UNED Escuela Internacional de Doctorado EIDUNED

TESIS DOCTORAL

2019/20

Supporting the Statistical Analysis of Variability Models by Processing Binary Decision Diagrams

Sergio Bra Gutiérrez

PROGRAMA DE DOCTORADO EN INGENIERÍA DE SISTEMAS Y CONTROL

Rubén Heradio Gil
David Fernández Amorós

Acknowledgements

I want to give thanks to my family and my partner Anabel for their moral support and the economical effort made that allows me to grow.

To Rubén and David, for each advice, suggestion and for encouraging me during all these years.

Contents

List of Figures

List of Tables

List of Algorithms

Resumen

Cuando uno de los principales objetivos de la Ingeniería de Software es el ahorro en costes y tiempo, las líneas de productos de software juegan un papel fundamental. En este campo, la clave reside en la identificación de componentes reutilizables o características que puedan ser aplicados en futuros proyectos.

La necesidad de proveer soluciones que se adapten a problemas específicos, satisfaciendo una amplia variedad de requerimientos no funcionales, como la eficiencia en tiempo de ejecución, consumo de memoria del dispositivo en el que se ejecuta determinada aplicación, requisitos a nivel de seguridad, etc. ha supuesto una ardua labor de investigación. La personalización se logra a través de un proceso de configuración donde se seleccionan las características deseadas. El espacio está restringido para evitar incompatibilidades entre características, garantizando que se satisfagan las dependencias definidas entre ellas. Hay muchas preguntas interesantes acerca de este espacio restringido: ¿Están todas las configuraciones libres de error? ¿Hay código muerto que no pueda ser ejecutado por las restricciones entre las características? ¿Cuál es rango de reutilización de componentes? ¿Cuál es el tamaño típico de un producto final en términos de sus componentes? Etc.

El tamaño del espacio de configuración potencialmente puede ser 2^n para n características. Así, procesar el conjunto completo de soluciones es imposible excepto para los casos más triviales. Después de eliminar las configuraciones no válidas que no satisfacen las restricciones entre características, el espacio resultante se reduce sustancialmente. Sin embargo, normalmente sigue siendo demasiado grande como para poder trabajar con él.

Una estrategia para abordar estas dificultades es seleccionar una muestra representativa del espacio y, una vez que los cálculos se han realizado, extrapolar las conclusiones al conjunto completo. En este punto, el problema reside en la obtención de una muestra aleatoria fiable para evitar interpretaciones erróneas.

Esta tesis presenta un conjunto de algoritmos que permiten (i) trabajar con la totalidad de la población de configuraciones válidas usando una estructura Booleana altamente optimizada conocida como Diagrama de Decisión Binario, y (ii) generar muestras aleatorias a partir del espacio de configuración. Además, se aporta una completa infraestructura en dos lenguajes de programación (C++ y **R**) con la que incorporar nuevos algoritmos sobre Diagramas de Decisión Binarios sin apenas esfuerzo.

Finalmente, esta tesis muestra la validación empírica de nuestros algoritmos y marco de trabajo a través de una prueba de rendimiento compuesta de modelos de variabilidad reales, cuyo número de características va desde 45 a 17 365.

Palabras clave: Líneas de productos de software, Modelos de características, Diagramas de decisión binarios, Funciones Booleanas.

Abstract

When one of the main aims of the Software Engineering is saving costs and time, the software product lines play an essential role. In this field, the key lies in identifying reusable components or features that can be applied in future projects.

Much research is being done to provide customizable solutions that match specific problems and thus satisfy a variety of non-functional requirements, such as runtime efficiency, computer memory consumption, security level, etc. Customization is often accomplished through a configuration process where the desired features are selected. The space of possible configurations is usually constrained to avoid features incompatibilities, and guaranteeing that feature inter-dependencies are satisfied. There are many questions of interest inside this restricted space: Are all configurations free of errors? Is there any dead code that cannot be activated because of the inter-feature constraints? What is the component reusability range? What is the typical size of a final product in terms of components? Etc.

The configuration space may potentially be 2^n for n features. Hence, processing the complete set of solutions is impossible except for the most trivial cases. After removing the invalid configurations that do not satisfy the inter-feature constraints, the solution space is reduced substantially. Nevertheless, it is usually still large enough to be suitable to work with it.

One strategy to address these difficulties is selecting a representative sample of the space and, once the computation is done, extrapolating the conclusions to the whole problem. At this point, the problem turns into the way of obtaining a reliable random sample to avoid wrong interpretations.

This thesis presents a set of algorithms that support both (i) working with the whole population of valid configurations by using a highly-optimized Boolean structure known as Binary Decision Diagrams, and (ii) generating random samples from the configuration space. Moreover, a whole infrastructure in two programming languages (C++ and **R**) is provided to incorporate new algorithms on Binary Decision Diagrams effortlessly.

Finally, this thesis reports the empirical validation of our algorithms and framework on a benchmark composed of real variability models, whose number of features ranges from 45 to 17 365.

Keywords: Software product lines, Feature models, Binary decision diagrams, Boolean functions.

Chapter 1

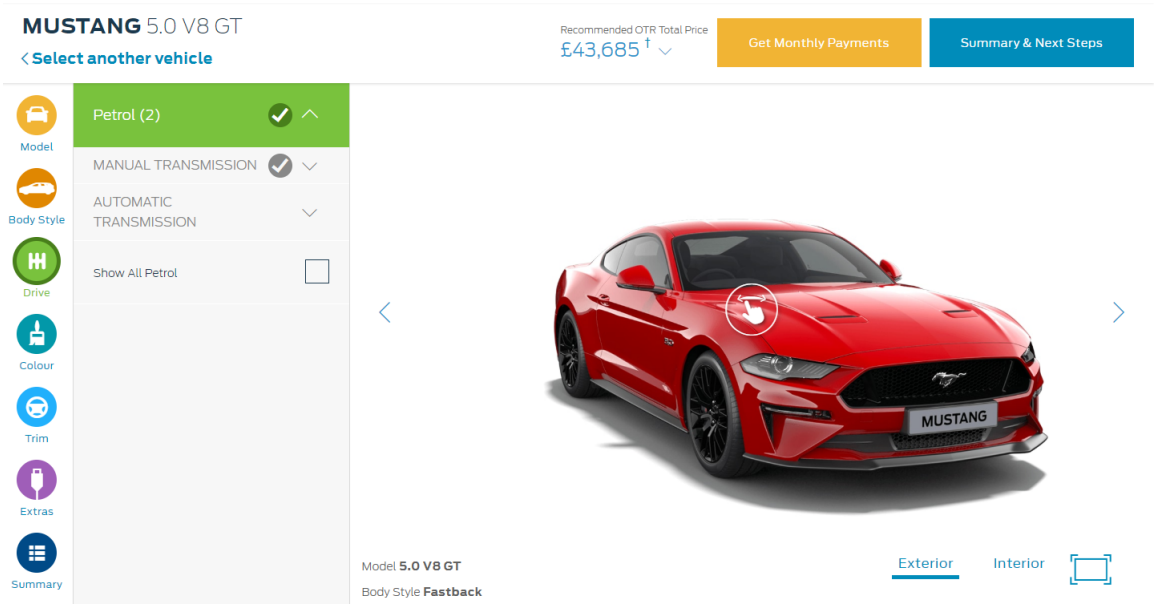
Introduction

A great variety of research areas and business models are based on combining features to provide successful solutions adapted to fulfill specific demands. The following situation could be imagined to illustrate this statement: when someone is willing to buy a car, one of the possible alternatives is checking the options and conditions that the manufacturers offer [1]. At some point in the inquiry, some choices have to be made to define the set of cars that could fit with the needs and distinctive features that the person is looking for. The factors that the customer has to decide about go from basic aspects as the engine (petrol, diesel, electric, Liquefied Petroleum Gas (LPG), etc.), transmission (manual or automatic), number of doors and seats, to esthetic qualities such as the color and type of paint, trim, wheels, lights and a growing amount of extras related to security, comfort, driving and connectivity.

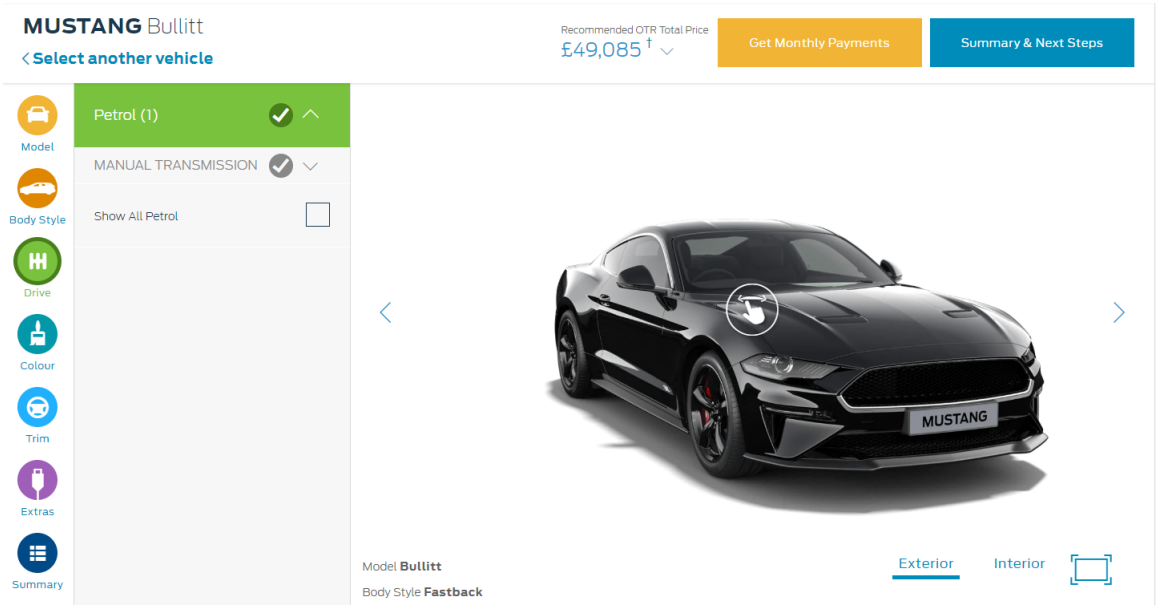
The huge amount of possible combinations can easily overwhelm indecisive people, or complicate the task of finding the relation of valid products, so it is crucial to simplify the process of adding and rejecting combinations according to the choices made. This is viable because each feature has dependencies with a set of features that have to be included in the solution when it is chosen, or items that are automatically rejected due to incompatibility issues with the selected one. As models can have thousands of features, it is important to be able to implement this mechanism in an efficient way to obtain the simplification in a reasonable execution time.

Figure 1.1 shows an example based on the Ford car configurator [2]. As not all the cars have the same features available, every time an item is selected, further options will be calculated according to the previous decisions. In Figure 1.1a a Mustang 5.0 has been selected, and, in this case, there are two possible transmission options: manual or automatic. However, if a Mustang Bullitt is chosen (Figure 1.1b), there is only the manual transmission model, so the configurator is smart enough to take into account that restriction and thus it simplifies the interaction with the customer.

This behaviour is comparable with the design phases of products like smartphones, home automation devices, industry equipment and is also applicable to other scientific fields like medicine [3]. In short, any industrial or research development that implies a series of combinations of features with the aim of obtaining the set of valid results that satisfies certain conditions is a candidate to benefit from this procedure [4].



(a) A model with two options of transmission



(b) A model with one transmission option

Figure 1.1: Example of dynamic calculation of features

In Software Engineering, the strategy of dividing a design problem into features, to analyze their interdependencies and subsequently trying to find out relations between these elements is more and more common. Kang et al. [5] define a feature as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system". This concept is essential for the Software Product Line (SPL) engineering paradigm, which efficiently enables the development of families of software systems that share a common set of assets.

The usage of feature models to represent all the products of a SPL in terms of its features, is very common. This representation was introduced by Kang et al. [5] as part of Feature-Oriented Domain Analysis (FODA). In particular, it is useful to determine reusable software components with a reduction of specification, development and testing tasks, saving costs and improving the time to market, two of the most coveted factors when a project comes to start.

Another example from the SPL domain is Busybox [6], a tool that provides some GNU's Not Unix (GNU) utilities in a light executable. To implement all its functionality in a compact format, it allows a custom selection of its 613 features, and these features and their interrelationships are defined making use a specific language called *Kconfig*[1].

The totality of the configuration space goes up to $\approx 3.4 \cdot 10^{184}$ combinations, but only $7.428 \cdot 10^{146}$ of them lead to a valid configuration due to the restrictions between features. Despite the fact that it is a small percentage of the total amount of possibilities (just a $2.185 \cdot 10^{-36}$ %), it is still a really large number, and unmanageable to test all the configurations.

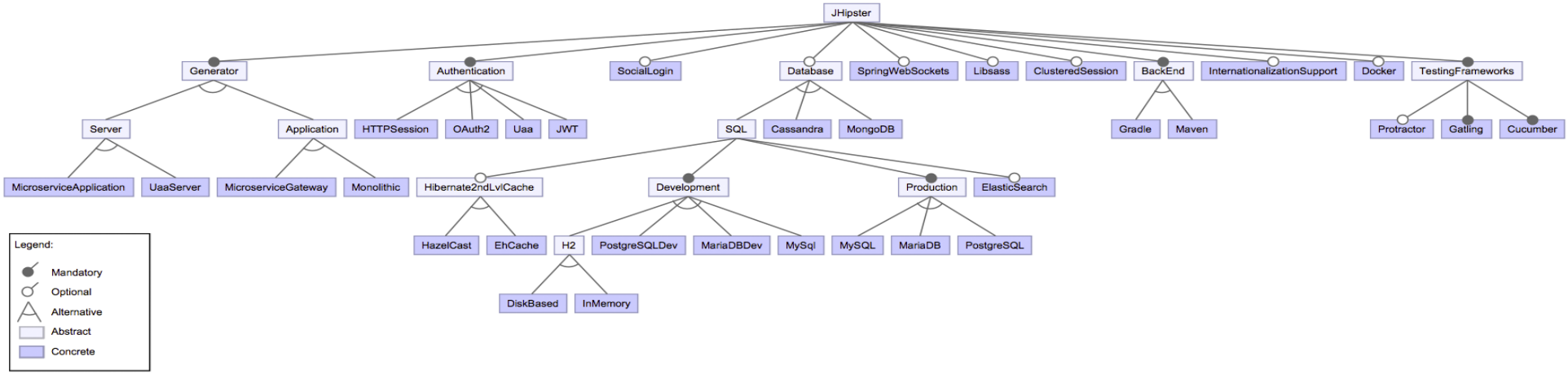
As an additional example, *JHipster* [7] is a development platform that generates, develops and deploys web applications and microservices. It only has 45 features, and the feature model is depicted in Figure 1.2[2] (Halin et al. [8]). It encompasses $2^{45} \approx 3.5 \cdot 10^{13}$ potential configurations, but barely 26 000 of them result on valid solutions. Testing this number of configurations could appear as a reasonable task, but it really has a high computational cost: $\sim$ 182 days of CPU time and 5.2 TB of disk space are needed for finishing this work [8].

Based on these figures, it turns as a critical research field finding a mechanism to obtain a representative sample of configurations. The idea is testing determinated behaviours on a set of a manageable size, and to extrapolate the conclusions to the absolute space of the problem. Through this thesis, some algorithms are proposed and evaluated to handle this kind of problems and extracting relevant conclusions in a reliable way.

These algorithms combine different areas of Mathematics, such as Probability and Combinatorics in order to calculate indicators like the probability of a feature to be included in a valid product or the number of products that contains a certain amount of features. Analyzing this information can help simplifying the problem of testing configuration spaces with huge dimensions and taking important design decisions in order to improve the model.

[1] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

[2] https://github.com/xdevroey/jhipster-dataset/blob/master/v3.6.1/featuremodel/FM-3.6.1.png



Constraints: (values in *italic* are literals)

Hibernate2ndLvlCache $\neq Hibernate2ndLvlCacheFalse \Rightarrow$ Database $= SQL$, Development $\neq DevelopmentFalse \Rightarrow$ Database $= SQL$, Database $= SQL \Rightarrow$ Development $\neq DevelopmentFalse$, Production $\neq ProductionFalse \Rightarrow$ Database $= SQL$, Database $= SQL \Rightarrow$ Production $\neq ProductionFalse$, ElasticSearch $\Rightarrow$ Database $= SQL$, Protractor $\Rightarrow$ TestingFrameworks, Gatling $\Rightarrow$ TestingFrameworks, Cucumber $\Rightarrow$ TestingFrameworks, Server $\neq ServerFalse \Rightarrow$ Generator $= Server$, Generator $= Server \Rightarrow$ Server $\neq ServerFalse$, Application $\neq ApplicationFalse \Rightarrow$ Generator $= Application$, Generator $= Application \Rightarrow$ Application $\neq ApplicationFalse$, H2 $\neq H2False \Rightarrow$ Development $= H2$, Development $= H2 \Rightarrow$ H2 $\neq H2False$, (Authentication $= OAuth2 \wedge$!SocialLogin $\wedge$ Server $\neq MicroserviceApplication) \Rightarrow$ (Database $= SQL \vee$ Database $= MongoDB$), SocialLogin $\Rightarrow$ ((Authentication $= HTTPSession \vee$ Authentication $= JWT) \wedge$ (Server $= ServerApp \vee$ Application $= Monolithic) \wedge$ (Database $= SQL \vee$ Database $= MongoDB$)), Server $= UaaServer \Rightarrow$ Authentication $= Uaa$, Production $= Oracle \Rightarrow$ (H2 $\neq H2False \vee$ Development $= Oracle12c$), (Authentication $\neq OAuth2 \wedge$!SocialLogin $\wedge$ Server $\neq MicroserviceApplication) \Rightarrow$ (Database $\neq DatabaseFalse$), Server $\neq ServerFalse \Rightarrow$!Protractor, Production $= MySQL \Rightarrow$ (H2 $\neq H2False \vee$ Development $= MySqlDev$), (Server $= MicroserviceApplication \vee$ Application $= MicroserviceGateway) \Rightarrow$ (Authentication $= JWT \vee$ Authentication $= Uaa$), Application $= Monolithic \Rightarrow$ (Authentication $= JWT \vee$ Authentication $= HTTPSession \vee$ Authentication $= OAuth2$), Production $= MariaDB \Rightarrow$ (H2 $\neq H2False \vee$ Development $= MariaDBDev$), Production $= PostgreSQL \Rightarrow$ (H2 $\neq H2False \vee$ Development $= PostgreSQLDev$), (Server $\neq ServerFalse \vee$ Application $\neq ApplicationFalse) \Rightarrow$ (BackEnd $\neq BackEndFalse \wedge$ Authentication $\neq AuthenticationFalse$), (SpringWebSockets $\vee$ ClusteredSession) $\Rightarrow$ Application $\neq ApplicationFalse$, Generator $= Client \Rightarrow$!Gatling$\wedge$!Cucumber $\wedge$ BackEnd $= BackEndFalse \wedge$ Authentication $= AuthenticationFalse$, Libsass $\Rightarrow$ (Application $\neq ApplicationFalse \vee$ Generator $= Client$)

Figure 1.2: JHipster feature model

To work with these algorithms, feature models are encoded as Boolean expressions, so the valid configurations of each model are the set of solutions that satisfy those formulas. Obviously, representing expressions that could lead to unmanageable configuration spaces like the exposed previously requires some specific data structures capable of working with that kind of information. Amongst the possible alternatives, Binary Decision Diagrams (BDDs) have been chosen because of their adaptability to represent a diversity of models that fits with the needs brought up in this thesis and, more importantly, in a very efficient way.

1.1 Scope of this Thesis

This research will discuss the suitability of a specific structure in order to encode expressions that represent feature models. To understand the theoretical framework where the investigation is located appropriately, a broad explanation of the context is provided.

The validity of the proposed solution will be demonstrated with a large enough set of models, varied in terms of size and constitution. All of them represent configurations, schemas and patterns of real feature models, because if the intended approach is proved as an acceptable alternative, it will be applied to diagrams of a similiar nature.

The intention of the investigation is the ratification of the aptitude of certain theories and algorithms to identify how well a feature model is built and other characteristics of the diagram.

1.2 Aims of the Thesis

Keeping in mind the situation described, the following goals have been defined:

- To provide a tool to manage structures able to represent feature models.
- To implement the algorithms analyzed in the work using the designed utility.
- To make a comparison between different versions of the implementations and discussing which version fits the best for diverse scenarios.

The designed tool will be explained with detail, but its guidelines have been:

- **Easy to use**. It must allow the simple use of its basic operations, and provide more advanced options for those cases not covered with the default settings and parameters.
- **Extensible**. If it is required to add any functionality, it should be achieved without a great investment of time.
- **Well documented**. The operations provided by the utility must provide a full explanation of its usage and functionality, and complete it with examples that illustrates the behaviour.

1.3 Research Questions

The current research is conceived to analyze the result of the application of some operations available in the literature, in order to obtain valuable information about models and its structure. Furthermore, our work hypothesis, which was validated in this thesis, is that most operations of interests on configuration models follow a common structure that can be generalized. Encoding this structure into a programming language, new algorithms can be effortlessly implemented on a high abstraction level. On the basis that the algorithms are studied in depth, providing theoretical basis and results that support these principles, the following research questions has been considered:

- **RQ1**: Can the designed tool provide results that allow to figure out how well a model is built?
- **RQ2**: How relevant is the implementation of the algorithms when they are executed over real models?

1.4 Hypotheses

Concerning **RQ1**, it is expected that with the contemplated implementation of the algorithms, the results obtained can lead to interpretations of how is the model represented in a given model.

Regarding **RQ2**, depending on the kind of programming language chosen to implement the functions, and the framework selected to execute the operations, the efficiency should vary. Similar performance is to be expected across small models, but as the size of the diagrams grows, the difference between the times needed to obtain the solutions should increase in favor of the most appropriate implementation.

1.5 Methodology

To develop the research, the usual methodology in the scientic field has been followed:

i. Wording of hypotheses.

ii. Collection of observations.

iii. Contrast between the hypotheses and the observations.

iv. Readjustment of the initial hypotheses in the light of the obtained results.

1.6 Personal Motivation

This thesis is conceived as a continuation of a previous work undertaken in an M.Sc. degree on the Software and Systems Engineering Group of the Superior Technical School of Cumputer Engineering of the Universidad Nacional de Educación a Distancia (UNED).

The possibilities of the work started in that project, warrant the expansion of the proposed architecture, and verifying its validity when it is applied over real models.

1.7 Thesis Outline

The thesis is structured in three blocks. The first of them attemps to contextualize the framework where the research is located. In this part, key concepts are defined and discussed, supported by the specialized literature.

Next, the second part is focussed on the implementation issues of the problem to tackle. The design decisions and construction details are broadly justified and explained in-depth.

The last block addresses the answer to the research questions through the confrontation of the theoretical expectations with the results obtained after the application of the algorithms and implementation object of the study.

Finally, a brief explanation of each chapter of the thesis is given below.

Chapter 2: Related Work

In this chapter, core concepts such as Boolean logic, SPLs, feature models, BDDs and SAT-solvers are introduced. In addition, the reasons why BDDs are chosen as the diagrams to represent feature models are explained and some details about their lifecycle are presented.

The last sections of this chapter discuss some algorithms present in the literature from a theoretical point of view, regarding the benefits of their application over the BDDs when they are used to represent feature models.

Chapter 3: Functional Programming on BDDs to Support the Statistical Reasoning on Variability Models

This chapter presents the architecture of the BDD library proposed in this thesis as well as some details of its operations and usage.

Moreover, an alternative implementation of the algorithms discussed in the previous chapter is provided in order to show the usefulness of our library.

Chapter 4: Experimental Validation

This chapter reports the results obtained when a benchmark is applied to the aforementioned algorithms. To this end, the library developed provides two ways of implementing these algorithms: a direct call to a C++ function that represents each procedure, and an alternative encoding the corresponding version on **R** through the functions available by the package.

In light of these results, a comparison between both versions is made, considering which of them satisfies the problems they are expected to solve for different use cases.

Chapter 5: Conclusions and Future Work

This chapter summarizes the primary conclusions of this thesis, the connection between the theoretical expectations of solution proposed and the evidences of the validation when

it is applied to real models. Finally, some lines are suggested to widen the work started in this thesis.

1.8 Main Contributions

After the time invested researching and studying the related literature, the contribution to this field of knowledge, besides this thesis, has been an open-source library and an article, *"Using Extended Logical Primitives for Efficient BDD Building"*, published in *Mathematics* [9], a journal ranked in the Q1 Journal Citation Report (JCR).

The `rbdd` library is an **R** package that allows the management of BDDs, as well as the application of all the functions needed to encode Boolean expressions making use of these structures. The Application Programming Interface (API) of this package follows an **R**-like style, to be as natural as possible to the community of programmers, and easy to extend if some new features are required.

Among the full set of instructions available in the library, there are functions related to the lifecycle of the BDDs, different options to load models supporting the most extended formats, debugging facilities, and some interesting algorithms and operations related to this kind of diagrams, that will be thoroughly explained in further chapters of the thesis.

As it will be explained in Section 2.2, the order of the make up that compound these diagrams is a key factor to achieve a successfully implementation of feature models. For these reason, the package provides the most common ordering algorithms to modify the structure, just as the possibility of applying different heuristics when the BDD is being built.

`rbdd` is available in a public repository[3] and it has been well documented with the aim of explaining the available functions and providing examples and, additionally, of simplifying the adaption of the package to extend its default functionality. The code has embedded all its dependencies, that are compiled automatically when the package is built, making the package compatible with most operating systems and architectures.

Working with `rbdd` on real models, we realized that feature models with extensive use of alternative constraints had a very negative impact on BDD sizes. As `rbdd` implemented in **R**, the statistical analysis of the results helped us to work out the problem is. We published our solution in the journal article referenced at the beginning of the section [9], which explores a way to synthesize BDDs more efficiently based on the technique of the identification of XOR groups in the Boolean expressions with the aim of simplifying the process of building the structure. The translation from an XOR-group of a feature model to its equivalent in propositional logic can lead to large formulas, with a great impact in the time required to form the diagram.

That paper [9] proposes an extension of the BDD engine to build the XOR-groups and its application over feature models and Kconfig configuration systems. Analyzing the obtained results, the theoretical expected benefits of this procedure have been confirmed, with a reduction of the time required to build the BDD up to orders of magnitude for real models.

[3] https://github.com/braguti/rbdd

Along this thesis, one of the points to which more attention has been paid is the efficiency when it comes to build a BDD that represents a real model (usually with a great amount of features) and to execute algorithms traversing the nodes that make up the diagram. For this reason, any technique that allows to speed up this process is taken into account, because this improvement could be one of the crucial factors that allow that the proposed alternative in this thesis to represent feature models be finally considered as a feasible possibility.

Chapter 2

Related Work

Although the study of logic was introduced by the ancient Greeks as part of Philosophy and Maths [10], it was not until the second half of the twentieth century when it turned into a fundamental knowledge in the technical science field with the born of the Computational Logic [11]. Before that, in the first half of that century, the fundamental pillars about which this area would be supported were founded. The first mainframe computers appeared, based on vacuum tubes first, and on transistors later, over the decades from 1930s to 1950s [12]. Other electronic instrumental appeared, such as electric calculators based on relay tchnology [13] or first integrated circuits [14]. The applications of this new field of knowledge have grown over time, covering major areas of research such as Artificial Intelligence (AI) [15], Software Engineering [16] or Formal Verification [17] [18] [19].

Computational Logic is the usage of hardware and software utilities to establish facts in a logical formalism [20]. Figure 2.1 represents some of the most important milestones reached on Computer Science that, among other things, would set up the basis of Computational Logic. The Boolean algebra was introduced by George Boole [21], and its importance lies in its application on switching circuits, as Shannon observed in 1938 [22]. Other important contributions were the Turing Machine, capable of assigning values to symbols following a series of defined rules, or the invention of the transistor, the magnetic core memory and the integrated circuit, basic elements that established the modern Computer Science.

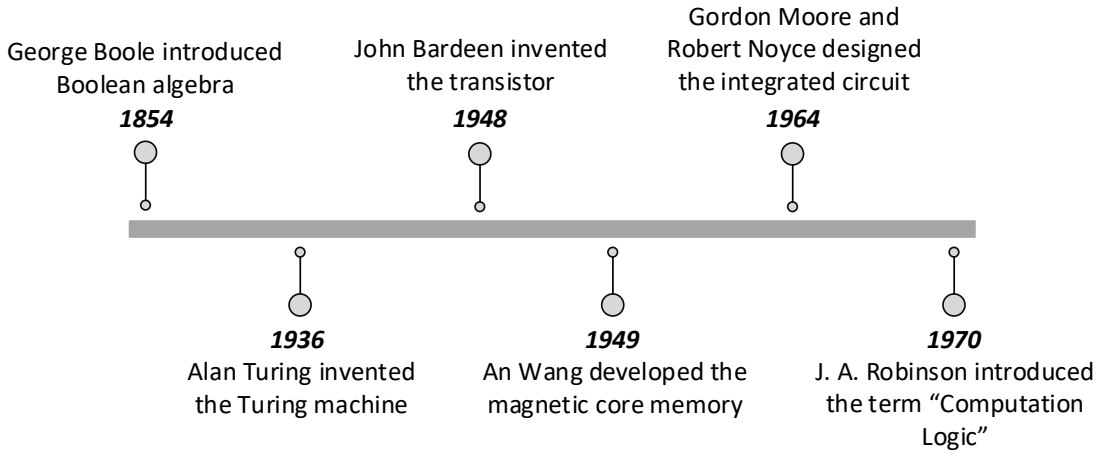


Figure 2.1: Main milestones on Computer Science and Computational Logic

In the early 1970's Alain Colmerauer, Philippe Roussel and Robert Kowalski conceived Prolog [23] [24]. That was the first application of logic resolution to represent and execute computer programs, being the origin of logic programming [25] [26].

Focussing on the Software Engineering research area, the Software Engineering Body of Knowledge (SWEBOK) (Bourque et al. [27]) quoted its definition from the ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering software" [28]. This description provides a framework in which well structured processes and procedures can be established and applied on the design and development of software, following the strategy that fits the best with the problem to deal with.

Software Engineering is a wide concept, so it is usually divided into fields known as Knowledge Areas (KAs). However, there is not an only structuring of the matter, each standard defining its customized parts. Serve as an example the SWEBOK (Table I.1.) [27], that defines 15 KAs, or the Project Management Body of Knowledge (PMBOK) [29], which makes 10 divisions. Despite these differences, the most popular alternatives cover the needs of the involved actors along the software lifecycle.

2.1 Software Product Lines and Variability Models

Among all the different ramifications in that Software Engineering is divided, the present work is focussed on the SPLs. According to Clements et al. [30], "a SPL is a set of software-intensive systems sharing a common, managed set of features that satisfies needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".

SPLs are particularly useful for that organizations or tasks in which the resource shortage is pointed out as a critical factor for inclining the resolution of the projects to the success or the failure. The Software Engineering Institute of the Carnegie Mellon University notes in its *Software Product Lines Collection* [31] that the main benefits for organizations which follow a skillfully product line implementation include:

- Improved productivity.
- Increased quality.
- Decreased cost.
- Reduced time to market.
- Ability to move into new markets quickly.

The basis of Software Product Line Engineering (SPLE) is the explicit modeling of the commonalities and differences between product variants (Beuche et al. [32]). There are different possibilities in order to represent that information, e.g., Feature Models or Domain Specific Languages (DSLs) [33].

The Institute of Electrical and Electronics Engineers (IEEE) defines a feature as "a distinguishing characteristic of a software item (e.g., performance, portability or functionality)" [34]. The way of representing these features to approach to the optimal solution

for the code reusability is by the feature models and its visual notation through feature diagrams. A feature diagram is a hierarchically ordered set of features, where each parent-child relationship is one from those described in Table 2.1 [35]. Figure 2.2 (Arcaini et al. [35], Section II.A) depicts a feature model representing the characteristics of a mobile phone. According to the figure's legend, each relationship between features can be transformed into a logical operation [36], and the conversion to propositional logic is shown in Table 2.1 [37] [38]. This is a key factor to be chosen as the representation of SPLs, because having the problem described as a logic formula, allows the application of different operations and algorithms to order the variables and simplify the problem.

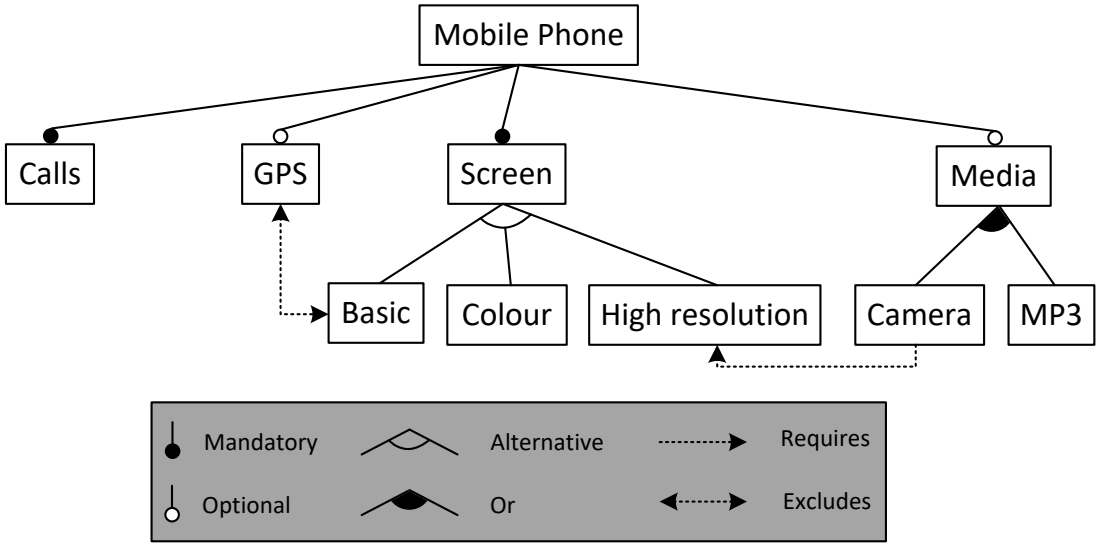
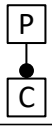
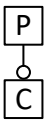
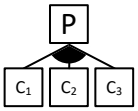
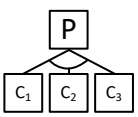
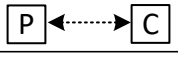


Figure 2.2: Example of a feature model

Table 2.1: Correspondence between feature relationships and propositional formulas

Relationship	Propositional Logic Mapping
	$P \leftrightarrow C$
	$C \rightarrow P$
	$P \leftrightarrow (C_1 \vee C_2 \vee \ldots \vee C_n)$
	$C_1 \leftrightarrow (\neg C_2 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$ $C_2 \leftrightarrow (\neg C_1 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge \ldots \wedge$ $C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{n-1} \wedge P))$
	$P \rightarrow C$
	$\neg(P \wedge C)$

Variability management comes into play at the search of variability between multiple applications [39] [40] [41]. Management of variability in a product line means that all the differences between products must be defined, represented, exploited, implemented and evolved throughout the lifecycle of the SPLE [39].

There are different alternatives for representing feature models having in mind the variability management. One of the most extended is the FODA method, that consists in the development of domain products widely applicable within a domain. It is reached abstracting away those factors exclusive to an application and including them as a part of the domain knowledge. The output of the method is a set of domain products that evolve through applications, not being considered ends unto themselves [5]. Nowadays, one of the most popular alternatives to this method is the Czarnecki-Eisenecker's notation [42] [43].

There is a lot of research work in the sphere of the variability modeling [44]. Jansen et al. [45] propose a methodology to define features at the first stages of the product's lifecycle (requirement level), defining a formal notation for expressing features and applying it to composition problems. A similar solution can be studied in the proposal of Hendrickson et al. [46].

2.2 Binary Decision Diagrams and SAT-Solvers

Traditionally, Boolean functions have been represented making use of different techniques, propositional logic being the most broadly used [47] because of its simplicity and flexibility. However, from the perspective of efficiency, there are better alternatives such as BDDs or Negation Normal Forms (NNFs) [48].

BDDs were introduced by Lee [49] and Akers [50], but it was Bryant [51] who demonstrated to the community their potential as data structures representing Boolean functions. They have been chosen as the structure to represent feature models for some strategic reasons. First of all, their ability to obtainresults more eficiently than other solutions, has been broadly demonstrated, as several academic works reveal [52] [53] [54]. Another important characteristic of BDDs is the possibility of applying different algorithms across the diagram, in order to get relevant information about the diagram and its features, as will be explained later in this chapter.

A BDD is a rooted, directed, acyclic graph, where each node has two child nodes (decision nodes), called high and low child, and there are two terminal nodes: 1-terminal and 0-terminal. It is common to use the BDD term to refer to Reduced Ordered Binary Decision Diagrams (ROBDDs), which are BDDs to which reduction and ordering operations have been applied. Reduction implies merging the isomorphic subgraphs of the complete graph and suppressing nodes whose children are isomorphic. Ordering operation modifies the structure until all the variables are represented in the same order on all paths, starting from the root node [55] [56]. Figure 2.3 [38] shows an example of how a BDD looks like.

The ordering applied to the graph has a dramatic impact in terms of efficiency, because it determines the size of the BDD (in addition to the proper logical function represented by the graph). Such is the case that the number of nodes of a BDD can fluctuate from a linear relation at the best scenario to exponential at the worst one. Figure 2.4 represents the increment of the number of nodes of a BDD when logical expressions are added to the diagram. If non-optimal ordering is applied to the BDD, the number of nodes grows

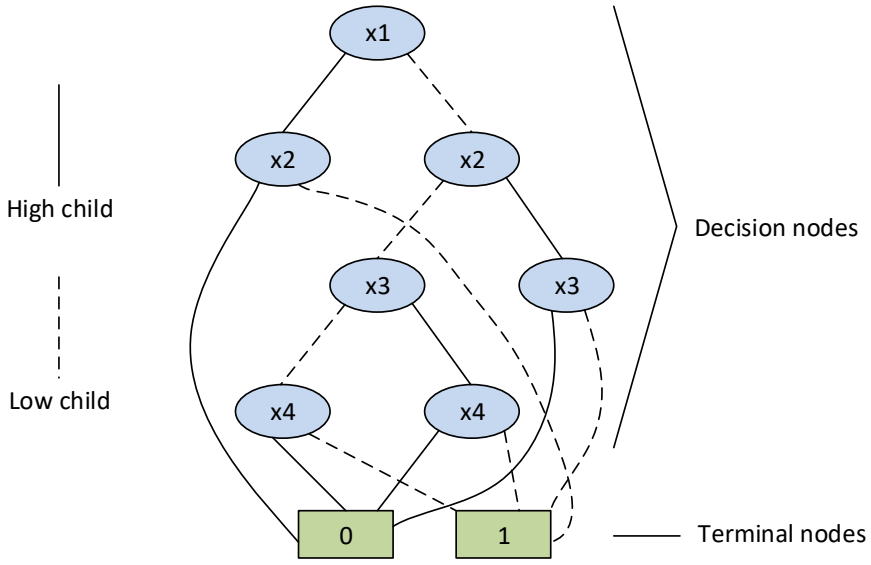


Figure 2.3: Structure of a BDD

until reaching unmanageables numbers, not being useful for practical purposes. However, when an ordering algorithm is employed, the size of the graph keeps small enough to work with. In the referenced example, the chosen ordering methods are sifting and random, but any other could be applied. The problem of getting the optimal ordering is not trivial. In fact, it is a NP-complete problem [57]. In practice, heuristics are used in order to have convenient execution times.

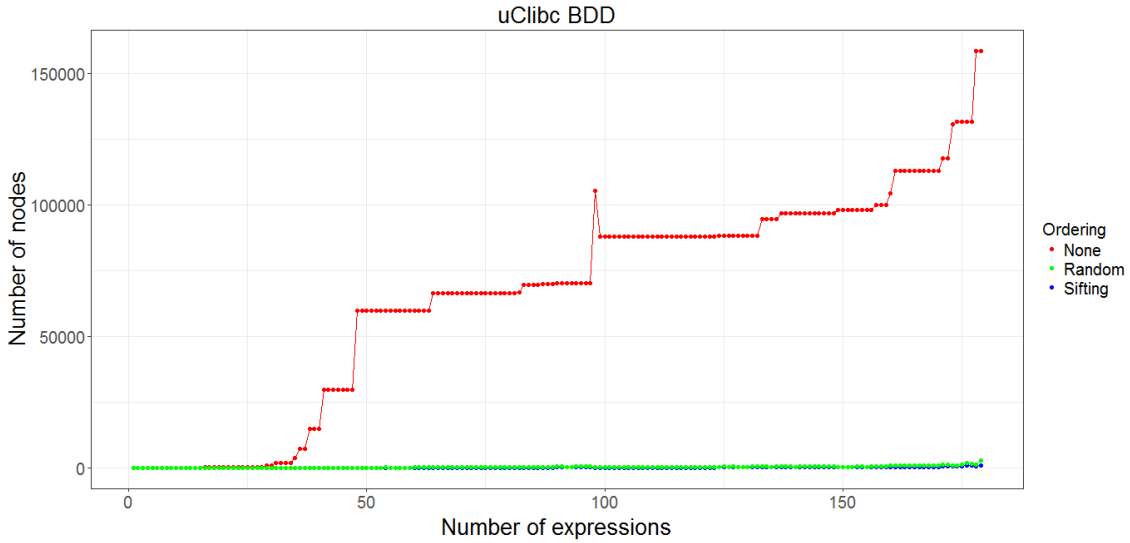


Figure 2.4: Comparison of BDD ordering algorithms

The boolean SATisfiability problem (SAT) is the problem of determining the existence of a solution that satisfies a given function. Boolean expressions are compounded by variables and logic operators (AND $\wedge$, OR $\vee$, NOT $\neg$). A formula is satisfiable if there is a boolean asignment of the variables which satisfies the formula.

The SAT problem is an NP-complete problem, as demonstrated in Cook's theorem [58],

and there is not a known algorithm that can reach the satisfiability of the Boolean function on a polynomial time. However, there are some restricted versions of the problem that allow a better performance to find a solution for a given formula. One example is the 2-satisfiability (2-SAT) problem, a version of SAT where the constraints consist of two variables each, expressed in Conjunctive Normal Form (CNF) [59]. In addition, there are some operations that can be solved in a satisfactory time, and considering SPL and variability models, checking product validity is one of them [60].

BDDs are a good option to implement Boolean functions and test the satisfiability of a formula [61]. There are some engines developed to manage BDDs, assign Boolean expressions and apply ordering algorithms. In the subsequent subsections are going to be reviewed two of them: CUDD and BuDDy.

2.2.1 CUDD

The CUDD package [62] [63] provides functions for the manipulation of BDDs, Algebraic Decision Diagrams (ADDs) and Zero-suppressed Binary Decision Diagrams (ZBDDs). It is written in C, and provides a C++ wrapper.

Its main advantage is its efficiency considering memory usage and performing logical operations between BDDs. In addition, its API is simple, and with a reduced set of instructions most logical operations can be carried out. Figure 2.5 shows a program that executes the AND Boolean function, and the BDD represented is depicted in Figure 2.6.

```
1  int main (int argc, char *argv[]) {
2      DdManager *manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS, CUDD_CACHE_SLOTS, 0);
3      DdNode *bdd, *x1, *x2;
4      x1 = Cudd_bddNewVar(gbm);
5      x2 = Cudd_bddNewVar(gbm);
6      bdd = Cudd_bddAnd(manager, x1, x2);
7      Cudd_Ref(bdd);
8      Cudd_Quit(manager);
9      return 0;
10 }
```

Figure 2.5: Implementation of the AND function with CUDD

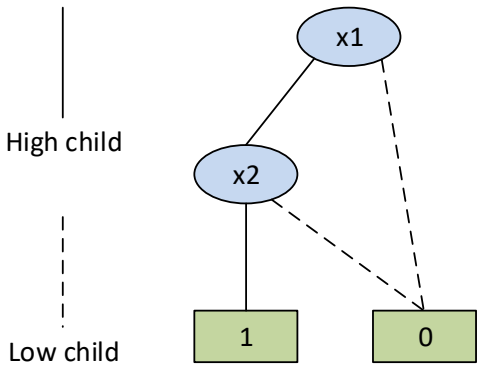


Figure 2.6: BDD representing the AND operation

In order to perform complementation operations, it implements *complement arcs*. Although they could simplify certain operations, these arcs add complexity because they modify the least significant bit of the else branch pointer of each node, making a bit harder working with pointers.

2.2.2 BuDDy

BuDDy [64] is another popular package developed in C created to check finite finite state machines, that allows programmers to manage the complete lifecycle of BDDs. This library stands out for its efficiency in resolving vectorized BDD operations, dynamic variable reordering and garbage collection [65]. Its interface is analogous than the API of CUDD, so most of the possible operations with BDDs could be done either one, equally.

To demonstrate the similarities between both packages, Figure 2.7 shows the implementation of the AND operation. If it is compared with Figure 2.5, the equivalence between both versions could be verified. They have the initialization of the BDD manager, the set of the logic values to the variables, the AND operation and the storage of the result in another variable and the free up of the memory used by the variables and the manager.

```
1  int main (int argc, char *argv[]) {
2      bdd x,y,z;
3      bdd_init(1000, 100);
4      bdd_setvarnum(5);
5      x = bdd_ithvar(0);
6      y = bdd_ithvar(1);
7      z = bdd_addref(bdd_apply(x, y, bddop_and));
8      bdd_delref(z);
9      bdd_done();
10     return 0;
11 }
```

Figure 2.7: Implementation of the AND function with BuDDy

Unlike CUDD, BuDDy does not support other types of structures such us ADDs or ZBDDs. In addition, the number of available dynamic ordering algorithms is lower than the provided by CUDD[1].

2.3 Synthesis of Binary Decision Diagrams

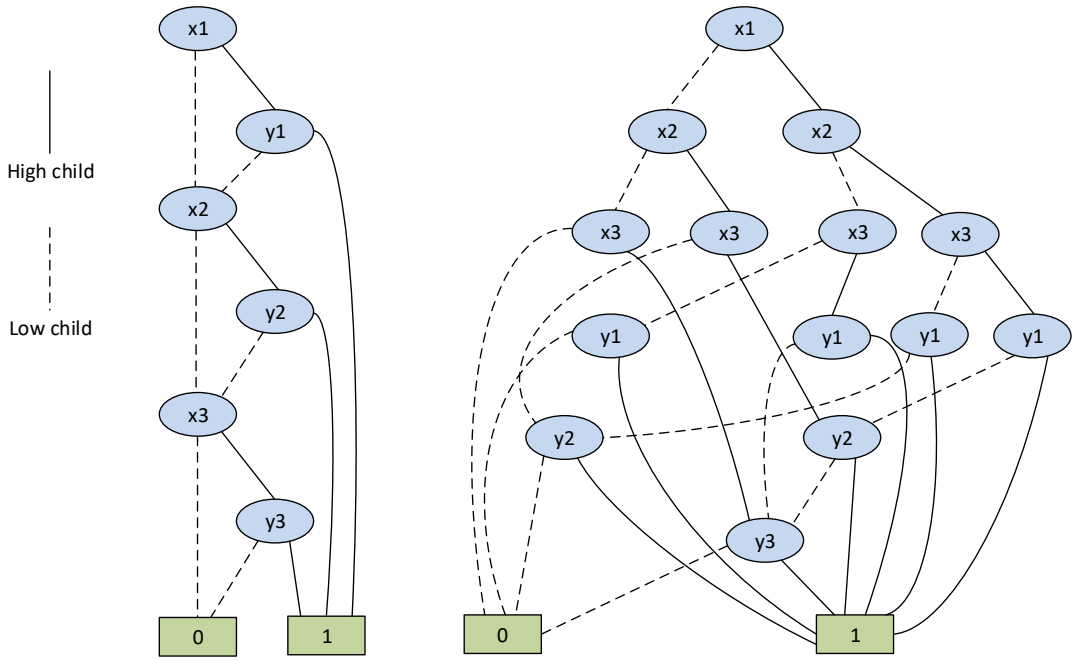
The process of building a BDD implies some actions in order to get a structure representing a Boolean function as efficient as possible. Otherwise, when the size of the diagram is large enough, working with the BDD would result unmanageable due to the required time to make the computations and, especially, its memory requisites.

The first step is the creation of the manager, that is, the entity which will handle the lifecycle of the BDD, the main settings, the control of the memory and resources, etc. At this point, and depending on the selected library, some additional parameters could be informed, such as the initial number of nodes or the size of the cache, in order to make an initial memory management and improve the performance of the program.

[1] BuDDy offers 7 heuristics, compared to the 18 implemented in CUDD.

Next, Boolean variables could be added to the BDD, and moreover, the logical operations between them. In most cases, the variables and clauses must be assigned to the diagram one by one, sequentially. As this is not practical in real programs, because the BDDs that represent feature models have thousands of relationships, Boolean functions are usually represented following standard formats, like CNF or Software Product Lines Online Tools (SPLOT) [66], to be processed before loading this information to the BDD.

As explained before, ordering the BDD is crucial to be able to operate with the structure. For this reason, when the diagram is built, the final step is applying an optimal reordering. It could be reached making use of heuristics [67], that changing the order of the variables in the BDD comes to a simplification of the model [68]. However, as previously mentioned, looking for the optimal ordering is not a simple task. Figure 2.8 depicts the same Boolean function, $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$, with a good variable ordering (Figure 2.8a) and a bad ordering (Figure 2.8b) [68]. Even in this simple example the importance of ordering these structures could be seen, and extrapolating to a real-life model with thousand times the size of the shown BDD, can be understood that ordering turns into a key concept to study about.



(a) Good variable ordering

(b) Bad variable ordering

Figure 2.8: Different orderings applied to the same expression

2.4 Core and Dead Features

In real feature models, features seldom have the same importance in the overall diagram. Some features are implied in the most demanded possibilities, or are dependencies of other important features. Moreover, there are features that are present in all the possible combinations that produce a valid model, and they are called *core features*, whereas the ones that are not included in any of the combinations are the *dead features*. The importance of this

topic in the SPL is exposed in Pérez Morago et al. [69] and its referenced literature [60] [70].

The criteria to determine if a feature is core or dead is broadly demonstrated by Pérez Morago et al. [69], and it is based on the fact that only the valid products are interesting, that is, the paths from the root to the 1-terminal node. The implications of this affirmation could be summarized on the following points ([69], Section IV-B):

1. If the 1-terminal could be reached from both high child and low child of the nodes labelled with the feature analyzed means that it is included in some valid products, but not in all of them, so it is not core nor dead feature.
2. If the 1-terminal is reached by the high child, but not by the low child of the nodes labelled with the feature, all the valid products include the feature, so it is core.
3. If the 1-terminal is reached by the low child, but not by the high child of the nodes labelled with the feature, all the paths that reach the 1-terminal node go across a low edge, so they represent non valid models, therefore the feature is dead.

As it is exposed by Pérez Morago et al. [69], Algorithm 1 obtains the core and dead features of a given BDD representing a variability model. It is based on updating two arrays in order to store if the children of each node reach the 1-terminal.

Algorithm 1 get_core_and_dead_features

Input: bdd: array,
var_ordering: array
Output: list with all the core features,
list with all the dead features
var: core_features, dead_features: list,
i: int,
through_high_array, through_low_array: array$[0..n-1]$ of boolean,
it_reaches_the_1_terminal_array: array$[0..m-1]$ of boolean

```
1:  begin
2:      core_features ← [ ]
3:      dead_features ← [ ]
4:      through_high_array ← [false, false, ..., false]
5:      through_low_array ← [false, false, ..., false]
6:      it_reaches_the_1_terminal_array ← [false, false, ..., false]
7:      does_it_reach_the_1 − terminal?(length(bdd) − 1, through_high_array,
8:          through_low_array, it_reaches_the_1_terminal_array)
9:      for i ← 0 to length(var_ordering) step 1 do
10:         if through_high_array[i] ∧ ¬through_low_array[i] then
11:             core_features.insert(var_ordering[i])
12:         else if ¬through_high_array[i] ∧ through_low_array[i] then
13:             dead_features.insert(var_ordering[i])
14:         end if
15:     end for
16:     return core_features, dead_features
17: end
```

These arrays are computed with Algorithm 2. This procedure is based in Bryant's traverse function (Section 3.1) to perform operations recursively on the nodes of the BDD. For each node, the algorithm checks if the 1-terminal node is reached.

Algorithm 2 does_it_reach_the_1-terminal?

Input: v: index of a node in the bdd (i.e., $0..m-1$),
through_high_array, through_low_array: array$[0..n-1]$ of boolean,
it_reaches_the_1_terminal_array: array$[0..m-1]$ of boolean
Output: *true* if v can reach the 1−terminal. Otherwise, it returns $false$
Global: through_high_array and through_low_array: array (by reference)

```
1:  begin
2:      if ¬bdd[v].mark then
3:          bdd[v].mark ← true
4:          // does v reach the 1-terminal through high?
5:          if bdd[v].high == 1 then // 1-terminal is reached
6:              through_high_array[bdd[v].index] ← true
7:              it_reaches_the_1_terminal_array[v] ← true
8:              update_reduced_nodes(v, "high", through_high_array, through_low_array)
9:          else if bdd[v].high ≠ 0 then // keep searching
10:             it_reaches_the_1_terminal_array[v] ←
11:                 does_it_reach_the_1 − terminal?(bdd[v].high, through_high_array,
12:                 through_low_array, it_reaches_the_1_terminal_array)
13:             if it_reaches_the_1_terminal_array[v] then
14:                 through_high_array[bdd[v].index] ← true
15:                 update_reduced_nodes(v, "high", through_high_array, through_low_array)
16:             end if
17:         end if
18:         // does v reach the 1-terminal through low?
19:         if bdd[v].low == 1 then // the 1-terminal is reached
20:             through_low_array[bdd[v].index] ← true
21:             it_reaches_the_1_terminal_array[v] ← true
22:             update_reduced_nodes(v, "low", through_high_array, through_low_array)
23:         else if bdd[v].low ≠ 0 then // keep searching
24:             it_reaches_the_1_terminal_array[v] ←
25:                 does_it_reach_the_1 − terminal?(bdd[v].low, through_high_array,
26:                 through_low_array, it_reaches_the_1_terminal_array)
27:             if it_reaches_the_1_terminal_array[v] then
28:                 through_low_array[bdd[v].index] ← true
29:                 update_reduced_nodes(v, "low", through_high_array, through_low_array)
30:             end if
31:         end if
32:     end if
33:     return it_reaches_the_1_terminal_array[v]
34: end
```

If the high terminal node is reached, Algorithm 3 is called. It updates the two auxiliary arrays, *through_high_array* and *through_low_array*, considering those nodes that could be suppressed from the BDD as part of the reduction process.

Once the arrays have been populated with the right values, core and dead features are

Algorithm 3 update_reduced_vertices

Input: v: $0..m-1$,
direction: string $\in$ "*high*", "*low*",
through_high_array, through_low_array: array$[0..n-1]$ of boolean
Output: through_high_array and through_low_array (by reference)
var: i: int

```
1:  begin
2:      if direction == "high" then
3:          for i ← bdd[v].index + 1 to bdd[bdd[v].high].index − 1 step 1 do
4:              through_high_array[i] ← true
5:              through_low_array[i] ← true
6:          end for
7:      else // direction == ''low''
8:          for i ← bdd[v].index + 1 to bdd[bdd[v].low].index − 1 step 1 do
9:              through_high_array[i] ← true
10:             through_low_array[i] ← true
11:         end for
12:     end if
13: end
```

identified according to two principles:

1. A feature f is core if and only if the high child reachs the 1-terminal and the low child does not reach it.
2. A feature f is dead if and only if the high child does not reach the 1-terminal and the low child does it.

2.5 Feature Probabilities

As it has been pointed out previously, variability models are comprised of a certain number of features, fact that defines the size and characteristics of the BDD designed to represent them. The importance of knowing the probability of a feature to be selected is emphasized in Heradio et al. [71], and it includes decisions such as if a feature f_1 is selected, which other features must be selected or rejected due to their interdependencies? Moreover, what could be the size of the product if a feature f_2 is selected or not? These are key questions, and obtaining a satisfactory answer could lead to a successful product design or to the need for a reconsideration of the variability model.

Knowing the features' probability of a variability model can help to make an accurate idea of how a model is, i.e., understanding the range of the number of features needed to obtain valid models and, linking with the previous section, what percentage of the total amount of features are core and dead.

Algorithm 4 obtains the ratio of valid products that includes each feature [71]. This algorithm mixes the node probability and the joint probabilities in what is called *conditional probability*, $p(\Phi \mid n)$. It is the probability of a feature of being included in a result knowing

that another parameter is effectively satisfied, and its value is obtained as is defined by Equation (2.1):

$$p(\Phi \mid n) = \begin{cases} 0 & \text{if } n_{HI} = n_0 \\ 1 & \text{if } n_{HI} = n_1 \\ \dfrac{p(\Phi, n_{HI}) + p(\Phi, \overline{n_{HI}})}{2 * p(n_{HI})} & \text{otherwise} \end{cases} \tag{2.1}$$

being n_0 and n_1 the terminal nodes. This formula applies analogously for the low path, n_{LO}.

Algorithm 4 get_feature_probabilities

Input: bdd: array of nodes,
var: list of variables of the BDD
Output: array of probabilities

```
1:  begin
2:      p(bdd root) ← 1/2
3:      p(n_i) ← 0 for all nodes n_i except the BDD root
4:      get_node_pr(bdd root)

5:      p(x_j, Φ) ← 0 for all variables x_j
6:      get_joint_pr(bdd root)

7:      p(Φ) ← p(n_1)
8:      for each x_j do
9:          p(x_j|Φ) ← p(x_j, Φ) / p(Φ)
10:     end for
11:     return p
12: end
```

The probabilities $p(n)$ and $p(\overline{n})$ for a node n are defined as the number of paths that go from the root to the terminal nodes by traversing n through its high and low outgoing edges, respectively, divided by the total amount of paths (Heradio et al. [71], Section III.B). The implementation of this definition can be found in Algorithm 5 and, combined with the Bryant's traverse method, returns the relation between each node of the BDD and its probability to be present in a valid product.

Similarly to the concept of node probability, joint probability must be calculated to obtain the desired probability of each feature to be present in a valid product. This joint probability, $p(x,\ \Phi)$, is the number of combinations where both x and Φ are valid divided by the total number of combinations. The procedure to calculate it is represented in Algorithm 6 and, again, this algorithm relies in the Bryant's recursive method to visit each node and compute the result in the entirety of the BDD, making sure each node it is visited just once, avoiding misleading conclusions.

Algorithm 5 get_node_pr

Input: n: node of the BDD
Output: array of node probabilities (by reference)

```
1:  begin
2:      mark(n) ← ¬mark(n)
3:      if n is non − terminal then
4:          // explore low
5:          if n_LO is terminal then
6:              p(n_LO) ← p(n_LO) + p(n)
7:          else
8:              p(n_LO) ← p(n_LO) + p(n)/2
9:          end if
10:         if mark(n) ≠ mark(n_LO) then
11:             get_node_pr(n_LO)
12:         end if

13:         // explore high
14:         if n_HI is terminal then
15:             p(n_HI) ← p(n_HI) + p(n)
16:         else
17:             p(n_HI) ← p(n_HI) + p(n)/2
18:         end if
19:         if mark(n) ≠ mark(n_HI) then
20:             get_node_pr(n_HI)
21:         end if
22:     end if
23: end
```

Algorithm 6 get_joint_pr

Input: n: node of the BDD
Output: array of joint probabilities (by reference)
Global:var: list of variables of the BDD (by reference)

1: **begin**
2: $mark(n) \leftarrow \overline{mark(n)}$
3: **if** $n\ is\ non-terminal$ **then**
4: `// explore low`
5: **if** $n_{LO} == n_0$ **then**
6: $p(\Phi|\overline{n}) \leftarrow 0$
7: **else if** $n_{LO} == n_1$ **then**
8: $p(\Phi|\overline{n}) \leftarrow 1$
9: **else**
10: **if** $mark(n) \neq mark(n_{LO})$ **then**
11: $get\_joint\_pr(n_{LO})$
12: **end if**
13: $p(\Phi|\overline{n}) \leftarrow \frac{p(\Phi,\ n_{LO} \vee \overline{n_{LO}})}{2p(n_{LO})}$
14: **end if**
15: $p(\overline{n}, \Phi) \leftarrow p(\Phi|\overline{n})p(n)$

16: `// explore high`
17: **if** $n_{HI} == n_0$ **then**
18: $p(\Phi|n) \leftarrow 0$
19: **else if** $n_{HI} == n_1$ **then**
20: $p(\Phi|n) \leftarrow 1$
21: **else**
22: **if** $mark(n) \neq mark(n_{HI})$ **then**
23: $get\_joint\_pr(n_{HI})$
24: **end if**
25: $p(\Phi|n) \leftarrow \frac{p(\Phi,\ n_{HI} \vee \overline{n_{HI}})}{2p(n_{HI})}$
26: **end if**
27: $p(n,\ \Phi) \leftarrow p(\Phi|n)p(n)$

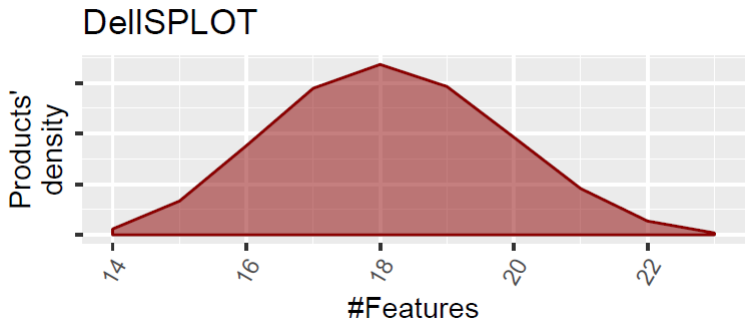
28: `// combine both low and high`
29: $p(\Phi,\ n \vee \overline{n}) \leftarrow p(\Phi,\ n) + p(\Phi,\ \overline{n})$
30: $p(var(n),\ \Phi) \leftarrow p(var(n)) + p(n,\ \Phi)$

31: `// add joint probabilities of the removed nodes`
32: **for each** x_j **between** $var(n)$ **and** $var(n_{HI})$ **do**
33: $p(x_j,\ \Phi) \leftarrow p(x_j,\ \Phi) + \frac{p(n,\ \Phi)}{2}$
34: **end for**
35: **for each** x_j **between** $var(n)$ **and** $var(n_{LO})$ **do**
36: $p(x_j,\ \Phi) \leftarrow p(x_j,\ \Phi) + \frac{p(\overline{n},\ \Phi)}{2}$
37: **end for**
38: **end if**
39: **end**

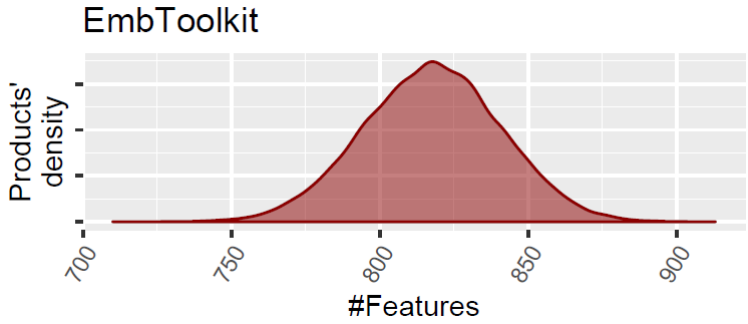
2.6 Product Distribution

Another important indicator of how a variability model is built is what is denominated as product distribution (Heradio et al. [71], Section III.C). It is defined as the amount of products that have a certain number of features, and analyzing this distribution can lead to some relevant conclusions. There are models in which almost the totality of valid products can be reached selecting a minimal number of features, or models whose distribution reveals that a very high percentage of features is necessary to get a relevant number of satisfying solutions.

With the aim of illustrating this behaviour, serve as example Figure 2.9. The plots depicted in Figure 2.9a and Figure 2.9b represent the product distribution of the DellSPLOT [72] and EmbToolkit [73] models, respectively. The first model has 118 variables, while the second has 2331 (Table 4.5). It shows that in the case of the DellSPLOT model, the majority of the products are obtained making use of just a range of 14-23 features. However, the EmbToolkit model needs near of 800 features in order to get a significant number of valid results.



(a) DellSPLOT product distribution



(b) EmbToolkit product distribution

Figure 2.9: Different ordering applied to the same expression

Algorithm 7 shows the method suggested by Heradio et al. [71] that returns a list with the product distribution of a given BDD. It starts by initializing that list with the cases of n_0, that represents no products at all, and n_1, that is, a single product with no features. Starting from these base cases, the algorithm calls the auxiliary function described in Algorithm 8 making recursive iterations over the different nodes. For achieving this behaviour, similarly the other algorithms discussed in this work, it takes advantage of the traverse method designed by Bryant.

Algorithm 7 product_distribution

Input: bdd: array of nodes
Output: array with the product distribution
Global: var: list of variables of the BDD

```
1: begin
2:     dist(n_0) ← [ ] // no products
3:     dist(n_1) ← [1] // one product with no features
4:     get_prod_dist(bdd root)
5:     return dist(bdd root)
6: end
```

The basis of this algorithm can be summarized in the three observations pointed to by the authors of the algorithm:

1. Including new features into all products is achieved by shifting the dist vector to the right.
2. Combining dist vectors is accomplished by adding them.
3. Removed nodes require taking into account both previous observations, and blending them by means of combinatorial numbers.

A mathematical proof can be found in Heradio et al. [71], Section III.C

Algorithm 8 get_prod_dist

Input: n: node of the BDD,
var: list of variables of the BDD (by reference)
Output: array of distribution (by reference)
var: removedNodes: int,
lowDist: vector of int,
highDist: vector of int

```
1:  begin
2:      mark(n) ← ¬mark(n)
3:      if n is non − terminal then
4:          // traverse
5:          if mark(n) ≠ mark(n_LO) then
6:              get_prod_dist(n_LO)
7:          end if
8:          // compute lowDist to account for the removed nodes through low
9:          removedNodes ← var(n_LO) − var(n) − 1
10:         let lowDist be a vector with removedNodes + length of dist(n_LO) zeros
11:         for i ← 0 to removedNodes step 1 do
12:             for j ← 0 to length of dist(n_LO) − 1 step 1 do
13:                 lowDist[i + j] ← lowDist[i + j] + dist(n_LO)[j] · C(removedNodes, i)
14:             end for
15:         end for
16:         // traverse
17:         if mark(n) ≠ mark(n_HI) then
18:             get_prod_dist(n_HI)
19:         end if
20:         // compute highDist to account for the removed nodes through high
21:         removedNodes ← var(n_HI) − var(n) − 1
22:         let highDist be a vector with removedNodes + length of dist(n_HI) zeros
23:         for i ← 0 to removedNodes step 1 do
24:             for j ← 0 to length of dist(n_LO) − 1 step 1 do
25:                 highDist[i + j] ← highDist[i + j] + dist(n_HI)[j] · C(removedNodes, i)
26:             end for
27:         end for

28:         // combine low and high distributions
29:         if lowDist is longer than highDist then
30:             distLength ← length of dist(n_LO)
31:         else
32:             distLength ← length of dist(n_HI) + 1
33:         end if
34:         let dist(n) be a vector of length distLength filled with zeros
35:         for i ← 0 to length of lowDist step 1 do
36:             dist(n)[i] ← lowDist[i]
37:         end for
38:         for i ← 0 to length of highDist step 1 do
39:             dist(n)[i + 1] ← dist(n)[i + 1] + highDist[i]
40:         end for
41:     end if
42: end
```

2.7 Uniform Random Sampling

In the area of testing, an important barrier that has to be faced when a certain set of configurations of a variability model must be checked, is the fact that the number of variables grows exponentially as new features are added to the diagram. A good strategy to deal with this behaviour is to sample some interesting configurations before testing over the whole structure [74].

Currently, there are samplers that offer acceptable results [75] [76], but there is not an optimal solution. Some of the existing options produce solutions with a poor performance, and the faster samples generate distributions that are not guaranteed to be uniform (Plazar et al., Section II.B [74]).

The algorithm proposed for obtaining the sample is exposed in Algorithm 9, and it consists in a calculation of the probability of each feature to be present in a valid product (analogous to the procedure explained in Section 2.5), and the application of a custom method to select the presence of the features randomly, but taking into account the probabilities of being selected obtained previously.

Algorithm 9 uniform_sampling

Input: bdd: the BDD
Output: array of boolean that indicates the presence of each feature
var: probabilities: array of numeric

1: **begin**
2: *probabilities* ← *get_node_pr*(*bdd root*)
3: *apply custom random function making use of the probabilities array*
4: **return** *result of the previous execution*
5: **end**

In the literature, a great variety of alternatives to generate a random sample can be found, applying it to different reasearch topics. Muñoz et al. [77] make use of Uniform Random Sampling (URS) into Bit-Blasted Propositional Formula (BBPF), and Oh et al. [78] employ a technique to locate sub-optimal products in colossal spaces[2].

In Section 7.1.4 of [80], Knuth showed how to accomplish uniform random sampling by subsequently partitioning the SAT solution space on variable assignments, and then counting the number of solutions of the resulting parts. Conceptually, the procedure works as follows: first, the number of solutions $\#\mathrm{SAT}(\varphi)$ of the input formula φ with v variables $x_1, x_2, \ldots, x_v$ is computed. Then, the number of solutions where x_1 is true is counted : $\#\mathrm{SAT}(\varphi \wedge x_1)$. Therefore, x_1 follows a Bernoulli distribution with probability $p_1 = \frac{\#\mathrm{SAT}(\varphi \wedge x_1)}{\#\mathrm{SAT}(\varphi)}$, and accordingly, its random assignment is generated. For instance, imagine that x_1 is assigned to false. Then, x_2 would follow a Bernoulli distribution with probability $p_2 = \frac{\#\mathrm{SAT}(\varphi \wedge \bar{x}_1 \wedge x_2)}{\#\mathrm{SAT}(\varphi \wedge \bar{x}_1)}$, and it would be randomly assigned. The procedure advances until the last variable x_v is assigned, and thus the random solution is completed.

The original algorithm by Knuth is specified on BDDs very efficiently, as the probabilities required for all the possible SAT solutions are computed just once with a single BDD

[2]Here, the authors use the term *colossal* to refer to those random samplings with $\gg 10^{60}$ search spaces. For further information can be consulted Batory et al. [79].

traversal, and then reused every time a solution is generated. Nevertheless, Knuth's algorithm can be easily adapted to SAT technology. In particular, *Spur*[3] [81] and *Smarch*[4] [82] rely on a *#SAT-solver* named *sharpSAT* [83].

[3] https://github.com/ZaydH/spur
[4] https://github.com/jeho-oh/Kclause_Smarch

Chapter 3

Functional Programming on BDDs to Support the Statistical Reasoning on Variability Models

Once a formal contextualization of the framework in which the current work is established has been done, a broad explanation of the technical aspects of the proposed solution it is given in this chapter. Some design decisions have been made throughout the entire project lifecycle, involving technologies, architecture issues and implementation matters, according to achieve the objectives defined as much as possible with regard to reaching satisfying results in reasonable thresholds.

Since BDDs were chosen as the structure to deal with the boolean expressions which represent variability models, some options were considered in order to handle the information stored efficiently and provide the solutions in an exploitable way for a further analysis. In this sense, it is necessary to be able to operate statistically with the results provided by the different algorithms and functions, so those languages and technologies that did not fit with this requirement could be automatically rejected.

3.1 Traverse

The algorithms object of study in this work have something in common: they make use of an implementation of the traverse function designed by Bryant [51]. Algorithm 10 shows Bryant's design of the traversing method. It works with the `vertex` structure, as a representation of a node, and which definition is defined as follows:

```
type vertex = record
    low: vertex
    high: vertex
    index: 1..n+1
    val: (0,1,X)
    id: integer
    mark: boolean
end
```

where `low` and `high` are pointers to the low and high branches of the vertex, `index` is the level of the node in the BDD, `val` is used to difference between terminal and non-terminal nodes, `id` is a unique identifier of the vertex and `mark` is a flag to indicate if the vertex

has been visited in another iteration of the algorithm.

Algorithm 10 Bryant's traverse design

Input: v: the vertex (by reference)

```
1:  begin
2:      v.mark ← not v.mark
3:      apply logic to v
4:      if v.index ≤ n ∧ v is non-terminal then
5:          if v.mark ≠ v.low.mark then
6:              traverse(v.low)
7:          end if
8:          if v.mark ≠ v.high.mark then
9:              traverse(v.high)
10:         end if
11:     end if
12: end
```

The function has to be invoked with the vertex that represents the root node of the BDD as argument, and it iterates recursively over its children. In every iteration, the vertex currently visited is flagged to prevent the execution of the function more than once for each node. Over each iteration custom logic can be applied to the BDD, fact that is leveraged to execute the proposed algorithms in the current work.

As Bryant dissects in [51] (Section 4.1), the performance of the algorithm is dependent on the size of the BDD, so if the logic applied to each vertex requires constant time, the complexity of the traverse algorithm is $O(\mid G \mid)$ (G represents the graph defined by the BDD).

Being able to use this algorithm is one of the key reasons to select BDDs as the structures for representing logic functions among other alternatives. It is easily implementable and it allows to apply custom logic over varied nature BDDs on a uniform way.

Due to its flexibility at being adapted to solve custom problems, it is recurrent to make use of traverse procedure in different fields such as Searching Algorithms (Jensen et al. [84]), Number Theory (Fefferman et al. [85]) or Testing (Miczo [86]).

3.2 The rbdd Package

The developed library, that has been called **rbdd**, has well-defined objectives, such as the management of the lifecycle of the BDDs, by virtue of associating logical expressions to the diagrams and executing custom algorithms over the models. Furthermore, the computed results have to be presented in a mode that allows their statistical analysis.

After evaluating the plausible alternatives, finally **R** was chosen, because it "provides an environment for statistical computing and graphics" [87], something fundamental to accomplish the goals of the project. Its power resides in the ability to compute statistical functions along large data sets faster than other possibilities, because of being conceived

for that specific purpose.

3.2.1 Architecture

R supplies an ideal environment to obtain information about the execution of the algorithms proposed, making comparatives and taking advantage of its graphical techniques to extract the desired conclusions. Moreover, the platform allows to extends its capabilities through additional packages, and in the same way, to develop packages implementing custom functions that cover certain needs not included in the base distribution.

After searching for a package that provides the functionality to work with BDDs in the Comprehensive R Archive Network (CRAN), and verifying that there is not a library that allows to manage this kind of diagrams from the **R** environment, it was decided to implement a custom package so the required operations can be achieved as it is needed to accomplish the proposed objectives.

However, due to the nature of **R** of specific purpose language, there is the need to extend its abilities combining it with another more generalist language, as could be C++. In this sense, some functions that could be hard to implement in **R**, are easily developed in C++, getting reasonable behaviour results. Thus, making use of both technologies, the benefits of each one can be applied to the final model and obtaining a balanced solutions which allows working with BDDs and design algorithms under a reliable environment.

The way to link **R** and C++ and, more importantly, to do it as transparent as possible to the users is using an existing **R** package, called `Rcpp` [88]. This library gives to the developers a manageable alternative to make C++ functions available from the **R** side, in an **R**-like syntax and without a perceptible loss of performance [89].

`Rcpp` provides dynamic casting between **R** data types to C++ objects and vice versa by means of the functions `Rcpp::as<T>(obj)` and `Rcpp::wrap(obj)`, respectively [90]. Furthermore, other non-primitive data types such as vectors, lists and matrices, and even custom classes can be passed through both languages because of the usage of the **R** internal pointers (SEXP) [91]. This is a key factor to accomplish the implementation of the BDDs management library and the exposure of the functions to the **R** side. In addition, the `cxxfunction` instruction provides a mechanism to call a C++ function from the **R** side, establishing the data types conversion automatically.

Figure 3.1 [92] shows an example of how the `Rcpp` library works. That code implements the Gibbs Sampler [93], a Monte Carlo Markov Chain (MCMC) algorithm that aproximates a sequence of observations from a specified multivariable probability distribution [94]. It encodes the following Gibbs sampler for a bivariate distribution (Equation (3.1)):

$$f(x, y) = k * x^2 * e^{-x*y^2 - y^2 + 2y - 4x} \tag{3.1}$$

and the conditional distributions are (Equation (3.2) and Equation (3.3)):

$$f(x|y) = x^2 * e^{-x*(4+y^2)} \qquad \#\# \text{ a Gamma density kernel} \quad (3.2)$$

$$f(x|y) = e^{-1*(x+1)*y^2 - \frac{2*y}{x+1}} \qquad \#\# \text{ a Gaussian kernel} \quad (3.3)$$

```
gibbscode <- '
  using namespace Rcpp;
  // n and thin are SEXPs which the Rcpp::as function maps to C++ vars
  int N   = as<int>(n);
  int thn = as<int>(thin);
  int i,j;
  NumericMatrix mat(N, 2);
  RNGScope scope;          // Initialize Random number generator
  double x=0, y=0;

  for (i=0; i<N; i++) {
    for (j=0; j<thn; j++) {
      x = ::Rf_rgamma(3.0,1.0/(y*y+4));
      y = ::Rf_rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
  }

  return mat;              // Return to R
'
# Compile and Load
RcppGibbs <- cxxfunction(signature(n="int", thin = "int"),
                         gibbscode, plugin="Rcpp")
```

Figure 3.1: Implementation of the Gibbs Sampler with `Rcpp`

As has been explained in the previous chapter, the libraries selected to handle the BDDs have been CUDD and BuDDy. With the purpose of making the solution adaptable to be extended with other managers, the C++ code has been designed as a wrapper. This architecture has an additional advantage, since it gives an API independent of the selected manager and provides a transparent environment. Figure 3.2 depicts the overall architecture designed, and the relations between the different actors could be examined. The user interacts with the wrapper's exposed methods through the `Rcpp` facility, and depending on the considered choice, the wrapper will call to the pertinent BDD manager functions and returning back the result of the computations.

3.2.2 API of `rbdd`

The implemented methods of the `rbdd` package that are available from the **R** side can be organized according to their goals, being organized into the categories described in the subsequent subsections. Figure 3.3 represents the main phases of the lifecycle of a BDD, in order to clarify the stage which the instructions apply to.

To easily identify the instructions provided by the `rbdd` library, all the methods start by the `bdd_` prefix, followed by a descriptive word(s) of the functionality implemented by them.

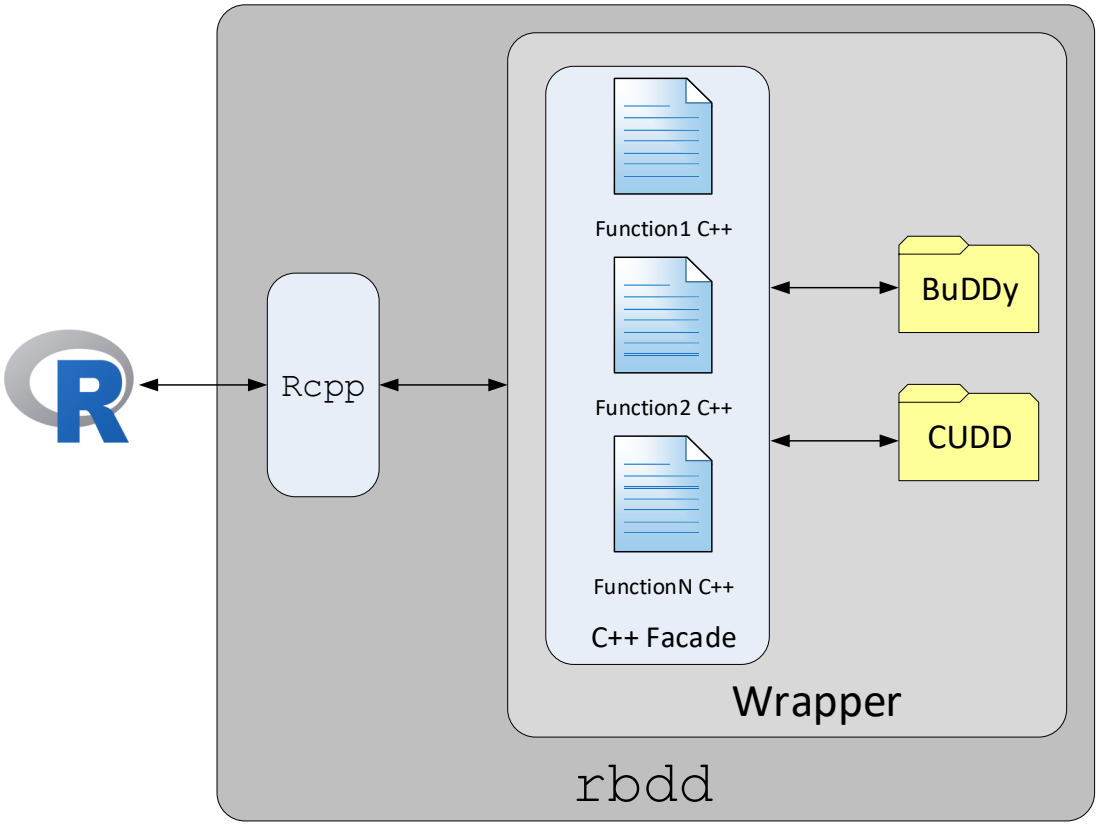


Figure 3.2: Architecture of the designed library

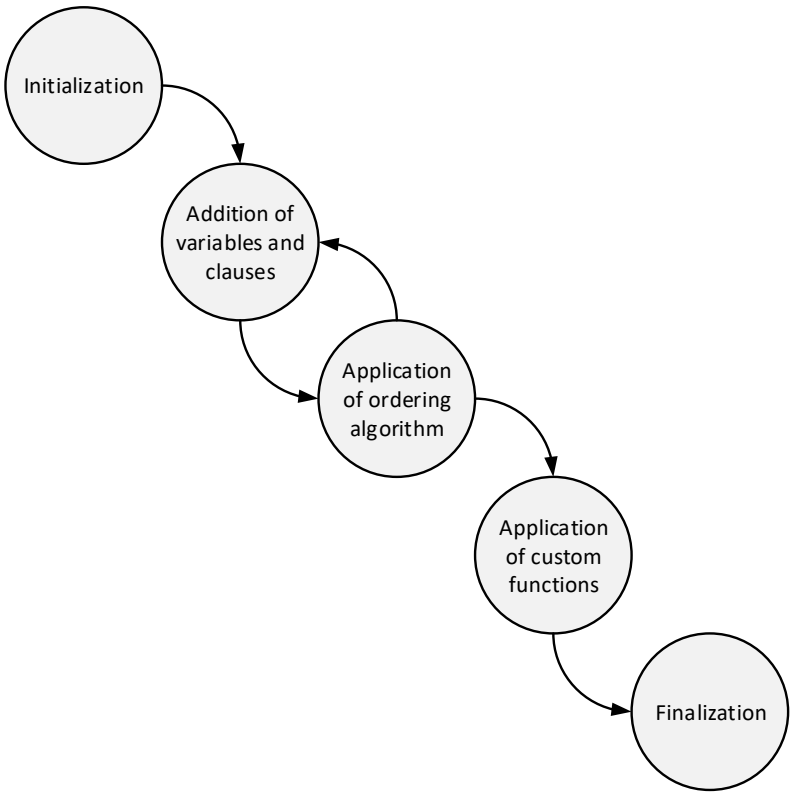


Figure 3.3: Lifecycle of a BDD

3.2.2.1 Initialization and finalization

These operations manage the main stages of the BDD lifecycle: its initialization and what it involves: setting up auxiliary structures, memory mananagement configuration, etc. (Table 3.1), and its finalization (Table 3.2), freeing the required resources by the diagram. There is also a function that resets the manager, what means a complete return of the manager structures to their initial states (Table 3.3).

CUDD works with an entity denominated *manager* [95], that is the element that assigns the resources for each BDD, keeps the relation of variables created, and all the other processes needed to perform the tasks related to the diagrams. These operations act directly over this entity, creating or destroying it, just as setting an initial configuration. On the other hand, BuDDy implements a similar concept, but in this library is called *package* but, in general terms, it represents an analogous concept [96].

Table 3.1: bdd_manager_init command

bdd_manager_init(library, node_num, cache_size, bdd_name)

This function creates an instance of the BDD manager. The user can choose the BDD manager library to work with, and its possible values are "buddy" and "cudd".

Also, the number of nodes and the size of the cache can be provided. If BuDDy is selected as manager, these values are set as 1000, and if the manager selected is CUDD, both values are 32 767.

Arguments:

`library`: (Optional) The library to use in order to implement the BDD operations. The possible values of this argument are "buddy" or "cudd". Any other value prompts an error message. If this value is omitted, "cudd" manager will be chosen.

`node_num`: (Optional) Number of nodes availables to allocate variables in the BDD. If BuDDy is selected as BDD manager, the default value is 1000 and for CUDD its value is 32 767.

`cache_size`: (Optional) Size of the cache of the factory, it improves the speed of the operations when instructions are executed repeatedly. The default value is 1000 for BuDDy and 32 767 for CUDD.

`bdd_name`: (Optional) Name of the BDD. It can contain letters, numbers and underscore.

Returned value:

N/A.

Examples:

```
bdd_manager_init()

bdd_manager_init(''buddy'')

bdd_manager_init(''cudd'')

bdd_manager_init(''buddy'', 2000)

bdd_manager_init(''cudd'', 2000)

bdd_manager_init(''buddy'', 2000, 5000)

bdd_manager_init(''cudd'', 2000, 5000)

bdd_manager_init(''buddy'', 2000, 5000, ''bdd_1'')
```

Table 3.2: bdd_manager_quit command

bdd_manager_quit(bdd_name)
This command finishes a BDD, liberating the memory space that it was using. **Arguments:** bdd_name: (Optional) Name of the BDD. **Returned value:** N/A. **Examples:** bdd_manager_quit() bdd_manager_quit(''bdd_1'')

Table 3.3: bdd_manager_reset command

bdd_manager_reset(bdd_name)
It ends the BDD factory and starts it again with the same BDD manager that was chosen in the bdd_manager_init command and the same parameters. **Arguments:** bdd_name: (Optional) Name of the BDD. **Returned value:** N/A. **Examples:** bdd_manager_reset() bdd_manager_reset(''bdd_1'')

3.2.2.2 Setting logical formulas functions

Once the BDD has been set up, it is time to populated it with a Boolean expression. To allow this task, rbdd provides different ways. For small BDDs, probably for testing purposes, the variables and clauses can be added manually, but that method is not practical in real models. In those cases, the BDD can be build passing the information through an input file. The ways available for adding items to the diagram are:

- Adding variables individually (Table 3.4).
- Inserting Boolean formulas manually, with the command exposed by Table 3.5.
- Declaring the logical expression in a Center of Discrete Mathematics and Theoretical Computer Science (DIMACS) CNF file [97] (Table 3.6). An example of a DIMACS CNF file is provided in Figure 3.4. Basically, it is composed by optional

comment lines, a problem line where the number of variables and clauses that the BDD contains is defined, and the clauses that define the logical operations between variables. The expression encoded in the referenced example is equivalent to the formula $(x_3 \vee x_2) \wedge (x_1 \vee \neg x_2)$.

- Informing the BDD in SPLOT format [66] [98], as is shown in Table 3.7. This method expects an eXtensible Markup Language (XML) file as input, Figure 3.5 can be consulted as a reference of a valid example, but basically it defines two blocks, the *feature_tree* with the hierarchical relationship between the variables, and the *clauses* section, with the Boolean operators between the previous defined variables.

```
c example.cnf
c
p cnf 3 2
3 2 0
1 -2 0
```

Figure 3.4: Example of a CNF file

```
<feature_model name="FeatureIDE model">
<feature_tree>
:r F_1
    :m F_2
        :m F_3
          :m F_4
              :g [1,*]
                  : F_5
                  : F_6
          :o F_7
          :o F_8
</feature_tree>
<constraints>
C1:~F_1 or F_2
C2:~F_3 or F_4 or F_8
C3:~F_5 or ~F_6 or ~F_7
</constraints>
</feature_model>
```

Figure 3.5: Example of a SPLOT file

Table 3.4: bdd_new_variable command

bdd_new_variable(variable_name, var_type, bdd_name)

This command creates a new variable to be used for the BDD manager.

Arguments:

`variable_name`: The name of the variable. It can only contain letters and numbers.

`var_type`: (Optional) Type of the variable. The possible values are "boolean" and "tristate". The default value is "boolean".

`bdd_name`: (Optional) Name of the BDD.

Returned value:

Index of the variable created. It returns -1 in case of error.

Examples:

```
bdd_new_variable(‘‘x’’)

bdd_new_variable(‘‘x1’’, ‘‘boolean’’)

bdd_new_variable(‘‘x2’’, ‘‘tristate’’)

bdd_new_variable(‘‘x3’’, ‘‘boolean’’, ‘‘bdd_1’’)
```

Table 3.5: bdd_parse_boolstr command

bdd_parse_boolstr(expression, bdd_name)

This instruction is used to populate a BDD after evaluating a logical expression.

Arguments:

`expression`: The expression to evaluate.

If the variables used do not exist in the factory, the method will create them. It also allows the use of parenthesis "()" to indicate the priority of the operations.

The logical operators implemented are:

- and ("x *and* y")
- or ("x *or* y")
- not ("*not* x")
- xor ("$xor(x\ y)$")
- if then ("$if\ x\ then\ y$")
- if then else ("$if\ x\ then\ y\ else\ z$")
- implies ("x - > y")
- equal ("$x = y$")

`bdd_name`: (Optional) Name of the BDD.

Returned value:

It returns 0 if the expression is parsed successfully, or -1 if something went wrong.

Examples:

```
bdd_parse_boolstr(''x and y or (not z and x)'')
bdd_parse_boolstr(''x and y or (not z and x)'', ''bdd_1'')
```

Table 3.6: bdd_parse_cnf command

bdd_parse_cnf(file, bdd_name, score, window, reorder, min_nodes, line_length, dyn_comp)

This function allows the user to inform a BDD defined in CNF.

Arguments:

`file`: The BDD defined in CNF.

`bdd_name`: (Optional) Name of the BDD.

`score`: (Optional) The scoring algorithm. Possible values are "none", "perm", "sifting", "force", "forceblocks", "random", "other" and "file". The default value is "none".

`window`: (Optional) The window size when the scoring algorithm selected is "perm". The default value is 1.

`reorder`: (Optional) The reorder algorithm to apply to the constraints. The possible values are "minspan", "minimax", "remember" and "none". The default value is "minspan".

`min_nodes`: (Optional) Parameter to indicate the mininum number of nodes that may exist to apply a reordering method. The default value is 100 000.

`line_length`: (Optional) Parameter to format the output. The default value is 27.

`dyn_comp`: (Optional) Flag to indicate if the algorithm has to calculate the connected variables and it reorders each component individually. The default value is TRUE.

Returned value:

It returns 0 if it creates the BDD successfully, −1 otherwise.

Examples:

```
bdd_parse_cnf(''cnfFile.cnf'')

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'')

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''sifting'')

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''perm'', 5)

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''force'', ''minimax'')

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''other'', ''remember'',
120000)

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''sifting'', ''none'',
100000, 30)

bdd_parse_cnf(''cnfFile.cnf'', ''bdd_1'', ''sifting'', ''none'',
100000, 30, FALSE)
```

Table 3.7: bdd_parse_splot command

bdd_parse_splot(file, bdd_name, score, window, reorder, min_nodes, line_length, dyn_comp)

This function allows the user to inform a BDD defined in SPLOT format, in a XML file.
Arguments:
`file`: The BDD defined in SPLOT format.
`bdd_name`: (Optional) Name of the BDD.
`score`: (Optional) The scoring algorithm. Possible values are "none", "perm", "sifting", "force", "forceblocks", "random", "other" and "file". The default value is "none".
`window`: (Optional) The window size when the scoring algorithm selected is "perm". The default value is 1.
`reorder`: (Optional) The reorder algorithm to apply to the constraints. The possible values are "minspan", "minimax", "remember" and "none". The default value is "minspan".
`min_nodes`: (Optional) Parameter to indicate the mininum number of nodes that may exist to apply a reordering method. The default value is 100 000.
`line_length`: (Optional) Parameter to format the output. The default value is 27.
`dyn_comp`: (Optional) Flag to indicate if the algorithm has to calculate the connected variables and it reorders each component individually. The default value is TRUE.
Returned value:
It returns 0 if it creates the BDD successfully, −1 otherwise.
Examples:
`bdd_parse_splot(''splotFile.xml'')`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'')`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''sifting'')`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''perm'', 5)`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''force'', ''minimax'')`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''other'', ''remember'', 120000)`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''sifting'', ''none'', 100000, 30)`
`bdd_parse_splot(''splotFile.xml'', ''bdd_1'', ''sifting'', ''none'', 100000, 30, FALSE)`

3.2.2.3 Ordering

It has been explained the criticality of the ordering issue in BDDs. For this reason, it appears as an imperative necessity to provide to the developed package the mechanism

to afford this functionality. Despite the different managers do not implement the same algorithms, both include the most common alternatives, and these are the set of possible options that can be chosen when the command is used. Table 3.8 shows the details of the function that allows to apply a reordering algorithm the diagram.

Table 3.8: bdd_order command

bdd_order(reorder_method, bdd_name)

This instruction allows to reorder the BDD depending on the method specified on the input parameter (if it is informed).
Arguments:
`reorder_method`: (Optional) The method for reordering the BDD. Possible values are "none", "window2", "window3", "sift" and "random". The default value is "sift".
`bdd_name`: (Optional) Name of the BDD.
Returned value:
N/A.
Examples:
`bdd_order()`
`bdd_order(''window2'')`
`bdd_order(''window2'', ''bdd_1'')`

3.2.2.4 I/O operations

When it is required to work with a defined benchmark where the BDDs are always the same, can be useful exporting and importing methods to save and load the existing structures, saving time when it would be necessary to recover them again.

For that purpose a function to save a created BDD to a file has been designed (Table 3.9) and another to recover it when it has been previously exported (Table 3.10). CUDD uses the DDDMP package [99] for its I/O instructions, while BuDDy implements itself that functionality.

Table 3.9: bdd_write command

bdd_write(file_name, bdd_name)

Instruction to save a BDD to a file. If BuDDy is chosen as BDD manager, the output extension is ".buddy". If CUDD is the manager, the extension will be ".dddmp".
 	
The file is saved in the current R's working directory.
 	
Arguments:
 	
`file_name`: The name of the output file.
 	
`bdd_name`: (Optional) Name of the BDD.
 	
Returned value:
 	
N/A.
 	
Examples:
 	
`bdd_write(''model_1'')`
 	
`bdd_write(''model_1'', ''bdd_1'')`

Table 3.10: bdd_read command

bdd_read(file_name, bdd_name)

Instruction to read a BDD from a file. If a name of BDD is provided, the content of the file will be load on a BDD with that name.
 	
Arguments:
 	
`file_name`: The name of the input file. The file must end in ".buddy" to store a BuDDy BDD or in ".dddmp" to store a CUDD BDD.
 	
`bdd_name`: (Optional) Name of the BDD.
 	
Returned value:
 	
N/A.
 	
Examples:
 	
`bdd_read(''buddyBDD.buddy'')`
 	
`bdd_read(''cuddBDD.dddmp'')`
 	
`bdd_read(''cuddBDD.dddmp'', ''bdd_1'')`

3.2.2.5 Applying functions to BDDs

The rbdd package has included some essential BDD functions. The interest of providing these methods on the library lies in their frequent use when this kind of diagram is chosen to represent Boolean expressions. These useful functions include:

- Comparison between BDDs (Table 3.11).

- Restriction [100] [101] [102]. This operation fixes a variable to a constant value, providing the possible solutions of the BDD considering that condition. The definition of this function is shown in Table 3.12.
- Traverse. rbdd includes a custom implementation of Bryant's traverse method (Table 3.13), allowing the execution of functions defined on the **R** side over the BDD nodes. Morover, this instruction entails the addition of some auxiliary functions, for obtaining information about the BDD of the node visited in each iteration on execution time:
  - Getting a pointer to the root node of the BDD (Table 3.14).
  - Checking if a node is the 0-terminal (Table 3.15).
  - Checking if a node is the 1-terminal (Table 3.16).
  - Getting the level of a node (Table 3.17).
  - Returning the variable in a given position of the diagram (Table 3.18).
  - Obtaining the children of a given node of the diagram (Table 3.19).

Table 3.11: bdd_identical command

bdd_identical(name_bdd_1, name_bdd_2)

This function compares two BDDs. The BDDs could be BDDs created with the bdd_manager_init and populated with the bdd_read or the bdd_parse commands, expressions which involve BDDs or in the case of the second expression, the logical constants "true" and "false".

The logic operations allowed between BDDs are:

- ! ("!bdd_1")
- && ("bdd_1 && bdd_2")
- || ("bdd_1 || bdd_2")
- != ("bdd_1 != bdd_2")
- == ("bdd_1 == bdd_2")
- < ("bdd_1 < bdd_2")
- > ("bdd_1 > bdd_2")

Arguments:

`name_bdd_1`: The name of the first BDD.

`name_bdd_2`: The name of the second BDD.

Returned value:

The result of comparing the BDDs.

Examples:

```
bdd_identical(``bdd_1'', ``bdd_2'')
bdd_identical(``!bdd_1 && bdd_2'', ``bdd_3'')
bdd_identical(``!bdd_1'', ``true'')
bdd_identical(``!bdd_1'', ``false'')
```

Table 3.12: bdd_restrict command

bdd_restrict(restriction, restriction_name, expression, positive_form, bdd_name)
This command creates a new variable to be used for the BDD factory. It restricts the value of a variable. **Arguments:** `restriction`: Name of the variable to restrict in the expression. `restriction_name`: The name of the variable. It can only contain letters and numbers. `expression`: Index of the expression to apply the restriction. Only required when the BDD manager is BuDDy. `positive_form`: (Optional) Indicates if the value to restrict is in its positive or negative form. `bdd_name`: (Optional) Name of the BDD. **Returned value:** Index of the variable created. It returns -1 in case of error. **Examples:** `bdd_restrict(''x'', ''bdd_1'')`

Table 3.13: bdd_traverse command

bdd_traverse(value_zero, value_one, function_to_apply, return_node, num_threads, bdd_name, trace)

Function that executes a **R** function across the BDD following the traverse algorithm. The **R** function must contain 5 or 6 input args, being this condition checked (and the existence of function in the **R** global environment) on execution time.

Arguments:

`value_zero`: The value(s) of the negative path.

`value_one`: The value(s) of the positive path.

`function_to_apply`: Name of the function to apply to the BDD.

`return_name`: (Optional) Indicates if the node pointer is returned. The default value is FALSE.

`num_threads`: (Optional) The number of threads to execute the method on multi-threading mode. The default value is 1.

`bdd_name`: (Optional) Name of the BDD.

`trace`: (Optional) If it is TRUE, it prints the results of intermediate steps of the traverse algorithm. The default value is FALSE.

Returned value:

The result of applying the traverse algorithm to the BDD.

Examples:

```
bdd_traverse(0, 1, ‘‘Rfunction’’)

bdd_traverse(c(0), c(1), ‘‘Rfunction’’, 2)

bdd_traverse(c(0, 5, 4), c(1, 0, 9), ‘‘Rfunction’’)

bdd_traverse(c(0, 4, 5, 1, 12), c(2, 5, 0, 3, 1), ‘‘Rfunction’’, 4,
‘‘bdd_1’’)

bdd_traverse(c(0, 4, 5, 1, 12), c(2, 5, 0, 3, 1), ‘‘Rfunction’’, 4,
‘‘bdd_1’’, TRUE)
```

Table 3.14: bdd_traverse_root_node command

bdd_traverse_root_node(bdd_name)

Function that gets the traverse root node of the BDD.

Arguments:

`bdd_name`: (Optional) Name of the BDD.

Returned value:

The pointer to the root node.

Examples:

```
bdd_traverse()
```

```
bdd_traverse(''bdd_1'')
```

Table 3.15: bdd_traverse_is_node_zero command

bdd_traverse_is_node_zero(node, bdd_name)

Function that checks if the given node is the 0-terminal node.

Arguments:

`node`: Index of the node to check.

`bdd_name`: (Optional) Name of the BDD.

Returned value:

Result of the query.

Examples:

```
bdd_traverse_is_node_zero(0)
```

```
bdd_traverse_is_node_zero(0, ''bdd_1'')
```

Table 3.16: bdd_traverse_is_node_one command

bdd_traverse_is_node_one(node, bdd_name)

Function that checks if the given node is the 1-terminal node.

Arguments:

`node`: Index of the node to check.

`bdd_name`: (Optional) Name of the BDD.

Returned value:

Result of the query.

Examples:

```
bdd_traverse_is_node_one(0)
bdd_traverse_is_node_one(0, ‘‘bdd_1’’)
```

Table 3.17: bdd_traverse_get_level command

bdd_traverse_get_level(node, bdd_name)

Function that returns the level in the BDD of a node.

Arguments:

`node`: Index of the node to compute its level.

`bdd_name`: (Optional) Name of the BDD.

Returned value:

Level of the node.

Examples:

```
bdd_traverse_get_level(0)
bdd_traverse_get_level(0, ‘‘bdd_1’’)
```

Table 3.18: bdd_traverse_get_var_at_pos command

bdd_traverse_get_var_at_pos(pos, bdd_name)

Function that returns the var which is in a given position.
Arguments:
`pos`: Position to be consulted.
`bdd_name`: (Optional) Name of the BDD.
Returned value:
Variable that is in the position.
Examples:
`bdd_traverse_get_var_at_pos(0)`
`bdd_traverse_get_var_at_pos(0, ''bdd_1'')`

Table 3.19: bdd_traverse_get_children command

bdd_traverse_get_children(ndoe, bdd_name)

Function that obtains the children of a node.
Arguments:
`node`: Index of the node.
`bdd_name`: (Optional) Name of the BDD.
Returned value:
Children of the node.
Examples:
`bdd_traverse_get_children(0)`
`bdd_traverse_get_children(0, ''bdd_1'')`

3.2.2.6 Debugging functions

Something usual when working with BDDs is checking some parameters of the diagram when it is built, such as the number of variables (Table 3.20) and nodes (Table 3.21) that compound it, to have a good measure of how it grows when new variables and clauses are added, or how it changes when different ordering algorithms are applied. Furthermore, printing functions are provided to obtain the value of the variables (Table 3.22) and clauses (Table 3.23) associated to the structure or even the solution of the Boolean expression that represents the whole BDD (Table 3.24).

Some operations at manager level have been added too. Thus, it is possible to consult if the manager has been initialized (Table 3.25) or the BDD library selected when the

manager has been created (Table 3.26).

Table 3.20: bdd_info_variable_number command

bdd_info_variable_number(bdd_name)

Function that gets the number of variables of a BDD.
Arguments:
`bdd_name`: (Optional) Name of the BDD.
Returned value:
Number of variables associated to a BDD.
Examples:
`bdd_info_variable_number()`
`bdd_info_variable_number(''bdd_1'')`

Table 3.21: bdd_info_node_number command

bdd_info_node_number(bdd_name)

Function that gets the number of nodes of a BDD.
Arguments:
`bdd_name`: (Optional) Name of the BDD.
Returned value:
Number of active nodes associated to a BDD.
Examples:
`bdd_info_node_number()`
`bdd_info_node_number(''bdd_1'')`

Table 3.22: bdd_info_variables command

bdd_info_variables(xverbose, bdd_name)

This function prints a table showing the index and the content of the variables created and returns a list with the variable list.

Arguments:

`xverbose`: (Optional) Flag to print the variables in the terminal. The default value is TRUE.

`bdd_name`: (Optional) Name of the BDD.

Returned value:

List containing the variables of the BDD.

Examples:

```
bdd_info_variables()

bdd_info_variables(‘‘bdd_1’’)

bdd_info_variables(FALSE, ‘‘bdd_1’’)
```

Table 3.23: bdd_info_boolstr command

bdd_info_boolstr(expression, bdd_name)

With this command the content of an expression is printed.

Arguments:

`expression`: The index of the expression to print.

`bdd_name`: (Optional) Name of the BDD.

Returned value:

N/A.

Examples:

```
bdd_info_boolstr(2)

bdd_info_boolstr(1, ‘‘bdd_1’’)
```

Table 3.24: bdd_print command

bdd_print(bdd_name)
This instruction prints the solution of a BDD. **Arguments:** bdd_name: (Optional) Name of the BDD. **Returned value:** N/A. **Examples:** bdd_print() bdd_print(‘‘bdd_1’’)

Table 3.25: bdd_manager_is_initialized command

bdd_manager_is_initialized(bdd_name)
This instruction allows to the user to know if the BDD manager has been initialized. **Arguments:** bdd_name: (Optional) Name of the BDD. **Returned value:** TRUE if the factory is initialized and FALSE if it is not. **Examples:** bdd_manager_is_initialized() bdd_manager_is_initialized(‘‘bdd_1’’)

Table 3.26: bdd_info_manager_library command

bdd_info_manager_library(bdd_name)

This instruction returns the name of the BDD manager chosen.
Arguments:
bdd_name: (Optional) Name of the BDD.
Returned value:
N/A.
Examples:
`bdd_info_manager_library()`
`bdd_info_manager_library(''bdd_1'')`

3.2.2.7 Customizing the environment of the BDD

At the time of the creation of a BDD, parameters as the cache ratio or the number of nodes can be passed as argument. However, it is possible that when the BDD increased in size those values could not be enough to contain the entire diagram. For this reasson, the instructions described in Table 3.27 and Table 3.28 have been included in rbdd, in order to allow increasing the size of the cache and the maximum number of nodes suitable in the structure, respectivelly.

Table 3.27: bdd_set_cache_ratio command

bdd_set_cache_ratio(cache_ratio, bdd_name)

This instruction allows to increase the cache ratio of the BDD.
Arguments:
cache_ratio: The increase to apply at the current cache ratio, used in order to improve the speed of the execution of the operations storing them in a temporary memory.
bdd_name: (Optional) Name of the BDD.
Returned value:
N/A.
Examples:
`bdd_set_cache_ratio(10)`
`bdd_set_cache_ratio(10, ''bdd_1'')`

Table 3.28: bdd_set_max_node_num command

bdd_set_max_node_num(size, bdd_name)

With this command the user can modify the maximum number of nodes of the created BDD.
Arguments:
`size`: The maximum number of nodes to set to the BDD factory, meaning the number of nodes that can be allocated in the structure.
`bdd_name`: (Optional) Name of the BDD.
Returned value:
N/A.
Examples:
`bdd_set_max_node_num(100)`
`bdd_set_max_node_num(100, ''bdd_1'')`

3.2.2.8 Algorithms implementation

Finally, an implementation of the algorithms studied in the previous chapter has been added to the `rbdd` package. These functions have been added because it has been considered that the information obtained is very relevant when BDDs are used to represent feature models in the field of SPLs. These algorithms are:

- Core and Dead (Table 3.29).
- Feature Probability (Table 3.30).
- Product Distribution (Table 3.31).
- Uniform Random Sampling (Table 3.32).

Table 3.29: bdd_get_core_dead command

bdd_get_core_dead(num_threads, trace, fast, bdd_name, restriction)

Function which applies the core and dead algorithm across the BDD following the traverse algorithm.

Arguments:

`num_threads`: (Optional) The number of threads to execute the method on multi-threading mode. The default value is 1.

`trace`: (Optional) If it is TRUE, it prints the results of intermediate steps of the traverse algorithm. The default value is FALSE.

`fast`: (Optional) Flag that indicates if the algorithm must be run in fast mode. The default value is FALSE.

`bdd_name`: (Optional) Name of the BDD.

`restriction`: (Optional) Restriction BDD to apply the algorithm.

Returned value:

The list with the core and dead features of the BDD.

Examples:

```
bdd_get_core_dead()

bdd_get_core_dead(1)

bdd_get_core_dead(1, TRUE)

bdd_get_core_dead(1, TRUE, TRUE)

bdd_get_core_dead(1, TRUE, TRUE, ‘‘bdd_1’’)

bdd_get_core_dead(1, TRUE, TRUE, ‘‘bdd_1’’, ‘‘restriction’’)
```

Table 3.30: bdd_get_var_probabilities command

bdd_get_var_probabilities(num_threads, trace, fast, bdd_name, restriction)

Function which applies the variables probability algorithm across the BDD following the traverse algorithm.

Arguments:

`num_threads`: (Optional) The number of threads to execute the method on multi-threading mode. The default value is 1.

`trace`: (Optional) If it is TRUE, it prints the results of intermediate steps of the traverse algorithm. The default value is FALSE.

`fast`: (Optional) Flag that indicates if the algorithm must be run in fast mode. The default value is FALSE.

`bdd_name`: (Optional) Name of the BDD.

`restriction`: (Optional) Restriction BDD to apply the algorithm.

Returned value:

The list with the probability of each variable of the BDD.

Examples:

```
bdd_get_var_probabilities()

bdd_get_var_probabilities(1)

bdd_get_var_probabilities(1, TRUE)

bdd_get_var_probabilities(1, TRUE, TRUE)

bdd_get_var_probabilities(1, TRUE, TRUE, ‘‘bdd_1’’)

bdd_get_var_probabilities(1, TRUE, TRUE, ‘‘bdd_1’’, ‘‘restriction’’)
```

Table 3.31: bdd_get_sat_distribution command

bdd_get_sat_distribution(num_threads, trace, fast, bdd_name, restriction)

Function which applies the SAT distribution algorithm across the BDD following the traverse algorithm.

Arguments:

`num_threads`: (Optional) The number of threads to execute the method on multi-threading mode. The default value is 1.

`trace`: (Optional) If it is TRUE, it prints the results of intermediate steps of the traverse algorithm. The default value is FALSE.

`fast`: (Optional) Flag that indicates if the algorithm must be run in fast mode. The default value is FALSE.

`bdd_name`: (Optional) Name of the BDD.

`restriction`: (Optional) Restriction BDD to apply the algorithm.

Returned value:

The list with the distribution of each variable of the BDD.

Examples:

```
bdd_get_sat_distribution()

bdd_get_sat_distribution(1)

bdd_get_sat_distribution(1, TRUE)

bdd_get_sat_distribution(1, TRUE, TRUE)

bdd_get_sat_distribution(1, TRUE, TRUE, ‘‘bdd_1’’)

bdd_get_sat_distribution(1, TRUE, TRUE, ‘‘bdd_1’’, ‘‘restriction’’)
```

Table 3.32: bdd_get_uniform_random_sampling command

bdd_get_uniform_random_sampling(num_threads, trace, fast, bdd_name, restriction)

Function which applies the URS algorithm across the BDD following the traverse algorithm.
Arguments:
`num_threads`: (Optional) The number of threads to execute the method on multi-threading mode. The default value is 1.
`trace`: (Optional) If it is TRUE, it prints the results of intermediate steps of the traverse algorithm. The default value is FALSE.
`fast`: (Optional) Flag that indicates if the algorithm must be run in fast mode. The default value is FALSE.
`bdd_name`: (Optional) Name of the BDD.
`restriction`: (Optional) Restriction BDD to apply the algorithm.
Returned value:
The list with the URS algorithm applied to the BDD.
Examples:
`bdd_get_uniform_random_sampling()`
`bdd_get_uniform_random_sampling(1)`
`bdd_get_uniform_random_sampling(1, TRUE)`
`bdd_get_uniform_random_sampling(1, TRUE, TRUE)`
`bdd_get_uniform_random_sampling(1, TRUE, TRUE, ‘‘bdd_1’’)`
`bdd_get_uniform_random_sampling(1, TRUE, TRUE, ‘‘bdd_1’’, ‘‘restriction’’)`

3.2.3 Installation an Usage of the rbdd Package

As any other **R** package, `rbdd` can be installed and loaded as follows from the **R** environment:

```
R> install.packages(‘‘rbdd’’)
R> library(rbdd)
```

To work with multiple-precision numbers, the library has defined the packages `gmp` [103] and `Rmpfr` [104] as dependencies, so they must be present for a correct installation. To install the package from the source code, it can be compiled with the command

```
$ R CMD INSTALL --build rbdd
```

The source code not only contains the files that implements the wrapper in what consists the library, but also the code of the C libraries that manage the BDDs, BuDDy and CUDD

taht is also compiled qhen the library is installed. It allows to generate shared libraries according to the operating system and architecture where the package is being installed, making possible to execute it in most environments without compatibility problems.

The package and its functions are documented, and this information can be checked using the `help` command:

```
R> help(rbdd)
R> help(bdd_manager_init)
```

Consider Figure 3.6 as an example of real usage of the `rbdd` library. It can be observed how some of the provided functions are utilized, in this case, for obtaining the list of core and dead features of a BDD representing a feature model, in this case the diagram that models the axTLS embedded Secure Sockets Layer (SSL) project [105]. The library is loaded in line 2, and the function called in the traverse method, `updateCoreDead`, is defined in line 7. Then, main function is implemented in line 38, and the first operation it process is the load of the BDD by file, through the `bdd_read` method, in line 40. To initilize the structures that are going to store the flags to indicate if a feature is core or dead, it is necessary to know how many variables has the BDD, so the `bdd_info_variable_number` is called (line 41). At this moment, the traverse call can be done to get what of the features are core or dead, informing the name of function to execute in each node and the initial values of the high and low paths (line 45). Finally, when the traverse method ends and the list are fully updated with the definitive values, the BDD manager can be close releasing the resources taken for the computations and storing the diagram. This is achieved with the command `bdd_manager_quit`, in line 46.

The execution from **R** of this code would be done with the instruction

```
R> getCoreDead(``axTLS.dddmp'')
```

where "axTLS.dddmp" represents a file storing a BDD, in this case in the format of the CUDD manager. This command would produce the following output:

```
R> getCoreDead(``axTLS.dddmp'')
$core
[1] 1

$dead
NULL
```

what represents that there is a core feature (the feature with index 1), and there is not dead features[1].

[1] This result concurs with the expected behaviour for the axTLS feature model, see Table 4.4.

```
# Loads the rbdd library
library(rbdd)

# Flag to show the trace in the traverse method
enableTrace <- FALSE

updateCoreDead <- function(plevel, tlevel, elevel, tr, er) {
        i <- plevel + 1
    while (i < tlevel) {
        core[[i + 1]] <<- 0
        dead[[i + 1]] <<- 0
                i <- i + 1
    }

        i <- plevel + 1
    while (i < elevel) {
        core[[i + 1]] <<- 0
        dead[[i + 1]] <<- 0
                i <- i + 1
    }

    if (tr == TRUE && er == TRUE) { # Both are true
        core[[plevel + 1]] <<- 0
        dead[[plevel + 1]] <<- 0
    }

    if (tr == TRUE && er == FALSE) {
        dead[[plevel + 1]] <<- 0
    }

    if (tr == FALSE && er == TRUE) {
        core[[plevel + 1]] <<- 0
    }

    return(TRUE) # This is not the zero node.
}

getCoreDead <- function(bdd_file) {
        # Creates and initializes the BDD manager from file
        bdd_read(bdd_file)
        numVariables <- bdd_info_variable_number()
        core <<- as.list(as.numeric(rep(1, length(numVariables))))
        dead <<- as.list(as.numeric(rep(1, length(numVariables))))

        bdd_traverse(FALSE, TRUE, "updateCoreDead", trace = enableTrace)
        bdd_manager_quit()

        realCore <- c()
        realDead <- c()

        for (i in 1:length(core)) {
                if (core[[i]] == 1) {
                        realCore[[length(realCore) + 1]] <- i
                }

                if (dead[[i]] == 1) {
                        realDead[[length(realDead) + 1]] <- i
                }
        }

        res <- list(core = realCore, dead = realDead)

        return(res)
}
```

Figure 3.6: Example of usage of the rbdd package

3.3 Core and Dead Features

Given a Boolean formula representing a variability model, modeled in a BDD, Algorithm 11 gives the complete list of core and dead features. Despite it will be explained in the next chapter (Section 3.1), a function called *traverse* is introduced. It is an implementation of the method defined by Bryant [51] and detailed in Algorithm 10. Simplifying, it is an efficient way to apply an algorithm recursively to the BDD, monitoring if a node has been visited, i.e., if the function has been already executed in that node and its children nodes.

Algorithm 11 core_and_dead

Input: bdd: the BDD
Output: list with the name of the core features,
list with the name of the dead features
Global: core_traverse: list [0..*size of variables* − 1] of int,
dead_traverse: list [0..*size of variables* − 1] of int

```
1:  begin
2:      initialize core_traverse and dead_traverse list elems to 1
3:      variables ← the variable list of the bdd
4:      traverse(bdd, false, true, update_core_dead)
5:      core and dead ← lists of string
6:      for i ← 0 to size of variables step 1 do
7:          if core_traverse[i] then
8:              push variables[i] in core
9:          end if
10:         if dead_traverse[i] then
11:             push variables[i] in dead
12:         end if
13:     end for
14:     return core and dead
15: end
```

This algorithm invokes the *traverse* function, in order to execute Algorithm 12 in every node of the BDD, that is the function that checks if the current node represents a core or a dead feature. After getting the lists of variables, they are iterated to get the name of the variables, making easier visualization tasks.

The output of the execution are two lists, with the entirety of core and dead features, and they could be useful to reduce the complexity of the real models, dismissing those variables because they are not relevant for building valid configurations. This is feasible because the fact that if core features are present in all the valid models, and dead features never take part on them, there is no reason to include these variables in the set of combinations.

Algorithm 12 update_core_dead

Input: plevel: int,
tlevel: int,
elevel: int,
tr: boolean (by reference),
er: boolean (by reference)
Output: boolean to indicate if the node is different to zero
Global: core_traverse: list $[0..size\ of\ variables-1]$ of int,
dead_traverse: list $[0..size\ of\ variables-1]$ of int

```
1:  begin
2:      for i ← plevel + 1 to tlevel − 1 step 1 do
3:          core_traverse[i] ← 0
4:          dead_traverse[i] ← 0
5:      end for
6:      for i ← plevel + 1 to elevel − 1 step 1 do
7:          core_traverse[i] ← 0
8:          dead_traverse[i] ← 0
9:      end for
10:     if tr ∧ er then
11:         core_traverse[plevel] ← 0
12:         dead_traverse[plevel] ← 0
13:     end if
14:     if tr∧ !er then
15:         dead_traverse[plevel] ← 0
16:     end if
17:     if !tr ∧ er then
18:         core_traverse[plevel] ← 0
19:     end if
20:     return true
21: end
```

3.4 Feature Probabilities

Algorithm 13 shows the proposed procedure to get the percentage of valid products that includes each feature and it is based on the work of Heradio et al. [71], Section III.B. To design the algorithm, two key concepts have been considered: the *node probability*, $p(n)$, that is the number of possible paths that go from the root node of the BDD to a terminal node passing through n divided by the total amount of paths, and the *joint probabilities*, $p(n, \Phi)$ the number of paths from the root to a terminal node where two conditions are satisfied divided by the number of possible combinations.

The algorithm first computes the variations between levels of each node and its high and low children, in order to know what is the maximum level difference. Algorithm 14 implements that functionality, iterating over each node being called through the Bryant's traverse function, ensuring that every node it is visited once. The result of this algorithm is used further as a helper structure to simplify the complexity of the arithmethic operations utilized to determine the number of viable products associated with every node of the BDD.

Next, the number of products in which each feature is present is computed by Algo-

Algorithm 13 get_var_probabilities

Input: bdd: the BDD
Output: list of mult. prec. float with the feature probabilities
Global: level_jump: set of int

```
1:  begin
2:      traverse(bdd, −1, 0, comp_level_jump)
3:      push 0 in level_jump
4:      r_zero and r_one ← list of mult. prec. int
5:      push 0 twice in r_zero
6:      push 1 twice in r_one
7:      rec_res ← traverse(bdd r_zero, r_one, up_products)
8:      i ← 1
9:      res ← list [0..size of rec_res − 3] of mult. prec. int
10:     while i < size of rec_res − 2 do
11:         push rec_res[i]/rec_res[0] in res
12:     end while
13:     return res
14: end
```

rithm 15. It considers the possible products traversing each children and stores then in the resulting list.

Finally, probabilities of each feature are obtained dividing the number of valid products in which every feature is present by the total number of products. The structures defined in the algorithm are designed to work with multiple-precision numbers, in order to keep as decimal precision as possible. Thus, the information remains unaltered and its statistical exploitation could redound to highly precise conclusions.

Algorithm 14 comp_level_jump

Input: plevel: int,
tlevel: int,
elevel: int,
tr: int (by reference),
er: int (by reference)
Output: the maximum difference between node levels
Global: level_jump: set of int

```
1: begin
2:     max ← −1
3:     if tr > max then
4:         max ← tr
5:     end if
6:     if er > max then
7:         max ← er
8:     end if
9:     if tr ≠ −1 then
10:        if level_jump does not contain (tlevel − plevel − 1) then
11:            push tlevel − plevel − 1 in level_jump
12:        end if
13:        if tlevel − plevel − 1 > max then
14:            max ← tlevel − plevel − 1
15:        end if
16:    end if
17:    if er ≠ −1 then
18:        if level_jump does not contain (elevel − plevel − 1) then
19:            push elevel − plevel − 1 in level_jump
20:        end if
21:        if elevel − plevel − 1 > max then
22:            max ← elevel − plevel − 1
23:        end if
24:    end if
25:    if max == −1 then
26:        max ← 0
27:    end if
28:    return max
29: end
```

Algorithm 15 up_products

Input: plevel: int,
tlevel: int,
elevel: int,
tr: list of mult. prec. int (by reference),
er: list of mult. prec. int (by reference)
Output: list of mult. prec. int
Global: level_jump: set of int

```
1:  begin
2:      then_part and else_part ← 0
3:      res_t and res_e ← list of mult. prec. int
4:      if tr[0] ≠ 0 then
5:          then_part ← tr[0] * 2^level_jump[tlevel−plevel−1]
6:          push then_part twice in res_t
7:          for i ← plevel + 1 to tlevel − 1 step 1 do
8:              push then_part/2 in res_t
9:          end for
10:         for i ← 1 to size of tr step 1 do
11:             push tr[i] * temp_then in res_t
12:         end for
13:     end if
14:     if er[0] ≠ 0 then
15:         else_part ← er[0] * 2^level_jump[elevel−plevel−1]
16:         push else_part in res_e
17:         push 0 in res_e
18:         for i ← plevel + 1 to elevel − 1 step 1 do
19:             push else_part in res_e
20:         end for
21:         for i ← 1 to size of er step 1 do
22:             push er[i] * temp_else in res_e
23:         end for
24:     end if
25:     if then_part ≠ 0 ∧ else_part ≠ 0 then
26:         push then_part + else_part in res
27:         i ← 1
28:         while i <= size of res_t − 1 do
29:             push res_t[i] + res_e[i] in res
30:             i ← i + 1
31:         end while
32:         return res
33:     end if
34:     if then_part == 0 then
35:         return res_e
36:     end if
37:     return res_t
38: end
```

3.5 Product Distribution

The algorithm proposed to obtain the product distribution is shown in Algorithm 16. The first step is the computation of the maximum level difference between a node and its children and it is reached with the function defined by Algorithm 14.

Algorithm 16 get_sat_distribution

Input: bdd: the BDD
Output: list of mult. prec. int with the variable distribution

```
1: begin
2:     max_jump ← traverse(bdd, −1, 0, comp_level_jump)
3:     make_combinations(max_jump)
4:     d_zero and d_one ← list [1] of mult. prec. int
5:     push 0 in d_zero
6:     push 1 in d_one
7:     return traverse(bdd, d_zero, d_one, dist_combine)
8: end
```

After obtaining this value, it is utilized to calculate a 2-dimension list containing the set of combinatorial numbers, $\binom{n}{m}$, where n goes from 1 to that maximum level computed previously, and m goes from 1 to n. The procedure to calculate that list in an efficient way is described in Algorithm 17 and the generated structure is used in the calculation of the distribution. Again, multiple-precision arithmetic is selected to represent the numbers, because in real BDDs the amount of products can exceed the limit easily that native data types support and would imply a considerable loss of precision that could affect the conclusions extracted from the analysis of the results.

Once the combinatorial numbers have been obtained and stored to reduce the computational effort, the distribution can be calculated. Again, the traverse function is useful to iterate over the BDD recursively, applying Algorithm 18 in each node. This algorithm computes the different combination of nodes for each path (low and high), because this is how the addition of features is represented mathematically (Heradio et al., Observations in Section III.C [71]).

Each iteration in this function modifies the structure where the distribution of the products is stored, so the list returned in the last execution contains the definitive computations. To study the distributions, it is usually to represent the figures in density plots, as the one presented in Figure 2.9 and Figure 4.6.

Algorithm 17 make_combinations

Input: n: int
Global: combinations: list $[0..n+1]$ of lists of mult. prec. int

```
1:  begin
2:      last_row ← list of mult. prec. list
3:      push 1 in last_row
4:      push last_row in combinations
5:      clear last_row
6:      push 1 twice in last_row
7:      push last_row in combinations
8:      x ← 2
9:      while x <= n do
10:         current ← list of mult. prec. int
11:         last_num ← 0
12:         k ← 0
13:         while k < x/2 + 1 do
14:             if k < (x − 1)/2 + 1 then
15:                 trav ← combinations[x − 1][k]
16:             else
17:                 trav ← combinations[x − 1][x − 1 − k]
18:             end if
19:             push last_num + trav in current
20:             last_num ← trav
21:             k ← k + 1
22:         end while
23:         push current in combinations
24:         x ← x + 1
25:     end while
26: end
```

Algorithm 18 dist_combine

Input: plevel: int,
tlevel: int,
elevel: int,
tr: list of mult. prec. int (by reference),
er: list of mult. prec. int (by reference)
Output: list $[0..elevel - plevel + size\ of\ er]$ of mult. prec. int
Global: combinations: list $[0..n + 1]$ of lists of mult. prec. int

```
1: begin
2:     initialize info_dist to 0
3:     if size of er ≠ 1 ∨ er[0] ≠ 0 then
4:         for a ← 0 to elevel − plevel + size of er − 1 step 1 do
5:             for b ← 0 to elevel − plevel − 1 step 1 do
6:                 if a − b >= 0 ∧ a − b < size of er then
7:                     if b < (elevel − plevel − 1)/2 + 1 then
8:                         dist ← combinations[elevel − plevel − 1][b]
9:                     else
10:                        dist ← combinations[elevel − plevel − 1][elevel − plevel − 1 − b]
11:                    end if
12:                    dist ← dist ∗ er[a − b]
13:                    dist ← dist + info_dist[a]
14:                    push dist in info_dist
15:                end if
16:            end for
17:        end for
18:    end if
19:    if size of tr ≠ 1 ∨ tr[0] ≠ 0 then
20:        for a ← 1 to tlevel − plevel + size of tr − 1 step 1 do
21:            for b ← 0 to tlevel − plevel − 1 step 1 do
22:                if a − b >= 0 ∧ a − b − 1 < size of tr then
23:                    if b < (tlevel − plevel − 1)/2 + 1 then
24:                        dist ← combinations[tlevel − plevel − 1][b]
25:                    else
26:                        dist ← combinations[tlevel − plevel − 1][tlevel − plevel − 1 − b]
27:                    end if
28:                    dist ← dist ∗ tr[a − b − 1]
29:                    dist ← dist + info_dist[a]
30:                    push dist in info_dist
31:                end if
32:            end for
33:        end for
34:    end if
35:    return info_dist
36: end
```

3.6 Uniform Random Sampling

The proposed alternative for generating these samples is reflected in Algorithm 19. It reads the variables associated with the BDD and obtains a list of Boolean indicators to determine if the variable that takes up each position is present or not in the sample configuration.

Algorithm 19 uniform_random_sampling

Input: bdd: the BDD
Output: list [0..*number variables bdd*] of boolean
Global: probabilities: map [0..*number variables bdd*] of nodes, mult. prec. float

```
1: begin
2:     create map of probabilities
3:     traverse(bdd, 0, 1, get_probabilities)
4:     return gen_random(bdd)
5: end
```

It calls the auxiliary function defined in Algorithm 20, relying on the Bryant's traverse procedure [51], to calculate the probability of each variable to be present in the sample. This probability is defined as is indicated in Equation (3.4):

$$\frac{2^{tlevel-plevel-1} * tr}{2^{tlevel-plevel-1} * tr + 2^{elevel-plevel-1} * er} \tag{3.4}$$

where *plevel* is the level of the node analyzed in the current iteration, *tlevel* is the level of the high child, *elevel* is the level of the low child, *tr* is the result processed by the high path, and *er* is the result obtained by the low path.

Algorithm 20 get_probabilities

Input: plevel: int,
tlevel: int,
elevel: int,
tr: mult. prec. int (by reference),
er: mult. prec. int (by reference),
node: current node of the bdd
Output: mult. prec. int
Global: probabilities: map [0..*number variables bdd*] of nodes, mult. prec. float

```
1: begin
2:     then_part ← 2^(tlevel−plevel−1) * tr
3:     else_part ← 2^(elevel−plevel−1) * er
4:     probabilities[node] ← then_part/(then_part + else_part)
5:     return then_part + else_part
6: end
```

Finally, once the probabilities have been obtained, Algorithm 21 goes through the BDD nodes. To guarantee that all the variables have a value assigned, it first generates a Boolean flag and is assigned to the list that is going to be returned as the result of the

algorithm. Then, it visits each node of the high path from the root node of the diagram and it updates the presence of the variables in the final result depending on a new random value and the probabilities computed before of this variable to be selected. When the entirety of the BDD has been visited, the solution is returned, containing the relation of variables and if they are chosen as a uniform random sample.

Algorithm 21 gen_random

Input: bdd: the BDD
Output: list [0..*number variables bdd*] of boolean
Global: probabilities: map [0..*number variables bdd*] of nodes, mult. prec. float

```
1:  begin
2:      exemplar ← list [0..number of variables of bdd] of boolean
3:      initialize elems of exemplar to false
4:      initialize trav to bdd root node
5:      initialize one to constant 1
6:      initialize zero to constant 0
7:      if trav == zero then
8:          return exemplar
9:      end if
10:     pos ← 1
11:     index ← level of trav
12:     for i ← 0 to index − 1 step 1 do
13:         random ← generate random number
14:         rand_num ← random/maximum value of random numbers
15:         exemplar[node in pos] ← rand_num < 0.5
16:         pos ← pos + 1
17:     end for
18:     while trav ≠ one do
19:         random ← generate random number
20:         rand_num ← random/maximum value of random numbers
21:         if rand_num <= probabilities[trav] then
22:             trav ← then branch of trav
23:             exemplar[node in pos] ← true
24:             pos ← pos + 1
25:         else
26:             exemplar[node in pos] ← false
27:             pos ← pos + 1
28:             trav ← else branch of trav
29:         end if
30:         for i ← index + 1 to level of trav step i do
31:             random ← generate random number
32:             rand_num ← random/maximum value of random numbers
33:             exemplar[node in pos] ← rand_num < 0.5
34:             pos ← pos + 1
35:         end for
36:         index ← level of trav
37:     end while
38:     return exemplar
39: end
```

Chapter 4

Experimental Validation

To test the usefulness and performance of the functional programming extension for BDDs this work proposes, a complete benchmark has been designed. The aim of the test is to prove the validity of the traverse algorithm implementations not only in terms of precision of the results but also in the reasonableness of execution time. When the size of the BDD increases, a bad design of the algorithm can lead to extremely high latency, giving as a result an unmanageable approach when working under real conditions.

The main purpose of this validation is, firstly, checking if the algorithms satisfy the minimum required conditions to consider them as a valid alternative to other known solutions and, additionally, to get an accurate comparison between both implementations, **R** and C++, in order to know in which conditions it is appropriate to utilize each one of them.

The benchmark has been executed in a 4 cores i7-6700HQ 2.60 GHz with 16 GB of RAM, 1 TB of disk and it has been run under Ubuntu 18.04.3 LTS [106], over a 64 bits architecture. The version of the **R** environment used to compile the library and to execute the benchmark has been the 3.6.2 [107].

4.1 Designed Benchmark

To elaborate the intended benchmark, a broad set of BDDs representing variability models and SAT problems coming from different domains (e.g., circuits) has been selected. These models have been gathered from previous works in the fields such as the studies of Heradio et al. [71] and Plazar et al. [74].

The BDDs in the bechmark are characterized in terms of their number of nodes of the BDDs, clauses, and variables. The graph depicted in Figure 4.1 shows the relation between the number of clauses and variables of the BDDs. The chosen sample goes from very small diagrams with barely a couple of dozens of nodes, to the biggest with about three and a half million, as it is represented in Figure 4.2.

Table 4.1 summarizes the descriptive statistics regarding the number of nodes of the benchmark models to help to understand how diverse is the set of BDDs selected, something essential to get representative values and reach a valid result in the current work. The satisfaction degree of the benchmark can be measured by paying attention to the diversity of the elements, varying on a significant scale on the size of the components of the diagram.

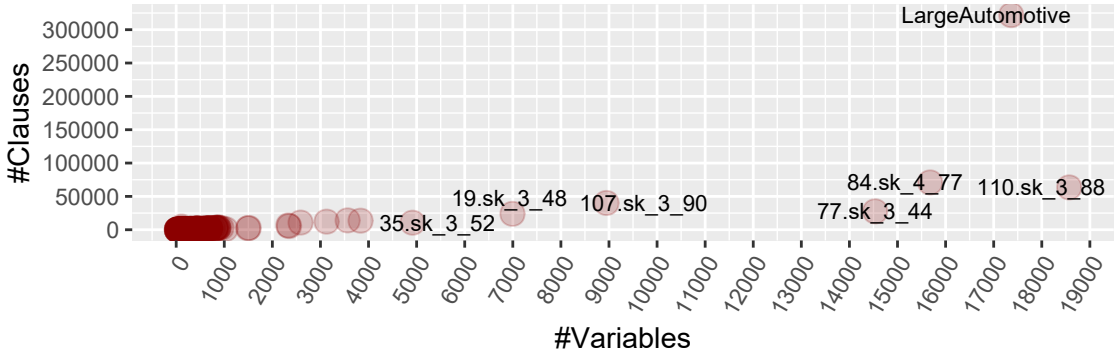


Figure 4.1: Size, in terms of the number of variables and clauses, of the benchmark models

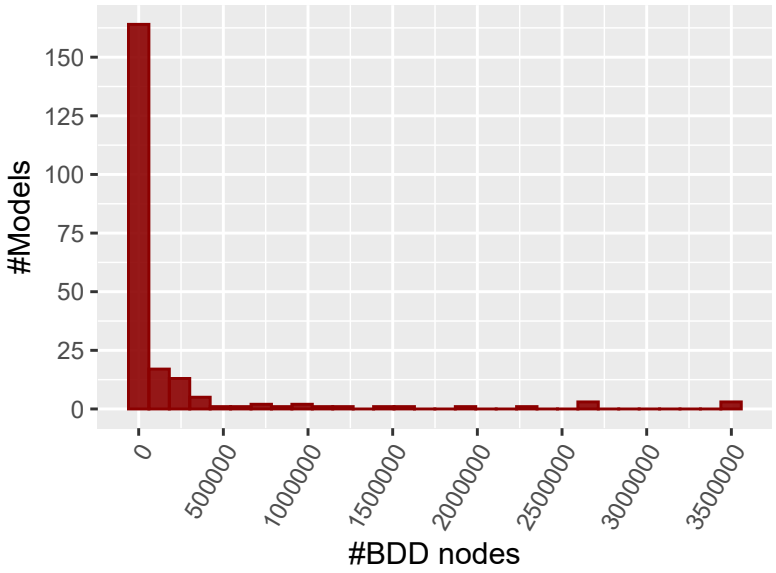


Figure 4.2: BDDs by number of nodes

Table 4.1: Statistical information about the benchmark

Minimum	1st **Quartile**	**Median**	**Mean**	3rd **Quartile**	**Maximum**
27	2546	12 319	190 756	58 880	3 498 303

Among the full group of diagrams, it is important to analyze a specific subset of them made up of those representing variability models. Table 4.2 shows the information of the variability models included in the benchmark. In the manner of the complete set of BDDs, the selection of variability models is as wide as possible, including a spread spectrum in terms of the main characteristics of these kinds of diagrams, such as the number of variables, clauses and nodes.

Table 4.2: Variability models included in the benchmark

Model	**#Variables**	**#Clauses**	**#BDD nodes**
JHipster	45	104	113
axTLS	64	96	116
Fiasco	113	4717	229
DellSPLOT	118	2181	2144
uClibc	298	903	2935
ToyBox	544	1020	703
BusyBox	613	530	1475
EmbToolkit	2331	6437	606 522
LargeAutomotive	17 365	321 897	30 432

4.2 Analysis of the Results

In order to validate the implementation of the selected algorithms, the explained benchmark has been run over the versions developed in **R** and C++. The expected result is that the performance in C++ should be better than the **R** implementation, because C++ is an imperative language [108], closer to the machine code, and where the programmer needs to manage the computer memory manually. In contrast, R is a functional language, more abstract than C++, that manages the computer memory automatically [109].

To reach a reasonable accuracy in the results, the usage of an external library to provide multiple-precision arithmetic has been required. By default, both C++ and **R** have a limitation on the range of numbers that can be represented by their core numeric data types[1], so for really big BDDs, in the order of thousands of nodes, that would be a real restriction. The library used for that purpose has been the GNU Multiple Precision (GMP) arithmetic library, in its version for C++ [110] and **R** [103]. Basically, the **R** version is a wrapper of the C++ one, adapting the interface to its framework, which adds latency in the operations, making them slower than when the original library is used in C++ directly.

The usage of this library has a cost on the performance of the algorithm resolution because of handling the arithmetic operations by the software utility, instead of solving it by the hardware, as happens when the base integer or float data types are used. Table 4.3 represents a comparison of the execution time and the loss of precision between using primitive types and the GMP library in C++ when some arithmetic operations are made (additions, subtractions, multiplications and divisions, where the magnitude of the initial numbers is indicated in the scale column). The execution time in the case of the

[1] Up to 18 446 744 073 709 551 615 and 2 147 483 647, respectively.

C++ primitive data types remains stable and lower than when the GMP is used, but that is because it is always working with smaller numbers, with the consequent loss of precision in the results computed.

Table 4.3: Comparison between hardware and software execution

Scale	Primitive types		GMP library	
	Execution time (ns)	**Loss precision**	**Execution time (ns)**	**Loss precision**
10^6	167	0	18 236	0
10^9	254	10^{19}	26 997	0
10^{12}	164	10^{24}	28 401	0
10^{15}	184	10^{30}	29 222	0
10^{18}	187	10^{37}	31 579	0
10^{21}	171	10^{41}	33 881	0

4.2.1 Core and Dead Features

The algorithm designed to obtain the core and dead features is the only one of the presented in this work that does not depend on the use of arithmetic operations with large precision (it can be checked on Algorithm 11), so the execution time is the lowest compared with the other functions. Table 4.4 shows the execution times when the algorithm is applied to the variability models included in the benchmark. The number of core and dead features has been included because it helps to get a rough idea of how well is the expression represented by the BDD built, as it is deeply explained in Pérez Morago et al. [69].

Table 4.4: Core and dead execution results

Model	#BDD nodes	#Core Features	#Dead Features	C++ time (seconds)	R time (seconds)
JHipster	113	7	0	0.001992941	0.032933
axTLS	116	1	0	0.0009958744	0.03294683
Fiasco	229	0	39	~ 0	0.06248498
DellSPLOT	2144	1	0	0.02493596	0.158576
uClibc	2935	0	19	0.031914	0.25033
ToyBox	703	4	365	0.008975983	0.10176492
BusyBox	1475	5	20	0.01595712	0.1496
EmbToolkit	606 522	59	619	6.576404	99.29467
LargeAutomotive	30 432	1686	6	0.4757569	32.83624

Figure 4.3 depicts the linear regression model of the runtimes of both implementations of the Core and Dead algorithm for the whole benchmark. It is clear the linear relation between the number of nodes of the BDD and the execution time. The ratio between the C++ version and the **R** implementation is 1:10, so that indicates whenever the number of nodes increases, the **R** version gets worse, but it is still spending an acceptable time to

obtain the solution for the largest diagrams.

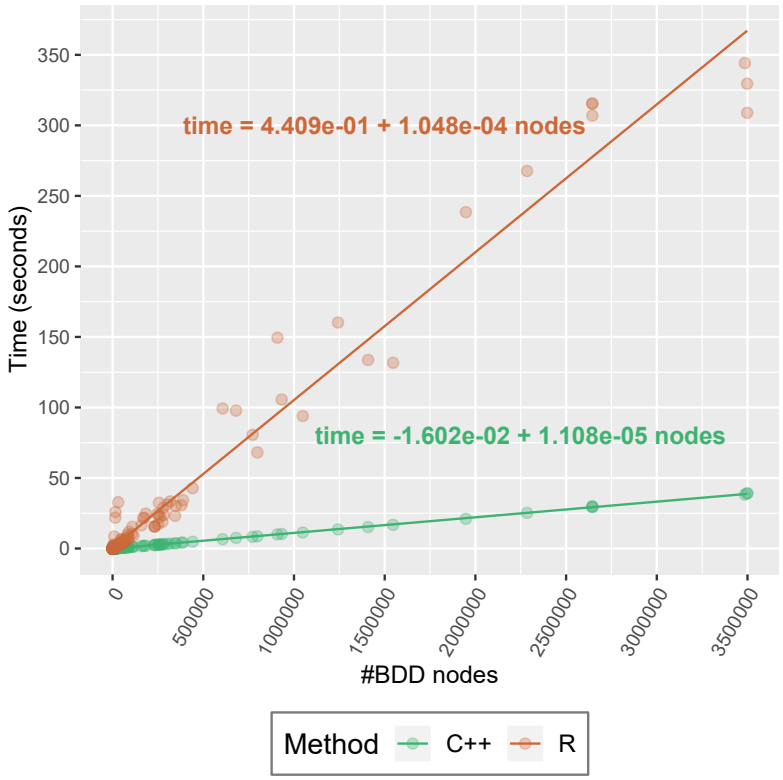


Figure 4.3: Core and dead runtimes

4.2.2 Feature Probabilities

Coming back to the subject of the analysis of the variability models included in the benchmark, Algorithm 13 allows getting the probability of each feature to be selected, which is an important property when a large model has to be managed.

The result of the execution of the algorithm in each variability model is represented in Figure 4.4. It shows how some of the models have several features that will never be included in a product (those that have a great number of features with a 0 probability of being selected). Keeping this information in mind can lead to reduce the complexity of the diagram, considering only the effective features and reducing the magnitude of the problem.

Paying attention to the runtimes of the implementations of the algorithm, the differences are insignificant for those diagrams with a low density of nodes, getting the solution in a very reasonable time for both implementations. However, when the size of the BDD increases, that difference grows dramatically depending on the language, reaching the **R** execution time up to 16 times slower than the C++ alternative. Table 4.5 exemplifies the runtimes of the algorithm when it is executed in the C++ and **R** versions, and the nodes of the BDD, making it easy to relate the evolution of the times exposed with the size of the structures.

Comparing these times with the ones obtained in the execution of the dead and core algorithm, the main difference resides in the fact of making use of the multiple-precision

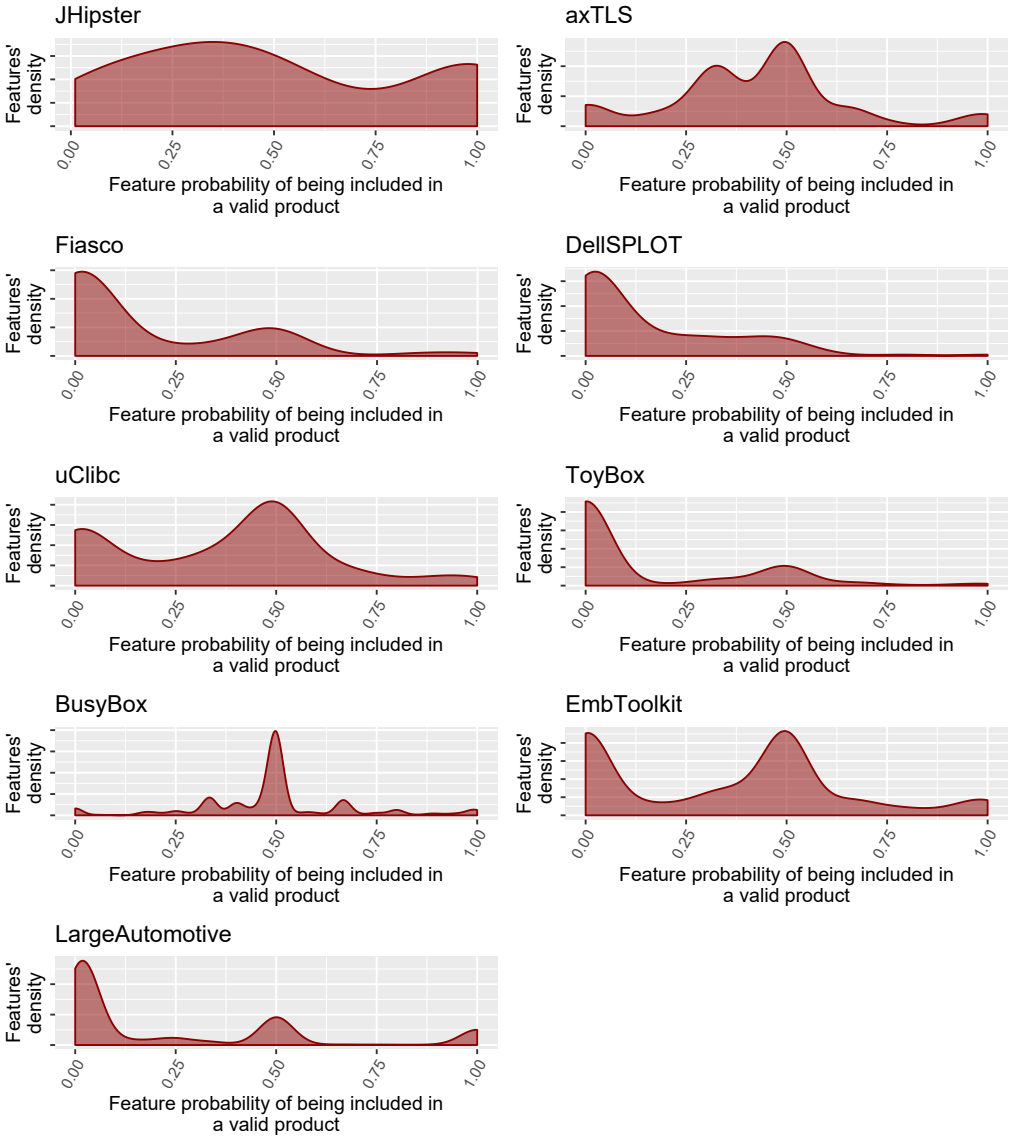


Figure 4.4: Variable probabilities of the variability models

Table 4.5: Variable probabilities execution times

Model	**#BDD nodes**	**C++ time (seconds)**	**R time (seconds)**
JHipster	113	0.27145	0.403857
axTLS	116	0.26861	0.481344
Fiasco	229	0.2811849	0.7436872
DellSPLOT	2144	0.2536831	2.186338
uClibc	2935	0.760865	6.939845
ToyBox	703	0.5129941	2.957633
BusyBox	1475	0.6535871	6.123531
EmbToolkit	606 522	451.5049	7641.154
LargeAutomotive	30 432	472.8244	4680.473

arithmetic operations. For the extreme cases, additions, multiplications and divisions are made over increasing integers as long as the number of nodes grows. As a result, the final iterations operates with really large numbers (on the order of 10^{180}), which comes from an important computational cost.

Furthermore, the gap between the times according to the technology can be explained if the different memory management, data structures and the GMP library behaviour for both environments is kept in mind.

To finish the study of the results of the feature probabilities algorithm, the general performance of the function is depicted in Figure 4.5, where the conclusion that can be inferred for the whole benchmark is consistent with the previous analysis of the BDDs representing only variability models. Times for small diagrams are similar for the C++ and **R** methods, but when it is needed to work with larger structures, the runtime of the **R** implementation grows until not enough manageable thresholds.

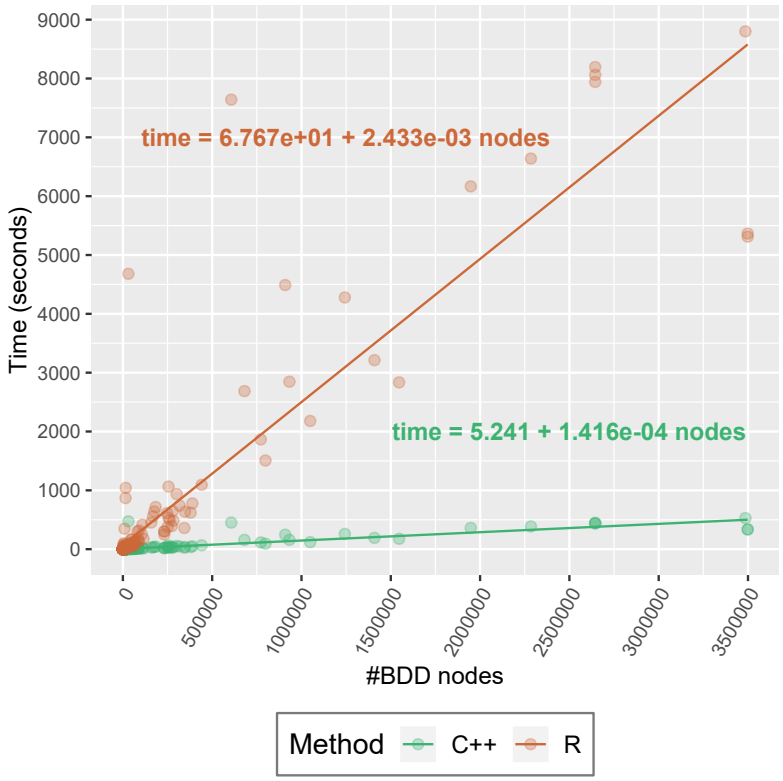


Figure 4.5: Variable probabilities runtimes

4.2.3 Product Distribution

The SAT assignments distribution, as it has been explained previously, gives a clear image of the main structure of the BDD and the relationship between the features that compound it. Regarding the variability models that concern the current work, the results of applying Algorithm 16 are shown in Figure 4.6.

In light of the normal distributions obtained with these models, some facts can be deduced. As a generalized comment, it could be proposed that it is not usual to get products

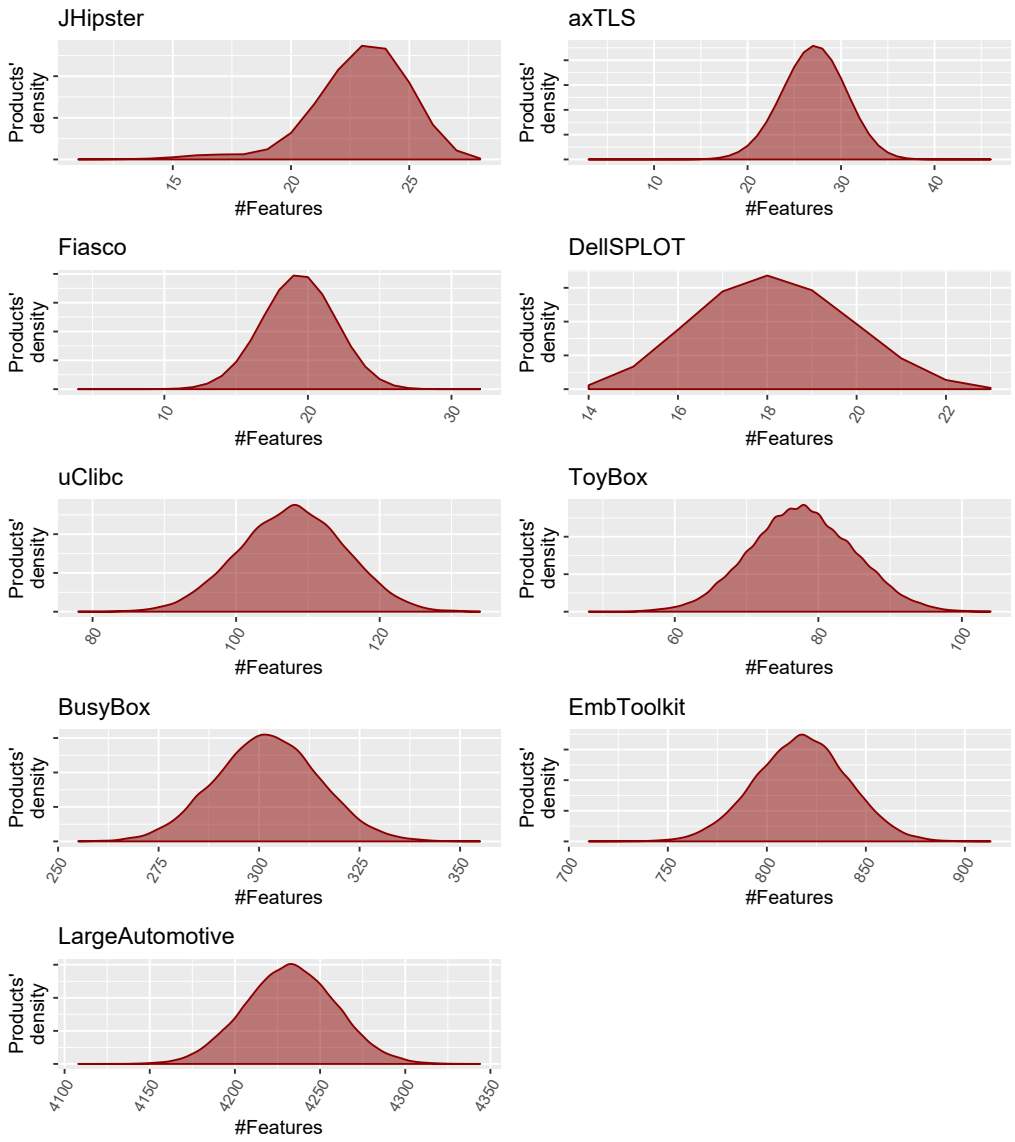


Figure 4.6: SAT assignments' distribution

with an extreme number of features, that is, with very few features or close to the total number of them. Instead of this, the behaviour of that kind of models is that the majority of the possible products are situated, to a greater or lesser extent, in an intermediate rank. Even more, nearly 100% of the potential products can be obtained with a number of features that fluctuate around 5%-20% of the total amount of the included features in the model. An extreme case is the model represented by LargeAutomotive, because it concentrates most of the possible products in a range of only 1% of the total number of features that it consists of.

In terms of performance of the algorithm for both versions, Table 4.6 reflects a comparative analysis, where the conclusions are concordant with those extracted for the Feature probabilities algorithm. In this case, the differences between C++ and **R** are even more pronounced, because the magnitude of the numbers and the amount and complexity of the multiple-precision operations are greater. The management of the data structures is another important factor in this case, especially for those diagrams with more than 1000 nodes, because the read and write operation in large lists, maps or vectors have an ad-

ditional penalty in terms of latency. In Table 4.7 the operations that take more time to be completed in the **R** version for the LargeAutomotive variability model are represented, with the average time and the reference to the respective line of the algorithm presented in Section 2.6.

Table 4.6: SAT assignments' distribution execution times

Model	**#BDD nodes**	**C++ time (seconds)**	**R time (seconds)**
JHipster	113	0.228837	0.3736188
axTLS	116	0.209224	0.476455
Fiasco	229	0.2655621	0.7029588
DellSPLOT	2144	0.183919	2.615166
uClibc	2935	0.7653599	11.63289
ToyBox	703	0.3842249	3.791659
BusyBox	1475	0.8007441	13.7684
EmbToolkit	606 522	612.1206	21 331.26
LargeAutomotive	30 432	326.815	7748.536

Table 4.7: Most expensive operations of the Product Distribution algorithm in **R**

Operation	**Average time (seconds)**	**Algorithm line**
Multiplication mult. prec. int	4337.240 540 1	12 and 28
Addition mult. prec. int	1544.730 256	13 and 29
Storage mult. prec. int	287.5610877	14 and 30

If the runtimes of the whole benchmark are analyzed, the resulting trend is analogous to the one obtained for the other algorithms, as Figure 4.9 shows. This is the expected behaviour, because the specific demand for the execution of this algorithm is proportioned for the two versions, so the regression model should be compensated.

4.2.4 Uniform Random Sampling

For the study of the results of this algorithm, the concept of *goodness-of-fit* will be introduced. The objective of this technique is to "examine how well a sample of data agrees with a given distribution as its population" [111]. The goodness-of-fit of a sample can be tested by comparing each feature's theoretical probability ft with its empirical proportion fe obtained from the sample. This can be done with the chi-squared statistic defined by Equation (4.1):

$$X^2 = \sum_{\forall \text{ non-dead feature } i} \frac{(\text{ft}_i - \text{fe}_i)^2}{\text{ft}_i} \tag{4.1}$$

Note that dead features have $\text{ft}_i = 0$, and thus they are omitted to guarantee that X^2 is a finite number. In fact, checking the empirical frequency of dead features would

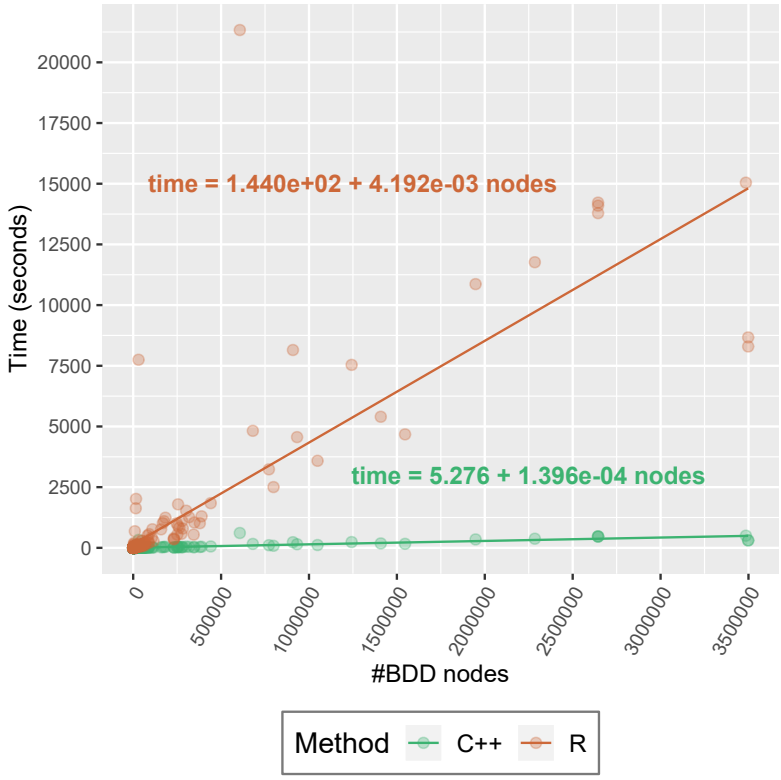


Figure 4.7: SAT assignments' distribution runtimes

be unnecessary because Knuth's algorithm [112] ensures that all generated products are valid, and so they do not include any dead feature. X^2 has approximately the χ^2 null distribution with the number of non-dead features minus one degrees of freedom in large samples, and thus the correspondent p-value can be calculated. If the p-value is greater than a threshold α, then evidence supports that the sample comes from the population (α is usually set to 0.05).

To determine the size of the sample, Cohen (Chapter 7) [113] defines the effect size w as "the *degree* to which the phenomenon is present in the population" or "the degree to which the null hypothesis is false" ($w = 0.1$ (small), $w = 0.3$ (medium), $w = 0.5$ (large)).

Cohen also proposes several power tables in Section 7.3 of [113] that summarizes the dependencies among Power (β), Significance level (1-α), Effect size (w), and Sample size (N). Having three of those parameters, the fourth one can be inferred. Table 4.8 and Figure 4.8 show the goodness-of-fit for $\beta = 0.05, \alpha = 0.05, w = 0.3$. For all the variability models the result obtained is near 1, which indicates the sampling implementation generates a result that agrees successfully with the distribution of the sample.

The runtimes reflect again the connection between the number of nodes of the BDD (its size) and the time needed to finish the computations. In this case, the times are closest to the ones obtained in the execution of the Core and Dead algorithm, because the mathematical operations are similar for both methods.

The same conclusion can be extrapolated for the entire benchmark, as demonstrates Figure 4.9, where the same tendency than for variability models is reproduced. The reason for the latency of this function is it only needs to make use of less arithmetical operations

Table 4.8: Uniform Random Sampling execution times

Model	**#BDD nodes**	**#Cases per sample**	**Goodness-of-fit p-value**	**C++ time for a single case (seconds)**	**R time for a single case (seconds)**
JHipster	113	427	0.919374	0.001994848	0.088763
axTLS	116	521	0.927117	0.00199604	0.09474707
Fiasco	229	556	0.994418	~ 0	0.2613189
DellSPLOT	2144	672	0.999713	0.02895284	1.590778
uClibc	2935	979	~ 1	0.03593993	2.339773
ToyBox	703	798	~ 1	0.009006977	0.5475621
BusyBox	1475	1370	~ 1	0.01994705	1.170869
EmbToolkit	606 522	2220	~ 1	7.338366	430.4086
LargeAutomotive	30 432	6591	~ 1	0.404907	21.94691

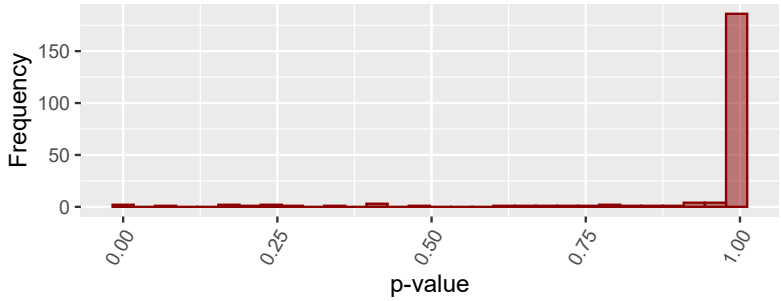


Figure 4.8: Goodness-of-fit p-values. The histogram includes all models

with numbers smaller than the required in other algorithms like the ones to obtain the feature probability or the SAT distribution.

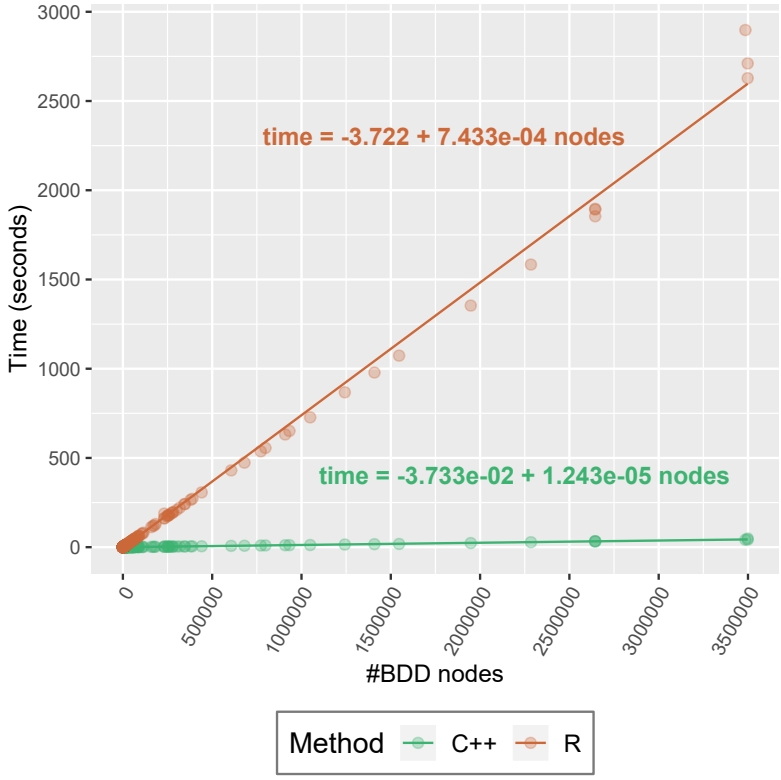


Figure 4.9: Uniform Random Sampling runtimes

4.3 Final Comments about the Experimental Validation

As a summary, and once the results have been evaluated, some general considerations can be made. Paying attention to the size of the diagram, the version chosen is not relevant when the BDD is small. However, as the number of nodes of the BDD grows, it becomes a critical factor. The C++ implementation is always faster because of some well-known reason, such as the dependency with the GMP library, the size of the data structures and the magnitude of the numbers.

The main reason to keep both versions, in spite of the differences in the performance of the operations, it is that **R**, due to its functional programming nature, supports writing algorithm prototypes much faster than C++. Moreover, there are circumstances where it could be more interesting to make use of this implementation, because the execution time it is not so important than the information given by **R**.

Finally, all the algorithms studied in this work utilize Bryant's traverse function [51], which is also included in the **R** library developed. It means that if someday there is the need of build new algorithms which are based in that function, it could be done in **R** with the current version of the wrapper, while to include the equivalent in C++ and the exposure of the function from the library, new developments and compilations must be done.

Chapter 5

Conclusions and Future Work

When BDDs are selected to represent a Boolean formula and that structure, in turn, depicts a variability model, some observations must be considered. Firstly, due to its nature, BDDs are a good option to encode logical expressions as a compressed representation of relations between nodes. In addition, it is easy to subsequently extend the diagram adding variables and clauses dynamically and, once it is built, Bryant's traverse function allows running custom functions in the nodes of the diagram. This feature has been demonstrated completely fundamental working with BDDs. On the other hand, the size of the diagram may grow exponentially if the right ordering algorithm is not applied. Another factor to study is the BDD manager that can be used for the purpose, because obtaining efficient implementations of the diagrams is complex and dependent on the library chosen. Regarding the previous points, the memory needed to store the information of the nodes that compound the diagram varies depending on the manager selected, and can be higher than other options of representing Boolean expressions. Table 5.1 summarizes the main advantages and disadvantages of the utilization of BDDs to reach the objectives defined in this work.

Table 5.1: Advantages and disadvantages of using BDDs

Advantages	**Disadvantages**
Suitable for logic expressions	Extremely sensitive on the ordering
Dynamic construction of the diagram	Complex efficient implementations
Bryant's traverse function	Potentially high memory cost

5.1 Conclusions

After the application of a benchmark composed of a sufficiently large number of real feature models to the designed library, it has been demonstrated that `rbdd` is capable of managing the lifecycle of a BDD successfully. `rbdd` provides methods to create the BDD manager and build all the auxiliary structures efficiently, exposing initialization and finalization features, wrapping the load and save functions of the BuDDy and CUDD libraries, and implementing custom CNF, SPLOT and Boolean parsers to include new variables and clauses, and applying heuristic techniques for ordering those variables [67].

The `rbdd` package has been designed to be as adaptable as possible, so if according to new requirements, additional functionalities or BDD managers are needed, it can be done

easily. The wrapper in which the architecture is based provides a flexible environment that allows the addition of new libraries without involving changes in the defined API, as long as those updates do not imply modifications on the signature of the operations.

The usage of this package has been proved as a valid tool to implement successfully a variety of algorithms to analyze configurable software system models. The results obtained after the execution of the operations over the proposed benchmark concur with the anticipated from the theoretical basis and the related work.

As it was expected, among the two available implementations of the algorithms, the best results are obtained when the execution is done using the C++ option, that is, utilizing the specific method exposed by `rbdd`. That makes sense because, as it has been explained previously, C++ is a general purpose programming language, with a more efficient management of the memory, the data structures and custom functions. However, **R** provides a good environment to explore the information statistically, which helps engineers and designers of the variability models to analyze how well is the diagram built, and under what conditions the model should be improved.

In addition, the fact of having in `rbdd` a version of Bryant's traverse algorithm gives the opportunity of implementing other functions that could be interesting for SPL engineers (or professionals of any other field that work with BDDs), thus covering new needs that might appear on the market. Making use of this function can lead to new designs without evolving the package, just encoding the functionality in **R** using the methods explained for operating with BDDs.

Considering the execution times, and the performance of applying the functions over the largest models present in the benchmark, it is revealed that the C++ implementation of the algorithms can reach the solutions of the problems on a reasonable time, while the **R** version efficiency worsens when the size of the diagram reaches medium-large values. This is mainly because of the handling of big numbers and their application on multiple-precision arithmetic, and the access and updating of large lists, despite of the improving process that the code has been gone through. Therefore, for large and complicated configuration models, the `rbdd` **R** version is more suitable for implementing algorithm prototypes that later, once they are validated, should be encoded into C++ to achieve their full potential.

With regard to other operations of the lifecycle of the BDDs, as creating the structure, populating the diagram with clauses and variables, consulting information such as the number of nodes or variables that compound the BDD, etc., there is not a real difference in terms of efficiency with the usage of the libraries directly in C or C++. The development of these operations in `rbdd` has not implied much more encoding than the appropriate call to the respective BDD manager library function, and for the most complex cases, the execution of parsers and the management of some intermediate data structure, but regardless without a relevant deterioration of the performance.

Finally, the usefulness of the results calculated with the library has been highlighted, with the capability to representing them into graphs and plots facilitating a deeper analysis of them. This depiction of the datasets allows understanding an information, that otherwise, in largest models (hundreds of thousands of nodes), could not be possible to afford through other alternatives.

5.2 Future Work

In the near future, we hope this work helps to obtain efficient and maintainable algorithms on BDDs. An example could be the analysis of multistate systems, proposed by Xing et al. [114], but the specific literature is plenty of candidates to be encoded with the designed tool.

The plots reported in Chapter 4 were been created a posteriori, once the functions returned a dataset with the solution of the problem. An option to enrich the `rbdd` package could be adding the automated generation of those types of plots when the execution of the algorithms is over.

Another point to improve, as it has been discussed previously, is the performance of the algorithms when they are implemented in **R**. The adaptation of the code to utilize some functions such as those belonging to the `apply` family [115], that allow executing functions over array margins. Some other advanced data structures could be taken into account to replace the standard lists and vectors included in the core environment of **R** [116].

The current version of the package allows the management of BDDs making use of two libraries: BuDDy and CUDD. There are other options for creating and working with these diagrams [117], such as CacBDD [118], Sylvan [119] or the BDD library developed by the Carnegie Mellon University [120].

Currently, when `rbdd` prints a BDD through the `bdd_print` function, it just returns the set of valid solutions that satisfies the Boolean expression contained in the diagram, but this is not the only way to depict a BDD. It could be returned as its graph representation, retrieving a visual model of the relationships between the nodes or features of the diagram. Figure 5.1 shows an example of the `igraph` package [121] [122] [123] to generate the diagram corresponding to the EmbToolkit configuration model [71]. Furthermore, `igraph` supports various analysis of interest for SPLs, such as identifying relevant features that have high *centrality*, or detecting inter-feature dependency topological patterns (e.g., the rich-club connectivity).

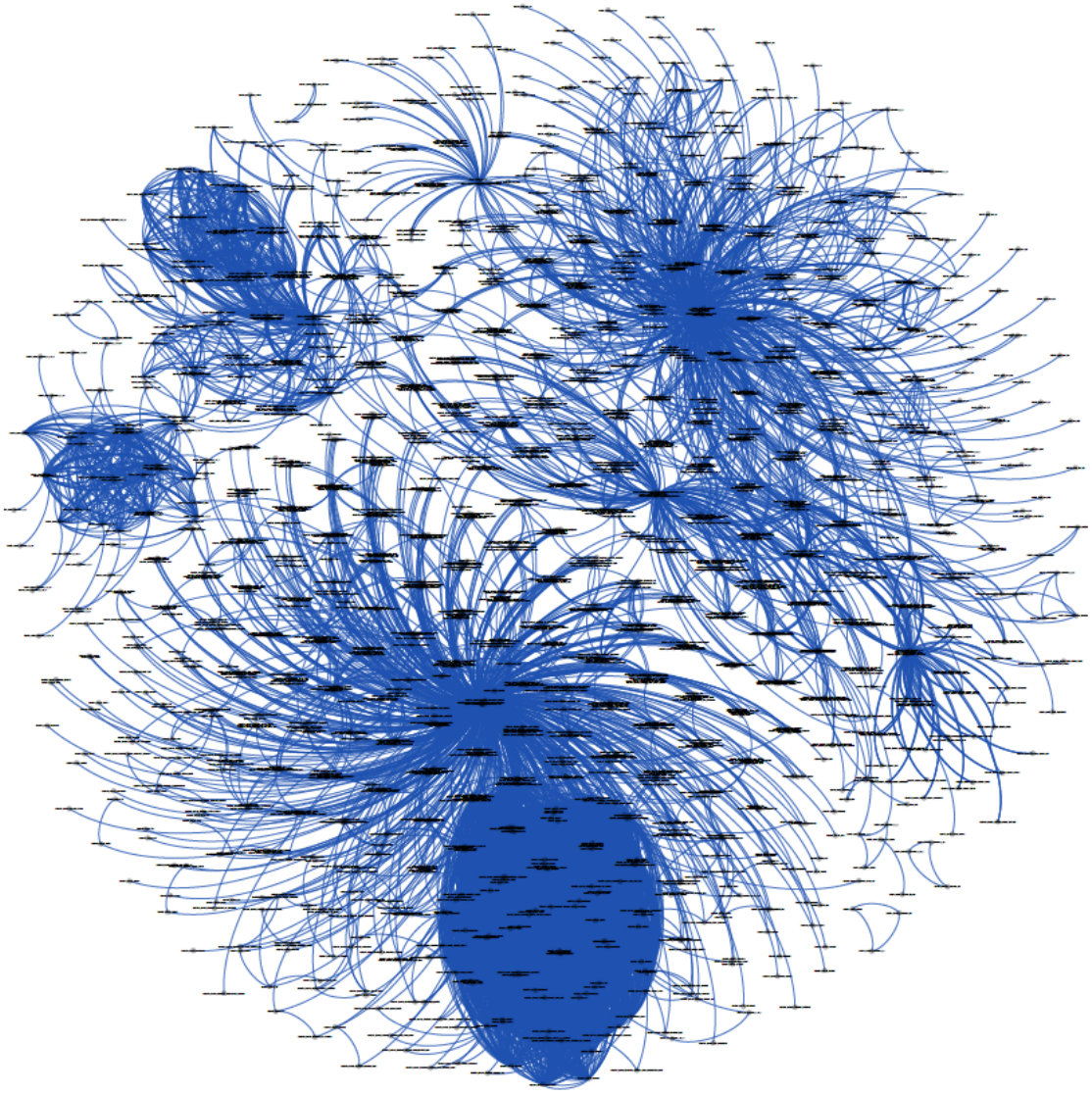


Figure 5.1: Example of a Kconfig configuration model representation using the `igraph` package

References

[1] C. Thörn and K. Sandkuhl, "Feature Modelling: Managing Variability in Complex Systems," in *Complex Systems in Knowledge-based Environments: Theory, Models and Applications* (A. Tolk, ed.), vol. 168 of *Studies in Computational Intelligence*, ch. 6, pp. 129 – 162, Springer Berlin Heidelberg, 2009.

[2] "Ford configurator." https://www.ford.co.uk/cars?bnpShowroom=on. Checked: 22/05/2020.

[3] M. Z. Nezhad, D. Zhu, X. Li, K. Yang, and P. Levy, "Safs: A deep feature selection approach for precision medicine," in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, (Shenzhen, China), pp. 501 – 506, 2016.

[4] K. Athanasopoulos, G. Theodoridis, C. Darisaplis, and I. Stamelos, "Towards a Software System for Facilitating the Reuse of Business Processes," in *Reuse in the Big Data Era: 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26 - 28, 2019, Proceedings* (X. Peng, A. Ampatzoglou, and T. Bhowmik, eds.), vol. 11602 of *Lecture Notes in Computer Science*, ch. 3, pp. 34 – 46, Cincinnati, USA: Springer International Publishing, 2019.

[5] K. C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[6] "Busybox." https://busybox.net/. Checked: 22/05/2020.

[7] "JHipster." https://www.jhipster.tech/. Checked: 22/05/2020.

[8] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack," *Empirical Softwere Engineering*, vol. 24, no. 2, pp. 674 – 717, 2019.

[9] D. F. Amoros, S. Bra, E. Aranda-Escolástico, and R. Heradio, "Using Extended Logical Primitives for Efficient BDD Building," *Mathematics*, vol. 8, no. 8, pp. 1 – 17, 2020.

[10] G. O'Regan, "A Short History of Logic," in *Concise Guide to Formal Methods: Theory, Fundamentals and Industry Applications*, Undergraduate Topics in Computer Science, ch. 5, pp. 93 – 108, Springer International Publishing, 2017.

[11] J. H. Siekmann, "Introduction - computational Logic," in *Computational Logic* (D. M. Gabbay, J. H. Siekmann, and J. Woods, eds.), vol. 9 of *Handbook of the History of Logic*, ch. 1, pp. 15 – 30, North-Holland, 2014.

[12] A. McGee, "Stating the Field: Institutions and Outcomes in Computer History," *IEEE Annals of the History of Computing*, vol. 34, pp. 104, 102 – 103, January 2012.

[13] "History of CASIO's Electronic Calculator Business." https://www.casio-intl.com/asia/en/calc/history/. Checked: 09/04/2020.

[14] C. Lécuyer, D. C. Brock, and J. Last, "Fairchild semiconductor, silicon technology, and military computing," in *Makers of the Microchip: A Documentary History of Fairchild Semiconductor*, The MIT Press, ch. 1, pp. 9 – 44, MIT Press, 2010.

[15] K.-S. Fu, "Learning control systems - Review and outlook," *IEEE Transactions on Automatic Control*, vol. 15, pp. 210 – 221, April 1970.

[16] F. L. Bauer, L. Bolliet, and H. J. Helms, "Software Engineering," (Garmisch, Germany), NATO Science Committee, Scientific Affairs Division, October 1968.

[17] E. Seligman, T. Schubert, and M. V. A. K. Kumar, "Formal verification: From dreams to reality," in *Formal Verification. An Essential Toolkit for Modern VLSI Design* (E. Seligman, T. Schubert, and M. V. A. K. Kumar, eds.), ch. 1, pp. 1 – 22, Boston: Morgan Kaufmann, 2015.

[18] E. Seligman, T. Schubert, and M. V. A. K. Kumar, "Basic formal verification algorithms," in *Formal Verification. An Essential Toolkit for Modern VLSI Design* (E. Seligman, T. Schubert, and M. V. A. K. Kumar, eds.), ch. 2, pp. 23 – 47, Boston: Morgan Kaufmann, 2015.

[19] C. Meinel and Thorsten Theobald, "Requirements on Data Structures in Formal Circuit Verification," in *Algorithms and Data Structures in VLSI Design*, ch. 5, pp. 77 – 86, Berlin: Springer-Verlag, 1988.

[20] L. C. Paulson, "A Brief History of Formal Logic," in *Computational Logic: Its Origins and Applications*, vol. 474, pp. 3 – 5, December 2017.

[21] G. Boole, "First Principles," in *The Mathematical Analysis of Logic*, ch. 2, pp. 14 – 18, Philosophical Library, 1847.

[22] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Transactions of the American Institute of Electrical Engineers*, vol. 57, no. 12, pp. 713 – 723, 1938.

[23] L. Sterling and E. Y. Shapiro, "Pure Prolog," in *The Art of Prolog: Advanced Programming Techniques*, Logic programming, ch. 6, pp. 119 – 128, MIT Press, 1994.

[24] I. Bratko, "An Overview of Prolog," in *Prolog Programming for Artificial Intelligence*, International computer science series, ch. 1, pp. 3 – 26, Addison Wesley, 2001.

[25] R. Kowalski, "Computational Logic," in *ESPRIT '90 - Proceedings of the Annual ESPRIT Conference Brussels* (Commission of the European Communities Directorate-General for Telecommunications, ed.), pp. 768 – 769, Springer Netherlands, 1990.

[26] R. Kowalski, "The logic of abductive logic programming," in *Computational Logic and Human Thinking: How to Be Artificially Intelligent*, ch. A6, pp. 280 – 295, Cambridge University Press, 2011.

[27] P. Bourque and R. E. Fairley, "Introduction to the Guide," in *Guide to the Software Engineering Body of Knowledge, Version 3.0*, pp. xxxi – xxxiii, IEEE Computer Society, 2014.

[28] "IEEE Standard Glossary of Software Engineering Terminology," (Washington, DC), Institute of Electronical and Electronics Engineers Computer Society, 1990.

[29] Project Manager Institute, "The Role of the Project Manager," in *A Guide to the Project Manager Body of Knowledge. PMBOK Guide®*, ch. 3, Project Manager Institute, 6th ed., 2017.

[30] P. C. Clements and L. M. Northrop, "Basic Ideas and Terms," in *Software Product Lines: Practices and Patterns*, ch. 1, pp. 5 – 15, Adison-Wesley Proffesional, 2001.

[31] Software Engineering Institute - Carnegie Mellon University, "Software Product Lines Collection." https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513819. Checked: 14/03/2020.

[32] D. Bouche and M. Dalgarno, "Software Product Line Engineering with Feature Models," in *Methods & Tools* (F. Martinig, ed.), vol. 14, (Vevey, Switzerland), pp. 9 – 17, Martinig & Associates, 2006.

[33] K. Czarnecki and U. Eisenecker, "Analysis and Design Methods and Techniques," in *Generative Programming: Methods, Tools and Applications*, pp. 17 – 59, Adison-Wesley, 2000.

[34] Institute of Electrical and Electronics Engineers, *829-2008 - IEEE Standard for Software and System Test Documentation*. 2008.

[35] P. Arcaini, A. Gargantini, and P. Vavassori, "Generating Tests for Detecting Faults in Feature Models," *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, pp. 1 – 10, May 2015. Graz, Austria.

[36] K. Czarnecki and A. Wasowski, "Feature Diagrams and Logics: There and Back Again," in *11th International Software Product Line Conference (SPLC 2007)*, (Kyoto, Japan), pp. 23 – 34, September 2007.

[37] R. Lopez-Herrejon and A. Egyed, "Detecting Inconsistencies in Multi-View Models with Variability," in *ECMFA 2010: Modelling Foundations and Applications*, (Paris, France), pp. 217 – 232, June 2010.

[38] S. Bra, R. Heradio, and D. Fernández, "Extending the R programming language to create and manage Boolean models encoded as BDDs," Master's thesis, Universidad Nacional de Educación a Distancia, Madrid, Spain, June 2017.

[39] L. Chen and M. A. Babar, "Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges," in *Software Product Lines: Going Beyond* (J. Bosch and J. Lee, eds.), (Berlin, Heidelberg), pp. 166 – 180, Springer Berlin Heidelberg, 2010.

[40] A. Santos, K. Koskimies, and A. Lopes, "A Model-Driven Approach to Variability Management in Product-Line Engineering.," *Nordic Journal of Computing*, vol. 13, pp. 196 – 213, January 2006.

[41] K. C. Kang and H. Lee, "Variability Modeling," in *Systems and Software Variability Management. Concepts, Tools and Experiences* (R. Capilla, J. Bosch, and K. C. Kang, eds.), ch. 2, pp. 25 – 42, Springer-Verlag Berlin Heidelberg, 2013.

[42] K. Czarnecki, Simon Helsen, and Ulrich Eisenecker, "Staged Configuration Using Feature Models," technical report, University of Waterloo and University of Applied Sciences Kaiserlautern, 2004.

[43] M. Šipka, "Exploring the Commonality in Feature Modeling Notations," technical report, Slovak University of Technology, 2005.

[44] J. Shi, "Use of Constraint Solving for Testing Software Product Lines," *Computer Science and Engineering: Theses, Dissertations, and Student Research*, pp. 41 – 48, December 2011.

[45] A. Jansen, R. Smedinga, J. van Gurp, and J. Bosch, "Feature-Based Product Derivation: Composing Features," *IEE Proceedings Software*, vol. 151, pp. 187 – 197, August 2004.

[46] S. A. Hendrickson and A. van der Hoek, "Modeling Product Line Architectures through Change Sets and Relationships," in *29th International Conference on Software Engineering (ICSE'07)*, (Minneapolis, USA), pp. 189 – 198, 2007.

[47] M. Makkai, C. C. Chang, and H. J. Keisler, "Model Theory," *Journal of Symbolic Logic*, vol. 56, pp. 1096 – 1097, September 1991.

[48] A. Degtyarev and A. Voronkov, "The Inverse Method," in *Handbook of Automated Reasoning* (A. Robinson and A. Voronkov, eds.), Handbook of Automated Reasoning, ch. 4, pp. 179 – 272, Amsterdam: North-Holland, 2001.

[49] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," *The Bell System Technical Journal*, vol. 38, pp. 985 – 999, July 1959.

[50] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509 – 516, June 1978.

[51] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," research note, California Institute of Technology and Carnegie-Mellon University, August 1986.

[52] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *27th ACM/IEEE Design Automation Conference*, (Orlando, USA), pp. 52 – 57, June 1990.

[53] Y. Mo, F. Zhong, H. Liu, Q. Yang, and G. Cui, "Efficient Ordering Heuristics in Binary Decision Diagram-based Fault Tree Analysis," *Quality and Reliability Engineering International*, vol. 29, no. 3, pp. 307 – 315, 2013.

[54] L. Xing, "An Efficient Binary-Decision-Diagram-Based Approach for Network Reliability and Sensitivity Analysis," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 38, pp. 105 – 115, January 2008.

[55] H. R. Andersen, "An Introduction to Binary Decision Diagrams," lecture notes, Department of Information Technology, Technical University of Denmark, October 1997.

[56] R. E. Bryant, "Binary Decision Diagrams," in *Handbook of Model Checking* (E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds.), ch. 7, pp. 191 – 217, Cham: Springer International Publishing, 2018.

[57] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on Computers*, vol. 45, pp. 993 – 1002, September 1996.

[58] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), pp. 151 – 158, Association for Computing Machinery, 1971.

[59] S. Prestwich, "CNF Encodings," in *Handbook of Satisfiability* (A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds.), Frontiers in Artificial Intelligence and Applications, ch. 2, pp. 75 – 97, IOS Press, 2009.

[60] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[61] S. Reda, R. Drechsler, and A. Orailoglu, "On the Relation between SAT and BDDs for Equivalence Checking," in *Proceedings International Symposium on Quality Electronic Design*, (San Jose, USA), pp. 394 – 399, March 2002.

[62] "CUDD Tutorials." https://davidkebo.com/cudd. Checked: 18/03/2020.

[63] F. Somenzi, "CUDD: CU Decision Diagram Package Release 3.0.0," research note, Department of Electrical, Computer and Energy Engineering, University of Colorado at Boulder, December 2015.

[64] "BuDDy: A BDD package." http://buddy.sourceforge.net/manual/. Checked: 20/03/2020.

[65] T. van Dijk, E. M. Hahn, D. N. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, "A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking," research note, University of Twente, Formal Methods & Tools, Enschede, The Netherlands, State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, China and Radboud Universiteit, Model-Based System Development, Nijmegen, The Netherlands, October 2015.

[66] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (Orlando, Florida, USA), pp. 761 – 762, ACM, 2009.

[67] N. Narodytska and Toby Walsh, "Constraint and Variable Ordering Heuristics for Compiling Configuration Problems," *International Joint Conference on Artificial Intelligence*, no. 7, pp. 149 – 154, 2006.

[68] P. Kissmann and J. Hoffmann, "BDD Ordering Heuristics for Classical Planning," research note, Saarland University, Saarbrücken, Germany, December 2014.

[69] H. Pérez Morago, R. Heradio, D. Fernández Amorós, R. Bean, and C. Cerrada, "Efficient Identification of Core and Dead Features in Variability Models," research note, Department of Software Engineering and Computer Systems, Universidad Nacional de Educación a Distancia, November 2015.

[70] R. Heradio-Gil, D. Fernandez-Amoros, J. A. Cerrada, and C. Cerrada, "Supporting commonality-based analysis of software product lines," *IET software*, vol. 5, no. 6, pp. 496 – 509, 2011.

[71] R. Heradio, D. Fernandez-Amoros, C. Dorn, and A. Egyed, "Supporting the Statistical Analysis of Variability Models," in *41st ACM/IEEE International Conference on Software Engineering (ICSE)*, (Montreal, Canada), May 2019.

[72] A. Nöhrer and A. Egyed, "C2O configurator: A tool for guided decision-making," *Automated Software Engineering*, vol. 20, pp. 265 – 296, June 2013.

[73] "EmbToolkit - Companion for your embedded system firmware." https://www.embtoolkit.org/. Checked: 04/04/2020.

[74] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, (Shaanxi, China), pp. 240 – 251, April 2019.

[75] S. Chakraborty, K. Meel, and M. Vardi, "A Scalable and Nearly Uniform Generator of SAT Witnesses," in *Computer Aided Verification. CAV 2013* (N. Sharygina and H. Veith, eds.), vol. 8044 of *Lecture Notes in Computer Science*, (Saint Petersburg, Russia), pp. 608 – 623, April 2013.

[76] W. Wei, J. Erenrich, and B. Selman, "Towards Efficient Sampling: Exploiting Random Walk Strategies," *AAAI'04: Proceedings of the 19th national conference on Artifical intelligence*, pp. 670 – 676, July 2004.

[77] D. J. Muñoz Guerra, J. Oh, M. Pinto, L. Fuentes, and D. Batory, "Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features," in *23rd International Systems and Software Product Line Conference*, (Paris, France), pp. 1 – 13, September 2019.

[78] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding Near-Optimal Configurations in Product Lines by Random Sampling," in *11th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Paderborn, Germany), pp. 61 – 71, August 2017.

[79] D. S. Batory, J. Oh, and M. Myers, "Percentile Calculations for Randomly Searching Colossal Product Spaces," research note, Department of Computer Science - University of Austin, Texas, 2018.

[80] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 2009.

[81] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, "Fast Sampling of Perfectly Uniform Satisfying Assignments," in *21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*, (Oxford, UK), pp. 135 – 147, 2018.

[82] J. Oh, P. Gazzillo, and D. Batory, "t-wise Coverage by Uniform Sampling," in *23rd International Systems and Software Product Line Conference (SPLC)*, (New York, NY, USA), pp. 84 – 87, ACM, 2019.

[83] M. Thurley, "sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP," in *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, (Seattle, WA, USA), pp. 424 – 429, 2006.

[84] R. M. Jensen, R. E. Bryant, and M. M. Veloso, "An Efficient BDD-Based A* Algorithm," technical report, Computer Science Department, Carnegie-Mellon University, January 2002.

[85] C. L. Fefferman and L. A. Seco, "Number Theory and Atomic Densities," in *Emerging Applications of Number Theory* (D. Hejnar, J. Friedman, M. Gutzwiller, and A. Odlyzko, eds.), vol. 109, ch. 8, pp. 205 – 218, New York: Springer, January 1999.

[86] A. Miczo, "Automatic Test Pattern Generation," in *Digital Logic Testing and Simulation*, ch. 4, pp. 165 – 231, New Jersey: Wiley-Interscience, 2nd ed., 2003.

[87] "The R Project for Statistical Computing." https://www.r-project.org/. Checked: 12/01/2020.

[88] "CRAN - Package Rcpp." https://cran.r-project.org/web/packages/Rcpp/. Checked: 18/01/2020.

[89] D. Eddelbuettel, "C++ for R Programmers," in *Seamless R and C++ Integration with Rcpp*, ch. 13, pp. 195 – 205, New York: Springer, 2013.

[90] D. Eddelbuettel and R. François, "Rcpp Extending," release notes, November 2019.

[91] "R Internals." https://cran.r-project.org/doc/manuals/r-release/R-ints.html. Checked: 25/01/2020.

[92] "Dirk Eddelbuettel - MCMC and faster Gibbs Sampling using Rcpp." http://dirk.eddelbuettel.com/blog/2011/07/14/. Checked: 13/04/2020.

[93] E. I. George, G. Casella, and E. I. George, "Explaining the Gibbs Sampler," *The American Statistician*, vol. 46, no. 3, pp. 167 – 174, 1992.

[94] B. A. Berg, "Markov Chain Monte Carlo," in *Markov Chain Monte Carlo Simulations and Their Statistical Analysis: With Web-Based Fortran Code*, ch. 3, pp. 128 – 195, World Scientific, 2004.

[95] "CUDD 3.0.0 - DdManager Struct Reference." https://add-lib.scce.info/assets/doxygen-cudd-documentation/structDdManager.html. Checked: 15/04/2020.

[96] "BuDDy - bdd_init command." http://buddy.sourceforge.net/manual/group__kernel_g77facd4e5f592e01bb64205a66aabaf3.html. Checked: 15/04/2020.

[97] Center of Discrete Mathematics and Theorical Computer Science (DIMACS), "Satisfiability Suggested Format," technical report, May 1993.

[98] "S.P.L.O.T.." http://www.splot-research.org/. Checked: 15/04/2020.

[99] "Tools - The DDDMP package." http://fmgroup.polito.it/quer/research/tool/tool.htm. Checked: 17/04/2020.

[100] Y. Hong, P. Beerel, J. Burch, and K. McMillan, "Sibling-substitution-based BDD minimization using don't cares," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, pp. 44 – 55, January 2000.

[101] Y. Hong, P. Beerel, J. Burch, and K. McMillan, "Safe BDD minimization using don't cares," in *DAC '97: Proceedings of the 34th annual Design Automation Conference*, (Anaheim, USA), pp. 208 – 213, June 1997.

[102] W. A. Hunt and F. Somenzi, "Rabbit: A Tool for BDD-Based Verification of Real-Time Systems," in *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, Lecture Notes in Computer Science, (Boulder, Colorado, USA), pp. 122 – 125, Springer Berlin Heidelberg, 2003.

[103] "GMP: Multiple Precision Arithmetic." https://cran.r-project.org/web/packages/gmp/index.html. Checked: 30/12/2019.

[104] "R MPFR: Multiple Precision Floating-Point Reliable." https://cran.r-project.org/web/packages/Rmpfr/index.html. Checked: 18/04/2020.

[105] "axTLS Embedded SSL." http://axtls.sourceforge.net/. Checked: 18/04/2020.

[106] "Ubuntu Release Notes." https://wiki.ubuntu.com/BionicBeaver/ReleaseNotes. Checked: 28/12/2019.

[107] "Changes in R 3.6.2." https://cloud.r-project.org/. Checked: 28/12/2019.

[108] "Standard C++." https://isocpp.org/. Checked: 30/12/2019.

[109] D. Dalpiaz, "Applied statistics with R," technical report, University of Illinois, October 2019.

[110] "GMP Library." https://gmplib.org/. Checked: 30/12/2019.

[111] R. B. D'Agostino, *Goodness-of-Fit Techniques*. CRC Press, 1986.

[112] D. E. Knuth, *The Art of Computer Programming*, vol. 2. Addison-Wesley Longman, 3rd ed., 1997.

[113] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 1988.

[114] L. Xing and Y. S. Dai, "A New Decision-Diagram-Based Method for Efficient Analysis on Multistate Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 3, pp. 161 – 174, 2009.

[115] "RDocumentation - apply." https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/apply. Checked: 02/05/2020.

[116] H. Wickham, *Advanced R, Second Edition*. Chapman & Hall/CRC The R Series, CRC Press, 2019.

[117] T. van Dijk, E. Hahn, D. Jansen, Y. Li, T. Neele, M. Stoelinga, A. Turrini, and L. Zhang, "A Comparative Study of BDD Packages for Probabilistic Symbolic Model Checking," vol. 9409, pp. 35 – 51, November 2015.

[118] G. Lv, K. Su, and Y. Xu, "CacBDD: a BDD Package with Dynamic Cache Management," in *Proceedings of the 25th International Conference on Computer Aided Verification*, vol. 8044, (Saint Petersburg, Russia), pp. 229 – 234, July 2013.

[119] T. van Dijk and J. van de Pol, "Sylvan: Multi-Core Decision Diagrams," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, (London, UK), pp. 677 – 691, 2015.

[120] "Carnegie Mellon - The BDD Library." https://www.cs.cmu.edu/~modelcheck/bdd.html. Checked: 02/05/2020.

[121] "igraph - The network analysis package." https://igraph.org/. Checked: 02/05/2020.

[122] E. D. Kolaczyc and G. Csárdi, *Statistical Analysis of Network Data with R*. New York: Springer, 2014.

[123] D. A. Luke, *A User's Guide to Network Analysis in R*. New York: Springer, 2015.

List of Acronyms

2-SAT 2-satisfiability.

ADD Algebraic Decision Diagram.

AI Artificial Intelligence.

API Application Programming Interface.

BBPF Bit-Blasted Propositional Formula.

BDD Binary Decision Diagram.

CNF Conjunctive Normal Form.

CRAN Comprehensive R Archive Network.

DIMACS Center of Discrete Mathematics and Theoretical Computer Science.

DSL Domain Specific Language.

FODA Feature-Oriented Domain Analysis.

GMP GNU Multiple Precision.

GNU GNU's Not Unix.

IEEE Institute of Electrical and Electronics Engineers.

JCR Journal Citation Report.

KA Knowledge Area.

LPG Liquefied Petroleum Gas.

MCMC Monte Carlo Markov Chain.

NNF Negation Normal Form.

PMBOK Project Management Body of Knowledge.

ROBDD Reduced Ordered Binary Decision Diagram.

SAT boolean SATisfiability problem.

SEVOCAB ISO/IEC/IEEE Systems and Software Engineering Vocabulary.

SPL Software Product Line.

SPLE Software Product Line Engineering.

SPLOT Software Product Lines Online Tools.

SSL Secure Sockets Layer.

SWEBOK Software Engineering Body of Knowledge.

UNED Universidad Nacional de Educación a Distancia.

URS Uniform Random Sampling.

XML eXtensible Markup Language.

ZBDD Zero-suppressed Binary Decision Diagram.