Universidad Nacional de Educación a Distancia

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Itinerario: Ingeniería de Software - 31105151

# Extending the R programming language to create and manage Boolean models encoded as BDDs

*Autor:*
Sergio Bra Gutiérrez

*Directores:*
Rubén Heradio Gil
David Fernández Amorós

Curso académico 2016/2017 - Convocatoria de junio

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

ITINERARIO: INGENIERÍA DE SOFTWARE - 31105151

---

# Extending the R programming language to create and manage Boolean models encoded as BDDs

---

TRABAJO DE TIPO A: TRABAJO ESPECÍFICO PROPUESTO POR UN PROFESOR

*Autor:*
Sergio Bra Gutiérrez

*Directores:*
Rubén Heradio Gil
David Fernández Amorós

*Espacio reservado para la hoja de calificaciones*

# DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 05/02/2017.

Quién suscribe:

Autor(a): Sergio Bra Gutiérrez
D.N.I/N.I.E/Pasaporte.: 72087621D

Hace constar que es la autor(a) del trabajo:

Extending the R programming language to create and manage Boolean models encoded as BDDs

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

**DECLARACIÓN:**

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

Sergio

## Impreso TFdM05_Autor. Autorización de publicación y difusión del TFdM para fines académicos

# Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Sergio

# Acknowledgements

Firstly, I want to give thanks to my parents for their moral support and the economical effort made during all these years.

To Anabel, for supporting me on the hard moments and be there when I need her.

To my sister Patricia and the friends who have made these years more pleasant.

And finally to Rubén and David, for giving me the chance of learning so much with the realisation of this project and an incredible help when problems appeared.

# Abstract

**R** has turned into a reference in the statistical computing field as time goes by. Also, its popularity is growing in other scopes, such as data mining, financial mathematics, biomedicine, etc. This is because of its nature of free software, among other factors, which has allowed the creation of a huge amount of libraries provided by the community to add usefulness to the basic implementation.

The **S** programming language, which **R** is based on, allows the development of functions outside the data analysis, but they are highly inefficient compared with the analogous in other languages like Java, C o C++. To solve this problem, some libraries offer the possibility of executing from **R** code written in other languages.

The aim of this project is the design and development of a wrapper made for **R** which implements the functions of a library built in C++, supplying some utilities to work with complex structures in a simple and efficient way.

Keywords: **R**, wrapper, BDD, software library, free software.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

According to the opinion of experts in the subject, like the professor of the Stanford University Donald E. Knuth [1], Binary Decision Diagrams (BDDs) are considered like one of the greatest advances in the sphere of the data structures in the last years. Such is the case that one of the most quoted pulications in the scientific field was the Randal Bryant's study, where the potential of these structures is analyzed in order to improve the efficiency of algorithms [2].

BDDs were introduced by C. Y. Lee [3] in 1959, and since then their contribution within the context of the engineering and mathematics has been very prolific. Partly, this is due to their hardware implementation does not present a high complexity, coming out as a key issue for their adoption.

Some of the domains where BDDs have been used to a greater extent are the scope of circuits and configurators. Their use combined with Software Product Lines (SPLs) is thoroughly discussed in the Marcilio Mendonça's thesis [4]. In that research, the reader can spot information about the application of BDDs to improve automated support for reasoning on feature models[1] and product configuration.

These structures can be fairly quick to count the number of valid configurations, checking the equivalence of feature models and providing foundation to the interactive procedure where the customer is able to choose a value for a decision variable at run time, all of that based on the compiling of combinatorial spaces of the configuration problem.

However, one of the main warhorse issues related to the use of BDDs is the fact that the size of those elements can grow exponentially, depending strongly on the length of the input and its ordering (that in the Mendonça's research matter would be the length of the feature model). This factor is greatly affected by the BDD variable ordering (finding the optimal order is an NP-hard problem), which has been typically approached by heuristics. At this moment, when we are concerned about the order of their elements, we can refer to these structures as Ordered Binary Decision Diagrams (OBDDs) or even Reduced Ordered Binary Decision Diagram (ROBDD).

Various authors have discussed over the matter of constraint and variable ordering. To the Bollig et al.'s work [5] previously quoted, other researchers have contributed to the cause like Narodytska et al. [6], in whose study up to three heuristic solutions could be analysed in order to reduce the time and space required to compile solutions to configuration problems into a decision diagram. Exploiting the properties of those problems, they

---

[1] In software development, a feature model is a compact representation of all the products of the SPL.

proposed algorithms based on ensuring monotonic growth in the size of the BDD, static variable ordering and dynamic variable grouping.

The best results were obtained by the first algorithm, keeping the size of the resulting BDD smaller than the other two ordering algorithms on all of their steps. As a result, this algorithm significantly reduced the time to construct the target BDD, from one to two orders of magnitude improvement in compilation time. One of the conjectures the researchers obtained with these results is that the original ordering usually reflects the natural structure of the problem, taking advantage of the grouping of constraints describing single components.

Meinel et al.'s book [7] is another work about OBDDs where the usage of heuristics for building efficient variable orders is investigated. Firstly, they considered some premises to design good methods to construct good orders, like the considerated functions may be given in form of net lists or the additional information may be provided and exploitable. Afterwards, the main idea is to deduce information concerning suitable positions of the variables in the ordering from the topological structure of the combinational circuit studied.

They compared these techniques with other dynamic reordering algorithms, and the conclusion that the researchers came to is that the dynamic ones require extremely much computation time if the BDD's nodes are not optimized.

For this reason, it does not turn out to be strange that main programming languages include libraries implementing that structure, as well as the basic ordering functions to work with it. Some of the most popular examples are JavaBDD [8] and JDD [9] in Java, CUDD [10] and BuDDy [11] in C, etc.

However, it has been detected that a very popular tool as **R** [12] does not include any kind of support of BDDs, neither integrated in the core version of the tool nor complemented by extensions made by the comunity of programmers who increases its funcionality.

It is interesting to be able to work with these diagrams in that tool, because since Ross Ihaka and Robert Gentleman, Auckland University, conceived **R** as an implementation of the **S** [13] programming language in 1992, its popularity has not ceased to grow. To reach the current situation of the tool (far from being definitive), it has experimented many and constant revisions, supplying a bigger funcionality, versatility and, what is most important, stability to the project.

Inside the software depelopment world, where everything changes extremely fast and new tools, frameworks, etc. are developed almost daily, to have as adaptability as possible is an essential factor for a product to be successful in a more and more demanding market and with a growing competition. This necessity of flexibility is covered by **R** with the use of libraries made by the large community of **R** users, that following the philosophy of the free development, makes possible to adapt the available functionalities of the tool based on the different needs.

Currently, the number of available libraries through the **R** repository is nearly 10 000 [14], that allows to get an idea of the enrichment given to the platform as soon as new needs appear to the developers.

Due to performance problems when general purpose functions are executed with this tool, as will be explained in further chapters of this document, it will be considered the use of libraries which allow to run code written in other languages, with a barely penalization in the performance.

Another **R**'s strength is the huge efficiency that presents in statistical modelling tasks, because writing a few lines of code, good results con be obtained, providing sophisticated solutions. Programming the same behaviour in other languages would mean a bigger computing cost, produced by the considerable increase of the complexity of the algorithm.

For all those reasons, it becomes as a perfect tool to execute a great variety of operations. Furthermore, **R** turns out as the ideal environment to work with an optimal structure like BDDs, providing the chance of making complex logic reasoning. Moreover, it would solve the the main problem of BDDs, that is to order the structure to be able to work efficiently with it.

Therefore the idea of generating a wrapper for **R** arises, whose purpose is to encapsulate the methods of the libraries previously described for modelling BDDs and making possible the systematic analysis of the use of algorithms which require these structures. In this way it would be possible to work with them in **R**, but without having altered the efficiency that is given by the codification in languages uncoupled to the statistical calculation.

In the Figure 1.1 it could be seen the suggested architecture, with its main interactions between the components the system is formed by. It is compound by a wrapper, which surrounds a C++ facade to make accesible the functions related to the management of BDDs from **R**. The main benefit of the defined system is its high extensibility, being very simple adding new functionality without a real impact on the earlier development.

Because of the nature of open source that surrounds **R** and its maintenance by the community of programmers, the construction tasks of the wrapper, tests, documentation, etc. have been accomplished under free tools and environments. Thus, the development of the library has been carried out in a machine working with Ubuntu 16.04.1 LTS - 64 bits [15], GNU's Not Unix (GNU)'s text editors and LaTeX [16] for the generation of official documents.

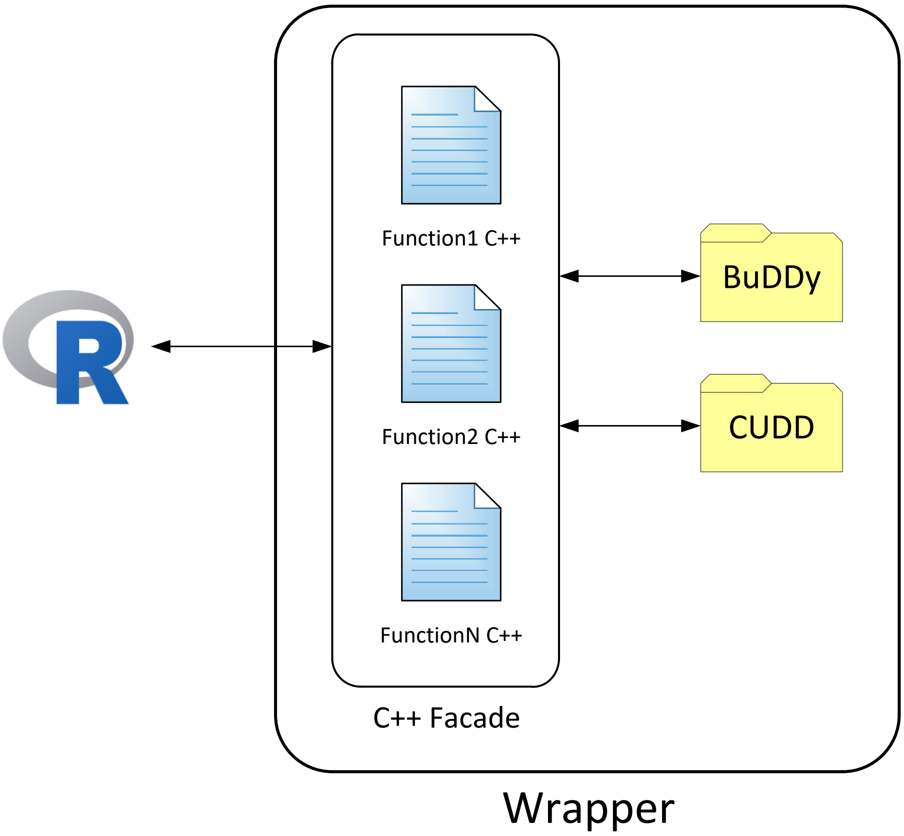


Figure 1.1: Interaction between the components of the suggested solution

## 1.1 Development framework

The present development is located on the Software and Systems Engineering Group of the Superior Technical School of Computer Engineering of the Universidad Nacional de Educación a Distancia (UNED).

Paying attention to the different lines in which the department is working on, the suggested project belongs to that one called "Software Development with Reuse Techniques". In lots of contexs, being the development of SPLs an example, it is wanted the systematic reuse of software. To reach this goal, it is essential to model the common components and variables of each product of a family, that is usually done through a feature diagram [17]. The management of these variability models is made translating the feature model to a propositional logic formula. To process that formula, boolean SATisfiability problem (SAT) solvers and BDDs are frequently utilised.

The aims of this work are:

- Providing a way to build the BDD of a very big family. **R** will be the tool used to choose what order of the variables is going to produce a compact BDD, meaning which one that needs the less possible memory.
- Being able to work with the selected BDD.

## 1.2 Goals and motivation

Based on the situation previously described, it is expected to deal with the implementation of a library for **R** that allows to define and to modify BDDs as simply and fast as possible, having the chance of applying some operations over them efficiently, too.

The development has been focused in the ease of its use from the point of view of a **R** programmer, as well as an user coming from the BDDs field. The conceived instructions are simple and with default parameters, minimising the need of informing too many elements to work easily, but keeping it highly configurable for that situations when it is required a more complex setting up of the paramenters. In addition, the library is modelled under a multiplatform environment, being available for computers working with Windows and Linux.

The notation of the expressions passed as input to the functions has been conceived following the same principle of keeping it as simple as possible, being able to define them as logic expressions or in Conjunctive Normal Form (CNF).

To reach it, the following goals have been set up:

- Designing and Developing a library for **R** which manages BDDs in a very efficient way. These package will be simple, extensible and robust.
- Generating a full and high-quality documentation, allowing an Open Source develop and simplifying the task for adding funcionality by the community of programmers.

## 1.3 Content of the document

To explain the process of reaching the defined goals, the thesis made is divided in chapters addressing the different facets that the project is made of:

- **Background and related work**: This section makes a description of the current situation over the considered work is based on and the related developments.
- **Analysis and design of the solution**: When the development frame is known, the solution to introduce, the specification of the model and the phases of the implementation are defined.
- **Development of the proposed solution**: Once the library is well-defined, it is described the followed process in order to build the wrapper and modify it in further updates.
- **Experimental validation**: After the development of the application, the validity of the solution will be shown by the resolution of some studied problems.
- **Conclusions and future work**: Finally, when the goals of the project are met, the final conclusions obtained after its realization will be explained, as well as the lines that could be followed for a improvement of the current work.

# Chapter 2

# Background and related work

SPLs have increased their popularity in the software industry due to the capability of achieving the reuse of code and structures. As a result, companies which apply those techniques have reduced the cost of software development, its maintenance and the time to market [18]. The key of this concept is the identification of common parts among the different products on a family in order to be able to model a reusable entity.

The main difference with respect to a traditional development is the logical separation between the core of the application, reusable software assets and actual application code [19].

Figure 2.1 represents the differences on costs when a company uses SPLs and when it follows a traditional system of development [20]. It could be discovered that at first the cost of using a SPL is higher than when a current practice is used, but when the number of products starts to increase, the investment it is quickly compensated because of the high reusability of the code.

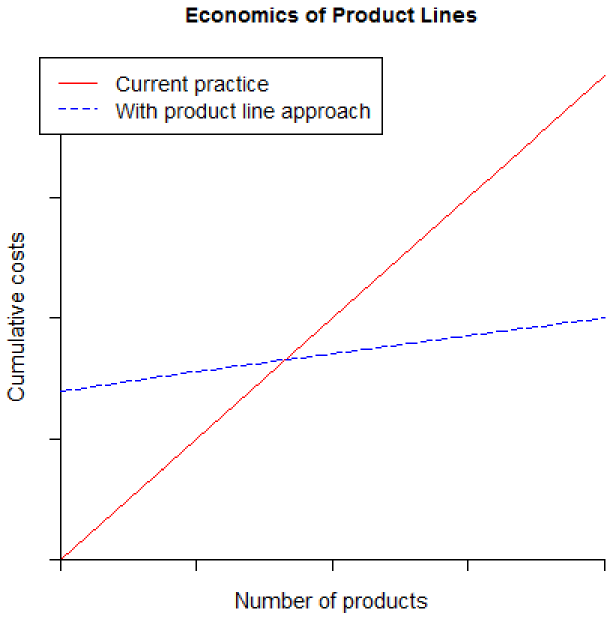


Figure 2.1: Economic comparison of the usage of traditional development practices versus SPLs

In view of this analysis, it is worth the effort of a methodology change and to invest in that concept, because at the end results are better even thought the initial penalisation. At first, when an organization decides to use SPLs, the design and development are less efficient because of it has to be thought for general purpose, not only for solving a specific problem.

## 2.1 Feature models

According to the Institute of Electrical and Electronics Engineers (IEEE) [21], a feature is "a distinguishing characteristic of a software item (e.g., performance, portability, or functionality)". In order to approach to the optimal solution for the code reusability problem, feature models are used to expose those characteristics of the system, being reflected on propositional logic formulas. Once a logic structure is built, it is possible to apply as many operations as is needed to reach a good solution to the explained casuistry.

Informally, a feature model is a simple, hierarchical representation that captures the commonality and variability os a product line. Every of these relevant characteristics of the problem space is translated as a feature in the model, which can be considered like something relevant for some stakeholder.

Domain analysis has as aim defining the different capabilities of interfaces to exploit commonality through the systematic exploration of software environments [22]. Some of the most relevant works in this field [23] [24] suggest the Domain Analysis and Reuse Environment (DARE) as a technique to execute a successful domain analysis, which has the following phases:

1. Context analysis: Returns the context of the domain providing the required inputs and outputs and identifying relationships with other interfaces.
2. Domain modelling: Describes the problems addressed by software in the domain, matching the different features in the domain.
3. Architecture modelling: Defines the implementation of software in the domain. The created structures are used as architectural models for generating applications from the domain model.

In practice, feature models are represented with a specific notation which allows capturing the different elements of the problem. The first widely extended notation was the Feature-Oriented Domain Analysis (FODA) [22], but nowadays the most accepted one is Czarnecki-Eisenecker's notation [25] [26].

## 2.2 Binary decision diagrams

As a result of the explained process, a propositional logic formula can be built representing the feature model, making the optimization tasks easier. The solution in which the present work is focused on is the usage of BDDs in order to handle the logic sentence.

A BDD is a data structure used to represent boolean functions, reflecting the relationships between boolean variables. Graphically, a BDD could be represented as a finite

Directed Acyclic Graph (DAG) with a unique initial node (rooted), where there are defined decision nodes and terminal nodes (or leaf nodes) [27]. Generally speaking, when the term BDD is used, it usually refers to ROBDD, that are BDDs to which some ordering and reducing techniques have been applied to. Figure 2.2 depicts a very simple ROBDD, showing the different parts which is made up of.

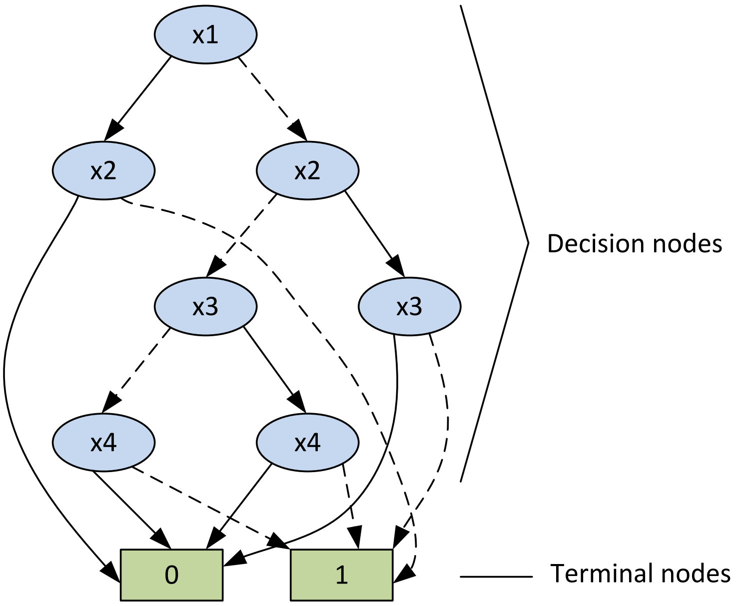


Figure 2.2: Example of a BDD

In practice, every binary decision tree can be transformed to a BDD by maximally reducing it applying two rules:

1. Merging any isomorphic subgraphs[1].
2. Deleting any node whose two children are isomorphic.

As a boolean function, many logical operations can be executed to a BDD, being conjuction, disjunction, negation or implication some examples. Individually, each operation takes a polynomial time, but the repetition of these functions several times can result in an exponentially big BDD [28]. The reason of this behaviour is that any of the operations between two BDDs return a BDD with a size proportional to the product of the BDDs' sizes. This is a real problem, and the amount of researches [5] [29] around that aspect is an evidence of its relevance.

Consequently, it becomes crucial to care about the variable ordering when BDDs are used. The problem of finding the best variable ordering is NP-hard, but there are some efficient heuristic algorithms to tackle it. In the Table 2.1 are represented the most widely used algorithms and their implementation in the selected BDD managers on this project [30] [31].

---

[1] In graph theory, an isomorphism is an edge-preserving vertex bijection which preserves an equivalence mapping between vertices.

Table 2.1: Description and implementation of the main BDD ordering algorithms

| **Algorithm** | **Description** | **BuDDy** | **CUDD** |
|---|---|---|---|
| Win2 | Sliding window of size 2 | Yes | Yes |
| Win2ite | Repeated Win2 until no further progress is done | Yes | No |
| Win2con | Converging variant of Win2 | No | Yes |
| Win3 | Sliding window of size 3 | Yes | Yes |
| Win3ite | Repeated Win3 until no further progress is done | Yes | No |
| Win3con | Converging variant of Win3 | No | Yes |
| Win4 | Sliding window of size 4 | No | Yes |
| Win4con | Converging variant of Win4 | No | Yes |
| Sift | Blocks are moved through all possible positions | Yes | Yes |
| SiftIte | Repeated Sift until no further progress is done | Yes | No |
| SiftCon | Converging variant of Sift | No | Yes |
| SiftSym | Symmetric variant of Sift | No | Yes |
| SiftSymCon | Converging variant of SiftSym | No | Yes |
| SiftGr | Implementation of group sifting | No | Yes |
| Random | Select a random position for each variable | Yes | Yes |
| RandomPv | Same as Random but pivoting over a variable | No | Yes |
| Annealing | Simulated annealing | No | Yes |
| Genetic | Genetic algorithm | No | Yes |
| Exact | Programming approach to exact oredering | No | Yes |

# Chapter 3

# Analysis and design of the solution

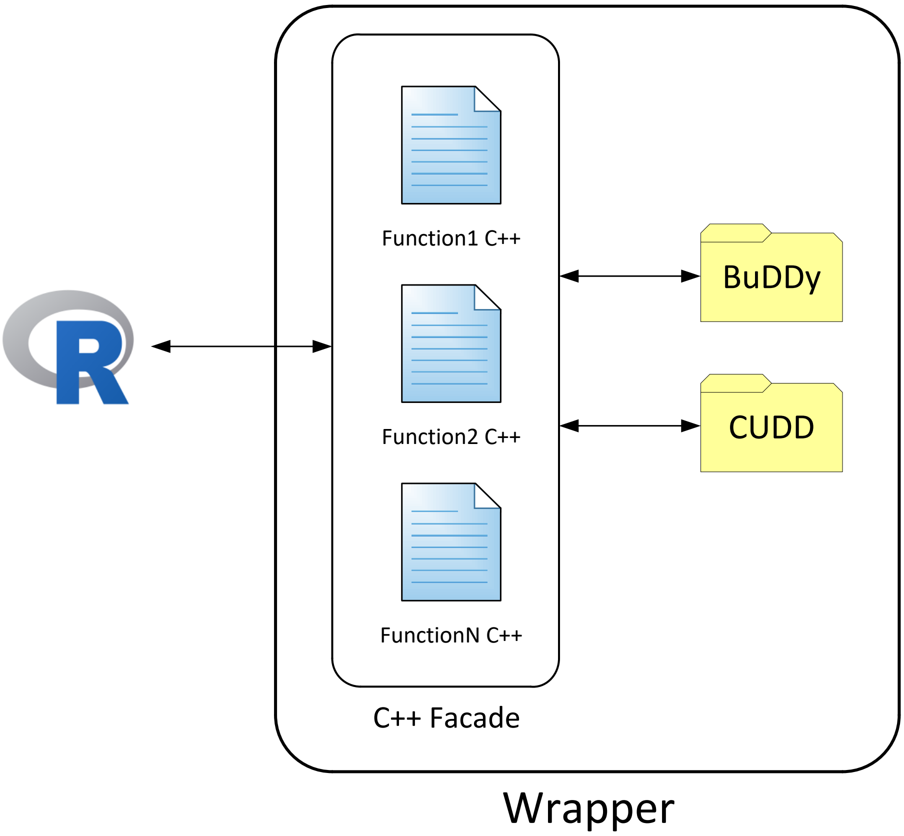
Once the background that surrounds the suggested development has been studied, it is time to start thinking about a solution capable of solving a series of well-defined requirements:

- The package must be able to create, custimize and manage BDDs in **R**.
- The implemented operations have to be efficient and simple, not being necessary a deep knowledge about the library to work with it, but keeping the chance of adding as complexity as the programmer could need.
- Firstly, the package will use BuDDy and CUDD as BDDs managers, but it must be designed in such a way that new managers could be added without a great impact in the design and development tasks.
- Also, if new dependencies are needed to include to the library, the process must not mean a great effort to the developers. The Chapter 4 shows the process to build the package in Linux and Windows, explaining how to add external code.
- As ordering a BDD turns into a key factor in terms of efficiency, some of the algorithms explained in Chapter 2 have to be available.
- The project must have a complete and easy to understand documentation, not being complicated to discover the usage of the different functions developed. Also, this paper could be seen as a guide to know the library and the way to use it and, taking advantage of its open source facet, to modify it.

## 3.1 API of the developed package

In Figure 1.1, reproduced again below, it was shown the suggested architecture, in order to accomplish the aforementioned premises. The base of the created library, called `rbdd`, is a C++ facade which encapsulates the interfaces to access to the different methods of the BDDs managers implemented.

From the **R** side, as many functions as methods are in the mentioned facade have been designed, and they could be classified depending on their goals in the categories explained as an Application Programming Interface (API) in the following subsections.



Interaction between the components of the suggested solution

### 3.1.1 Creating and setting up BDDs

These operations provide functionality to create a new BDD and to change the default configuration in order to custimize the structure and adapt according to the necessity of the users.

At this point the BDD manager is selected (being BuDDy the default manager), and it is mandatory to execute the `init_bdd` command before running any other instruction.

The desgined functions are shown in Tables 3.1 to 3.4:

Table 3.1: init_bdd command

**init_bdd(string bdd_name, string library, int node_num, int cache_size)**

This function creates an instance of the BDD factory. The user can choose the BDD manager 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 1 000, and if the manager selected is CUDD, both values are 32 767.

It is mandatory to execute this instruction before executing any other command of this library.

**Arguments:**

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

`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 ommited, “buddy” 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 1 000 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 1 000 for BuDDy and 32 767 for CUDD.

**Returned value:**

N/A.

**Examples:**

```
init_bdd(‘‘bdd_1’’)

init_bdd(‘‘bdd_1’’, ‘‘buddy’’)

init_bdd(‘‘bdd_1’’, ‘‘cudd’’)

init_bdd(‘‘bdd_1’’, ‘‘buddy’’, 2000)

init_bdd(‘‘bdd_1’’, ‘‘cudd’’, 2000)

init_bdd(‘‘bdd_1’’, ‘‘buddy’’, 2000, 5000)

init_bdd(‘‘bdd_1’’, ‘‘cudd’’, 2000, 5000)
```

Table 3.2: set_max_node_num command

**set_max_node_num(string bdd_name, int size)**

With this command the user can modify the maximum number of nodes of the created BDD.

**Arguments:**

`bdd_name`: Name of the BDD.

`size`: The maximum number of nodes to set to the BDD factory, meaning the number of nodes that can be allocated in the structure.

**Returned value:**

N/A.

**Examples:**

```
set_max_node_num(‘‘bdd_1’’, 100)
```

Table 3.3: set_cache_ratio command

**set_cache_ratio(string bdd_name, int cache_ratio)**

This instruction allows to increase the cache ratio of the BDD.

**Arguments:**

`bdd_name`: Name of the BDD.

`cache_ratio`: The increasement 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.

**Returned value:**

N/A.

**Examples:**

```
set_cache_ratio(‘‘bdd_1’’, 10)
```

Table 3.4: reset_bdd command

**reset_bdd(string bdd_name)**

It ends the BDD factory and starts it again with the same BDD manager that was chosen in the init_bdd() command.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

N/A.

**Examples:**

`reset_bdd(‘‘bdd_1’’)`

### 3.1.2 Creating and managing variables

Once the BDD factory is created, another important functionality is the ability of adding logic variables in order to build the desired structure.

With the instructions described in Tables 3.5 to 3.8, users can add and manage variables to the created BDDs.

Table 3.5: new_variable command

**new_variable(string bdd_name, string variable_name, string var_type)**

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

**Arguments:**

`bdd_name`: Name of the BDD.

`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”.

**Returned value:**

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

**Examples:**

`new_variable(‘‘bdd_1’’, ‘‘x’’)`

`new_variable(‘‘bdd_1’’, ‘‘x1’’, ‘‘boolean’’)`

`new_variable(‘‘bdd_1’’, ‘‘x2’’, ‘‘tristate’’)`

Table 3.6: new_variable_from_expression command

**new_variable_from_expression(string bdd_name, string expression)**

This instruction is used to create a new variable after evaluating a logical expression. The expression could be introduced explicitly (informing the name of the variables and the logical operations) or using the CNF.

**Arguments:**

`bdd_name`: Name of the BDD.

`expression`: The expression to evaluate.

If the expression is set from the explicit form, the variables used must exist in the factory, showing an error if some of them do not. 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 ("x xor y")
- nand ("x nand y")
- nor ("x nor y")
- xnor ("x xnor y")
- if then ("if x then y")
- if then else ("if x then y else z")
- implies ("x -> y")
- if and only if <-> ("x iff y")
- equal ("x = y")
- true ("x = true")
- false ("x = false")

With the CNF way, the expression could be informed introducing the name of a file (.cnf) that contains the expression following the syntax rules of that files, or entering the clauses manually, where the variables are informed by their index, that can be consulted with the print_variables() command. It is mandatory to end the expression with a 0.

**Returned value:**

`index_var`: Index of the variable created. It returns -1 if case of error.

**Examples:**

```
new_variable_from_expression(‘‘bdd_1’’, ‘‘x and y or (not z and x)’’)
```

```
new_variable_from_expression(‘‘bdd_1’’, ‘‘1 2 0 -1 3 2 0’’)
```

```
new_variable_from_expression(‘‘bdd_1’’, ‘‘cnfFile.cnf’’)
```

Table 3.7: add_cnf_var command

| **add_cnf_var(string bdd_name, string name)** |
|---|
| This command adds an intermediate CNF variable that is not the result of the evaluation of the CNF expression.<br>**Arguments:**<br>`bdd_name`: Name of the BDD.<br>`name`: The name of the variable.<br>**Returned value:**<br>`index_var`: Index of the variable created. It returns -1 in case of error.<br>**Examples:**<br>`add_cnf_var(‘‘bdd_1’’, ‘‘1_1’’)` |

Table 3.8: restrict_bdd command

| **restrict_bdd(string bdd_name, int expression, string var_to_restrict, string variable_name, bool positive_form)** |
|---|
| This command creates a new variable to be used for the BDD factory. It restricts the value of a variable.<br>**Arguments:**<br>`bdd_name`: Name of the BDD.<br>`expression`: Index of the expression to apply the restriction.<br>`var_to_restrict`: Name of the variable to restrict in the expression.<br>`variable_name`: The name of the variable. It can only contain letters and numbers.<br>`positive_form`: (Optional) Indicates if the value to restrict is in its positive or negative form.<br>**Returned value:**<br>`index_var`: Index of the variable created. It returns -1 in case of error.<br>**Examples:**<br>`restrict_bdd(‘‘bdd_1’’, 1, ‘‘x’’, ‘‘restrictVariable’’)`<br>`restrict_bdd(‘‘bdd_1’’, 2, ‘‘y’’, ‘‘restrictVariable’’, FALSE)` |

### 3.1.3 Consulting operations

The following block of methods, represented in Tables 3.9 to 3.13 and **??**, offers operations to know the state of the BDDs, its configuration or the assigned variables.

Table 3.9: print_bdd command

**print_bdd(string bdd_name)**

This instruction prints the solution of a BDD.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

N/A.

**Examples:**

`print_bdd(‘‘bdd_1’’)`

Table 3.10: get_bdd_library command

**get_bdd_library(string bdd_name)**

This instruction returns the name of the BDD manager chosen.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

N/A.

**Examples:**

`get_bdd_library(‘‘bdd_1’’)`

Table 3.11: get_node_num command

**get_node_num(string bdd_name)**

Gets the number of active nodes in use.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

`node_num`: Number of active nodes in use.

**Examples:**

`get_node_num(‘‘bdd_1’’)`

Table 3.12: is_initialized command

**is_initialized(string bdd_name)**

This instruction allows to the user to know if the BDD factory has been initialized.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

`is_initialized`: It is true if the factory is initialized and false if it is not.

**Examples:**

`is_initialized(‘‘bdd_1’’)`

Table 3.13: print_variables command

**print_variables(string bdd_name)**

This function prints a table showing the index and the content of the variables created.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

N/A.

**Examples:**

`print_variables(‘‘bdd_1’’)`

Table 3.14: expression_to_string command

**expression_to_string(string bdd_name, int expression)**

With this command the content of a variable is printed.

**Arguments:**

`bdd_name`: Name of the BDD.

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

**Returned value:**

N/A.

**Examples:**

`expression_to_string(‘‘bdd_1’’, 1)`

### 3.1.4 Operations over BDDs

There are some functions implemented to work with the created (and configured) BDDs. As a result of the execution of these instructions, the structure of the BDD might change, so the user must be completely sure about the commands are going to be executed. These intructions are explained in Tables 3.15 to 3.18.

Table 3.15: apply_bdd command

**apply_bdd(string bdd_name, int expression)**

This function executes a logical operation expressed as a variable and assocaited to the BDD manager through the new_variable() or new_variable_from_expression() instruction.

**Arguments:**

`bdd_name`: Name of the BDD.

`expression`: The index of the variable with the expression to execute.

**Returned value:**

N/A.

**Examples:**

```
apply_bdd(‘‘bdd_1’’, 1)
```

Table 3.16: done_bdd command

**done_bdd(string bdd_name)**

This command finishes a BDD, liberating the memory space that it was using.

**Arguments:**

`bdd_name`: Name of the BDD.

**Returned value:**

N/A.

**Examples:**

```
done_bdd(‘‘bdd_1’’)
```

Table 3.17: reorder_bdd command

**reorder_bdd(string bdd_name, string reorder_method)**

This instruction allows to reorder the BDD depending on the method specified on the input parameter (if it is informed). The possible methods are:

- “none”
- “window2”
- “window3”
- “sift”
- “random”

**Arguments:**

`bdd_name`: Name of the BDD.

`reorder_method`: (Optional) The method for reordering the BDD. The default value is “sift”.

**Returned value:**

N/A.

**Examples:**

```
reorder_bdd(‘‘bdd_1’’)

reorder_bdd(‘‘bdd_1’’, ‘‘window2’’)
```

Table 3.18: same_bdd command

**same_bdd(string name_bdd_1, string name_bdd_2)**

This function compares two BDDs. The BDDs could be BDDs created with the init_bdd() or the read_bdd() commands, expressions which involve BDDs or in the case of the second expression, the constant BDDs “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:**

`result`: The result of comparing the BDDs.

**Examples:**

```
same_bdd(‘‘bdd_1’’, ‘‘bdd_2’’)

same_bdd(‘‘!bdd_1 && bdd_2’’, ‘‘bdd_3’’)

same_bdd(‘‘!bdd_1’’, ‘‘true’’)

same_bdd(‘‘!bdd_1’’, ‘‘false’’)
```

### 3.1.5 I/O operations

The last instructions (Tables 3.19 and 3.20) are related with saving an existing BDD into a file and loading a BDD from a record.

Table 3.19: read_bdd command

**read_bdd(string bdd_name, string file_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:**

`bdd_name`: Name of the BDD.

`file_name`: The name of the input file. The file must end in “.buddy” to store a BuDDy BDD or in “.blif” to store a CUDD BDD.

**Returned value:**

N/A.

**Examples:**

```
read_bdd(‘‘bdd_1’’, ‘‘buddyBDD.buddy’’)
```

```
read_bdd(‘‘bdd_1’’, ‘‘cuddBDD.blif’’)
```

Table 3.20: save_bdd command

**save_bdd(string bdd_name, string file_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 “.blif”.

The file is saved in the current R’s working directory.

**Arguments:**

`bdd_name`: Name of the BDD.

`file_name`: The name of the output file.

**Returned value:**

N/A.

**Examples:**

```
save_bdd(‘‘bdd_1’’, ‘‘buddyExecution’’)
```

## 3.2 Usage of the `rbdd` library

As it was explained, one of the main aims of the present work is to provide a tool which allows creating and managing BDD as simple as possible. Firstly, it is well documented, and in case of error, it prints a descriptive message informing about what is the reasson of the problem. The documentation of the library could be checked executing the instruction `help(package = rbdd)`. An example of that behaviour could be verified executing the code shown in the Figure 3.1 and Figure 3.2. The first one fails because before executing any designed instruction, command `init_bdd` must be run. The second example shows what happens when a function is call with a non-valid parameter.

```
> library(rbdd)
> new_variable("bdd_1", "x")
There is not a BDD created with name bdd_1. Create it with the
    init_bdd command.
[1] -1
```

Figure 3.1: Execution of a command without running the `init_bdd` instruction

```
> library(rbdd)
> init_bdd("bdd_1", "invalid_manager")
Unknown BDD library: invalid_manager. Expected values are "buddy"
    and "cudd"
```

Figure 3.2: Execution of a method with an invalid input

A complete execution using the commands explained previously can be checked in the Figure 3.3. Line 1 imports the created library and line 2 initializes the BDD manager with the default parameters, that is using BuDDy functions. The way of adding new variables to the BDD is shown in line 3 to line 5, and a variable built as a result of a logic expression is assigned in line 6. To consult the logic variables added, the `printVariables()` command could be utilized, as in line 7, which prints a table like that one shown from line 8 to line 14. Line 15 solves the BDD, and the result could be saved in a file running the command of the line 16. The last instructions check the library used and terminate the BDD, liberating the disk space. The content of the generated file which stores the BDD is printed in Figure 3.4 and it represents the solutions that satisfy the configured BDD.

```
1  > library(rbdd)
2  > init_bdd("bdd_1")
3  > x = new_variable("bdd_1", "x")
4  > y = new_variable("bdd_1", "y")
5  > z = new_variable("bdd_1", "z")
6  > expression = new_variable_from_expression("bdd_1", "x and y or (y
      and not z)")
7  > print_variables("bdd_1")
8  ++++++++++++++++++++++++++++
9  Index variable -> Expression
10 ++++++++++++++++++++++++++++
11 Variable 1 -> x
12 Variable 2 -> y
13 Variable 3 -> z
14 Variable 4 -> (x and y) or (y and not z)
15 > apply_bdd("bdd_1", expression)
16 > save_bdd("bdd_1", "buddyExacution")
17 > get_bdd_library("bdd_1")
18 [1] "BuDDy"
19 > done_bdd("bdd_1")
```

Figure 3.3: Full example of usage of the `rbdd` library

```
4 4
0 1 2 3
7 2 1 0
11 1 0 7
4 1 0 1
14 0 11 4
```

Figure 3.4: Result of saving the BDD created

## Chapter 4

# Development of the proposed solution

**R** is well-known by its capability to apply statistical functions to a huge set of data in a very efficient way, but its performance decreases dramatically when not that specific code is called. To fix this issue, a library to execute general purpose methods could be chosen, improving in this way compilation and execution times.

Rcpp [32] is a library which provides functions available in **R** to execute code developed in C++. To carry out that task, the package relies on the direct conversion between **R** data types with the equivalent structures in C++ and viceversa, in such a way that invocations between both parts could be as simple as possible. So, next functions are offered to the user in order to make a type conversion [33]:

- `Rcpp::as` carries out the input conversion of the **R** functions to be used from the C++ code.
- `Rcpp::wrap` obtains the analogous data type to that one returned by a C++ function, which will be provided when the execution of the instruction ends.

Using this library corresponds perfectly with the aim of developing an open source application, easily extensible. Adding new functions is as simple as including the code in a single file and executing an instruction, as it will be explained below.

To make the creation of a package automatically, providing these functions available from the moment when the library is built, the instruction `Rcpp.package.skeleton()` can be run, giving to the programmer the basic skeleton of a **R** package with the detailed Rcpp imports. This process generates the directory depicts in the Figure 4.1, and each subdirectory and file have the following function:

- `R`: Subdirectory which contains the file `RcppExports.R`, where the available functions from **R** are defined in.
- `man`: Folder where documentation of the package and the **R** defined methods are placed.
- `src`: Contains the source code written in C++ to be accessed from the **R** side.
- `DESCRIPTION`: File with the basic information of the package.
- `NAMESPACE`: File where indicate the dependencies with other **R** libraries.

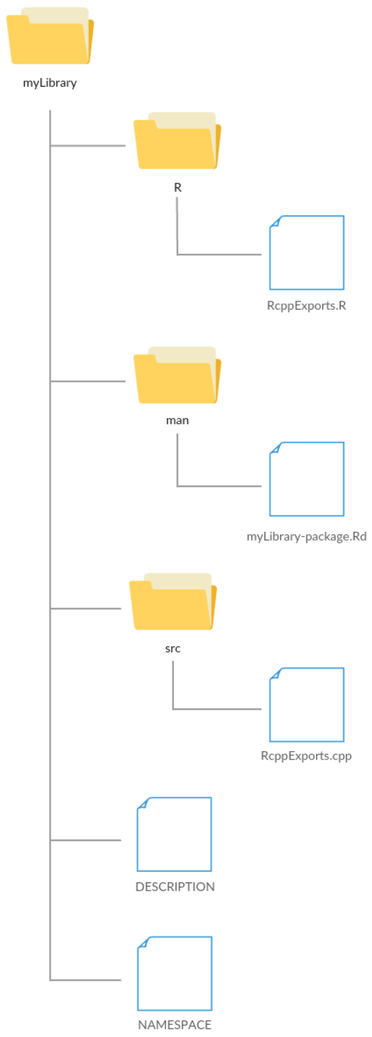


Figure 4.1: Structure of the library created with the instruction `Rcpp.package.skeleton()`

Another feature of the Rcpp library is the possibility of automate the process of *casting* between data, just adding the following line of code

```
//[[Rcpp::export]]
```

before each C++ function, and using the instruction

```
Rcpp::compileAttributes(‘‘package_name’’)
```

After the execution of that process the file `RcppExports.R` is updated with the corresponding code to make the conversion between parameters, and it is used as a link with the definition of the **R** functions and the C++ methods.

## 4.1 Adding own functions

The code that the programmer wants to be available from **R** must be in `.cpp` files inside the `src` folder. Adding the line explained before, the data conversion between **R** types and C++ types is made automatically, simplifying the tasks of the developers.

In the `rbdd` library, the methods of the C++ side have been designed to use as an intermediate layer between **R** and the BDD managers functionalities. In order to achieve this aim, the functions explained in Chapter 3 has been added to the `rbddFunctions.cpp` file as well as multiple internal structures in order to keep the information available during the session.

In addition, several auxiliary classes have been developed to provide supporting functions, like the ones related to parsing functions, to give CNF utilities, to exploit the BDD managers posibilities, etc.

If future developments are implemented and it is needed to add aditional functions to the library, the steps to follow are to:-

1. Include the `//[[Rcpp::export]]` line before the declaration of the method in the `rbddFunctions.cpp` file,
2. Add the code of the function which is called from **R** in the `rbddFunctions.cpp` file,
3. If there are other needed files, include them in the `src` folder,
4. Program the desired behaviour in the parent class of the BDD managers (`vBDDFactory`), adding a new method for each new function, in the `vBDDFactory.hpp` file,
5. Write the code of the function in the specified BDD managers classes, such as `buddyFactory` and `cuddFactory`,
6. Execute the `Rcpp::compileAttributes(‘‘package_name’’)` instruction in order to updete the `RcppExports.R` and `RcppExports.cpp` files.

## 4.2 Dealing with external dependencies

If external dependencies are needed, **R** does not advise to incorporate dynamic libraries, represented with files with extension `.dll` or `.so` for libraries developed in Windows and Linux, respectively [34]. To develop a package which uses some functionalities available in an external library, it must be included a `Makevars` file, and in that file the following variable has to be informed:

```
PKG_LIBS=dependency1.o dependency2.o ...
```

where files with `.o` extension are the result of the compilation of the source files of the library that is the consumed dependency.

This implies that if a change is made over some of the source files of the extern dependency, it would have to obtain the static file and replace it in the source directory. If the modification is the addition of source files, it is enought to include them in the `PKG_LIBS` variable described before.

The compilation of the source files of the library under a Linux-based environment is made with GNU [35] tools like `make`, `gcc`, `g++`, etc.

In most cases it will be necessary to follow the typical way to compile a C++ library, which is showed in the Figure 4.2. At the end of that process the `.o` files would be generated and ready to include in the `src` folder of the library.

```
cd dependency_directory
./configure CC=gcc CXX=g++ CXXFLAGS="-fPIC -std=c++11" CFLAGS="-
   fPIC -std=c11"
make
```

Figure 4.2: Compilation process of an external library

However, if it is expected to generate a package to use it in a Windows system, that compilation must be done in an equivalent system, too. So, it is required to use environments which provides these GNU utilities. A good example is MSYS [36], that includes a Linux *bash* and the main tools needed to compile the C and C++ code.

It is worth nothing that if a library with compatibility with 32 and 64 bits systems it is wanted to be develop, it is necessary the compilation of the dependencies under compiler of each architecture, and to include the `.o` files as are provided as the result of this process into the source directory of the developed package. The Figure 4.3 shows how to indicate in a unique `Makevars` file the location of the dependencies files depending on the architecture and the operative system.

```
ifeq ($(OS),Windows_NT)
  ifeq "$(WIN)" "64"
    PKG_ROOT=./include/windows/x64
  else
    PKG_ROOT=./include/windows/x86
  endif
else
  UNAME_S := $(shell uname -s)
  ifeq ($(UNAME_S),Linux)
    PKG_ROOT = ./include/linux
  endif
endif
```

Figure 4.3: Location of the depencies depending of the architecture of the machine

To build the **R** library it is necessary to use the command:

```
R CMD INSTALL --build --compile-both package_name
```

# Chapter 5

# Experimental validation

The final step once the library has been built is to demonstrate it works as it is expected. To achieve this aim some real examples will be implemented just utilising the developed wrapper. Internet can provide a wide set of real usages of BDDs for solving a great variety of problems.

To enrich the presentation of the results, a real example of a representition of an SPL will be showed, as a proof of concept of the relationship between the main topics involved on this work.

## 5.1 Propagation of a signal

The first example is the implementation of a signal propagator from the input to the output of a circuit [37]. Sometimes it is needed to fix all the inputs of a gate except one of them, so the signal can be propagated from that input to the output, in such a way that a change on that signal will always have an effect on the output.

In order to simplify the problem, a gate of three inputs and an only output it is assumed, representing the boolean function of the Equation (5.1):

$$z(a, b, c) = (a \quad \vee \quad b) \quad \wedge \quad c \tag{5.1}$$

and the truth table is described in Table 5.1.

The desired behaviour is to propagate the value of **b** to the output. So what it is needed is to find the values of **a** and **c** which allow a change in the output when the value of **b** changes. To found the solution of this problem, the function shown in the Equation (5.2) that fulfill the requirement could be built.

$$p(a, c) = z(a, 0, c) \quad \oplus \quad z(a, 1, c) \tag{5.2}$$

Table 5.1: Truth table of the propagator circuit

| **a** | **b** | **c** | **z** |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

With this expression and checking on the truth table, it could be set that the premise it is satisfied only when $\mathbf{a} = 0$ and $\mathbf{c} = 1$.

To solve the problem using the developed library, the code illustrated in the Figure 5.1 could be used. The instructions initialize the BDD and the respective variables, print them to check everything is correct, and applies the BDD in order to get the solution which solves the set out problem. The last lines print the content of the solution and free the memory space used.

The output obtained after executing this script is shown in the Figure 5.2. The last line is the solution of the BDD, which means that the conditions that satisfy the premise are those that $\mathbf{a} = 0$ and $\mathbf{c} = 1$, what is exactly the expected solution.

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

# Creates the BDDfactory
> init_bdd("signal_propagator")

# Creates variables in the factory
> new_variable("signal_propagator", "a")
> new_variable("signal_propagator", "b")
> new_variable("signal_propagator", "c")

# Creates a variable from an expression
> z = new_variable_from_expression("signal_propagator", "(a or b)
   and c")

# Creates variables restricting the value of a variable
> restrict_bdd("signal_propagator", z, "b", "restrict1")
> restrict_bdd("signal_propagator", z, "b", "restrict2", FALSE)
> fixed_b = new_variable_from_expression("signal_propagator", "
   restrict1 xor restrict2")

# Prints the defined variables
> cat("The defined variables are")
> print_variables("signal_propagator")

# Applies the final expression in order to be computed
> apply_bdd("signal_propagator", fixed_b)

# Prints the solution of the BDD
> cat("\nThe solved bdd is:\n")
> print_bdd("signal_propagator")

# Frees the space used by the BDD
> done_bdd("signal_propagator")
```

Figure 5.1: Implementation of the signal propagator with `rbdd`

```
The defined variables are:
+++++++++++++++++++++++++++
Index variable -> Expression
+++++++++++++++++++++++++++
Variable 1 -> a
Variable 2 -> b
Variable 3 -> c
Variable 4 -> (a or b) and c
Variable 5 -> restrict1
Variable 6 -> restrict2
Variable 7 -> restrict1 xor restrict2

The solved bdd is:
<0:0, 2:1>
```

Figure 5.2: Output of the execution of the signal propagator example

## 5.2 Modified Condition / Decision Coverage

When the behaviour of a condition is going to be tested in order to prove that it satisfies the expected results, one way to be sure everything is correct is to check that for all the possible combinations of the inputs, the calculated output is the right one. That technique is known as Multiple Condition Coverage (MCC), and it can not be used in real critical software projects, where the number of combinations grows exponentially. For that reason, MCC is not a real possibility when the reliability of a software is pretented to be checked.

For that reason, it comes to the conclusion that it is necessary to follow some criteria which allow to cover as many options as it is possible with the less number of combinations. In this way, the following approachs could be considered:

- **Condition Coverage**: Every logic variable is tested for all its possible values (0 or 1).
- **Decision Condition Coverage (DCC)**: This technique increases the previous method adding the tests which include all the possible options of the output.
- **Modified Condition / Decision Coverage (MC/DC)**: It verifies that it is checked the effect of changing the value of each variable independently of the value of the other variables.

MC/DC is widely extended for testing critical software applications, like the software of planes, which must have a high reliability [38]. The task of choosing a set of tests that satisfied the MC/DC could be complex, because it has to be granted a sufficient coverage of the branches and when the number of variables is big enough it could be almost impossible to achive.

One way to achieve it could be to use the signal propagator explained in the previous section could, applying it for every logic variable and selecting the result of each iteration. The obtained output after each iteration is a set of the those tests which allow the signal propagator of the variables. After that, the test designer has to choose those ones that provide independence pairs for each condition.

In order to illustrate that use of the developed library with an example, it is going to be calculated the test cases which satisfy the MC/DC of the Equation (5.3) [38].

$$z(a,b,c,d) = (a \quad \vee \quad b) \quad \wedge \quad (c \quad \vee \quad d) \tag{5.3}$$

The truth table of that logic expression is shown in the Table 5.2.

Looking at the truth table, the test cases could be calculated studying the independent effects of each variable, as it is highlighted in Table 5.3, Table 5.4, Table 5.5 and Table 5.6 for variables **a**, **b**, **c** and **d**, respectively. It has to be considered that the more number of inputs the system has, the more difficult to select the appropriate set of conditions, so it becomes critical to automate this process.

Table 5.2: Truth table of the expression to apply the MC/DC technique

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Table 5.3: Independent effect of the variable **a**

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Table 5.4: Independent effect of the variable **b**

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 |

Table 5.5: Independent effect of the variable **c**

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |

Table 5.6: Independent effect of the variable **d**

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |

Finally, the Table 5.7 shows the union of the solutions for every single variable, that represents the actual set of test for the MC/DC of the proposed logical expression.

Table 5.7: Union of the independent effect of each variable

| **a** | **b** | **c** | **d** | **z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |

Figure 5.3 describes how to implement that functionality with the `rbdd` library, applying the logic of the signal propagator for each variable.

The output after executing that script it is showed in the Figure 5.4, and it could be observed that the different calculated sets include the expected results compared with the theorical solution.

According to the Hayhurst el al.'s tutorial [38, Chapter 2, Table 3], the truth table is like the one shown in the Table 5.8. Columns shaded in gray indicate the independence pairs for each conditions.

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

# Creates the BDDfactory
> init_bdd("mcdc")

# Creates variables in the factory
> new_variable("mcdc", "a")
> new_variable("mcdc", "b")
> new_variable("mcdc", "c")
> new_variable("mcdc", "d")

# Creates a variable from an expression
> z = new_variable_from_expression("mcdc", "(a or b) and (c or d)")

# Creates variables restricting the value of the variable a
> restrict_bdd("mcdc", z, "a", "restrict1")
> restrict_bdd("mcdc", z, "a", "restrict2", FALSE)
> fixed_a = new_variable_from_expression("mcdc", "restrict1 xor
   restrict2")

# Applies the final expression in order to be computed
> apply_bdd("mcdc", fixed_a)

# Prints the solution of the BDD
> cat("\nCases when variable a is fixed are:\n")
> print_bdd("mcdc")

# Creates variables restricting the value of the variable b
> restrict_bdd("mcdc", z, "b", "restrict3")
> restrict_bdd("mcdc", z, "b", "restrict4", FALSE)
> fixed_b = new_variable_from_expression("mcdc", "restrict3 xor
   restrict4")

# Applies the final expression in order to be computed
> apply_bdd("mcdc", fixed_b)

# Prints the solution of the BDD
> cat("\nCases when variable b is fixed are:\n")
> print_bdd("mcdc")

# Creates variables restricting the value of the variable c
> restrict_bdd("mcdc", z, "c", "restrict5")
> restrict_bdd("mcdc", z, "c", "restrict6", FALSE)
> fixed_c = new_variable_from_expression("mcdc", "restrict5 xor
   restrict6")

# Applies the final expression in order to be computed
> apply_bdd("mcdc", fixed_c)

# Prints the solution of the BDD
> cat("\nCases when variable c is fixed are:\n")
> print_bdd("mcdc")
```

```
# Creates variables restricting the value of the variable d
> restrict_bdd("mcdc", z, "d", "restrict7")
> restrict_bdd("mcdc", z, "d", "restrict8", FALSE)
> fixed_d = new_variable_from_expression("mcdc", "restrict7 xor
   restrict8")

# Applies the final expression in order to be computed
> apply_bdd("mcdc", fixed_d)

# Prints the solution of the BDD
> cat("\nCases when variable d is fixed are:\n")
> print_bdd("mcdc")

# Frees the space used by the BDD
> done_bdd("mcdc")
```

Figure 5.3: Getting test cases for MC/DC with `rbdd`

```

Cases when variable a is fixed are:
<1:0, 2:0, 3:1><1:0, 2:1>

Cases when variable b is fixed are:
<0:0, 2:0, 3:1><0:0, 2:1>

Cases when variable c is fixed are:
<0:0, 1:1, 3:0><0:1, 3:0>

Cases when variable d is fixed are:
<0:0, 1:1, 2:0><0:1, 2:0>
```

Figure 5.4: Output of the execution of the MC/DC test cases

Table 5.8: Truth table of the problem with the independence effect of each variable

| **a** | **b** | **c** | **d** | **z** | **a** | **b** | **c** | **d** |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | | | | |
| 0 | 0 | 0 | 1 | 0 | X | X | | |
| 0 | 0 | 1 | 0 | 0 | X | X | | |
| 0 | 0 | 1 | 1 | 0 | X | X | | |
| 0 | 1 | 0 | 0 | 0 | | | X | X |
| 0 | 1 | 0 | 1 | 1 | | X | | X |
| 0 | 1 | 1 | 0 | 1 | | X | X | |
| 0 | 1 | 1 | 1 | 1 | | X | | |
| 1 | 0 | 0 | 0 | 0 | | | X | X |
| 1 | 0 | 0 | 1 | 1 | X | | | X |
| 1 | 0 | 1 | 0 | 1 | X | | X | |
| 1 | 0 | 1 | 1 | 1 | X | | | |
| 1 | 1 | 0 | 0 | 0 | | | X | X |
| 1 | 1 | 0 | 1 | 1 | | | | X |
| 1 | 1 | 1 | 0 | 1 | | | X | |
| 1 | 1 | 1 | 1 | 1 | | | | |

## 5.3 Implementing a SPL

As it was explained at the begining of this work, one of the aims of the line which the project belongs to is the systematic reuse of software. If it is put the development of SPLs under the spotlight, it could be found that modeling the common structural parts and variables of every product turns becomes crucial. To do it, feature diagrams are usually utilised, for using them as a propositional logic formaula.

One method to solve the resultant logic formula applied to a feature model is to represent it as a BDD, and at which point the `rbdd` library could be useful. It has as advantages the simple way to define the problem with a few number of instructions, the capability of the package to reach the solution in a reasonable time and the posibility of reordering the clauses in order to improve the execution speed of the program.

Benavides et al. [39] propose an example of a feature model, depicted in the Figure 5.5. It is inspired by the mobile phone industry and it illustrates the way features are used to design and build software for mobile phones. The software of the device is determined by the features which it supports, so analysing the model it could be pointed that all the telephones must include support for calls, and the posibility of displaying the information in either a basic, colour or high resolution screen. Also, some optional features like the avalability of Global Positioning System (GPS) or camera are described, too.

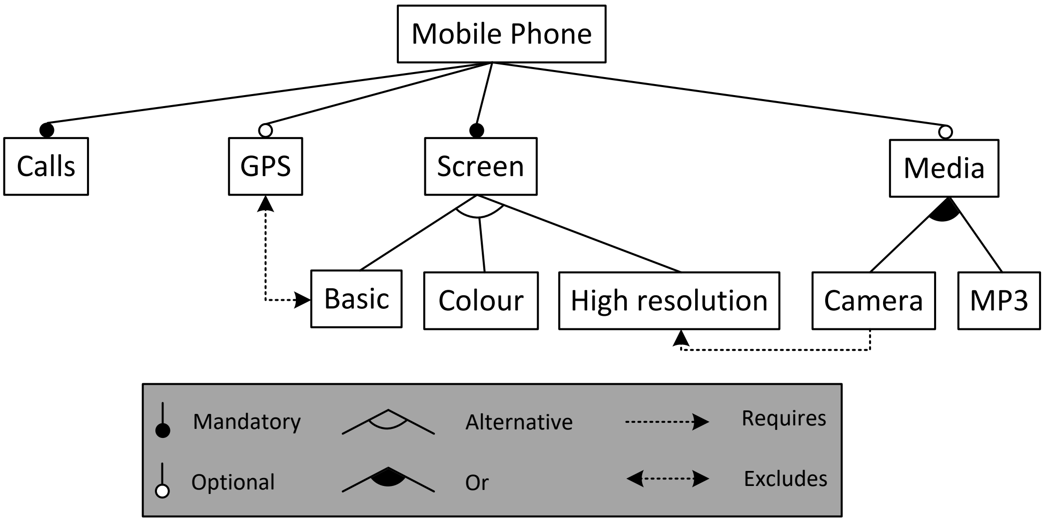


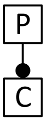
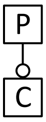
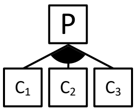
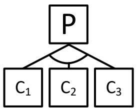
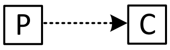
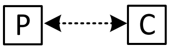
Figure 5.5: Feature model which describes features of a mobile phone

The translation of a feature model into a propositional logic formula might follow the following steps [39]:

1. Each feature of the feature model maps to a variable of the propositional formula,
2. Each relationship of the model is mapped into one or more small formulas depending on the type of relationship,
3. The resulting formula is the conjunstion of all the resulting formulas of the previous step plus and additional constraint assigning `true` to the variable that represents the root.

The rules for getting the equivalence between propositional formulas and relations in the feature model are explained in Table 5.9.

Table 5.9: Mapping between features and propositional formulas

| **Relationship** | **Propositional Logic Mapping** |
|---|---|
|  | $P \leftrightarrow C$ |
|  | $C \rightarrow P$ |
|  | $P \leftrightarrow (C_1 \vee C_2 \vee \ldots \vee C_n)$ |
|  | $C_1 \leftrightarrow (\neg C_2 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge$<br>$C_2 \leftrightarrow (\neg C_1 \wedge \ldots \wedge \neg C_n \wedge P)) \wedge \ldots \wedge$<br>$C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge \neg C_{n-1} \wedge P))$ |
|  | $A \rightarrow B$ |
|  | $\neg(A \wedge B)$ |

Following those guidelines, the code of the Figure 5.6 implements the problem exposed previously in the Figure 5.5. Firstly, the variables of the model are created and the logic expression is built. The generated output is shown in the Figure 5.7 and represents the different scenarios which satisfy the premises defined in the designed system.

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

# Creates the BDDfactory
> init_bdd("feature_model")

# Creates variables in the factory
> new_variable("feature_model", "mobilePhone")
> new_variable("feature_model", "calls")
> new_variable("feature_model", "gps")
> new_variable("feature_model", "screen")
> new_variable("feature_model", "media")
> new_variable("feature_model", "basic")
> new_variable("feature_model", "colour")
> new_variable("feature_model", "highResolution")
> new_variable("feature_model", "camera")
> new_variable("feature_model", "mp3")

# Creates a variable from an expression
> expression = new_variable_from_expression("feature_model", "(
   mobilePhone = true) and (mobilePhone iff calls) and (gps ->
   mobilePhone) and (mobilePhone iff screen) and (media ->
   mobilePhone) and ((basic iff (not colour and not highResolution
   and screen)) and (colour iff (not basic and not highResolution
   and screen)) and (highResolution iff (not basic and not colour
   and screen))) and (media iff (camera or mp3)) and not(gps and
   basic) and (camera -> highResolution)")

# Applies the expression in order to be computed
> apply_bdd("feature_model", expression)

# Prints the solution of the BDD
> cat("\nThe solution of the feature model is:\n")
> print_bdd("feature_model")

# Frees the space used by the BDD
> done_bdd("feature_model")
```

Figure 5.6: Code which implements the feature model

```
The solution of the feature model is:
<0:1, 1:1, 2:0, 3:1, 4:0, 5:0, 6:0, 7:1, 8:0, 9:0><0:1, 1:1, 2:0,
   3:1, 4:0, 5:0, 6:1, 7:0, 8:0, 9:0><0:1, 1:1, 2:0, 3:1, 4:0, 5:1,
    6:0, 7:0, 8:0, 9:0><0:1, 1:1, 2:0, 3:1, 4:1, 5:0, 6:0, 7:1,
   8:0, 9:1><0:1, 1:1, 2:0, 3:1, 4:1, 5:0, 6:0, 7:1, 8:1><0:1, 1:1,
    2:0, 3:1, 4:1, 5:0, 6:1, 7:0, 8:0, 9:1><0:1, 1:1, 2:0, 3:1,
   4:1, 5:1, 6:0, 7:0, 8:0, 9:1><0:1, 1:1, 2:1, 3:1, 4:0, 5:0, 6:0,
    7:1, 8:0, 9:0><0:1, 1:1, 2:1, 3:1, 4:0, 5:0, 6:1, 7:0, 8:0,
   9:0><0:1, 1:1, 2:1, 3:1, 4:1, 5:0, 6:0, 7:1, 8:0, 9:1><0:1, 1:1,
    2:1, 3:1, 4:1, 5:0, 6:0, 7:1, 8:1><0:1, 1:1, 2:1, 3:1, 4:1,
   5:0, 6:1, 7:0, 8:0, 9:1>
```

Figure 5.7: Conditions which satisfied the proposed feature model

## 5.4 Increasing the number of nodes of a BDD

Finally, to complete the validation of the developed library, it is going to be analysed how the number of nodes of a BDD changes when logical expressions are added to the system and, therefore, its complexity increases. Also, it will be checked how the number of nodes changes applying different reording algorithms.

To achieve that last behaviour, a BDD is going to be build reading expressions in CNF and getting the number of nodes after each iteration, that is, when a new logic statement is added to the structure. The files which contains the expressions are read sequentially, adding a new sentence in each iteration and calculating the number of nodes at that moment.

The benchmark is designed building three BDDs, which features are described in the Table 5.10. Figure 5.8 explains the set of instructions executed to obtain the number of nodes of the first BDD studied, and its structure is analogous to the other BDDs.

Table 5.10: Features of the analysed BDDs

| **BDD** | **Number of variables** | **Number of expressions** |
|---|---|---|
| axtls | 56 | 64 |
| fiasco | 93 | 95 |
| uClibc | 204 | 178 |

```
> library(rbdd)
> library(ggplot2)

> axtls_file = file("axtls.cnf")
> axtls_length = length(readLines(axtls_file))
> clauses = readLines(axtls_file, n = 1)
> last_space = gregexpr(" ", clauses, fixed=TRUE)
> loc<-last_space[[1]]
> space<-loc[length(loc)]
> clauses = substr(clauses, 0, space - 1)
> number_nodes_axtls_none = 0
> number_nodes_axtls_sift = 0
> number_nodes_axtls_rand = 0
> index = 1

> for (line_number in 1:(axtls_length - 1)) {
>       init_bdd("axtls")
>       clauses_new = paste(clauses, line_number)
>       liness = readLines(axtls_file, n = line_number + 1)
>       liness[1] = clauses_new
>       fileCon<-file("output_axtls.cnf")
>       writeLines(liness, fileCon)
>       close(fileCon)
```

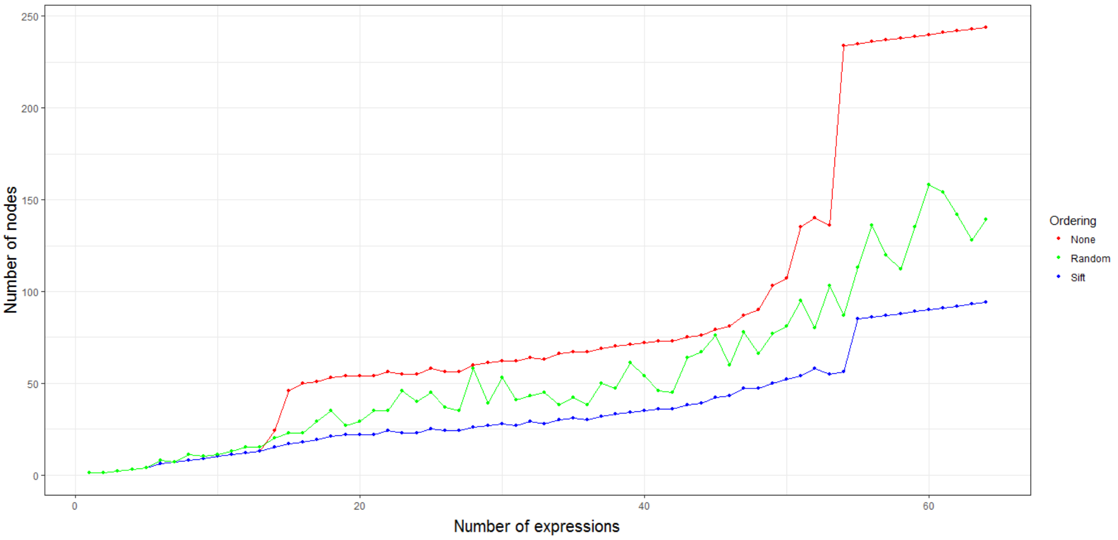
```
>       axtls_exp = new_variable_from_expression("axtls", "
  output_axtls.cnf")
>       apply_bdd("axtls", axtls_exp)
>       reorder_bdd("axtls", "none")
>       number_nodes_axtls_none[index] = get_node_num("axtls")
>       reorder_bdd("axtls", "sift")
>       number_nodes_axtls_sift[index] = get_node_num("axtls")
>       reorder_bdd("axtls", "random")
>       number_nodes_axtls_rand[index] = get_node_num("axtls")
>       index = index + 1
>       done_bdd("axtls")
> }
> close(axtls_file)

> number_expressions <- seq(1, axtls_length - 1)
> axtls_none.data <- data.frame(number_expressions,
  number_nodes_axtls_none)
> axtls_sift.data <- data.frame(number_expressions,
  number_nodes_axtls_sift)
> axtls_rand.data <- data.frame(number_expressions,
  number_nodes_axtls_rand)
> axtls_graph <- ggplot() +
>       geom_line(data=axtls_none.data, aes(x=number_expressions, y
  =number_nodes_axtls_none, colour="None"), color="red") +
>       geom_point(aes(x=number_expressions, y=
  number_nodes_axtls_none, colour="None"), size=1) +
>       geom_line(data=axtls_sift.data, aes(x=number_expressions, y
  =number_nodes_axtls_sift, colour="Sift"), color="blue") +
>       geom_point(aes(x=number_expressions, y=
  number_nodes_axtls_sift, colour="Sift"), size=1) +
>       geom_line(data=axtls_rand.data, aes(x=number_expressions, y
  =number_nodes_axtls_rand, colour="Random"), color="green") +
>       geom_point(aes(x=number_expressions, y=
  number_nodes_axtls_rand, colour="Random"), size=1) +
>       scale_colour_manual(name="Ordering", values=c("None"="red",
   "Sift"="blue", "Random"="green")) +
>       labs(x = "Number of expressions", y = "Number of nodes") +
>       theme_bw() + theme(axis.title.x = element_text(size = 15,
  vjust=-.2)) + theme(axis.title.y = element_text(size = 15, vjust
  =0.3))
```

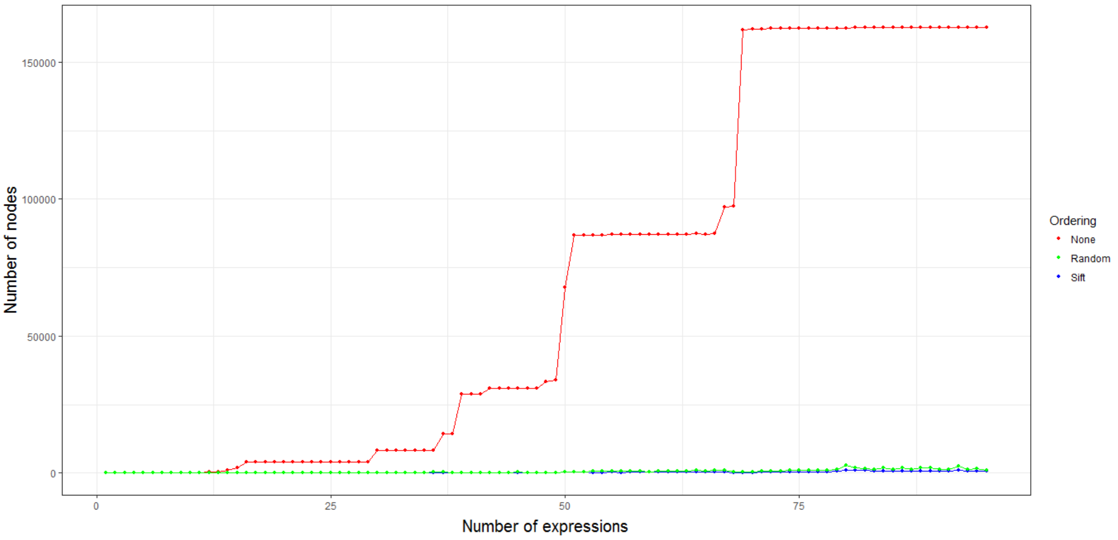
Figure 5.8: Code which obtains the number of nodes i every iteration applying different reorder algorithms

Figures 5.9a to 5.9c depict the result of executing the benchmark described. They show the tables with the relationship between the increase of nodes in the BDD when the logic expressions are added to the system for different reording algorithms.

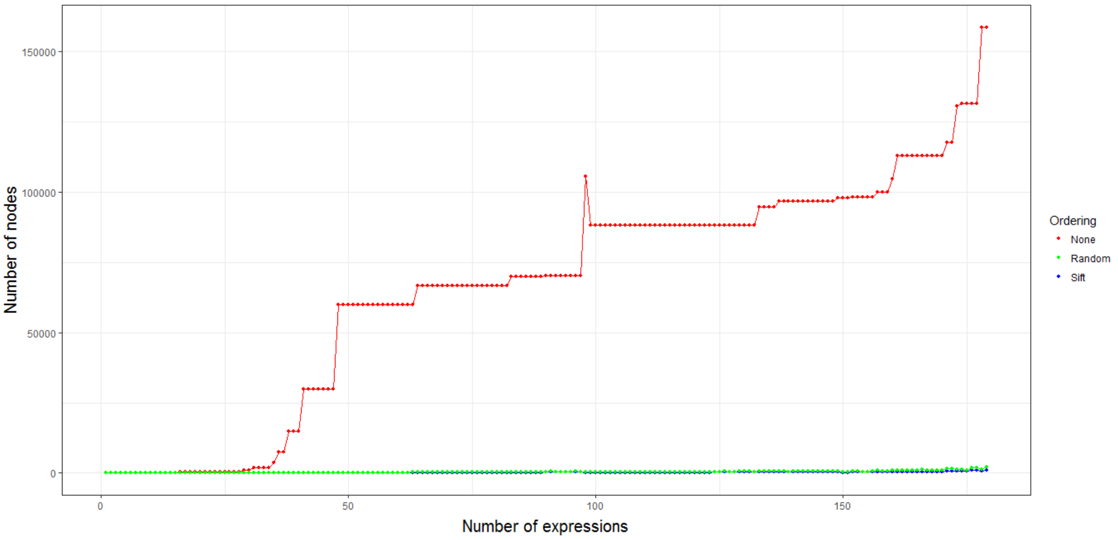
The conclusion that it could be extracted after those executions is that every system is different and its growth in terms of complexity varies a lot depending on how new elements are added to the system, but choosing the right reorder algorithm means a crucial impact in terms of memory space, especially when the system has a big number of expressions and nodes. For that reason this task could not be afforded carelessly, turning into a key factor in the development of BDDs phase.



(a) Reordering heurestic influence on the axtls BDD growth according to the formula size



(b) Reordering heurestic influence on the fiasco BDD growth according to the formula size



(c) Reordering heurestic influence on the uClibc BDD growth according to the formula size

Figure 5.9: Analysis of the raise of the number of nodes in a BDD when the number of logic expressions increases

## Chapter 6

# Conclusions and future work

This chapter provides some consluding remarks and direction for future research.

## 6.1 Conclusions

The main challenge this work has faced is the creation of a library which is able to create and manage BDDs, a kind of structure that allows to operate efficiently with complex logic expressions. In addition, it was expected a friendly interface, not having a large learning curve, but being possible to add features using instructions and changing the default parameters to customise the behaviour of the program.

It could be confirmed that the aim has been successfully fullfilled, considering that it has been made a complete specification to deal with the described problem and, taking it as the basis, the system has been developed and validated.

To validate the library is the desired, four case studies have been used and discussed the results after their execution.

Getting into detail, the obtained application allows to simplify the way an user can operate with BDDs from **R**, something not possible to do without the `rbdd` library because of the nonexistent support of those structures in that programming language. In addition, the library as well as its functions have been well documented, being designed `man` pages accesible from the software environment using the command `help(package = rbdd)`, showing high-detailed descriptions of the instructions and their inputs and outputs, examples of usage, etc.

Another accomplished goal has been the premise of keeping the developtment of the solution under the philosophy of the free software. The code and some examples are available in a public git repository [40], so any user who wants to expand its functionalities to adapt the library to cover specific requirements could do it easily, increasing the ability of the software and resulting beneficiary the whole community.

To achieve the aims of the project, a number of concepts acquired during the master have been applied, belonging to the main fields of the degree, such as the development of SPLs or the specification of software systems. The fact of being a distance learning system has promoted a new methodology of work, utterly different than the others followed on previous stages of the studying lifetime.

As BDDs have not been a topic studied during the previous degree and the master, it has been used the skills of researching in order to find useful information, finding valuable references and turning to the authors who discussed about the features and benefits of those structures.

Finally, project management has turned into a key factor related with the success in the attainment of the defined aims. The freedom offered by the project director has allowed to define a convenient schedule, completing each phase in the approppiate moment.

## 6.2 Future work

Due to the mentioned nature of free software of the implemented package, the possibilities to increase the functionality of the tool are countless. Every target user, meaning programmers coming from the logic field as well as **R** developers who want to utilise this solution to afford operations for which BDDs are the more suitable option. In this way, the structure of the library has been designed such that including new methods is a simple process, described in Chapter 4.

Main operations have been deployed on the package, but there are some functions not covered with the current version of the `rbdd` library. It could be added more BDD managers, like JDD [9] or CAL [41], but that is a more complex task because it means adapting the implemented methods to support those of the new manager.

Another point to work would be the publication of the package in the **R** repository [14]. The development of the library has followed the Comprehensive R Archive Network (CRAN) Repository Policy [42], so this process would not be so hard.

Finally, a new interface could be developed to handle the BDDs on an object-oriented way from the **R** side, providing a more **R** flavoured syntax to interact with the created variables.

# Bibliography

[1] D. E. Knuth, "Donald E. Knuth Lectures." http://scpd.stanford.edu/free-stuff/engineering-archives/donald-e-knuth-lectures. Checked: 27/09/2016.

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

[3] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," *Bell System Technical Journal*, no. 38, pp. 985–999, 1959.

[4] M. Mendonça, *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, Ontario, Canada, 2009.

[5] B. Bollig and Ingo Wegener, "Improving the Variable Ordering of OBDDs is NP-Complete," *IEEE Transactions on Computers*, vol. 9, no. 45, pp. 993–1002, 1996.

[6] 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.

[7] C. Meinel and Thorsten Theobald, *Algorithms and Data Structures in VLSI Design*. Berlin: Springer-Verlag, 1988. ISBN 3-540-64486-5.

[8] "JavaBDD." http://javabdd.sourceforge.net/. Checked: 27/09/2016.

[9] "JDD." https://bitbucket.org/vahidi/jdd/wiki/Home. Checked: 27/09/2016.

[10] "CUDD." http://vlsi.colorado.edu/~fabio/CUDD/. Checked: 27/09/2016.

[11] "BuDDy." https://sourceforge.net/projects/buddy/. Checked: 27/09/2016.

[12] "The R Project for Statistical Computing." https://www.r-project.org/. Checked: 27/09/2016.

[13] "The S System." http://ect.bell-labs.com/sl/S/. Checked: 27/09/2016.

[14] "Available CRAN Packages." https://cran.r-project.org/. Checked: 27/09/2016.

[15] "Ubuntu Release Notes." https://wiki.ubuntu.com/XenialXerus/ReleaseNotes. Checked: 30/09/2016.

[16] "The LaTeX Project." https://www.latex-project.org/. Checked: 30/09/2016.

[17] D. Fernández Amorós, Rubén Heradio, José A. Cerrada, and Carlos Cerrada, "A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models," research note, ETS de Ingeniería Informática, Universidad Nacional de Educación a Distancia, September 2014.

[18] J. Bosch, "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization," research note, University of Groningen, July 2002.

[19] D. Beuche and Mark Dalgarno, "Software Product Line Engineering with Feature Models," tech. rep., Pure Systems.

[20] P. Donohoe, "Introduction to Software Product Lines," (Pittsburgh, PA 15213), Software Engineering Institute, Carnegie Mellon University, 2014.

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

[22] 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, Software Engineering Institute - Carnegie-Mellon University, November 1990.

[23] K. A. Gilroy, Edward R. Comer, J. Kaye Grau, and Patrick J. Merlet, "Impact of Domain Analysis on Reuse Methods," technical report, U.S. Army Research Office, November 1989.

[24] R. Prieto Dï¿½az, "Domain Analysis for Reusability," (Washington DC), IEEE Computer Society, 1987.

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

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

[27] R. Kaun and Manu Bansal, "BDD Ordering and Minimization Using Various Crossover Operators in Genetic Algorithm," *International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering*, vol. 2, no. 3, pp. 1247–1250, 2014.

[28] H. Reif Andersen, "An Introduction to Binary Decision Diagrams," lecture notes, IT University of Copenhagen, 1999.

[29] D. Sieling, "The Nonapproximability of OBDD Minimization," *Information and Computation*, vol. 172, no. 2, pp. 103–138, 2002.

[30] "BuDDy - Variable reordering." `http://buddy.sourceforge.net/manual/group__reorder.html`. Checked: 19/01/2017.

[31] F. Somenzi, "CUDD: CU Decision Diagram Package Release 3.0.0," December 2015.

[32] "CRAN - Package Rcpp." `https://cran.r-project.org/web/packages/Rcpp/`. Checked: 26/09/2016.

[33] D. Eddelbuettel, *Seamless R and C++ Integration with Rcpp*. New York: Springer, 2013. ISBN 978-1-4614-6867-7.

[34] "Writing R Extensions." https://cran.r-project.org/doc/manuals/R-exts.html. Checked: 26/09/2016.

[35] "GNU operating system." http://www.gnu.org/home.en.html. Checked: 26/09/2016.

[36] "MSYS." http://www.mingw.org/wiki/msys. Checked: 26/09/2016.

[37] H. Cohen, "The BuDDy Library and Boolean Expressions." http://www.drdobbs.com/the-buddy-library-boolean-expressions/184401847. Checked: 17/03/2017.

[38] K. J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson, "A Practical Tutorial on Modified Condition / Decision Coverage," technical report, National Aeronautics and Space Administration (NASA), May 2001.

[39] D. Benavides, Sergio Segura, and Antonio Ruiz Cortés, "Automated Analysis of Feature Models 20 Years Later: A Literature Review," technical report, Department of Languages and Computer Systems, University of Seville, Feb 2010.

[40] "Git repository of the rbdd project." https://gitlab.com/rbdd-package/rbdd-package. Checked: 22/04/2017.

[41] "CAL BDD." https://embedded.eecs.berkeley.edu/Research/cal_bdd/. Checked: 22/04/2017.

[42] "CRAN Repository Policy." https://cran.r-project.org/web/packages/policies.html. Checked: 22/04/2017.

# List of Acronyms

**API** Application Programming Interface.

**BDD** Binary Decision Diagram.

**CNF** Conjunctive Normal Form.
**CRAN** Comprehensive R Archive Network.

**DAG** Directed Acyclic Graph.
**DARE** Domain Analysis and Reuse Environment.
**DCC** Decision Condition Coverage.

**FODA** Feature-Oriented Domain Analysis.

**GNU** GNU's Not Unix.
**GPS** Global Positioning System.

**IEEE** Institute of Electrical and Electronics Engineers.

**MC/DC** Modified Condition / Decision Coverage.
**MCC** Multiple Condition Coverage.

**OBDD** Ordered Binary Decision Diagram.

**ROBDD** Reduced Ordered Binary Decision Diagram.

**SAT** boolean SATisfiability problem.
**SPL** Software Product Line.

**UNED** Universidad Nacional de Educación a Distancia.