accepted by it. However, as there are only two initial states, all infinite sequences $(s_i)_{i>0} \in \mathcal{P}(AP)^\omega$ such that $a \notin s_2$ cannot be associated with a valid run of this automaton.

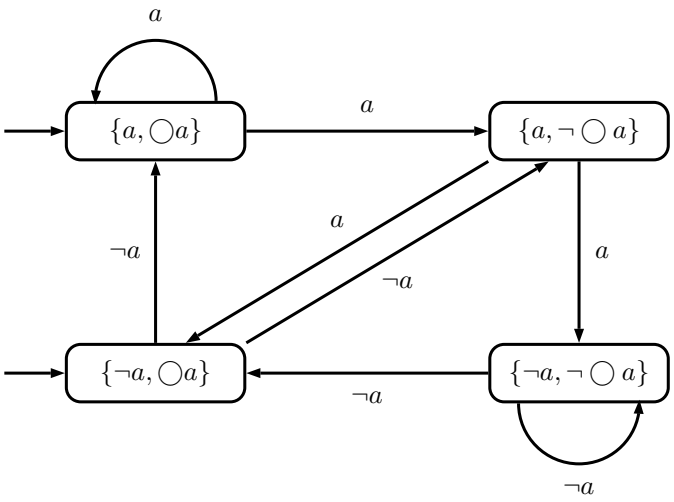


Figure 3.5: Büchi automaton for the LTL formula $\bigcirc a$

3.1.4 Verifying ω-regular Properties

Let $TS = \langle S, Act, Trans, I, AP, L\rangle$ be a transition system and P an ω-regular property over the set of atomic propositions AP. We want to verify if $TS \models P$, i.e. if $Traces(TS) \subseteq P$. This can be rewritten as $Traces(TS) \cap (\mathcal{P}(AP)^\omega \setminus P) = \varnothing$, or even further as $Traces(TS) \cap \neg P = \varnothing$. The set of ω-regular languages is closed under complementation. Consequently, $\neg P$ is also ω-regular. Let $A = \langle Q, \Sigma, \Delta, Q_0, F\rangle$ be a Büchi automaton recognizing $\neg P$. The verification comes down to verifying that $Traces(TS) \cap \mathcal{L}(A) = \varnothing$. To do so, we consider the synchronous product between the transition system and the Büchi automaton.

The *synchronous product* between TS and A is the transition system $TS \otimes A = \langle S', Act, Trans', I', AP', L'\rangle$ such that

1. $S' = S \times Q$
2. $AP' = Q$
3. $\forall (s, q) \in S' : L'(s, q) = \{q\}$
4. $Trans'$ is the smallest relation such that

$$\frac{(s, \alpha, s') \in Trans \qquad (q, L(s), q') \in \Delta}{((s, q), \alpha, (s', q')) \in Trans'}$$

5. $I' = \{(s, q) \in S' \mid s \in I\}$

Then, the verification problem is reduced to checking that

$$TS \otimes A \models \Diamond\Box \bigwedge_{q \in F} \neg q$$

This means that there must not exists, in $TS \otimes A$, any cycle reachable from an initial state s_0 in I' and including a state s_f such that $L'(s_f) \subseteq F$. We can verify this through an exhaustive exploration of the state space of $TS \otimes A$.

This search process can be realized by a state-space traversal algorithm. This method consists in two nested depth-first search procedures. The first one, called the *outer* depth-first search, is started from an initial state s_0. When it has fully expanded a state s such that $L'(s) \subseteq F$, the outer search calls the second procedure, called the *inner* depth-first search. The inner search then visits all states reachable from s that is has not visited before. If there exists a state s' reachable from s such that s' is in the path currently occupied by the outer search process, then there exists a cycle reachable from s_0 and including s, which is labeled by a state in F. The property P is thus violated by TS. If no such state s' is reachable, the inner search exits and the outer search continues.

This nested depth-first algorithm is presented in Algorithm 4. This algorithm calls the *outer-search* function on all initial states of $TS \otimes A$. The *outer-search* function is defined in Algorithm 5. It maintains a set of visited states *visitedInOuter* and a stack of states *outerStack*. When an unvisited state s' is reachable from the state s at the top of the stack, s' is added in both the stack and the set of visited states, and the process continues. When no such state s' exists, and if $L'(s) \subseteq F$, the algorithm calls the *inner-search* procedure. Afterwards, the procedure backtracks. The *inner-search* function is defined in Algorithm 6. It follows the same traversal strategy than the *outer-search* function. However, if it reaches a state contained in *outerStack*, the stack of states of the *outer-search* procedure, then it returns *true*, which means that a cycle has been found and that the property P is violated.

Algorithm 4 Nested Depth-First Search Algorithm

```
boolean ndfs()
  for all s0 ∈ I' do
    if outer-search(s0) then
      return true
    end if
  end for
  return false
```

During the past few years, researchers from the University of Namur have adapted these algorithms to software product lines. The following section introduces the notion of product line.

3.2 Product Lines and Feature Diagrams

A *product line* is a set of related products manufactured by a single organization. Variants of a product line share common pieces or behaviors, and are built

Algorithm 5 Outer Depth-First Search

```
boolean outer-search(State s_0)
  Stack outerStack ← emptyStack()
  Set visitedInOuter ← ∅
  Set visitedInInner ← ∅
  push(outerStack, s_0)
  visitedInOuter ← visitedInOuter ∪ {s_0}
  while not empty(outerStack) do
    State s ← top(outerStack)
    if ∃s', α : ((s, α, s') ∈ Trans') ∧ (s' ∉ visitedInOuter) then
      push(outerStack, s')
      visitedInOuter ← visitedInOuter ∪ {s'}
    else
      if L'(s) ⊆ F then
        if inner-search(s, visitedInInner, outerStack) then
          return true
        end if
      end if
      pop(outerStack)
    end if
  end while
  return false
```

Algorithm 6 Inner Depth-First Search

```
boolean inner-search(
State s_0, Set visitedInInner, Stack outerStack)
  Stack innerStack ← emptyStack()
  push(innerStack, s_0)
  visitedInInner ← visitedInInner ∪ {s_0}
  while not empty(innerStack) do
    State s ← top(innerStack)
    if ∃s', α : ((s, α, s') ∈ Trans') ∧ (s' ∉ visitedInInner) then
      if in(outerStack, s') then
        return true
      end if
      push(innerStack, s')
      visitedInInner ← visitedInInner ∪ {s'}
    else
      pop(innerStack)
    end if
  end while
  return false
```

from a common, managed set of assets. Reusing core assets from one product to another allows one to benefit from economies of scale when developing a large number of products. A *software product line* is a product line of software-intensive systems. The variability that can take place in a product line is usually expressed in terms of *features*. Simply said, a feature is an asset that can be included in a product and be missing in another. It usually defines some set of additional functionalities that can be inserted to a specific product to enhance it, or retrieved from another for incompatibility, economical or even simplicity reasons. Developing a software product line is like developing a highly configurable software system in which features can be activated or deactivated depending on the target's needs or specificities.

Relations between features, as dependencies or incompatibilities, are usually expressed in a *feature diagram*. Feature diagrams have first been introduced by Kang et al. [KCH$^+$90] and have received a formal semantic from Schobbens et al. [SHT06]. A feature diagram can be defined as a tuple $d = \langle F, r, DE, \lambda \rangle$ where

1. F is a set of features
2. $r \in F$ is the root features, also called the *concept*
3. $DE \subseteq F \times F$ is the *decomposition relation*
4. The structure $\langle F, DE \rangle$ defines a directed tree of root r
5. $\forall f \in F : \lambda(F)$ is a boolean function $\mathbb{B}^n \to \mathbb{B}$ where $n = |\{f' \in F \mid (f, f') \in DE\}|$

A product of a product line expressed by a feature diagram $d = \langle F, r, DE, \lambda \rangle$ is a set of feature $P \subseteq F$. A product P is *valid* with respect to d, denoted $P \models d$, if

1. The concept is in P : $r \in P$
2. All features of P, except the concept, must have their predecessors in P: $\forall (p, f) \in DE : (f \in P \Rightarrow p \in P)$
3. For all feature $f \in P$, the boolean function $\lambda(f)$ is satisfied: Let $\{f_i\}_{i=1}^n = \{f' \in F \mid (f, f') \in DE\}$, then $\lambda(f)(f_1 \in P, \dots f_n \in P) = \top$

The function λ thus restricts, for an included feature, the set of possible incorporated children.

The semantics of a feature diagram can be seen as a boolean formula over the set of features F. The formula translating the feature diagram d is indeed

$$r \wedge \Big(\bigwedge_{(p,f) \in DE} (f \Rightarrow p) \Big) \wedge \Big(\bigwedge_{f \in F} \mathcal{F}_{\lambda(f)} \Big)$$

where $\mathcal{F}_{\lambda(f)}$ is a boolean formula representing the boolean function $\lambda(f)$. A valid product is then as simple as a satisfying assignment for this formula. Feature diagrams are, however, often represented graphically as trees. The formal semantics introduced before is therefore important to bridge the gap between graphical representation and formal logic. Henceforth, a formula over a set of considered features is called a *feature expression*.

Example 3.2.1. Let us consider a product line of beverage vending machines. These machines can sell tea, coffee, or both. Each type of drinks is considered as a feature. If the corresponding feature is activated, it is possible to cancel an order. High-end products might also be equipped with a motion sensor that checks whether the user has effectively taken his cup before processing a new order. A machine may finally deliver beverages for free to the owners of special badges. The set of features of interest is here $\{Coffee, Tea, Cancel, Badge, Sensor\}$. The feature diagram $d = \langle F, r, DE, \lambda \rangle$ is presented in Figure 3.6. The set of features is $F = \{Vending\ Machine, Beverage, Coffee, Tea, Cancel, Badge, Sensor\}$. *Vending Machine* is the concept and *Beverage* a composite feature introduced for hierarchical decomposition. The feature expression describing this product line is

$$Vending\ Machine \wedge Beverage \wedge (Coffee \vee Tea)$$

A valid product is e.g.

$$\{Vending\ Machine, Beverage, Coffee, Badge, Sensor\}$$

which corresponds to the satisfying assignment

$$\{Vending\ Machine \to \top, Beverage \to \top, Coffee \to \top, \\ Tea \to \bot, Badge \to \top, Sensor \to \top\}$$

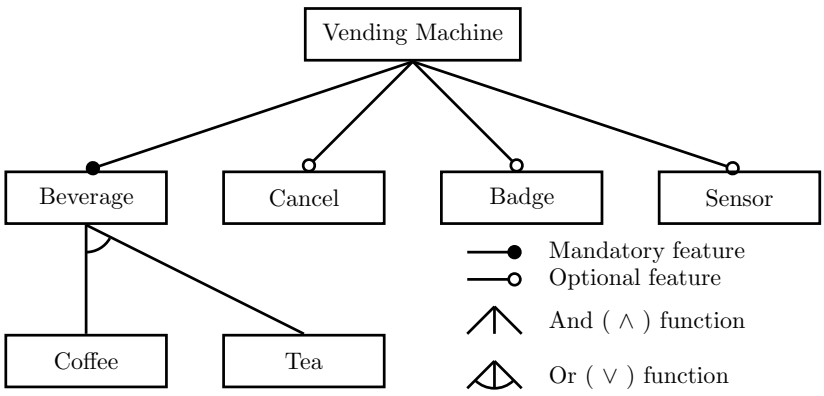


Figure 3.6: Feature diagram describing a product line of beverage vending machines

3.3 Product-Line Model Checking

3.3.1 Featured Program Graphs and Transition Systems

Verifying properties on a software product line using the model-checking algorithms introduced in Section 3.1.4 requires to specify the behavior of every single product of the product line and to verify them individually. However, the number of variants in a product line tends to grow exponentially with the number

of features. Indeed, each time a feature is added, the number of product is potentially doubled. Therefore, verifying product individually may require a huge amount of time and be unfeasible in practice. To address this problem, extensions of program graphs and transition systems have been proposed [CHS+10]. These extensions can be used to specify the behaviors of all the products of a product line in one single model.

A *featured program graph (FPG)* over a set of variable V is defined as a tuple $FPG = \langle Loc, Act, Effect, Trans, Loc_0, f_0, d, \gamma \rangle$ where

1. $\langle Loc, Act, Effect, Trans, Loc_0, f_0 \rangle$ is a program graph over V
2. $d = \langle F, r, DE, \lambda \rangle$ is a feature diagram
3. $\forall t \in Trans : \gamma(t)$ is a feature expression over the set of features F

The semantics of an FPG is defined as being a featured transition system. A *featured transition system (FTS)* is defined as a tuple $FTS = \langle S, Act, Trans, I, AP, L, d, \gamma \rangle$ where

1. $\langle S, Act, Trans, I, AP, L \rangle$ is a transition system
2. d a feature diagram
3. $\forall t \in Trans : \gamma(t)$ is a feature expression over the set of features F

Let $FTS(FPG)$ denote the transition system translating the featured program graph $FPG = \langle Loc, Act, Effect, Trans_{PG}, Loc_0, f_0, d, \gamma_{PG} \rangle$ over the set of variables V. It is defined as the tuple $\langle S, Act, Trans_{TS}, I, AP, L, d, \gamma_{TS} \rangle$ where

1. S, Act, I, AP, L are defined as in Section 3.1.1.
2. $Trans_{PG}$ is the smallest relation, and γ_{TS} the labeling function, such that

$$\frac{t = (l, f, \alpha, l') \qquad t \in Trans_{PG} \qquad a \models f}{t' = ((l, a), \alpha, (l', Effect(\alpha, a)) \qquad t' \in Trans_{TS} \qquad \gamma_{TS}(t') = \gamma_{PG}(t)}$$

Each transition in an FPG or an FTS is annotated with a feature expression that describes the set of products that can execute this transition. The *projection* of the featured transition system $FTS = \langle S, Act, Trans, I, AP, L, \gamma \rangle$ to a product P, denoted $FTS_{|P}$, is the transition system $TS = \langle S, Act, Trans_{|P}, I, AP, L \rangle$ obtained from FTS by removing all transition $t \in Trans$ such that P is not allowed to execute t, i.e. such that $P \not\models \gamma(t)$. The set $Trans_{|P}$ is thus defined as $\{t \in Trans \mid P \models \gamma(t)\}$. Therefore, the projection of an FTS describes the behavior of one product of the product line.

Example 3.3.1. Let us consider again the product line described in Example 3.2.1. The behaviors of the different beverage vending machines are expressed in the FTS exposed in Figure 3.7. The feature expressions labeling the transitions are expressed after forward slashes. For instance, an additional step after serving a cup is executed for the products with the feature $Sensor$. The projection of this featured transition system on the product $P = \{Vending\ Machine, Beverage, Coffee, Tea, Badge, Sensor\}$ is shown in figure 3.8.

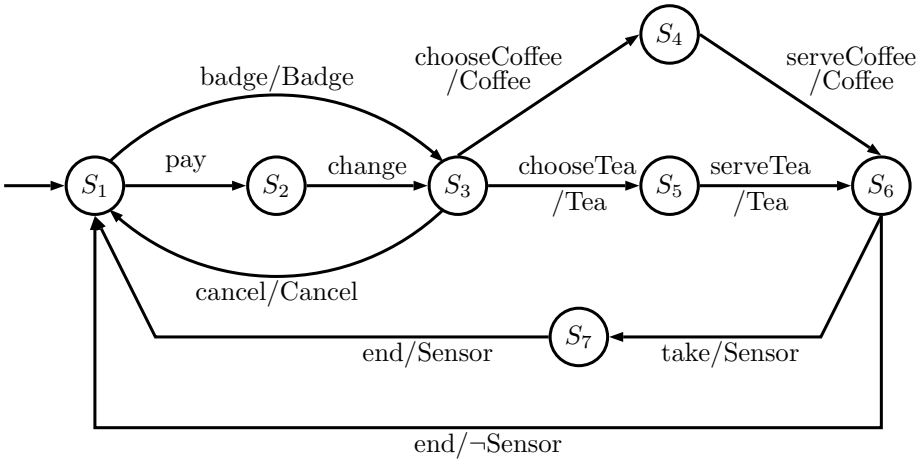


Figure 3.7: FTS for the product line of beverage vending machines described in Example 3.2.1

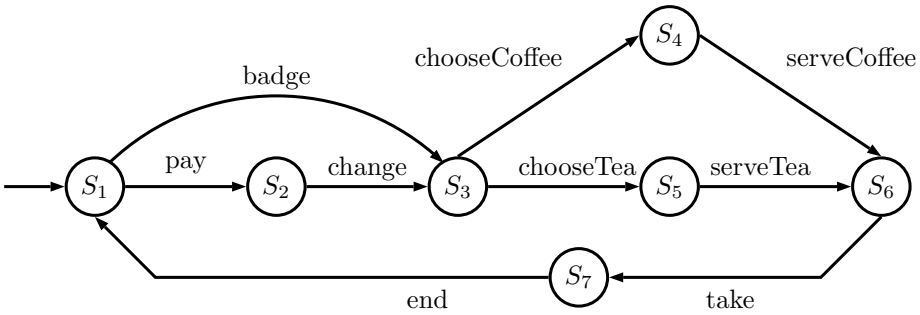


Figure 3.8: Projection of the FTS presented in Figure 3.7 on the product $\{Vending\ Machine, Beverage, Coffee, Tea, Badge, Sensor\}$

3.3.2 Verifying ω-regular Properties on Featured Transition System

Let $FTS = \langle S, Act, Trans, I, AP, L, d, \gamma \rangle$ be a featured transition system and P an ω-regular property over the set of atomic propositions AP. We want to verify if $\forall Prod \subseteq F$ such that $Prod \models d : TS_{|Prod} \models P$, i.e. if $Traces(TS_{|Prod}) \subseteq P$. We also want to avoid verifying products individually. To address this, Classen et al. defined an algorithm inspired from the one presented in Section 3.1.4. Let $A = \langle Q, \Sigma, \Delta, Q_0, F \rangle$ be a Büchi automaton recognizing the language $Words(\neg P)$. The *synchronous product* between FTS and A is the featured transition system $FTS \otimes A = \langle S', Act, Trans', I', AP', L', d, \gamma' \rangle$ such that

1. S', Act, I', AP', L' are defined as in Section 3.1.4
2. $Trans'$ is the smallest relation, and γ' the labeling function, such that

$$\frac{t = (s, \alpha, s') \in Trans \qquad t \in Trans \qquad (q, L(s), q') \in \Delta}{t' = ((s, q), \alpha, (s', q')) \qquad t' \in Trans' \qquad \gamma'(t') = \gamma(t)}$$

As in Section 3.1.4, a nested depth-first search algorithm is performed to look for a cycle reachable from an initial state s_0 in I' and including a state s_f such that $L'(s_f) \subseteq F$. However, only some products of the product line defined by the feature diagram d might be able to follow a specific path in the FTS. For this reason, the algorithm maintains a feature expression e representing the set of all products that can follow the currently expanded path. This feature expression is obtained by accumulating those labeling the taken transitions. Two differences takes place compared to Algorithm 4:

1. Any time a transition is taken, we must verify that there exists at least one product that can actually follow the path currently expanded by the algorithm. Such product exists if the feature expression $e \wedge d$ is satisfiable, i.e. if there exists a satisfying assignment for both the current feature expression e and the feature diagram d.

2. When a state s is reached for a second time, it may happen that it has already been visited for a different set of products. For this reason, the feature expression f for which s is visited is stored during the process. When s is reached again by the algorithm, the procedure continues for the set of products satisfying $e \wedge d$ but not satisfying f, i.e. those satisfying $e \wedge \neg f \wedge d$.

The resulting algorithm is presented in Algorithm 7. This algorithm calls the *featured-outer-search* function on all initial states of $FTS \otimes A$ and returns a feature expression representing the set of products violating the property P. The *featured-outer-search* function is presented in Algorithm 8. This algorithm is similar to Algorithm 5. It maintains a set of visited states $visitedInOuter$, a stack $outerStack$, a mapping $outerTable$ between visited states and feature expressions, and a feature expression $badProducts$ representing the set of products violating the property P. Elements of the stack are composed from a state $s \in S'$, a feature expression e, and the set of taken transitions from the state s. When an unvisited state s' is reachable from the state s at the top of the stack, and the transition is executable by a valid product, s' is added in the stack $outerStack$, the set of visited states $visitedInOuter$, and the mapping $outerTable$, and the process continues. The existence of a valid product is verified by the *sat* function, performing a satisfiability check on the current feature expression. When s' has already been visited by a set of features f, $\neg f$ is added to the current feature expression, and a new satisfiability check is performed. When no such state s' exists, and if $L'(s) \subseteq F$, the algorithm calls the *featured-inner-search* procedure. Afterwards, the procedure backtracks. The *featured-inner-search* function is defined in Algorithm 9. It follows the same traversal strategy than the *featured-outer-search* function. However, if it reaches a state contained in $outerStack$, the stack of states of the *featured-outer-search* procedure, then it adds the current feature expression to the set of violating product $badProducts$. To avoid verifying the same products several times, satisfiability checks are also performed in conjunction with $\neg badProducts$.

These algorithms have successively been implemented in two model checkers: SNIP and ProVeLines 2.

Algorithm 7 Featured Nested Depth-First Search Algorithm

FeatureExp fndfs()
  FeatureExp $badProducts \leftarrow \bot$
  for all $s_0 \in I'$ **do**
    $badProduct \leftarrow badProducts \vee$ featured-outer-search(s_0)
  end for
  return $badProducts$

3.3.3 SNIP

SNIP has first been introduced by Classen et al [CCH+12] as a model checker for software product lines. Featured program graphs are specified using *fPromela*, a feature-aware extension of *Promela*, the input language of the SPIN model checker [Hol04]. The syntax of Promela is close to imperative programming languages such as C or Pascal. fPromela extends it with additional constructs to guard statements with feature expressions. Feature diagrams are represented using *TVL*, a text-based language for expressing variability [CBH11]. SNIP is coded in C and follows the architecture presented in Figure 3.9. Dashed edges represent dependencies. The core of SNIP is divided into several layers that have access to a set of libraries. These libraries are external tools called from the core of SNIP, implementing several aspects of the model checker. LTL formulas are parsed and translated into Büchi automata using the *LTL 2 BA* library [GO01]. Boolean functions are represented by BDDs from the *CU Decision Diagram*, or *CUDD*, package.

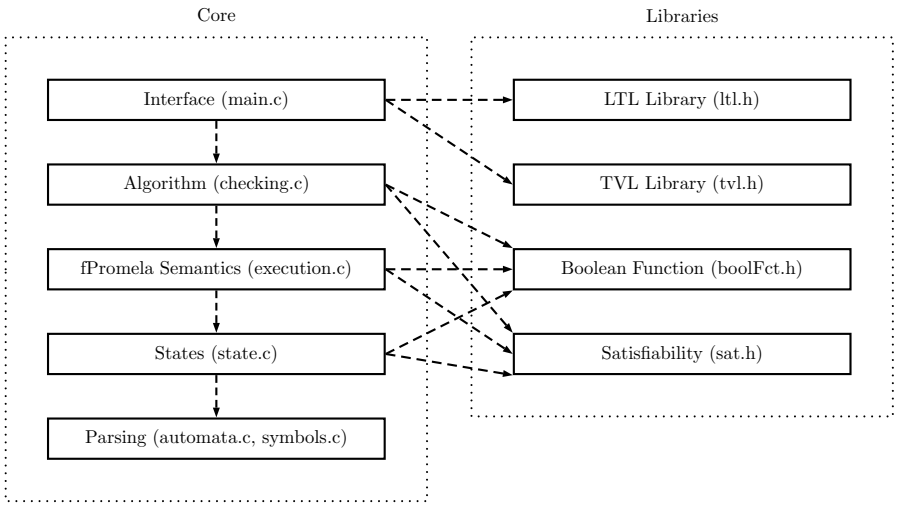


Figure 3.9: A simplified sketch of the architecture of the SNIP model checker

In the past few years, the functionalities of SNIP have been extended in many directions. The tool now includes the support for real-time [CSHL12] and stochastic [CHS+13] software product lines behavior. Support for non boolean variability [CSHL13] has also been introduced. These additional possibilities have transformed SNIP into a software product line, named *ProVeLines*, for

Algorithm 8 Featured Outer Depth-First Search

```
FeatureExp featured-outer-search(State s_0)
  Stack outerStack ← emptyStack()
  Set visitedInOuter ← ∅
  Map outerTable ← emptyTable()
  Set visitedInInner ← ∅
  Map innerTable ← emptyTable()
  FeatureExp badProducts ← ⊥
  push(outerStack, (s_0, ⊤, ∅))
  visitedInOuter ← visitedInOuter ∪ {s_0}
  outerTable[s_0] ← ⊤
  while not empty(outerStack) do
    (s, e, T) ← top(outerStack)
    if ∃s', α : ((s, α, s') ∈ Trans') ∧ ((s, α, s') ∉ T) then
      T ← T ∪ {(s, α, s')}
      pop(outerStack)
      push(outerStack, (s, e, T))
      e ← e ∧ γ(s, α, s')
      if sat(e ∧ d∧) then
        if s' ∈ visitedInOuter then
          e ← e ∧ ¬outerTable[s']
          if sat(e ∧ d∧) then
            push(outerStack, (s', e, ∅))
            outerTable[s'] ← e
          end if
        else
          push(outerStack, (s', e, ∅))
          visitedInOuter ← visitedInOuter ∪ {s'}
          outerTable[s'] ← e
        end if
      end if
    else
      if L(s) ⊆ F then
        badProducts ← featured-inner-search(
        s, e, visitedInInner, innerTable, outerStack, badProducts)
      end if
      pop(outerStack)
    end if
  end while
  return badProducts
```

Algorithm 9 Featured Inner Depth-First Search

```
boolean featured-inner-search(
State s_0, FeatureExp e_0,
Set visitedInInner, Map innerTable,
Stack outerStack, FeatureExp badProducts)
  Stack innerStack ← emptyStack()
  push(innerStack, (s_0, e_0, ∅))
  visitedInInner ← visitedInInner ∪ {s_0}
  innerTable[s_0] ← e_0
  while not empty(innerStack) do
    (s, e, T) ← top(innerStack)
    if in(s, outerStack) then
      badProducts ← badProducts ∨ e
      pop(innerStack)
    else
      if ∃s', α : ((s, α, s') ∈ Trans') ∧ ((s, α, s') ∉ T) then
        T ← T ∪ {(s, α, s')}
        pop(innerStack)
        push(innerStack, (s, e, T))
        e ← e ∧ γ(s, α, s')
        if sat(e ∧ d ∧ ¬badProducts) then
          if s' ∈ visitedInInner then
            e ← e ∧ ¬innerTable[s']
            if sat(e ∧ d ∧ ¬badProducts) then
              push(innerStack, (s', e, ∅))
              innerTable[s'] ← e
            end if
          else
            push(innerStack, (s', e, ∅))
            visitedInInner ← visitedInInner ∪ {s'}
            innerTable[s'] ← e
          end if
        end if
      else
        pop(innerStack)
      end if
    end if
  end while
  return badProducts
```

Product line of Verifiers for software product Lines.

In addition to standard boolean features, ProVeLines supports two additional types that are intensively used in practice: multi-features, i.e. features that may appear several times in a given product, and numeric features. The former entails changes only to the semantics of feature diagrams. The latter, however, also raises the need for non boolean feature expressions. To this end, the Z3 solver has been integrated into some variants of ProVeLines. In these variants, boolean formulas are represented using the built-in ASTs of the Z3 API, and the satisfiability problem is solved using the mentioned solver. However, Cordy's experiment revealed a large time overhead compared to the BDD implementation. Verifying LTL properties, even on a fully boolean model, can indeed take until 96,144 % more time when using the SMT solver instead of the BDD package.

3.3.4 ProVeLines 2

The large amount of additional possibilities introduced in ProVeLines have rendered the tool difficult to maintain. For this reason, a second version of the tool was developed. This development aimed at ensuring the maintainability of the tool by providing a clear and documented architecture. This architecture, presented in Figure 3.10, facilitates variability. In particular, it makes good use of the of the *Abstract Factory* design pattern. The *checker* package implements the model-checking algorithm presented in Section 3.3.2.

The major difference between ProVeLines and ProVeLines 2, in terms of functionalities, is that ProVeLines 2 relies on the *FSTM* input language. FSTM stands for *Featured State Machines*. It is a domain-specific language developed specifically for one of the industrial partners of the University of Namur. Its implementation can be seen in the *fts* package. The *math* package provides an implementation for boolean functions and satisfiability checking. The current implementation also relies on the CUDD package.

3.4 Objectives

As mentioned in Section 3.3.3, the use of SMT solvers to perform satisfiability checks in ProVeLines has raised an important performance issue. Cordy however states that

> "On the technical side, the use of SMT solvers might be a mandatory step. Still, it constitutes an interesting problem [...]."

The goal of this master thesis is therefore to investigate this problem. In a first time, we want to explore the root cause of it. In a second time, we want to propose an new integration approach to solve this issue, or at least to reduce it.

Conclusion

This chapter first introduced the basic concepts of LTL model checking before extending them to software product lines. It then presented a set of tools implementing these extensions. Finally, it stated the problem tackled in this master thesis.

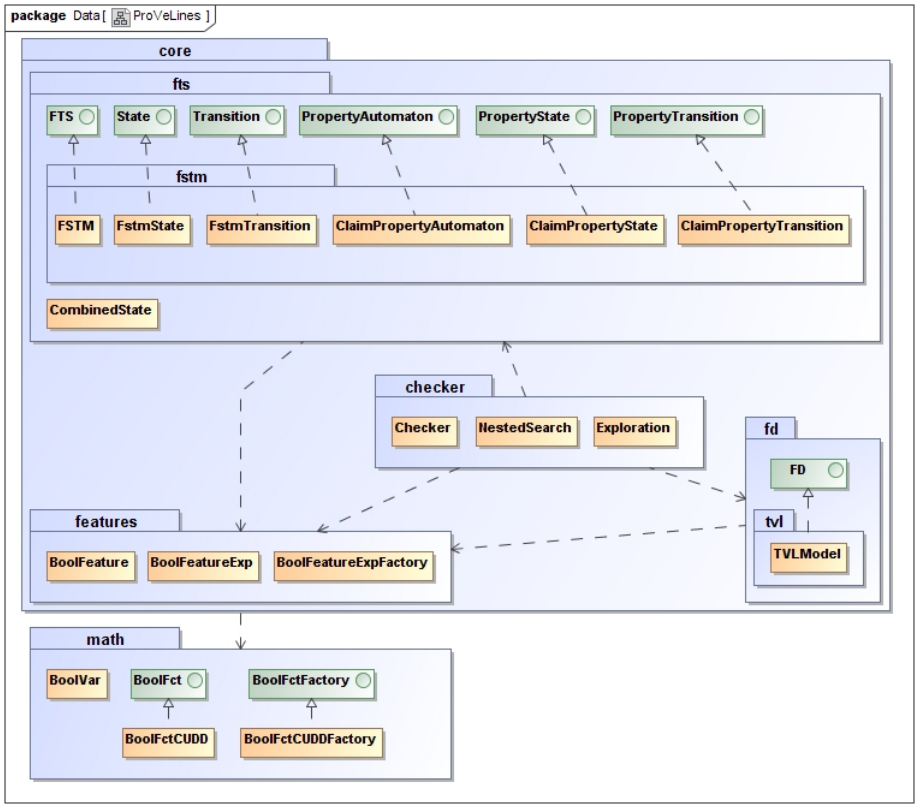


Figure 3.10: UML Class Diagram presenting a simplified version of the architecture of ProVeLines 2

Part II

Contribution

Chapter 4

Efficient Integration of a SAT Solver into ProVeLines

As we have seen in Chapter 2, an SMT solver is made of two main parts: A SAT solver and a theory specific solver. The time overhead mentioned in Section 3.3.3 has been revealed during an experiment where the SMT solver was used exclusively to solve boolean constraint. Therefore, our insight was that the SAT solver could be the root cause of it. To make sure of this, we decided to integrate a SAT solver into ProVeLines, and to measure the performance of the resulting tool.

This chapter goes through the theoretical developments behind this integration. As we shall see, what seemed to be a straightforward work at first has revealed to be a very ponderous task.

4.1 Clause Production

SAT solvers work on clause databases exclusively. The first step when integrating a SAT solver is therefore to represent boolean formulas using an ad-hoc implementation and to transform them into Conjunctive Normal Form.

4.1.1 Tree Representation of Boolean Formulas

When the model checker needs to build and manipulate boolean formulas, by creating their negations, conjunctions and disjunctions, these formulas can simply be represented as directed trees. Terminal vertices of these trees are labeled with either boolean variables or constants from the boolean domain $\mathbb{B} = \{\bot, \top\}$. Non-terminal vertices are labeled with boolean operators from the set $\{\neg, \wedge, \vee\}$. Non-terminal vertices labeled with $\neg$ have only one child while those labeled with $\wedge$ and $\vee$ have exactly two. The boolean formula represented by a directed tree of root node r, denoted $\mathcal{F}(r)$, can be recursively defined by

1. If r is a terminal vertex, then
   (a) If r is labeled with $\bot$, then $\mathcal{F}(r) = \bot$
   (b) If r is labeled with $\top$, then $\mathcal{F}(r) = \top$

(c) If r is labeled with a boolean variable x, then $\mathcal{F}(r) = x$

2. If r is a non-terminal vertex, then

   (a) If r is labeled with $\neg$ and has the vertex v as child, then $\mathcal{F}(r) = \neg\mathcal{F}(v)$

   (b) If r is labeled with $\wedge$ and has the vertices v_1 and v_2 as children, then $\mathcal{F}(r) = \mathcal{F}(v_1) \wedge \mathcal{F}(v_2)$

   (c) If r is labeled with $\vee$ and has the vertices v_1 and v_2 as children, then $\mathcal{F}(r) = \mathcal{F}(v_1) \vee \mathcal{F}(v_2)$

Example 4.1.1. The tree representing the formula $a \wedge (\neg a \vee b)$ is shown in figure 4.1.

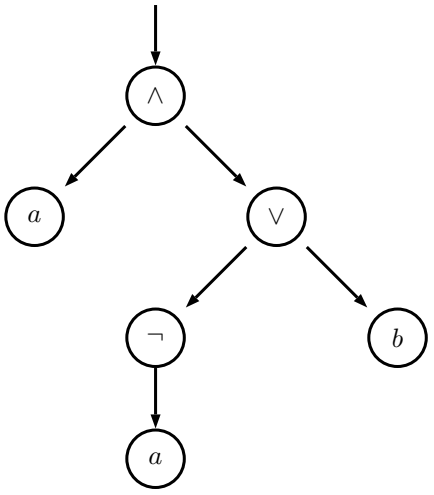


Figure 4.1: A directed tree representing the boolean formula $a \wedge (\neg a \vee b)$

4.1.2 Conversion into Equivalent CNF

As mentioned before, when the model checker needs to verify the satisfiability of an arbitrary formula using a SAT solver, this formula needs to be transformed into CNF. Let F be an arbitrary boolean formula. In order to produce a formula in CNF equivalent to F, one can apply the following procedure:

1. Iteratively apply De Morgan's laws to push the negation operations $\neg$ towards the literals of F.

$$\begin{aligned} \neg(\bigwedge_{i=1}^{n} F_i) &\longrightarrow \bigvee_{i=1}^{n} \neg F_i \\ \neg(\bigvee_{i=1}^{n} F_i) &\longrightarrow \bigwedge_{i=1}^{n} \neg F_i \end{aligned} \qquad \text{(De Morgan's laws)}$$

2. Iteratively apply the double negation rule to eliminate multiple negations and produce a formula in Negation Normal Form.

$$\neg\neg F \longrightarrow F \qquad \text{(Double negation rule)}$$

3. Iteratively apply the distribution of $\vee$ over $\wedge$ until the remaining formula consists of a conjunction of clauses.

$$f \vee (\bigwedge_{i=1}^{n} f_i) \longrightarrow \bigwedge_{i=1}^{n} (f \vee f_i) \qquad \text{(Distribution of } \vee \text{ over } \wedge\text{)}$$

4. Remove all tautological clauses, i.e. clauses containing both literals x and $\neg x$ for any boolean variable x, from the resulting conjunction.

Example 4.1.2. Let us consider the formula $F = \bigvee_{i=1}^{n}(x_i \wedge y_i)$ where $n > 0$. This formula is already in NNF, so that the two first steps of the above procedure can be skipped. The formula in CNF resulting from applying the third step is composed from the set of clauses of the form $\bigvee_{i=1}^{n} \alpha_i$ where for each $i \in \{1, \ldots n\}$, α_i is either x_i or y_i. The number of item in this set is equal to the number of arrangements of two elements in an ordered sequence of length n, i.e. 2^n. Also, each clause is of size n. The size of the original formula is therefore of size $\mathcal{O}(n)$ while the resulting clause database is of size $\mathcal{O}(n.2^n)$.

As Example 4.1.2 shows, the application of the aforementioned procedure can lead to a possible blow-up of the size of the generated CNF. This actually illustrates the inherent complexity of this approach. As mentioned in Section 1.2.2, a formula in CNF is valid if, and only if, it is reduced to the empty conjunction of clauses. As the validity problem is dual with the satisfiability problem, i.e. any boolean formula F is satisfiable if, and only if, $\neg F$ is not valid, it is easy to reduce the satisfiability problem to the application of the above procedure. Let F be an arbitrary boolean formula. Assume we negate it and use the defined procedure to obtain the formula in CNF $(\neg F)_{cnf}$. Then F is satisfiable if, and only if, $(\neg F)_{cnf}$ is not reduced to empty conjunction. In this case, a satisfying assignment for F is easy to find as an unsatisfying one for a clause of $(\neg F)_{cnf}$. This has mainly two consequences. First, the problem of finding a formula in CNF equivalent to an arbitrary boolean formula is NP-hard. Second, it is pointless to use a SAT solver on a formula in CNF F_{cnf} to determine the satisfiability of an equivalent formula F, as the SAT problem can easily be reduced to the problem of finding F_{cnf}. To workaround this, it is necessary to consider the notion of equisatisfiability.

4.1.3 Equisatisfiability and Tseitin Transformation

Two boolean formulas F_1 and F_2 are *equisatisfiable* if F_1 is satisfiable if, and only if, F_2 is satisfiable. However, their sets of satisfying assignments might not correspond. As a matter of fact, F_1 and F_2 could not even be defined on the same set of boolean variables.

Example 4.1.3. Let F be an arbitrary boolean formula over the set of variables X. Then the formula $F' = F \wedge x$, where x is a boolean variable such that $x \notin X$, is equisatisfiable with F. Indeed, A is a satisfying assignment for F if, and only if, $A \cup \{x \to \top\}$ is a satisfying assignment for F'.

Given an arbitrary boolean formula F, it is possible to find a formula in CNF F_{cnf} that is equisatisfiable with F. Such transformation has first been introduced by Tseitin [Tse68]. It is now often referred to as the *Tseitin Transformation*. This transformation can be described by a recursive procedure as follows. The procedure applies to a boolean formula and returns a new literal. If F, F_1 and F_2 are boolean formulas such that $F = F_1 \circ F_2$, where $\circ$ is any binary boolean operator, the Tseitin Transformation is first applied to both F_1 and F_2, introducing two literals l_{F_1} and l_{F_2}. A new literal l_F is then created such that $l_F \Leftrightarrow l_{F_1} \circ l_{F_2}$. This last equivalence is transformed into a set of clauses and l_F is returned by the procedure. The constraint $l_F \Leftrightarrow l_{F_1} \wedge l_{F_2}$ can be transformed into

$$\begin{aligned}&(\neg l_F \vee l_{F_1})\\ \wedge&(\neg l_F \vee l_{F_2})\\ \wedge&(l_F \vee \neg l_{F_1} \vee \neg l_{F_2})\end{aligned}$$

which can be interpreted as

$$\begin{aligned}&(l_F \Rightarrow l_{F_1})\\ \wedge&(l_F \Rightarrow l_{F_2})\\ \wedge&((l_{F_1} \wedge l_{F_2}) \Rightarrow l_F)\end{aligned}$$

which translates correctly the semantics of conjunction. l_F evaluates to $\top$ if, and only if, both l_{F_1} and l_{F_2} also do. The constraint $l_F \Leftrightarrow l_{F_1} \vee l_{F_2}$ can be transformed into

$$\begin{aligned}&(l_F \vee \neg l_{F_1})\\ \wedge&(l_F \vee \neg l_{F_2})\\ \wedge&(\neg l_F \vee l_{F_1} \vee l_{F_2})\end{aligned}$$

which can be interpreted as

$$\begin{aligned}&(l_{F_1} \Rightarrow l_F)\\ \wedge&(l_{F_2} \Rightarrow l_F)\\ \wedge&(l_F \Rightarrow (l_{F_1} \vee l_{F_2}))\end{aligned}$$

which translates correctly the semantics of disjunction. l_F evaluates to $\top$ if, and only if, either l_{F_1} or l_{F_2} also does. If F and F' are boolean formulas such that $F = \neg F'$, the procedure is first applied to F', introducing the literal $l_{F'}$, and $\neg l_{F'}$ is returned. If F is a boolean formula of the form $F = x$, where x is a boolean variable, then x is simply returned by the procedure. Finally, if F is a boolean formula of the form $F = v$, where $v \in \mathbb{B}$, then a fresh literal l_F is created. If $v = \bot$, the clause $\neg l_F$ is generated. If $v = \top$, the clause l_F is generated. In both case, the literal l_F is returned. The clause database equisatisfiable to the formula F is obtained by adding the unit clause l_F to the set of clauses previously produced by the procedure.

The overall process is formalized in Algorithm 10. This one calls the *produceClauses* function defined in Algorithm 11. Both these algorithms work under the assumption that the formula is represented as a tree, as explained in section 4.1.1. The function *newLiteral* returns a fresh literal while the procedure *addClause* adds a clause to the resulting formula in CNF.

Algorithm 10 Tseitin Transformation

tseitin(**Vertex** v)
  Literal $l \leftarrow$ produceClauses(v)
  addClause(l)

Example 4.1.4. Let us consider again Example 4.1.1. The result of the application of Algorithm 10 on the formula $a \wedge (\neg a \vee b)$ is presented in Figure 4.2. Each vertex is associated with the literal returned by the function *produceClauses* and the generated clauses are shown in dashed rectangles. The formula in CNF produced by the Tseitin Transformation is then

$$v_1 \quad \wedge (\neg v_1 \vee a) \quad \wedge (\neg v_1 \vee v_2) \quad \wedge (v_1 \vee \neg a \vee \neg v_2)$$
$$\wedge (v_2 \vee a) \quad \wedge (v_2 \vee \neg b) \quad \wedge (\neg v_2 \vee \neg a \vee b)$$

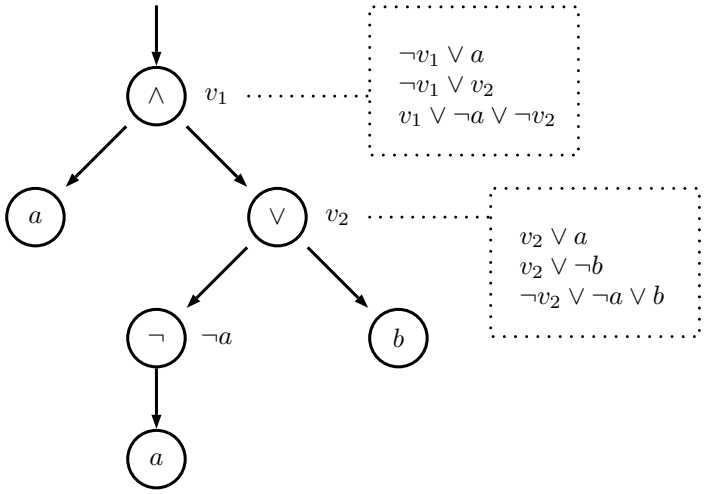


Figure 4.2: Application of Algorithm 10 on the boolean formula $a \wedge (\neg a \vee b)$

Example 4.1.4 clearly shows that the Tseitin Transformation yields formulas of increased size. However, the procedure goes through each sub-formula of the original formula F only once, and produces, for any one of them, at most three clauses containing at most three literals. Consequently, its worst-case time complexity, as well as the size of the output formula, are linear with respect to the size of F, i.e. in $\mathcal{O}(\|F\|)$, where $\|F\|$ represents the size of F. Furthermore, as additional variables are introduced during the process, it is quite clear that the formula F and its result by the Tseitin Transformation F_{cnf} are not strictly equivalent, but only equisatisfiable. However, their correlation is strong as a satisfying assignment for F can be retreived from any satisfying assignment for F_{cnf} by removing all variables introduced by the transformation. This transformation is effectively usable to produce formulas in CNF when solving the satisfiability problem with a SAT solver in ProVeLines. Still, its efficiency can be further improved.

Algorithm 11 Clause Production for Tseitin Transformation

```
Literal produceClauses(Vertex v)
  Literal result
  if v is a terminal vertex then
    if v is labeled with ⊥ then
      Literal l ← newLiteral()
      addClause(¬l)
      result ← l
    else if v is labeled with ⊤ then
      Literal l ← newLiteral()
      addClause(l)
      result ← l
    else if v is labeled with a variable x then
      result ← x
    end if
  else if v is a non-terminal vertex then
    if v is labeled with ¬ then
      Literal l ← produceClauses(child(v))
      result ← ¬l
    else if v is labeled with ∧ then
      Literal l ← newLiteral()
      Literal l1 ← produceClauses(leftChild(v))
      Literal l2 ← produceClauses(rightChild(v))
      addClause(¬l ∨ l1)
      addClause(¬l ∨ l2)
      addClause(l ∨ ¬l1 ∨ ¬l2)
      result ← l
    else if v is labeled with ∨ then
      Literal l ← newLiteral()
      Literal l1 ← produceClauses(leftChild(v))
      Literal l2 ← produceClauses(rightChild(v))
      addClause(l ∨ ¬l1)
      addClause(l ∨ ¬l2)
      addClause(¬l ∨ l1 ∨ l2)
      result ← l
    end if
  end if
  return result
```

4.1.4 Graph Representation of Boolean Formulas

Instead of directed trees, Directed Acyclic Graphs (DAGs) can be used to represent boolean formulas. Graphs are useful to avoid the redundancy that occurs in the presence of isomorphic sub-graphs. Two graphs are isomorphic if their roots also are. Let v_1 and v_2 be two vertices of a graph. Then, v_1 and v_2 are isomorphic if, and only if, one of the following conditions is satisfied:

1. v_1 and v_2 are both terminal vertices and are labeled with the same value, i.e. either $\bot$, $\top$, or a boolean variable x

2. v_1 and v_2 are both non terminal vertices, are labeled with the same operator and their children are also isomorphic

The isomorphism of non-terminal nodes is therefore a recursive property: it can only be determined on the basis of their children. In order to avoid redundancy, we make the hypothesis that the following restriction must hold on every graph representing a formula: Distinct but isomorphic sub-graphs are forbidden. This effectively means that two isomorphic sub-graphs must always share the same root vertex and, consequently, the same data structure. This can be ensured at build time by a so-called *unique table*. In a nutshell, when the application program needs a new vertex, it first looks in this table to find a vertex with the same label and children. If such node already exists, it is reused. If not, a new node is created and inserted into the unique table. Under the hypothesis that all existing nodes are isomorphic if, and only if, they are equal, a new node is indeed isomorophic to a previously created one if they share the same label and children.

One should note that we first found the description of a unique table in the context of an efficient BDD implementation [BRB90].

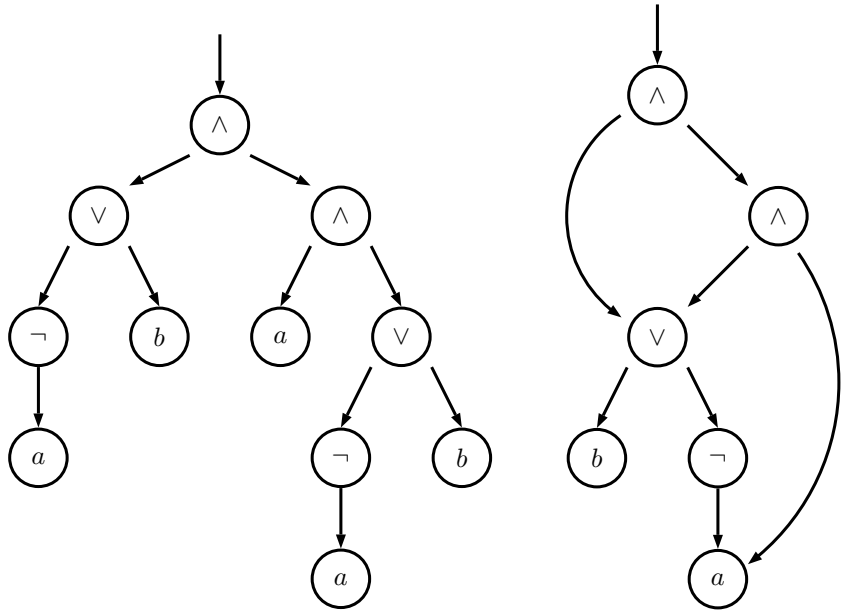
Example 4.1.5. Let us consider the formula $F = (\neg a \vee b) \wedge (a \wedge (\neg a \vee b))$. Figure 4.3a shows the formula represented as a tree while Figure 4.3b shows the same one represented as a DAG on which the mentioned restriction holds.

4.1.5 Cache Mechanism

The graph representation of boolean formulas can be used to reduce the size of a formula in CNF produced by the Tseitin Transformation. The mechanism is fairly simple. The procedure memorizes the literal that it outputs for each vertex during the transformation. When a vertex is encountered for the second time during the graph traversal, the memorized literal is reused. The process is formalized in Algorithm 12.

Example 4.1.6. Let us consider again Example 4.1.5. The results of the application of Algorithm 12 on both graphs are presented in Figure 4.4. The resulting formula of the process presented in Figure 4.4a is

$$
\begin{array}{lll}
v_4 \wedge (\neg v_4 \vee v_1) & \wedge(\neg v_4 \vee v_3) & \wedge(v_4 \vee \neg v_1 \vee \neg v_3) \\
\wedge (v_1 \vee a) & \wedge(v_1 \vee \neg b) & \wedge(\neg v_1 \vee \neg a \vee b) \\
\wedge (\neg v_3 \vee a) & \wedge(\neg v_3 \vee v_2) & \wedge(v_3 \vee \neg a \vee \neg v_2) \\
\wedge (v_2 \vee a) & \wedge(v_2 \vee \neg b) & \wedge(\neg v_2 \vee \neg a \vee b)
\end{array}
$$



(a) Tree (b) Graph

Figure 4.3: Two representation for the boolean formula $(\neg a \vee b) \wedge (a \wedge (\neg a \vee b))$

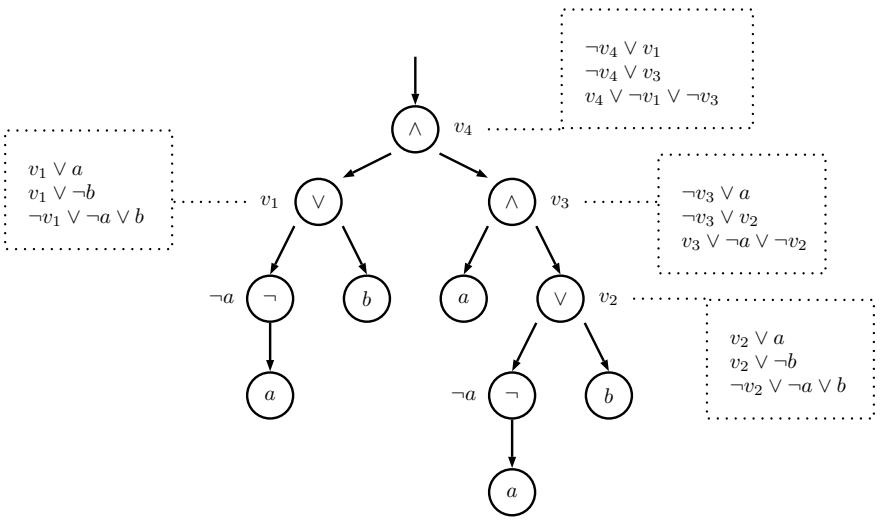
Algorithm 12 Clause Production for Tseitin Transformation (Caching Version)

```
Literal produceClauses(Vertex v)
  if contains(cache, v) then
    return cache[v]
  end if
  Literal result

                    ⟨ same as in Algorithm 11 ⟩

  cache[v] ← result
  return result
```

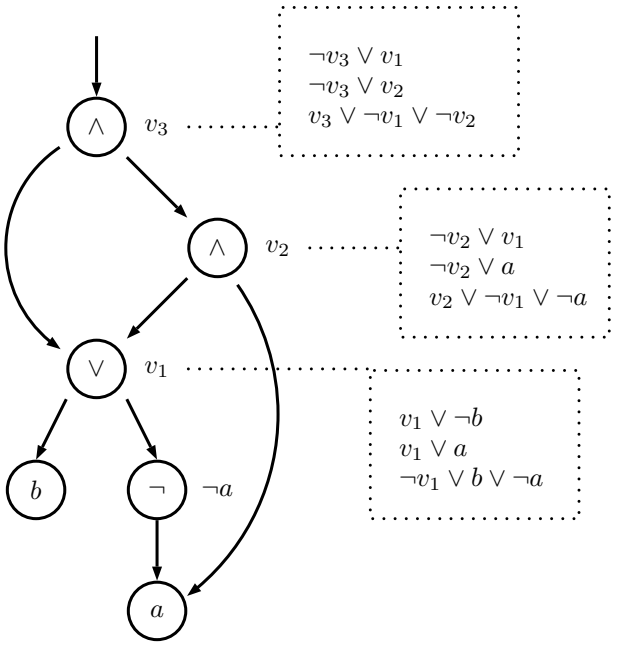
The resulting formula of the process presented in Figure 4.4b, on the other hand, is

$$\begin{array}{lll} v_3 \wedge (\neg v_3 \vee v_1) & \wedge(\neg v_3 \vee v_2) & \wedge(v_3 \vee \neg v_1 \vee \neg v_2) \\ \wedge (\neg v_2 \vee v_1) & \wedge(\neg v_2 \vee a) & \wedge(v_2 \vee \neg v_1 \vee \neg a) \\ \wedge (v_1 \vee \neg b) & \wedge(v_1 \vee \neg a) & \wedge(\neg v_1 \vee b \vee \neg a) \end{array}$$

In this example, the cache mechanism allows one to reduce the number of clauses from 13 to 10.



(a) Tree



(b) Graph

Figure 4.4: Application of Algorithm 12 on the two graphs of Example 4.1.5

4.2 Incremental Solving

In addition to defining an efficient clause production process, we designed some modifications in the model-checking algorithm introduced in Section 3.3.2 to be more consistent with the capabilities of a SAT solver.

4.2.1 Principle of Incremental Solving

When model checking the FTS of a software product line against an LTL property, the process presented in Algorithm 7 solves the satisfiability problem over a large set of closely related formulas. It actually accumulates constraints when taking transitions, forgets them when backtracking, and performs SAT checks for every intermediate formulas obtained following this strategy.

To illustrate this, let us consider a featured transition system $FTS = \langle S, Act, Trans, I, AP, L, d, \gamma \rangle$ such that

1. $\{S_1, S_2, S_3, S_4\} \subseteq S$
2. $\{(S_1, \alpha_1, S_2), (S_2, \alpha_2, S_3), (S_2, \alpha_3, S_4)\} \subseteq Trans$
3. $\gamma(S_1, \alpha_1, S_2) = f_1$, $\gamma(S_2, \alpha_2, S_3) = f_2$, $\gamma(S_2, \alpha_3, S_4) = f_3$

These requirements define the part of FTS illustrated in Figure 4.5. Let us suppose that Algorithm 7 first reaches the state S_1. At this stage, it has to solve the SAT problem over a formula F. Following the path going through S_3, the process then accumulates the feature expressions labeling the available transitions to consecutively create and solve the problems $F \wedge f_1$ and $F \wedge f_1 \wedge f_2$. When it has fully expanded this path, the process backtracks and takes the second visible path. Therefore, it also solves the SAT problem over $F \wedge f_1 \wedge f_3$.

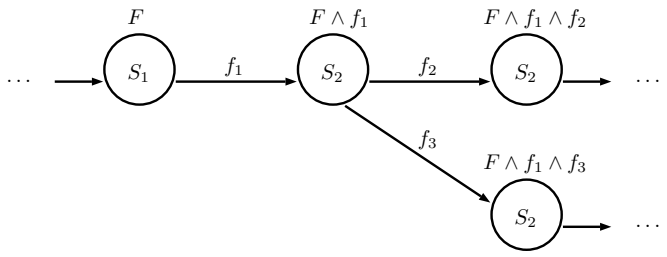


Figure 4.5: Accumulation of constraints in Algorithm 7

Let us assume that we use a SAT solver to solve the mentioned SAT problems. Then, it is inefficient to transform, at each step, the accumulated formula into a formula in CNF. Instead, it is preferable to transform the additional encountered feature expressions into a set of clauses, and to add or retrieve them to and from a managed clause database throughout the exploration. This way of proceeding indeed allows one to

1. Avoid the repetitive complete applications of the clause production process to a set of closely related formulas, which leads to repetitively create the same subsets of clauses

2. Reuse the cached results of previous transformations in the cache mechanism presented in Section 4.1.5

Furthermore, the chosen SAT solver could allow the addition and removal of clauses. This additional capability enables the reuse of the acquired knowledge of previous runs, as conflict clauses and activity scores, from one SAT check to another. To this end, several strategies of clauses management have been proposed, two of the most common being :

1. Clauses can be added and removed on demand. This strategy requires to maintain traceability links between original clauses and learned clauses, so that when a clause C is retrieved, all learned clauses induced by the presence of C during the analysis process can also be deleted. This approach was introduced by Whittemore et al. along with the SATIRE SAT solver [WKS01].

2. A *solve under assumptions* function is provided by the SAT solver. This procedure allows one to solve the satisfiability problem under the assumption that an arbitrary set of literals are assigned to $\top$. This comes down to the problem of finding a satisfying assignment for a formula that is an extension of an arbitrary partial assignment. This technique was introduced by Eén and Sörenson with the Minisat solver [ES03a]. The way in which this method can be used to virtually remove sets of clauses is explained in the next section.

Henceforth, we call *incremental solving* the technique that consists in solving the SAT problem over a set of closely related formulas by addition and retrieval of clauses.

4.2.2 Solving Under Assumptions

Let us suppose that we want to add a clause C to the clause database of a SAT solver and to remain able to remove it later. If the solver can solve the satisfiability problem under a set of assumptions, then it is possible to create a fresh literal l, add it to this set of assumptions, and insert the clause $C' = C \vee \neg l$ to the clause database. Under the new assumption, the clause C' will indeed behave like the clause C. When we need to remove this clause, l is removed from the list of assumptions and the unit clause $\neg l$ is added to the clause database. This clause will propagate the variable assignment $\{l \rightarrow \bot\}$ which satisfies the clause C'. From there, this clause can be safely ignored, or removed, by the solver. Obviously, for this to work correctly, the literal l needs be used solely to identify a clause, or a set of clauses, and to remove it. One should note the simplicity of the process, as all learned clauses are here safe to keep and that no traceability link is needed. This way of virtually add and remove clauses is presented in [ES03b].

Let us suppose that we use the Tseitin-Transformation. Another way to solve under assumptions can be used. This method relies on the fact that all intermediate variables introduced in Algorithm 11 are said to be *functionally dependent* on the original variables [ES03b]. Let F be a formula over a set of variable X, $F_{tseitin}$ the set of clauses produced by Algorithm 11 applied on F, I the set of intermediate variables introduced by the process, and $l \in I$

the returned literal. Then, any complete variable assignment on F, be it satisfying or unsatisfying, can be extended in exactly one satisfying assignment for $F_{tseitin}$. This means that, given an arbitrary assignment over the variables of X, the boolean values assigned to the elements of I are implied by $F_{tseitin}$. Consequently, F and $F_{tseitin}$ are obviously not equisatisfiable. Only the formula $F' = F_{tseitin} \wedge l$ is equisatisfiable with F by reducing the number of satisfying assignments. As a consequence, adding the set of clauses $F_{tseitin}$ to the clause database of a SAT solver does not modify, on its own, the set of satisfying assignment of it. However, solving under the assumption l acts as if the formula F was fully asserted. Using this property for incremental solving is quite straightforward. When F needs to be added to the set of constraint, $F_{tseitin}$ is inserted to the clause database and l to a managed set of assumptions. When backtracking, l is simply retrieved from the the mentioned set of assumptions, while $F_{tseitin}$ remains in the clause database. As these clauses cannot be removed by the SAT solver, it could considerably increase the size of its clause database. However, this approach leverage the cache mechanism presented in Section 4.1.5 by allowing the reuse of old clauses. Consequently, we believe than it could considerably fasten the model-checking process, notably when revisiting previously visited states.

4.2.3 Incremental Algorithms

As such, Algorithms 8 and 9 cannot benefit from incremental solving. To workaround this, we defined new model-checking algorithms meant to use incremental solving. The interface provided by the *Solver* class we defined is presented in Figure 4.6. The *addAssertion* method allows one to add a formula in the clause database of the SAT solver. The *push* method is used to create backtracking points while the *pop* method revert the state of the SAT solver to the last created backtracking point. This *push*/*pop* interface is inspired from the APIs of SMT solvers like Z3 [DMB08] and STP [GD07]. The *addContraint* method adds a formula into a set of background constraints that are not impacted by the creation and deletion of backtracking points. This set of managed constraints is meant to include the feature diagram d and the set of violating products *badProducts* of Algorithms 8 and 9, which must not be impacted when backtracking. The resulting algorithms are presented in Algorithms 13 and 14. These algorithms are very close to the original algorithms. Feature expressions accumulated during the state traversal problem are however managed through the solver interface. The feature expressions are asserted into the solver when taking transitions. When backtracking, instead of retrieving a feature expression from the stack, the state of the solver is returned to the last created backtracking point.

4.2.4 SMT Solvers and Incremental Solving

As mentioned before, the interface presented in Figure 4.6 is inspired from the APIs of SMT solvers like Z3 [DMB08] and STP [GD07]. Consequently, it is easy to use this interface as a wrapper around an SMT solver. Moreover, these SMT solvers also provide built-in representations for boolean-valued formulas. Therefore, the algorithms introduced in the previous section can be reused as such when using an SMT solver instead of a SAT solver.

```
/**
 * Modifies: this
 * Effects:  Creates a backtracking point
 */
void push()

/**
 *  Requires: There exists at least one
 *            backtracking point
 *  Modifies: this
 *  Effects:  Backtrack to the last created
 *            backtracking point and delete it
 */
void pop()

/**
 *  Modifies: this
 *  Effects:  Asserts the formula 'assertion'
 */
void addAssertion(Formula assertion)

/**
  *  Returns: A formula representing all the
  *           assertions made
  */
Formula getAssertions()

/**
 *  Returns: true iff the conjunction of all the
 *           assertions made is satisfiable with
 *           relation to the set of constraints
 */
Boolean solve();

/**
 *  Modifies: this
 *  Effects: Adds the constraint 'constraint' to
 *  the set of constraints
 */
void addConstraint(Formula constraint)
```

Figure 4.6: Solver class interface

Algorithm 13 Incremental Featured Outer Depth-First Search

```
FeatureExp incremental-featured-outer-search(
State s_0)
  Stack outerStack ← emptyStack()
  Set visitedInOuter ← ∅
  Map outerTable ← emptyTable()
  Set visitedInInner ← ∅
  Map innerTable ← emptyTable()
  FeatureExp badProducts ← ⊥
  push(outerStack, (s_0, ∅))
  visitedInOuter ← visitedInOuter ∪ {s_0}
  outerTable[s_0] ← ⊤
  Solver solver
  solver.addConstraint(d)
  solver.push()
  while not empty(outerStack) do
    (s, T) ← top(outerStack)
    if ∃s', α : ((s, α, s') ∈ Trans') ∧ ((s, α, s') ∉ T) then
      T ← T ∪ {(s, α, s')}
      pop(outerStack)
      push(outerStack, (s, e, T))
      solver.push()
      solver.addAssertion(γ(s, α, s'))
      if solver.solve() then
        if s' ∈ visitedInOuter then
          solver.addAssertion(¬outerTable[s'])
          if solver.solve() then
            push(outerStack, (s', ∅))
            outerTable[s'] ← solver.getAssertions()
          else
            solver.pop()
          end if
        else
          push(outerStack, (s', ∅))
          visitedInOuter ← visitedInOuter ∪ {s'}
          outerTable[s'] ← solver.getAssertions()
        end if
      else
        solver.pop()
      end if
    else
      if L'(s) ⊆ F then
        badProducts ← featured-inner-search(
        s, solver, visitedInInner, innerTable, outerStack, badProducts)
      end if
      pop(outerStack)
      solver.pop()
    end if
  end while
  return badProducts
```

Algorithm 14 Incremental Featured Inner Depth-First Search

```
boolean incremental-featured-inner-search(
State s_0, Solver solver,
Set visitedInInner, Map innerTable,
Stack outerStack, FeatureExp badProducts)
  Stack innerStack ← emptyStack()
  push(innerStack, (s_0, ∅))
  visitedInInner ← visitedInInner ∪ {s_0}
  innerTable[s_0] ← solver.getAssertions()
  solver.push()
  while not empty(innerStack) do
    (s, T) ← top(innerStack)
    if in(s, outerStack) then
      badProducts ← badProducts ∨ solver.getAssertions()
      solver.addConstraint(¬solver.getAssertions())
      pop(innerStack)
      solver.pop()
    else
      if ∃s', α : ((s, α, s') ∈ Trans') ∧ ((s, α, s') ∉ T) then
        T ← T ∪ {(s, α, s')}
        pop(innerStack)
        push(innerStack, (s, T))
        solver.push()
        solver.addAssertion(γ(s, α, s'))
        if solver.solve() then
          if s' ∈ visitedInInner then
            solver.addAssertion(¬innerTable[s'])
            if solver.solve() then
              push(innerStack, (s', ∅))
              innerTable[s'] ← solver.getAssertions()
            else
              solver.pop()
            end if
          else
            push(innerStack, (s', ∅))
            visitedInInner ← visitedInInner ∪ {s'}
            innerTable[s'] ← solver.getAssertions()
          end if
        else
          solver.pop()
        end if
      else
        pop(innerStack)
        solver.pop()
      end if
    end if
  end while
  return badProducts
```

Conclusion

In this chapter, we introduced several techniques that can be used to efficiently integrate a SAT solver in ProVeLines. Among others, we introduced model-checking algorithms that are meant to use incremental solving. It is noteworthy that these algorithms can be reused as such when working with SMT solvers. The goal of the next chapter is twofold. First, we want to measure the performance of ProVeLines with a SAT solver. Second, we want to evaluate the effectiveness of Algorithms 13 and 14 with both SAT and SMT solvers.

Chapter 5

Experimental Results

Now that we have introduced an approach to efficiently integrate a SAT solver in ProVeLines, we undertake a concrete empirical study. We mainly have two objectives:

1. Measure the performance of ProVeLines with a SAT solver
2. Evaluate the efficiency of the incremental algorithms introduced in Section 4.2.3, with both SAT and SMT solvers

To do so, we performed several implementation tasks. This chapter first goes through the description of the choices and rationale leading these tasks and, in a second time, undertakes a concrete empirical evaluation.

5.1 Implementation

In order to use its extensible architecture at our best convenience, and in agreement with the current maintainers of the tool, we decided to integrate our changes to ProVeLines 2. However, this choice had a major drawback. The lack of FSTM models available rendered it difficult to rely solely on this input language to realize a deep empirical study. For this reason, we also integrated the fPromela input language of ProVeLines 1 in ProVeLines 2. This section describes these two implementation tasks.

5.1.1 Integrating SAT and SMT solvers

Regarding SAT solvers, we choose to work with the Minisat project [ES03a]. Minisat has been conceived as a minimalistic SAT solver written in C++. Its original goal was to demonstrate the implementation details of a modern SAT solver and to serve as a basis for further research. The original source code was about 600 lines of code. Minisat is today considered as a milestone in the history of SAT and SMT solvers and has served as the starting point of different other projects as Glucose [AS09] and CVC4 [BCD+11]. For our purpose, we used the 2.2 version, which is the latest available update.

As for SMT solver, we decided to integrate the Z3 solver [DMB08]. Z3 is an SMT solver developed by Microsoft Research based on the DPLL(T) architecture. It is often considered as the most advanced project of its category. It was

also used during the first attempt of integration of an SMT solver in ProVeLines 1 [CSHL13].

Figure 5.1 shows the result of these two additions upon the architecture of ProVeLines 2, more precisely in the *math* package. The most noticeable modification is the appearance of the *BoolSolver* interface, which is the one presented in Section 4.2.3. This interface allows one to make use of incremental solving during the model-checking process. For the rest, the integration of both Minisat and Z3 was rendered rather straightforward due to the extensible architecture of ProVeLines 2, consisting mainly in providing concrete implementations for the three interfaces *BoolFct*, *BoolSolver* and *BoolFctFactory*. The concrete implementation using Minisat also makes use of a newly integrated package, called *dag*, implementing helper classes to build and manipulate boolean formulas as graphs.

Two new model-checking algorithms, based on incremental solving, have been implemented in the *checker* package. The result is presented in Figure 5.2. These algorithms make use of the interface of the newly introduced *FeatureSolver* class, which is basically a wrapper around the *BoolSolver* interface.

From this work, we defined six different configurations for ProVeLines 2, depending on how boolean functions are represented and which model-checking algorithms are used:

Cudd: This configuration relies on the BDD implementation from the CUDD package.

Minisat: This configuration relies on the Minisat solver. Boolean formulas are represented by DAGs as presented in Section 4.1.4, but the Tseitin transformation is applied without any caching mechanism. Moreover, incremental solving is not used.

Cache: This configuration is similar to the previous one. However, the Tseitin transformation is applied using the cache mechanism introduced in Section 4.1.5.

Inc: This configuration is similar to the previous one, but this time makes use of incremental solving.

Z3: This configuration relies on the Z3 SMT solver and does not use incremental solving.

Zinc: In this configuration, the Z3 solver and incremental solving are both used.

The Z3 set of classes introduced in this section actually implement the *BoolFct*, *BoolSolver* and *BoolFctFactory* interfaces. Consequently, our Z3-based implementation only allows one to build and manage boolean formulas. It is clearly not the goal of an SMT solver to be used to solve solely boolean problems. However, for the purpose of our experiment, this is sufficient.

5.1.2 Integrating fPromela in ProVeLines 2

Integrating the fPromela input language into ProVeLines 2 has required a large amount of work, although the high-level architecture resulting from it is quite simple. The main rationale behind it was the desire to reuse as much code

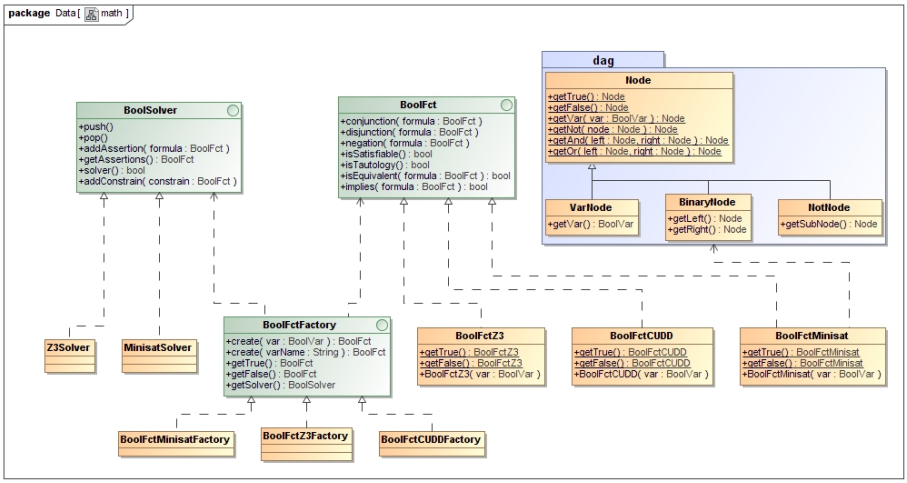


Figure 5.1: Class diagram describing the architecture integrating Minisat and Z3 into the *math* package

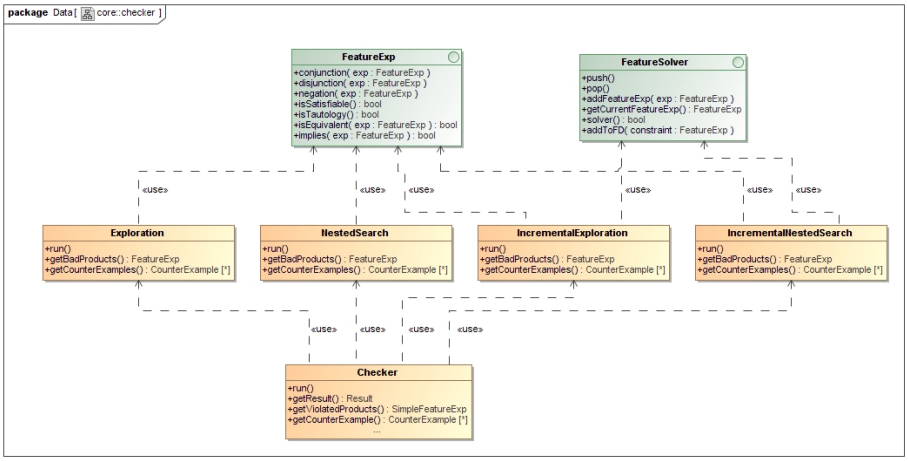


Figure 5.2: Class diagram describing the architecture integrating incremental algorithms

as possible from the first version of the tool. The solution basically consists in isolating the C code from ProVeLines 1 dedicated to the parsing and management of this input language, and wrapping it in a C++ interface. Again, the architecture of ProVeLines 2 demonstrated its extensibility. It has been sufficient to implement the interfaces defined in the *fts* package to allow the whole project to integrate the input language. The result of this work is shown in Figure 5.3. The *fPromela* package contains the C++ wrapper classes while the *C* sub-package includes the old C code. One should note the similarities between the latter package and the architectural sketch presented in Figure 3.9. The classes from this package refer to the C structures used in ProVeLines 1. Since this code still need to access the *boolFct* structure, allowing to build and manipulate boolean functions, the code of this structure has been rewritten to consists of a wrapper around the C++ *BoolFct* interface.

In order to be consistent with the architecture of ProVeLines 2, the C code needed some major modifications. Among others, the *neverState* structure appeared as a fork from the *state* structure. Also, in order for this solution to work properly, the C++ code needs to be compiled as C++ code while the C code needs to be compiled as C code. The application program is then created by correctly linking the compiled codes. For this step to be executed properly, special preprocessor instructions need to be used to mark the C code called from C++ or the C++ code called from C.

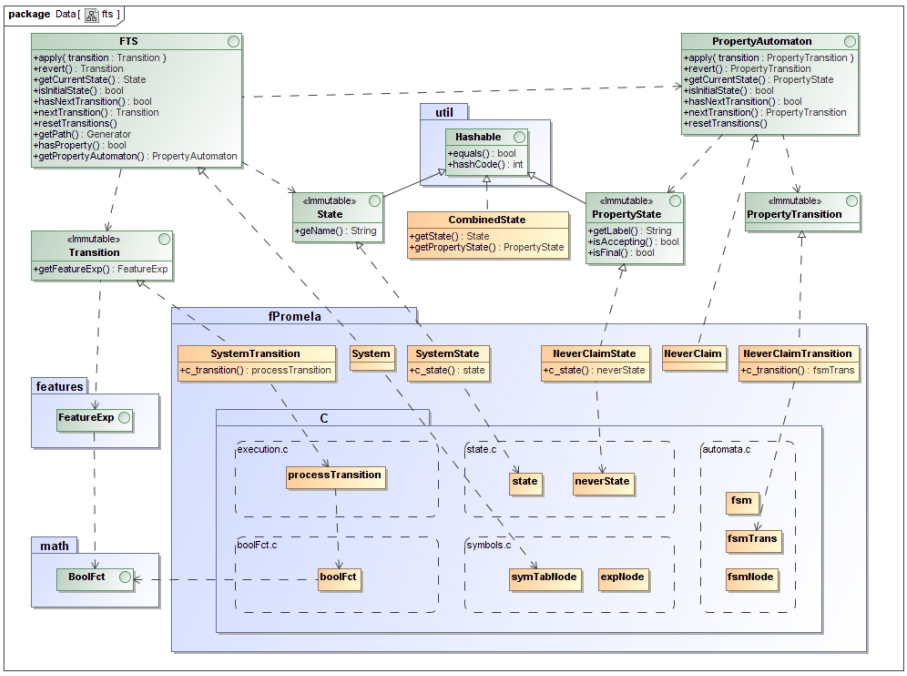


Figure 5.3: Class diagram describing the architecture integrating the fPromela input language

5.2 Empirical Evaluation

The objective of this section is to evaluate the performance of each of the six configurations introduced in Section 5.1.1 in terms of verification time. For the purpose of this evaluation, we used four different benchmarks.

mutex: This model describes a family of mutual exclusion algorithms. It is written in fPromela. It is the smallest benchmark available.

ano: This FSTM model, the only one available, consists of an anonymized file. The original model has been defined for an industrial partner of the University of Namur. Due to non-disclosure agreements between the University of Namur and the mentioned partner, we do not have any knowledge of its precise nature. We should however emphasis the atypical nature of the FTS it defined, presenting a small set of 28 states weaved together by no less than 626 transitions. At this time, we do not have access to its feature diagram. Therefore, during our experiment, all products are considered valid products.

minepump: This model describes a family of systems supposed to keep a mine shaft clear of water while avoiding methane related explosion. Its main component consists of a water pump which activates itself when the water reaches a defined threshold, but only if the methane level stays below a critical limit. It is written in fPromela.

cfdp: This model is based on a simplified version of the CCSDS (Consultative Committee for Space Data Systems) File Delivery Protocol (CFDP) recommended standard. It describes a scenario where two entities connected via a space data link want to share the content of an arbitrary file. It is written in fPromela.

For each one of these benchmarks, four properties have been selected and verified. Additionally, we performed an exhaustive exploration of the FTS without property. Such exploration allows one to verify the reachability of states or the absence of deadlocks. We provide statistics about these benchmarks in Figure 5.4. We provide the following statistics:

Size: Number of states defined by the FTS

Revisited: Number of revisited states during the exhaustive exploration process

Depth: The maximal depth reached by the depth-first search algorithm during the exhaustive exploration process

Features: The number of features defined in the feature diagram

The benchmarking process has been repeated six times, i.e. one time for each of the six configurations mentioned in Section 5.1.1. Execution times have been measured. A timeout was set to 3600 seconds. Memory usage was limited to 3 GB. The experiment has been run on a personal computer run by a processor Core i5 2450M (2.5GHz), 4GB of RAM and Ubuntu 13.10. The results are shown in Figure 5.5. In this table, "oom" stands for "out-of-memory".

Model	Size	Revisited	Depth	Features
mutex	287	116	89	4
ano	28	987	12	-
minepump	11,398	52,200	605	11
cfdp	2,384,616	2,622,355	648	13

Figure 5.4: Benchmarks statistics

5.2.1 Minisat-based Configurations

On the mutex benchmark, the Minisat configuration performs 9 to 12.5 times slower that the Cudd configuration. The Cache configuration takes on average 51% less time that the Minisat configuration. The Inc configuration almost behaves like the Cudd configuration. However, the last selected temporal property takes 72% more than time to be verified by the Inc configuration than by the Cudd configuration. The Minisat configuration falls out of memory on every other benchmarks. On the ano benchmark, the Cache configuration performs on average 149.75 times slower than the Cudd configuration. The Inc configuration performs on average 7.77 times slower that the Cudd configuration, but takes 95% less time than the Cache configuration. On the minepump benchmark, the Cache configuration performs on average 676.85 times slower than the Cudd configuration. The Inc configuration performs on average 202.10 times slower that the Cudd configuration, but takes 70% less time than the Cache configuration. On the cfdp benchmark, the Cache and Inc configuration both fall out of time.

5.2.2 Z3-based Configurations

On the mutex benchmark, the Z3 configuration performs on average 30.75 times slower than the Cudd configuration. The Zinc configuration performs on average 5.20 times slower that the Cudd configuration, but takes 83% less time than the Z3 configuration. On the ano benchmark, the Z3 configuration performs on average 994.21 times slower than the Cudd configuration. The Zinc configuration performs on average 179.47 times slower that the Cudd configuration and 23.10 times slower that the Inc configuration, but takes 82% less time than the Z3 configuration. On the minepump benchmark, the Z3 configuration performs on average 374.87 times slower than the Cudd configuration. The Zinc configuration performs on average 132.82 times slower that the Cudd configuration, but takes 65% less time than the Z3 configuration and 34% less time that the Inc configuration. Finally, one the cfdp benchmark, the Z3 and Zinc configurations both fall out of time during the exploration and on the fourth property. The Z3 configuration also timeout on the third property. On the two remaining properties, the Zinc configuration performs 97% faster than the Z3 configuration.

5.2.3 Observations

At this point, we can draw several observations:

1. The cache mechanism introduced to reduce the number of clauses produced when working with a SAT solver proves to have a huge beneficial impact.

2. The incremental algorithms also prove to have a huge beneficial impact, with both SAT and SMT solvers.

3. The performance of the SAT and SMT based configurations are not strongly correlated to the performance of the Cudd configuration. The Cudd configuration takes a similar time to verify the fourth property of the ano benchmark and the second property of the minepump benchmark, while the Inc configuration takes about 2000% more time. The performance of SAT and SMT based configurations seems to be more impacted by the size of the verified FTS.

4. The Z3-based configurations are slower than the Minisat-based configurations on small examples, i.e. on the mutex and ano benchmarks, but outperform them on large examples.

5. The five configurations embedding SAT and SMT solvers perform slower that the Cudd configuration, especially on large models. It is quite clear that those configurations scale really bad compared to the BDD-based implementation.

5.2.4 Limitation

It should be noted that the results observed in Figure 5.5 do not correlate with those published by Cordy [CSHL13]. For some reason, the SMT-based implementation of ProVeLines 1 is more efficient that the Z3 configuration we defined. However, the implementation of the Z3 configuration is highly similar to the one published by Cordy. This may require further investigation.

Conclusion

In this chapter, we showed the implementation details of the theoretical approach introduced in Chapter 4 and performed a concrete empirical evaluation. We showed that both the cache mechanism and the incremental model-checking algorithms we proposed have a huge beneficial impact on the performance of the tool. However, the use of SAT and SMT solvers suffers from a clear scalability issue compared to the BDD approach. This result allows us to state the following hypothesis: SMT solvers are inefficient with ProVeLines because SAT solvers also are.

Model	Property	Configuration					
		Cudd	**Inc**	**Cache**	**Minisat**	**Zinc**	**Z3**
mutex	exploration	20	20	80	180	110	3900
	property 1	20	20	100	250	80	2390
	property 2	30	30	160	310	160	3970
	property 3	50	60	410	600	230	7500
	property 4	110	190	430	1020	720	18,920
ano	exploration	160	1970	54,020	oom	45,110	367,630
	property 1	310	2570	42,670	oom	56,300	276,180
	property 2	410	2680	41,470	oom	58,840	256,450
	property 3	330	2690	41,430	oom	58,130	258,980
	property 4	910	3240	42,540	oom	57,900	268,540
minepump	exploration	1900	947,490	2,737,410	oom	621,800	950,630
	property 1	1210	143,670	731,280	oom	127,230	460,740
	property 2	970	65,610	225,670	oom	40,360	274,130
	property 3	2100	341,050	702,800	oom	146,180	620,660
	property 4	1450	236,420	1,119,120	oom	174,730	601,860
cfdp	exploration	84,860	timeout	timeout	oom	timeout	timeout
	property 1	13,660	timeout	timeout	oom	88,710	2,775,040
	property 2	8540	timeout	timeout	oom	68,450	1,957,940
	property 3	25,560	timeout	timeout	oom	218,960	timeout
	property 4	oom	timeout	timeout	oom	timeout	timeout

Figure 5.5: Benchmark results for six configurations of ProVeLines 2 against four distinct models. Time is given is milliseconds (ms). Timeout is set to 1h (3,600,000 ms). Memory usage is limited to 3 GB. "oom" stands for "out-of-memory".

Conclusion

The integration of an SMT solver into ProVeLines revealed a large time overhead in the verification process compared to a BDD implementation. Clearly, this problem is a complex one, mainly because it takes place in the crossroads of three research areas: computer-aided reasoning, and more especially BDDs and SMT solvers, software product lines, and model checking. Investigating the problem therefore require a good understanding of these three subjects, and assembling this knowledge constitutes itself a real challenge. In the first part of this thesis, we tried to present the knowledge and understanding we acquired about them during the last year. In the second part of this thesis, we tried to investigate this problem and to propose a more efficient integration approach.

As we showed in Chapter 2, SMT solvers are mainly build upon SAT solvers. For this reason, we decided to investigate the efficient integration of a SAT solver in ProVeLines 2. What seemed to be a very straightforward work at first actually revealed to be a really ponderous task. However, it was rewarding as some of the knowledge we acquired doing this job could be reused when working with SMT solvers. In particular, we defined incremental product-line model-checking algorithms, which are meant to better integrate themselves with both SAT and SMT solvers.

We then undertake a concrete empirical evaluation. We had two objectives. First, we wanted to measure the performance of ProVeLines 2 with a SAT solver. Second, we wanted to evaluate the actual efficiency of the incremental algorithms we introduced. This empirical study required a large amount of work. We integrated a SAT solver and a SMT solver in ProVeLines 2, and implemented the defined algorithms. Also, in order to benefit from a larger set of benchmarks, we integrated the fPromela input language of ProVeLines 1 into ProVeLines 2.

The results were both encouraging and disappointing. Indeed, the incremental algorithms that we defined clearly outperform the original algorithms of ProVeLines. However, both the SAT-based and SMT-based implementations we proposed, even using the incremental algorithms, are far slower than the BDD-based implementation. In particular, we noticed a huge scalability issue with large models.

Another disappointment come from the fact that the result we obtained with ProVeLines 2 do not correlate with the results obtained by Cordy et al. when working with ProVeLines 1 [CSHL13]. Our version was indeed far slower. At this moment, we do not have any explanation of that fact. This is clearly a limitation of our work. Another limitation is that we only tested our SMT-based implementation with boolean constraints. Moreover, we believe that it could have been beneficial to compare our work with other works combining formal

verification techniques and SAT solvers. Among others, we now believe that a strong correlation exists between our work and symbolic execution. Symbolic execution tools as KLEE [CDE08] exhaustively explore the state-space of a software process while representing and verifying the constraints over its variables using an SMT solver.

Still, incremental algorithms have proved to be efficient. Moreover, our results allow us to draw the following hypothesis: SMT solvers are inefficient with ProVeLines because SAT solvers also are. The performance issue of SMT solvers in this context has nothing to do with the theory-specific part of this kind of solvers. In this context, ProVeLines could benefit from some sort of SMT solvers where the boolean part would be managed by BDDs. We do believe that it might be an interesting topic for future work.

Bibliography

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[CBH11] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Sci. Comput. Program.*, 76(12):1130–1143, December 2011.

[CCH+12] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer (STTT), Springer-Verlag*, 14(5):589–612, 2012. DOI 10.1007/s10009-012-0234-1.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CHS+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems:

Efficient verification of temporal properties in software product lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[CHS+13] Maxime Cordy, Patrick Heymans, Pierre-Yves Schobbens, Amir Molzam Sharifloo, Carlo Ghezzi, and Axel Legay. Verification for reliable product lines. *CoRR*, abs/1311.1343, 2013.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[CSHL12] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 66–75, New York, NY, USA, 2012. ACM.

[CSHL13] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 472–481, Piscataway, NJ, USA, 2013. IEEE Press.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[ES03a] Niklas Een and Niklas Sörensson. An extensible sat-solver [ver 1.2], 2003.

[ES03b] Niklas Een and Niklas Sörensson. Temporal induction by incremental sat solving, 2003.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Cambridge, MA, USA, 1974.

[Gan07] Vijay Ganesh. *Decision procedures for bit-vectors, arrays and integers*. PhD thesis, Stanford University, 2007.

[Gan13a] Vijay Ganesh. First order logic: Syntax and semantics. ECE750T-28: Computer-aided Reasoning for Software Engineering, 2013.

[Gan13b] Vijay Ganesh. Overview of first-order theories. ECE750T-28: Computer-aided Reasoning for Software Engineering, 2013.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of Computer Aided Verification (CAV)*, 2007.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. pages 175–188. Springer, 2004.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[HS07] Hyojung Han and Fabio Somenzi. Alembic: An efficient algorithm for cnf preprocessing. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 582–587, New York, NY, USA, 2007. ACM.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, February 2011.

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sha38] Claude Elwood Shannon. *A Symbolic Analysis of Relay and Switching Circuits.* 1938.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *RE*, pages 136–145. IEEE Computer Society, 2006.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. Satire: A new incremental satisfiability engine. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 542–545, New York, NY, USA, 2001. ACM.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.