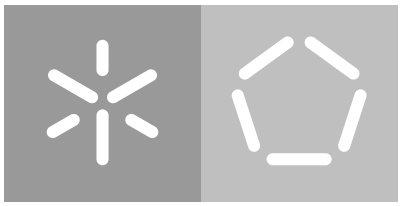


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Eduardo José Dias Pessoa

Parallel verification of Dynamic Systems with Rich Configurations



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Eduardo José Dias Pessoa

Parallel verification of Dynamic Systems with Rich Configurations

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Professor Doutor Nuno Filipe Moreira Macedo
Professor Doutor Manuel Alcino Pereira da Cunha

ACKNOWLEDGEMENTS

This work would never be possible without the unconditional support of my family and my girlfriend Sara. I am deeply grateful to them. A special thanks to Sara for the constant motivation during all the phases of my work.

Secondly, I would like to express my gratitude to my supervisor Professor Nuno Filipe Moreira Macedo and co-supervisor Professor Manuel Alcino Pereira da Cunha for their support, guidance and motivation on this dissertation.

Finally, I want express the my gratitude to all of friends and colleagues for their continuous support, motivation and fellowship during this journey.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.









ABSTRACT

Model checking is a technique used to automatically verify a model which represents the specification of some system. To ensure the correctness of the system the verification of both static and dynamic properties is often needed.

The specification of a system is made through modeling languages, while the respective verification is made by its model-checker. Most modeling frameworks are not ready to verify models rich in both kind of properties thereby limiting the verification of dynamic systems with rich configurations.

Electrum is a modeling language which mixes the best of the Alloy and TLA specification languages, with the capability of handling the problem mentioned above. This language is supported by two model-checking techniques – one bounded and one unbounded.

Nonetheless, the Electrum's bounded model-checker has limitations, thus, this dissertation aims to overcome them with the purpose of improving the analysis procedure of Electrum models, in particular, the definition of a new Electrum's semantics through a translation into Kodkod as well as the creation of a novel procedure of verifying Electrum models in parallel. Hence, in order to achieve these goals, a temporal extension to the Kodkod constraint solver was implemented.

Keywords: Alloy, Electrum, TLA, Formal Specification Language, Dynamic Systems, Rich Configurations, Model-Checking, Parallel Verification

RESUMO

Model-checking é uma técnica utilizada para verificar automaticamente modelos que representam uma determinada especificação de um sistema. Para assegurar a correcção de um sistema, é necessário verificar dois tipos de propriedades – dinâmicas e estáticas.

A especificação de um sistema é realizada através das linguagens de modelação, enquanto que a verificação do mesmo é feita pelo seu respectivo *model-checker*. A maior parte das *frameworks* de modelação não se encontram preparadas para verificar os dois tipos de propriedades, limitando assim a verificação de sistemas dinâmicos com configurações ricas.

O *Electrum* é uma linguagem de modelação que conjuga o melhor de Alloy e TLA, com a capacidade de contornar o problema acima enunciado. Esta linguagem tem associadas duas técnicas de *model-checking*, uma *bounded* e outra *unbounded*.

No entanto, o *bounded model-checker* do Electrum apresenta limitações, portanto, esta dissertação tem como objectivo resolver tais limitações a fim de melhorar o procedimento de análise dos modelos Electrum. Concretamente, a definição de uma nova semântica para o Electrum através de uma tradução para Kodkod, bem como a criação de um novo processo de verificação de modelos Electrum em paralelo. Consequentemente, estes dois últimos objectivos são alcançados através da construção de uma extensão temporal para o Kodkod.

Palavras-chave: Alloy, Electrum, TLA, Linguagem de Especificação Formal, Sistemas Dinâmicos, Configurações Ricas, Verificação de Modelos, Verificação Paralela

CONTENTS

Contents iii
1 INTRODUCTION 3
2 BACKGROUND 6
2.1 Alloy . . . 6
2.1.1 Alloy Language . . . 7
2.1.2 An Alloy Example . . . 8
2.1.3 Alloy Analyzer . . . 12
2.1.4 Alternatives to Dynamic Alloy Modeling . . . 14
2.2 TLA . . . 15
2.2.1 TLA$^+$ Language . . . 16
2.2.2 A TLA Example . . . 18
2.3 Kodkod . . . 19
2.3.1 Kodkod Language . . . 20
2.3.2 Translation to Propositional Logic . . . 20
2.3.3 Translation from Alloy to Kodkod . . . 23
2.4 Linear Temporal Logic . . . 25
2.4.1 LTL Operators . . . 26
2.4.2 LTL Bounded Model Checking . . . 27
2.4.3 Negation Normal Form . . . 28
2.4.4 Embedding of Temporal Formulas in Alloy . . . 29
2.5 SAT Solving Overview . . . 30
2.5.1 Conjunctive Normal Form . . . 31
2.5.2 DPLL SAT Solvers . . . 31
2.5.3 DPLL Optimizations . . . 32
3 STATE OF THE ART 34
3.1 SAT Solving Parallelization Strategies . . . 34
3.1.1 Cooperative Solving . . . 35
3.1.2 Competitive Solving . . . 38
3.1.3 Collaborative Solving . . . 38
3.1.4 Parallel SAT Solvers . . . 39
3.2 Verification of SPLs using Dynamic Algorithms . . . 42
3.2.1 fSMV Language and its Model-Checker . . . 43
3.2.2 Results Analysis . . . 45

3.3 Parallel Bounded Verification . . . 45
3.3.1 TranScoping Technique . . . 45
3.3.2 Ranger Technique . . . 48
4 KODKOD TEMPORAL EXTENSION 52
4.1 Temporal Kodkod Language . . . 53
4.2 Architecture . . . 53
4.3 Bounds Expansion . . . 55
4.4 Negation Normal Form Transformation . . . 56
4.5 Temporal Operators Translation . . . 58
4.6 Slicing Procedure . . . 61
5 ELECTRUM BOUNDED MODEL-CHECKER 63
5.1 Electrum Specification Framework . . . 64
5.1.1 Electrum Language . . . 64
5.1.2 An Electrum Example . . . 66
5.1.3 First Electrum Bounded Model-Checker . . . 67
5.1.4 New Electrum Bounded Model-Checker . . . 68
5.2 Translation Semantics from Electrum to Kodkod . . . 70
5.2.1 Signatures and Fields . . . 71
5.2.2 Temporal Operators . . . 74
5.3 Solution Renaming . . . 75
5.4 Visualizer . . . 77
5.5 Evaluator . . . 79
6 PARALLELIZATION STRATEGY 81
6.1 Decomposed Model Finding . . . 82
6.2 Decomposed Kodkod . . . 83
6.3 Decomposition Following an Example . . . 84
6.4 Empirical Evaluation . . . 86
7 CONCLUSIONS AND FUTURE WORK 90

LIST OF FIGURES

Figure 1 Alloy's syntax . . . 7
Figure 2 Static hotel room system example under Alloy . . . 9
Figure 3 Dynamic hotel room system example under Alloy . . . 11
Figure 4 Hotel room system counter-example under Alloy . . . 12
Figure 5 Hotel room system example under TLA . . . 19
Figure 6 Kodkod's syntax . . . 21
Figure 7 Kodkod architecture . . . 22
Figure 8 The two possible cases for a bounded path . . . 27
Figure 9 NNF rules . . . 29
Figure 10 Temporal operators expansion . . . 30
Figure 11 DPLL algorithm taken from (van Harmelen et al., 2008) . . . 32
Figure 12 Search space splitting example using guiding paths taken from (Martins et al., 2012) . . . 36
Figure 13 JaCk-SAT algorithm taken from (Singer and Monnet, 2007) . . . 41
Figure 14 SMV model taken from (Classen et al., 2014) . . . 43
Figure 15 An excerpt of an Alloy model to illustrate the tranScoping technique . . . 48
Figure 16 An excerpt of an Alloy model to modeling binary trees . . . 50
Figure 17 Graphical representation of a vecSpec taken from (Rosner et al., 2013b) . . . 50
Figure 18 Kodkod temporal extension architecture . . . 54
Figure 19 Excerpt of the NNF algorithm . . . 57
Figure 20 Temporal translation algorithm . . . 58
Figure 21 Slicing strategy . . . 61
Figure 22 Electrum's syntax . . . 65
Figure 23 Hotel room system under Electrum . . . 66
Figure 24 Electrum's BMC architecture . . . 70
Figure 25 Electrum model example . . . 71
Figure 26 Electrum temporal operators' translation . . . 74
Figure 27 Alloy model example . . . 75
Figure 28 Hotel room system counter-example under Electrum . . . 78
Figure 29 $P\downarrow$ of the hotel room system example . . . 84

LIST OF TABLES

Table 1 LTL operators 26
Table 2 Parameters retrieved from the splitting analysis 48
Table 3 Naming table produced by Kodkod 49
Table 4 Satisfiable models evaluation 87
Table 5 Unsatisfiable models evaluation 89

LIST OF LISTINGS

2.1 XML with an Alloy instance . . . 13
2.2 Logic . . . 16
2.3 Sets . . . 17
2.4 Functions . . . 17
2.5 Records . . . 17
2.6 Tuples . . . 17
2.7 Temporal operators . . . 17
2.8 Relations . . . 24
2.9 Bounds . . . 24
2.10 Formula . . . 25

4.1 `TimeConstraints` relations . . . 54
4.2 `TimeConstraints` formulas . . . 55

5.1 Original bounds following Alloy . . . 75
5.2 Solution returned by Kodkod in Alloy . . . 75
5.3 Original bounds following Electrum . . . 76
5.4 Solution returned by Kodkod in Electrum . . . 76

6.1 First configuration . . . 85
6.2 Second configuration . . . 85
6.3 The bounds of an integrated problem P_i . . . 85

1

INTRODUCTION

Software specification and verification have been increasingly used in software engineering, since they allow the developer to identify and fix possible and unexpected errors during software development. To achieve this goal, the analysis of two property classes is required – *static* and *dynamic*. Static properties are usually expressed through first-order logic, while the dynamic ones are expressed in some kind of temporal logic. The verification of both kind of properties is crucial to establish the correctness of the modeled system. However, there are few modeling languages which, accompanied by their model-checkers, can efficiently combine the analysis of these kind of properties simultaneously.

In most systems there is a notion of *configuration*, components of the state system which are initially arbitrary but remain unchanged as the system evolves, defined by the structural properties. Following that, the frameworks which excel only in analyzing either the static or the dynamic properties, are limited in the verification of dynamic systems with rich configurations – systems whose state space is characterized by rich structural properties whose evolution must satisfy certain temporal properties. Hence, to handle this kind of systems, modeling frameworks must satisfy certain requirements that include a clear distinction between the system configuration and the system evolution, and the capability of constraining configurations by rich structural properties. Distributed algorithms are an example of such systems, since the network topologies on which the algorithm is expected to execute are defined through structural properties and the evolution of its state through behavioral properties. However, there exists a reduced quantity of modeling languages capable of satisfying these requirements, since it is imperative that they be rich, flexible and expressive enough to define both static and dynamic properties.

Alloy (Jackson, 2006) and *TLA* (Temporal Logic of Actions) (Lamport, 1994) are two of the most popular specification languages nowadays – the former is a formal specification language based on first-order logic that provides automatic verification through its Analyzer, while the latter is a formal specification language that combines temporal logic with a logic of actions that provides automatic verification through the TLC model-checker. Nonetheless, these specification languages present some limitations – Alloy is not well-suited to verify temporal properties, whereas TLA lacks a type system with inheritance, among other features. To overcome these limitations the *Electrum* (Macedo et al., 2016) specification language was proposed. This specification language combines the best of Alloy, due to its structural concepts, and TLA, since it is capable of freely defining action predicates with primed variables, but also temporal properties.

For the purpose of allowing the automatic verification of temporal properties, this language is accompanied by two model-checking techniques – one unbounded and one bounded. The former is built over the nuXmv (Cavada et al., 2014) model-checker – a new symbolic model-checker (it examines a set of states for each step) for the analysis of synchronous finite- and infinite-state systems. The latter is built over Alloy's tool, Alloy Analyzer, which allows the user to generate all possible instances of a model and to iterate over each one. This verification process is achieved, at low level, by Kodkod (Torlak and Jackson, 2007) – a relational model finder that invokes off-the-shelf SAT solvers to try to find some solution, based on a set of boolean variables previously translated from the Alloy model.

In the first version of the Electrum's bounded model-checking tool, its models were expanded into standard Alloy, which would then be translated into Kodkod by the Analyzer. Such translation into Alloy was required since the Kodkod does not support the temporal component of Electrum, which leads to the creation of explicit *time* components at Alloy level. The constraints generated by the translation from Alloy to Kodkod are not capable of restricting the temporal signatures or fields, which entails the generation of several additional constraints. Considering that, the direct translation from Electrum into Kodkod would avoid a layer of complexity on which nothing relevant, for the proper functioning, is presented. This way, the direct translation from Electrum into Kodkod is one of this dissertation's goals, thereby avoiding the translation into Alloy as middle step and would also clarify the semantics of the language. Besides, with the purpose of supporting the temporal component of Electrum (variable signatures/fields, temporal operators) at Kodkod level, a temporal extension to the Kodkod constraint solver that support these components, is proposed as another of this dissertation's goals. Obviously, before solving a specific model, this extension translates the temporal components of a certain temporal Kodkod model into the standard Kodkod first-order logic (FOL), following well-known translation rules (Linear Temporal Logic (LTL) into FOL).

Besides the aforementioned goals, this dissertation also aims to achieve a novel verification procedure for Electrum models. More precisely, the novel parallelization strategy proposed in this work, first of all, starts by splitting the dynamic from the static properties in the model specification. After that, from the static component the configurations are generated, which are then integrated into the temporal problem and launched in parallel. Note that, the splitting of the static component into smaller parts and the respective generation of configurations, as well as their integration into the temporal component is only possible since the configurations remain fixed during this procedure. Such strategy is implemented at Kodkod level and the temporal extension has a fundamental role in this strategy, since it is responsible for splitting the dynamic from the static properties of the Kodkod models resulting from the translation of Electrum models.

In order to achieve these goals, it is necessary to specify and achieve particular goals. This way, concretely, this project is expected to deliver the following contributions:

- a survey of the existing Alloy extensions that are able to handle temporal models and the parallelization techniques, as well as parallel SAT solvers and parallel bounded model-checking techniques

- an implementation of a temporal extension of the Kodkod constraint solver that translates the temporal component into the standard Kodkod first-order logic and that implements the automated splitting of the model to apply the parallelization strategy
- a formulation of the Electrum's structural semantics (through the direct translation into temporal Kodkod)
- an implementation of the novel parallelization strategy
- an implementation of the bounded model-checker tool for Electrum using the Kodkod temporal extension and an adaptation of the Electrum Visualizer for showing the temporal models
- an evaluation of the tool by comparing the performance with the existing sequential procedure

The remainder of this dissertation is divided in several chapters with the purpose of explaining the goals presented above. The first one is the Background (Chapter 2) where the basic concepts related to the dissertation's goal are explored. In the next one, the State of Art (Chapter 3), both the existing techniques of parallelising SAT-solvers and the techniques to perform the verification of Alloy models in parallel are discussed. After that, the remainder dissertation is divided into three main chapters, which are basically the three main contributions of this Master's Thesis. In Chapter 4 the Kodkod temporal extension is approached. Besides, the automatic splitting of a specific Electrum model with the purpose of applying the new parallelization strategy is also presented. The Electrum's bounded model-checker framework is handled in Chapter 5, where an overview of the Electrum language is performed as well as a comparison between the two model-checking techniques. Besides, it shows how the Electrum tool was adapted to the new back-end. Moreover, the new parallelization strategy is handled in Chapter 6. Finally, the conclusions and future work are explained in Chapter 7.

2

BACKGROUND

This chapter serves as a warm up for the remainder of the dissertation. It will introduce basic concepts necessary to understand the document. As the Electrum language is based on Alloy and TLA, these two languages will be introduced in this chapter.

Moreover, an introduction to Kodkod is necessary since the creation of a temporal extension to Kodkod is one of the thesis's goals. Both the syntax and the semantic of the operators of such temporal extension follow the standard Linear Temporal Logic (LTL), thus, it is approached on this chapter as well. Since this dissertation aims to explore new model-checking techniques for Kodkod, it is also relevant to have an insight on how the underlying SAT solvers operate. Furthermore, the Alloy models are translated into Kodkod, in which the Kodkod problems are translated into propositional logic and then solved by SAT solver – these translations also motivate the structure of this chapter.

The remainder chapter is composed as follows: the first section (Section 2.1) approaches the Alloy specification language, the second one (Section 2.2) explores the TLA specification language, the third one (Section 2.3) covers the Kodkod constraint solver, the fourth one (Section 2.4) describes the LTL, and finally, the last one (Section 2.5) presents a SAT solving overview.

2.1 ALLOY

Alloy [1] (Jackson, 2006) is a formal modeling language based on first-order logic, accompanied by an Analyzer to validate and verify the model specification and its constraints. Besides that, the Alloy Analyzer provides some important features to support the user in designing the models and checking properties about them, such as: it allows the checking of assertions about the model's specification; it provides a visualizer capable of graphically displaying instances of the model; it is capable of simulating the execution of operations.

The Alloy Analyzer is a self-contained executable. Furthermore, it can easily be embedded into other applications, since it provides an API and the source code.

1 Available at http://alloy.mit.edu/alloy/download.html

2.1.1 *Alloy Language*

Figure 1 illustrates an Alloy specification (`alloySpec`), that is comprised by an optional module declaration (`moduleDecl`), a set of `imports` and `specifications`. A module declaration (`moduleDecl`) is the relative path-name where the model specification is defined, which in turn can `import` other modules. The `specification` can either be a signature declaration , a fact, a predicate, a function, an assertion or a command.

```
alloySpec ::= [moduleDecl] import* specification*
moduleDecl ::= module qualName [[name,+]]
import ::= open qualName [[qualName,+]] [as name]
specification ::= sigDecl | Constraints | assertDecl | cmdDecl
Constraints ::= factDecl | predDecl | funDecl
sigDecl ::= [abstract] [mult] sig name,+ [sigExt] {field,*}[block]
sigExt ::= extends qualName | in qualName [+ qualName]*
mult ::= lone | some | one
field ::= [disj] name,+ : [disj] expr
factDecl ::= fact [name] block
predDecl ::= pred [qualName .] name [paraDecls] block
funDecl ::= fun [qualName.] name [paraDecls] : expr { expr }
paraDecls ::= (field,*) | [field,*]
assertDecl ::= assert [name] block
cmdDecl ::= [name :] [run | check] [qualName | block] [scope]
scope ::= for number [but typescope,+] | for typescope,+
typescope ::= [exactly] number qualName
expr ::= const | qualName | @name | this
| unOp expr | expr binOp expr | expr arrowOp expr
| expr [ expr,* ]
| expr [! | not] compareOp expr
| expr (⟹ | implies) expr else expr
| let letDecl,+ blockOrBar
| quant field,+ blockOrBar
| { field,+ blockOrBar }
| ( expr ) | block
const ::= [-] number | none | univ | iden
unOp ::= ! | not | no | mult | set | # | ∽ | * | ^
binOp ::= ∨ | or | ∧ | and | ⟺ | iff | ⟹ | implies | &| + | - | ++ | <: | :> | .
arrowOp ::= [mult | set ] -> [mult | set ]
compareOp ::= in | = | < | > | =< | >=
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | bar expr
bar ::= |
quant ::= all | no | sum | mult
qualName ::= [this/ ] (name /)* name
```

Figure 1: Alloy's syntax

A signature declaration (`sigDecl`) represents a set of atoms, whereas an atom is an indivisible, immutable and uninterpreted entity. Signature inheritance can be introduced by the keyword `in` (subset signature) and the keyword `extends` (type signature). However, a type signature introduces a type or a subtype. Concretely, if a signature extends another, it is said to be sub-signature of the signature it extends. Another important feature about signature declaration is the keyword `abstract`, which means that a signature has no elements, except those belonging to its extensions. Inside the signature

it is possible to declare a set of `fields`, which represent sets of atom tuples, and its `name` is the relation between the signature and its `expr`.

`Constraints` are comprised by `facts`, `predicates` and `functions`. The first one, facts, represent the system's invariants (model constraints), which any instance of model must satisfy. Predicates are named constraints that define a formula (true or false). However, they hold when invoked inside a fact, whereas facts always hold. Finally, functions, consist in expressions capable of returning a relation, a set or an atom. For the purpose of expressing constraints over the model's specification, Alloy provides relational operators denoted by `unOp, binOp, arrowOp` and `compareOp`.

Considering the Alloy formulas, they are composed essentially by first order logic with relational operators and transitive closure. This way, one of the most important operator is the non-reflexive transitive closure, denoted by the symbol "^" – this operator applied to a certain binary relation R is the smallest transitive relation that contains R. Besides, another important is the reflexive-transitive closure, denoted by "$*$", in which, this operator applied to a certain binary relation R is the smallest relation that is transitive and reflexive and contains R. However, the expression `expr` are built from relation and relational operators. One of the most common relational operator used is the join, denoted by "." – this operator represents the composition of relations. Following that, the Alloy formulas are basically achieved recurring to `expr` with the typical first-order logic operators. Moreover, it also provides both quantification and multiplicity expressions, denoted by `quant` and `mult` respectively. considering the latter, they are used to constrain the cardinality of a certain relation. On the other hand, `assertions` express properties of the model that can be checked by the Analyzer. An assertion is checked by Alloy Analyzer to be true for all the examples in specified a scope.

Finally, the last components of `specification` are the `commands`. Alloy provides two kinds of commands – `run` and `check`. The run command provides instructions to Alloy Analyzer to return instances that satisfy the facts in the model, the `block` inside the run and the optional `scope` – the scopes bound the maximum number of objects of each type. The check command instructs the Alloy Analyzer to find counter-examples that do not satisfy the formula being checked or the assertions aforementioned.

Due to all above referred, Alloy is a first-order relational logic and its syntax is well-suited for structuring specifications in the logic.

2.1.2 *An Alloy Example*

To better understand the Alloy language, an example taken from (Jackson, 2006) will be presented and explained. However, as Alloy is more appropriated to modeling static systems, but, first of all, the static version of the previous problem will be explained. Thereafter, considering the same problem, it is presented and explained the dynamic version, in which the type *time* is introduced. Figure 2 illustrates the static version of the Hotel room system example. This system represents a card key system, used to lock and unlock guest rooms with recordable locks preventing, this way, previous

guests from entering a room, once its lock was assigned to other occupant, who has arrived after. To some next occupant a new key is assigned by the front desk, although the the lock is only recoded when the occupant uses the key to open the room's door, causing the expiration of the previous keys. There is no communication between the front desk and the locking systems, which are stand-alone. Nevertheless, the system works correctly, since they use the same pseudo-random generator, and are initially synchronized.

```
open util/ordering[Key] as ko

sig Key {}
sig Room {
  keys: set Key,
  currentKey: keys
}

fact DisjointKeySets {
  Room<:keys   in   Room lone-> Key
}

one sig FrontDesk {
  lastKey: Room -> lone Key,
  occupant: Room -> Guest
}

sig Guest {
  keys: Key
  }
```

```
fun nextKey [k: Key, ks: set Key]: set Key {
  min [k.nexts & ks]
  }
pred entry [ g: Guest, r: Room, k: Key] {
   k in g.keys
   let ck = r.currentKey |
       k = ck or
       (k = nextKey[ck, r.keys] and ck = k)
  }

pred checkin [g: Guest, r: Room, k: Key] {}
pred checkin [g: Guest] {}

assert NoBadEntry {
  all r: Room, g: Guest, k: Key |
    let o = FrontDesk.occupant[r] |
      entry [g, r, k] and some o ⟹ g in o
}

check NoBadEntry for 3 but 2 Room, 2 Guest
```

Figure 2: Static hotel room system example under Alloy

The model is specified by declaring the following *signatures* - `Key`, `Room`, `FrontDesk` and `Guest`. The `Room` signature is composed by two fields: `keys` and `currentKey`. The first one represents each room's set of `keys` and the second one ensures that each room has exactly only one `currentKey`. To guarantee that each key belongs to at most one (*lone*) room, a *fact* (`DisjointKeySets`) is created. Signature `Frontdesk` has multiplicity `one`, which means that there is one front desk, and two fields – `lastKey` and `occupant`. Field `lastKey` establishes the mapping from the front desk to each room and its most recent key combination, whereas the `occupant`, to each room and its current guest. `Guest` is the last declared signature and makes the mapping between itself and its keys.

Functions are used as model queries, being possible to reuse them in any place of the model. The function `nextKey` returns the smallest key in `ks` that follows `k`, given a key `k` and a set of keys `ks`. Predicate `entry` corresponds to the entrance of a guest in the room. Relative to this `entry` predicate, there is a pre-condition (`k` **`in`** `g.keys`) in which the guest must have the key that was used to open the lock. In this case, the post-conditions are also composed by pre-conditions to check whether or not the current key (**`let`** `ck = r.currentKey`) matches to its lock – the key on the

card either matches the lock's current key (`k = ck`) or matches its successor (`k = nextKey[ck, r.keys]`) and the lock is advanced (`ck = k`).

According to the hotel's model dimension, only the most important operations were included as depicted in Figure 2. Assertion `NoBadEntry` verifies whether there are unauthorized room entries, therefore, if a guest `g` enters room `r` and `r` is occupied (information given by the front desk), the room `g` will be a recorded occupant of `r`. Command **`check`** instructs the Analyzer to analyse the assertion `NoBadEntry`, providing the respective scope. The property presented on the assertion is invalid, since the Analyzer yield a counter-example.

As previously shown, with Alloy, static systems can be easily modeled. However, although Alloy allows modeling dynamic systems, such procedure is not easy since it is required to use "tricks" to simulate the evolution over the time. Figure 3 is the dynamic version of the first example here explained. Dynamic systems are based on the introduction of *states* in the relations and operations (*predicates*) that specify the relationship between pre- and post-states of the model. The two most common ways to specify dynamic systems in Alloy, are expressed by using *global* and *local* state *idioms*. The first one requires that all dynamic fields will be grouped in the same signature, being the *state* the domain in all dynamic relations. Whereas the declaration of the second one is made locally, adding an extra relation in the mutable fields, being the co-domain of relation. The Alloy example depicted in Figure 3 is presented according the local state idiom.

This dynamic version has one more signature `Time`, and the mutable fields (the ones that evolve in time) include it as range of them. Following that, the fields that evolve in time are: `currentKey`, `lastKey`, `occupant` and `keys`. Hence, for instance, the field `lastKey` establishes the mapping from the front desk to each room and its most recent key combination, whereas the `occupant`, to each room and its current guest at any time and the other fields follow the same idea.

Dynamic systems need operations to change its *state*, namely, the fact `traces` imposes the actions that are applied as the time evolves. However, since the first state must be defined, the initial state is defined invoking the `init` predicate for the first state – `init[first]`. This way, this predicate ensures that, initially, guests have no keys, the front desk has no rooms occupied and for all rooms its key at the front desk and the current combination of the lock itself are synchronized. However, to ensure this specifications it is required to apply relational operators between relations. For instance, in this predicate, the expression **`no`** `Guest.keys.t` uses the unary operator **`no`** and the relational composition operator (denoted by ".") – denotes the value of the relation `keys` to all guests (`Guest`) at instant `t`. Furthermore, with the purpose of imposing the actions over the trace, the fact ensures that there is some either `entry`, `checkin` or `checkout` for all states but the last one.

Operations are denoted by the well-known predicates. Such operations, in dynamic systems, often receive as input two special parameters – `t` and `t'` denoting respectively, the pre- and the post- state. An operation is composed of three parts: pre-, post- and frame-conditions. Pre-conditions denote the required conditions at instant `t` to proceed to the post-conditions, whereas the post-condition provides the state of the system at instant `t'`. Frame-conditions define parts of the system that remain

```
open util/ordering[Time] as to
open util/ordering[Key] as ko

sig Key {}
sig Time {}
sig Room {
    keys: set Key,
    currentKey: keys one -> Time
 }

fact DisjointKeySets {
    Room<:keys in Room lone-> Key
}

one sig FrontDesk {
    lastKey: (Room -> lone Key) -> Time,
    occupant: (Room -> Guest) -> Time
}
sig Guest {
    keys: Key -> Time
}


fact traces {
    init [first]
    all t: Time-last | let t' = t.next |
    some g: Guest, r: Room, k: Key |
      entry [t, t', g, r, k]
      or checkin [t, t', g, r, k]
      or checkout [t, t', g]
}


pred init [t: Time] {
    no Guest.keys.t
    no FrontDesk.occupant.t
    all r: Room |
    FrontDesk.lastKey.t [r] = r.currentKey.t
}
```

```
fun nextKey [k: Key, ks: set Key]: set Key {
    min [k.nexts & ks]
}

pred entry [t, t': Time, g: Guest, r: Room, k
    : Key] {
    k in g.keys.t
    let ck = r.currentKey |
    (k = ck.t and ck.t' = ck.t) or
    (k = nextKey[ck.t, r.keys] and ck.t' = k)
    noRoomChangeExcept [t, t', r]
    noGuestChangeExcept [t, t', none]
    noFrontDeskChange [t, t']
}

pred noFrontDeskChange [t,t':Time]{..}
pred noRoomChangeExcept [t,t':Time,rs:set
    Room]{..}
pred noGuestChangeExcept [t,t':Time,gs:set
    Guest] {..}
pred checkout [t,t':Time,g:Guest]{..}
pred checkin [t, t': Time, g: Guest, r: Room,
     k: Key] {..}
pred NoIntervening {..}


assert NoBadEntry {
    all t: Time, r: Room, g: Guest, k: Key |
    let t' = t.next,
    o = FrontDesk.occupant.t[r] |
    entry [t, t', g, r, k] and
    some o ⟹ g in o
}

check NoBadEntry for 3 but 2 Room, 2 Guest,
    10 Time
```

Figure 3: Dynamic hotel room system example under Alloy

unchanged along the state transition. For instance, relative to the `entry` predicate, there is a pre-condition (`k in g.keys.t`) in which the guest must have the key that was used to open the lock. In this case, the post-conditions are also composed by pre-conditions to check whether or not the current key (`let ck = r.currentKey`) matches to its lock – the key on the card either matches the lock's current key (`k = ck.t`) and the lock remains unchanged (`ck.t' = ck.t`) or matches its successor (`k = nextKey[ck.t, r.keys]`) and the lock is advanced (`ck.t' = k`). The last three conditions represent the frame-conditions.

Following the same idea, the assertion `NoBadEntry` verifies whether there are unauthorized room entries, therefore, if a guest `g` enters room `r` at time `t` and `r` is occupied (information given by the front desk), the room `g` will be a recorded occupant of `r`. Like the static version of this problem, the assertion `NoBadEntry` is invalid since the Analyzer yield a counter-example. However, in this case, the scope for the `Time` must be defined, more precisely, the number of states must be defined.

These two examples explained shows that Alloy is more appropriated to design static models rather than dynamic ones. Hence, such fact is one of the purposes of using the Electrum language to design dynamic models.

2.1.3 *Alloy Analyzer*

The Alloy Analyzer provides a Visualizer capable of showing counter-examples, for the user, that broke the specified properties on the model. Following that, the user has a wide perception of what may be the problems of the specification. Moreover, as the Analyzer not only generates a single instance, the user can iterate over all possible instances. Besides that, the user is able to customize the Visualizer's theme and this way, it can adapt the visualization as it wants. In this procedure, it is possible to assign several colors to nodes (signatures) as well as to the edges – which are the binding between the nodes (fields). In particular, for the purpose of a thorough analysis, the user can also change the color of some specific signature or field, as well as its font size. Despite of showing instances, the Alloy Analyzer provides an evaluator. This feature allows the user to specify predicates over the current instance on the Visualizer.

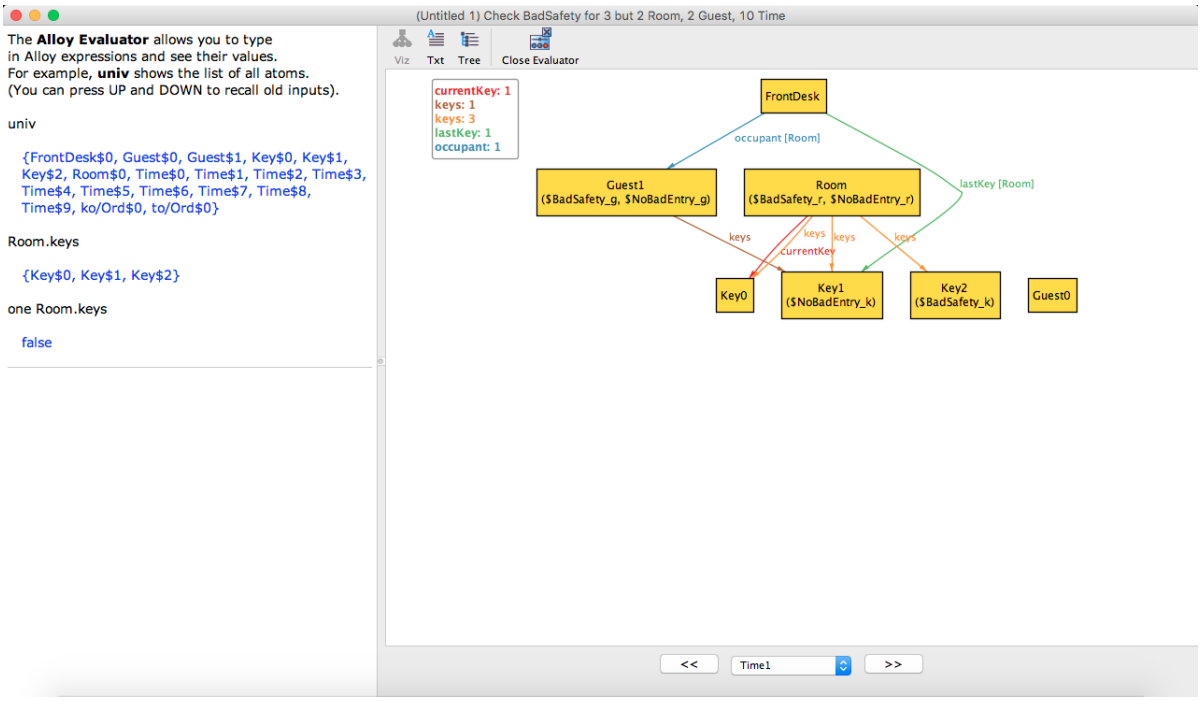


Figure 4: Hotel room system counter-example under Alloy

Figure 4 shows a counter-example resulting from the command **`check`** `NoBadEntry` of the hotel room system example, defined in Figure 3 under Alloy, but it also depicts the evaluator with some basic predicates over the counter-example. Sometimes, the analysing of a certain counter-example might be a tough task due to the quantity of signatures and the relations between them, namely, for

instance, in temporal models. To address this issue, a feature called projection is provided. Such feature allows the user to project over a certain signature with the purpose of reducing the arity of the drawn relations that contain such signature. Moreover, it is also possible to project over more than one signature at the same time. When the projection is performed, a combo-box is created on the bottom of the screen, enabling the user to project over all atoms belonging to the signature projected.

As previously referred, for temporal examples, the projection allows the user to iterate over the times present in the problem and, this way, showing as the system evolves over the time.

The Alloy's Visualizer shows its counter-examples by reading a xml file with a certain solution. When a certain model specification is verified, in of existing a certain instance to depict, the Analyzer serializes it into a xml file in order to the Visualizer read it.

```
<alloy>
<instance command="Check BadSafety for 3 but 2 Room, 2 Guest, 10 Time">

<sig label="univ" ID="2" builtin="yes"></sig>

<sig label="Key" ID="4" parentID="2">
   <atom label="Key$0"/> ... <atom label="Key$2"/>
</sig>

<sig label="Time" ID="5" parentID="2">
   <atom label="Time$0"/> ... <atom label="Time$9"/>
</sig>

<sig label="Room" ID="6" parentID="2">
   <atom label="Room$0"/>
</sig>

<field label="currentKey" ID="8" parentID="6">
   <tuple> <atom label="Room$0"/> <atom label="Key$0"/> <atom label="Time$0"/>
   </tuple>
   <tuple> ... </tuple>
   <types> <type ID="6"/> <type ID="4"/> <type ID="5"/> </types>
</field>
</instance>
</alloy>
```

Listing 2.1: XML with an Alloy instance

Listing 2.1 depicts an excerpt of the Alloy's solution of the hotel room system example (Figure 4). The instance's root is the element *alloy* and its child is the element *instance*, in which, the problem's atoms are presented. Inside of an instance, three kind of elements can be defined – *sig*, *fields* or *skolems*. Each *sig* element, that represents the typically signature, is comprised of the signature's name (*label*), an identifier (*ID*) and its parent's identifier (*parendID*). For instance, the signature *key*

is a child of *univ* ($ID = 2$), thus, its parent's identifier is 2. As every signature is a set of atoms, each *sig* is filled with *atom* elements.

Fields are represented by the $field$ element. As every field is composed by tuples, the $field$ element is composed by $tuple$ elements, in which, they are filled with *atom* elements.

Moreover, with the purpose of ensuring that, in a certain xml reading, each field typed properly, the last line of the $field$ represents the identifiers of the signatures that compose it. More precisely, as the field *currentKey* is typed as follows $Room(id = 6) \rightarrow Key(id = 4) \rightarrow Time(id = 5)$, its *types* element is comprised as follows: $<types><typeID = "6"/><typeID = "4"/><typeID = "5"/></types>$.

Every time a user requires some next instance, if there is any, the same procedure is performed for the new instance by rewriting the xml file.

2.1.4 *Alternatives to Dynamic Alloy Modeling*

Alloy is more suitable to static modeling, due to its nature. Alloy's language is simple and flexible, allowing the verification of a dynamic system by implementing the local or the global state idiom. Nonetheless, these approaches lead the developer to focus on the idiom design (which is too verbose) rather than on the conception of behavioral properties. To address this problem several extensions of alloy have been proposed, such as DynAlloy (Frias et al., 2005), Imperative Alloy (Near and Jackson, 2010) and Electrum (Macedo et al., 2016).

DynAlloy proposes an extension to the Alloy specification language and its syntax, as well as an extension to Alloy's tool. DynAlloy aims to describe dynamic properties of systems using actions, allowing the specification of dynamic properties over execution traces using a style inspired on dynamic logic. The extension of Alloy's syntax is based on notation for partial correctness assertions, becoming easier to express properties about executions. Alloy's tool was also extended with the purpose of an efficient verification of DynAlloy specifications, including specifications regarding the execution traces assertions and actions.

Imperative Alloy proposes a simple extension to Alloy's semantic. The mutable fields (dynamic) have a keyword associated, `dynamic`, declared as follows – `sig A { field : dynamic (B) }`. Dynamic operations are expressed by actions, where operators for imperative programming are included such as: loop, field update and sequential composition. Pre- and post- conditions are defined by `before` and `after` actions, respectively. For a better property verification, temporal quantifiers were also included, where the existential operator is expressed by `sometimes` and the universal operator by `always`.

Electrum is a modeling language that combines the better from Alloy, due to the structural concept, and the better from TLA, due to the capacity to freely define predicates with primed variables. TLA (Lamport, 1994) (Temporal Logic of Actions) is a logic that combines temporal logic and actions. It is used to describe behaviours on concurrent systems. An action is a logic expression that contains

primed (the value of variable is on the next state) and unprimed variables (the value of variable is on the previous state). In this case, fields and signatures can be tagged as variable (`var sig A{} or sig A {var field : B}`), meaning that they are dynamic. Electrum also extends the semantic of its operators, namely – **always, after, eventually** and **until**. Furthermore, this language has two model-checking techniques associated.

Most specification languages and their model-checkers are designed to verify either structural or behavioral properties. Nonetheless, to guarantee the total correctness of the software design it is required to verify both of them. *Configurations* represent components of the system state, that are initially arbitrary but remain unchanged as the system evolves, thus, as a consequence, the verification of dynamic systems, with rich configurations, is limited. These systems have their state space characterized by rich structural properties, which means that the initial state does not have a concrete valuation. Furthermore, the system evolution must satisfy some temporal properties. More precisely, this kind of systems, have underlying four requirements: a clear distinction between the system configuration and the system evolution; configurations constrained by rich structural properties; a declarative specification of the system evolution and, finally, the necessity to verifying both properties – safety and liveness.

In fact, the Electrum language is well-suited to satisfy the aforementioned requirements. However, contrary to Electrum, both DynAlloy and Imperative Alloy extensions break the third requirement – a declarative specification of the system evolution – since they compromise the flexibility that the Alloy users are accustomed to. Hence, Electrum is more appropriated to handle dynamic systems with rich configurations rather than these two Alloy extensions.

2.2 TLA

TLA (Lamport, 2002), or Temporal Logic of Actions, is a logic that combines temporal logic with a logic of actions. However, TLA has become a shorthand for referring two languages – the TLA$^+$ specification language and the PlusCal (Lamport, 2009) algorithm language. The latter is based on the former, since a PlusCal algorithm is automatically translated to a TLA$^+$ specification. It is an algorithm language that can be used to replace pseudo-code for concurrent or sequential algorithms.

TLA$^+$ is well-suited for designing and checking dynamic systems since it combines the temporal logic with logic of actions, by recurring to a mathematical notation – even though it is specified in typeset mathematical symbols, in favor of an easier modeling procedure, it has a tool that allows the user typing LaTeX symbols, which represent their definitions in ASCII. It follows the idea that it should contain just the minimum possible beyond what is needed to formally express a system with simple mathematics and logic. Nonetheless, to support the temporal logic, it provides the well-known LTL (that will be presented in Sub-Section 2.4.1) operators. Furthermore, the logic of actions is related to the well-known variables of TLA – primed and non-primed variables. Thus, an action is an expression which contains primed and non-primed variables. The primed variables denote the

variable's value. in the next state and are denoted by the symbol ' (prime), for instance, x' is the value of x in the next state. On the other hand, the non-primed variables are the variable's value in the current state, thus, they do not have any symbol to identify them. Due to these two features, the verification of both safety and liveness properties of a certain specification is easily encoded and verified. Note that, the safety properties assert that nothing bad happens, whereas the liveness ones assert that something good eventually happens.

This specification language is accompanied by an integrated development environment[2] built on top of Eclipse. To assist the user on the development task, it includes an editor which identifies the syntax errors, but also shows them in a highlighting way. For ensuring the correctness of its specification models, it provides several back-end tools to achieve it such as: TLAPS, TLATeX, PlusCal and the most important, the TLC model-checker[3]. This last one creates a finite state model of TLA$^+$ for checking invariance properties. It searches all defined state transitions by recurring a breadth-first search after to generate a set of the initial states which satisfying the specification. In such search, if some state violates the invariant defined in the specification, TLC returns the path from the root until such state.

2.2.1 *TLA$^+$ Language*

A TLA$^+$ specification is organized into modules. As previously referred, the TLA$^+$ combines temporal logic with logic of action by recurring to a mathematical notation. This way, every model specification typically can be split in four parts: the initial predicate, the possible next states, and the safety and liveness properties. Besides, the specification usually starts defining its name by using the keyword **`MODULE`**. Moreover, on the declaration section is possible to declare variables by recurring to the **`VARIABLE`** keyword, the constants are declared by the keyword **`CONSTANT`**, the **`ASSUME`** asserts the declared expression as an assumption, and finally, it is possible incorporate other modules into the current one by using the **`EXTENDS`** or the **`INSTANCE`** statements.

The TLA$^+$'s syntax is extensive due to the constant, temporal and action operators, as well as the *miscellaneous operators*. The constant operators are comprised of Listing 2.2, Listing 2.3, Listing 2.4, Listing 2.5 and Listing 2.6. Despite of these, the syntax also provides the strings and the numbers operators.

The language provides, from the logic operators, first-order logic operators. Furthermore, the $CHOOSE$ operator is also provided and represents the *Hilbert's* ϵ operator – this operator just selects an arbitrary set element.

$\wedge \quad \vee \quad \neg \quad \Rightarrow \quad \equiv$
$TRUE \quad FALSE \quad BOOLEAN \quad [the\ set\ \{TRUE, FALSE\}]$
$\forall x : p \quad \exists x : p \quad \forall x \in S : p \quad \exists x \in S : p$
$CHOOSE\ x : p \quad [An\ x\ satisfying\ p] \quad CHOOSE\ x \in S : p \quad [An\ x\ in\ S\ satisfying\ p]$

2 Available at http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html
3 Available at http://research.microsoft.com/en-us/um/people/lamport/tla/tlc.html

Listing 2.2: Logic

An important part of this language are the data structures, which are comprised of Sets (Listing 2.3), Functions (Listing 2.4), Records (Listing 2.5) and Tuples (Listing 2.6). Since it supports sets, it provides the corresponding operators which are applied to them, such as the intersection, the union, the difference, the subset, among others that are depicted below (Listing 2.3). As depicted, the sets can be built in a enumerated way ($\{e_1, ..., e_n\}$), but they can also be constructed from other sets (the two other ones on the second line).

$= \neq \in \notin \cup \cap \subseteq \setminus$
$\{e_1, ..., e_n\}$ $\{x \in S : p\}$ $\{e : x \in S\}$
$SUBSET\ S$ $UNION\ S$

Listing 2.3: Sets

Functions in TLA$^+$ are basically an assignment of a value in the range with an element on their domain – their domain is a set. Note that, $DOMAIN\ f$ returns a set which is the respective domain of the function. Records are a particular function in TLA$^+$ that assign a field to a certain value. The great advantage of such data structure is that the values can be accessed by applying the well-known join (.) operator. Tuples the follow the classical definition of tuple – a finite list of elements – and are denoted by the operators $\langle\langle e_1, ..., e_n\rangle\rangle$.

$f[e]$
$DOMAIN\ f$
$[x \in S \longmapsto e]$
$[S \to T]$
$[f\ \ EXCEPT\ \ ![e_1] = e_2]$

Listing 2.4: Functions

$e.h$
$[h_1 \longmapsto e_1, ..., h_n \longmapsto e_n]$
$[h_1 : S_1, ..., h_n : S_n]$
$[r\ \ EXCEPT\ \ !.h = e]$

Listing 2.5: Records

$e[i]$
$\langle e_1, ..., e_n\rangle$
$S_1 \times ... \times S_n$

Listing 2.6: Tuples

Temporal logic operators are an important feature of TLA$^+$. The temporal formula which denotes that a predicate F is always true is $\Box F$, whereas the formula $\Diamond F$ expresses that F eventually is true. Moreover, note that, $F \rightsquigarrow G$ means F *leads to* G and $F \xrightarrow{+} G$ means F *guarantees* G.

$\Box F$ $\Diamond F$ $WF_e(A)$ $SF_e(A)$ $F \rightsquigarrow G$ $F \xrightarrow{+} G$ $\exists x : F$ $\forall x : F$

Listing 2.7: Temporal operators

Note that, the action operators are the ones related to the primed and the non-primed variables. Miscellaneous operators are the conditional and the let statements.

2.2.2 *A TLA Example*

Figure 5 illustrates the hotel room system example, previously explored and explained in Section 2.1.2, modeled under TLA$^+$. Note that, this specification just depicts mathematical symbolism but in order to facilitate the modeling task, some symbols are directly encoded in the IDE, for instance: the symbol $\in$ can be encoded as `\in` and the $\forall$ as `\A`, among others.

There are two classes of variables: **`CONSTANT`** and **`VARIABLE`**. The former variables are known as flexible variables and the latter ones are known as rigid variables. Therefore, `Key`, `Room` and `Guest` variables are not variable as the system evolves. Besides, the remaining variables will vary as the system evolves.

Once the variables are defined, the first step is to define the invariant (`TypeInv`) – the constraints which the system must respect as it evolves. Therefore, the invariant is a formula composed by the conjunction of several sub-formulas. Furthermore, relations in TLA$^+$ are encoded this way: for instance, the keys variable that relates a room with a set of keys is expressed as follows `keys ∈ [Room -> SUBSET Key]`. In this case, the multiplicity of `Key` is the well-known *set* and that is the reason of using the **`SUBSET`**. For the cases with multiplicity *one*, as for instance the variable `currentKey`, the formula is expressed as follows `currentKey ∈ [Room -> Key]`.

The `Init` formula establishes the initial state of the system when it starts – the `lastKey` has the same value of `currentKey` and the `occupant` and the `gkeys` are assumed to be empty. Likewise Alloy, the `Next` predicate is comprised by a disjunction of the `Checkout` and `Checkin` predicates. The behavior of a certain TLA$^+$ model is the conjunction of three components: the `Init` predicate, the `Next` predicate which restricts a valid evolution recurring to the primed variables and finally the temporal formula, which restricts a valid evolution recurring to the temporal operators.

Note that, the classical frame conditions in TLA$^+$ are built by actions. The typical primed variable can be encoded by the keyword **`UNCHANGED`**. Considering the `Entry` predicate, **`UNCHANGED`**`<keys, lastKey, occupant, gKeys>` means that all the variables inside the tuple will remain unchanged in the next state. Nonetheless, considering functions, `currentKey' = [currentKey EXCEPT [r] = k]`! means that the `currentKey` variable remains unchanged except for its values that are different of `r`, which are updated to `k`.

`NoBadEntries` safety property verifies whether there are unauthorized room entries for all states ($\Box$). However, in TLA, a configuration file is necessary to run the TLC model-checker. Such configuration file is essentially composed of: the specification that the user wishes to verify, the properties to check and the values to substitute for constant parameters.

In fact, to this version of this example (hotel room system) here illustrated, the variables `Room` and `Key` must have a constant assignment, which can be solved by introducing the two additional not constant variables – `room` and `key`. Besides, these variables must be **`UNCHANGED`**, and in both `TypeInv` and `Init` some additional constraints must be encoded, such as `guest ∈ SUBSET Guest`. As the variable `guest` represents a set of guests, the previous constraint must be declared, meaning the

guest is contained in the power-set of Guest. Obviously, the room variable follows the same idea. Moreover, the remaining occurrences of Room and Guest must be replaced by room and guest, respectively.

```
------------------------------ MODULE Hotel ------------------------------
EXTENDS Naturals
CONSTANT Key, Room, Guest
VARIABLE keys, currentKey, lastKey, occupant, gKeys
--------------------------------------------------------------------------
TypeInv == ∧ keys ∈ [Room -> SUBSET Key]
           ∧ currentKey ∈ [Room -> Key]
           ∧ ∀ r ∈ Room : currentKey[r] ∈ keys[r]
           ∧ ∀ r1,r2 ∈ Room : (keys[r1] ∩ keys[r2]) # {} ⟹ r1 = r2
           ∧ lastKey ∈ [Room -> Key]
           ∧ occupant ∈ [Room -> SUBSET Guest]
           ∧ gKeys ∈ [Guest -> SUBSET Key]

Init    == ∧ keys ∈ [Room -> SUBSET Key]
           ∧ currentKey ∈ [Room -> Key]  ∧ ∀ r ∈ Room : currentKey[r] ∈ keys[r]
           ∧ ∀ r1,r2 ∈ Room : (keys[r1] ∩ keys[r2]) # {} ⟹ r1 = r2
           ∧ lastKey = currentKey
           ∧ occupant = [r ∈ Room |-> {}]
           ∧ gKeys = [g ∈ Guest |-> {}]

vars == <keys,currentKey,lastKey,occupant,gKeys>

nextKey[k ∈ Key, ks ∈ SUBSET Key] == {x ∈ ks : x > k ∧ (∀ y ∈ ks : y > k ⟹ x<=y)}

Entry(g,r,k) == ∧ (k = currentKey[r] ∨ {k} = nextKey[currentKey[r],keys[r]])
                ∧ k ∈ gKeys[g]
                ∧ currentKey' = [currentKey EXCEPT ![r] = k]
                ∧ UNCHANGED<keys,lastKey,occupant,gKeys>

Checkout(g) ==...
Checkin(g,r,k) ==...

Next == ∃ g ∈ Guest : Checkout(g) ∨ ∃ r ∈ Room, k ∈ Key : Entry(g,r,k)
∨ Checkin(g,r,k)

Spec == Init ∧ [Next]_vars
--------------------------------------------------------------------------
NoBadEntries == □ [∀ g ∈ Guest, r ∈ Room, k ∈ Key : Entry(g,r,k) ∧ occupant[r] # {} ⟹ g ∈
    occupant[r]]_vars

==========================================================================
```

Figure 5: Hotel room system example under TLA

2.3 KODKOD

Kodkod (Torlak and Jackson, 2007) is a constraint solver that combines first-order logic with relational algebra and transitive closure. There are many computational problems that can be solved using a constraint-solving engine, but to solve them it is required to express them as collections of restrictions. Due to the Kodkod's nature, the universe of a Kodkod problem has to be bounded, therefore the

number of atoms (elements in the problem) is fixed. The Kodkod implementation is open-source, accompanied by many examples and an extensive documentation, to make easier the incorporation into other tools, such as Alloy4 (Torlak and Dennis, 2006), Nitpick (Blanchette and Nipkow, 2010), Margrave (Nelson et al., 2010), TACO (Galeotti et al., 2010) or Forge (Dennis et al., 2006).

Kodkod is able to solve a wide variety of problems such as design and code analysis, test case generation, scheduling and planning due to its generic relational engine capability.

To accomplish the design analysis, there must exist the invariant preservation, providing the relational engine with a constraint of the form $\mathbf{S} \wedge \neg \mathbf{P}$ – considering that **P** defines a system property and **S** a system specification. Using the same method, it is also possible to make the code analysis by translating the code to a relational constraint. Another feature of the relational engine, from the system's invariants, is to produce unit tests for modules, by implementing intricate data-types such as red-black trees.

The relational engine is also capable of scheduling and planning, based on requirements and prerequisites of a software system. Basically it can yield the behaviour of the system through its specification.

2.3.1 *Kodkod Language*

A problem in Kodkod is represented by a tuple <`univDecl`, `relDecl`*, `formula`> as depicted in the Figure 6. The universe (`univDecl`) consists of a finite set of atoms that exist in the problem. On the other hand, the declaration of relations (`relDecl`) is the set of all relations in the problem, in which all of them must be bounded by a relational constant. Therefore, to each relation declaration it is specified its `arity` and two `constants` that represents its lower and upper bounds.

A Kodkod relational declaration must be limited by lower bounds – to specify partial instances – and upper bounds – to introduce types – , so that Kodkod's analysis searches an instance within bounds range. A `constant` is composed by a set of `tuples` that is drawn from the problem's universe.

The last component of the Kodkod representation – the formula – is composed by relational and arithmetic operators, to construct relational expressions. Moreover, it is also possible to introduce universal and existential quantifiers into formulas or expressions. As the expressions and the formulas (`expr` and `formula` respectively) are essentially the same that Alloy provides, it is not relevant explain it again.

Due to all these relational operators, Kodkod has a rich language for the formulation of relational constraints.

2.3.2 *Translation to Propositional Logic*

Kodkod problems are translated into propositional logic and then, by using a SAT solver, it is determined the satisfiability of those problems. A Kodkod problem analysis follows several steps, prior

```
problem := <univDecl,relDecl*,formula>

univDecl := { atom[, atom]*}
relDecl := relVar :arity [constant, constant]
varDecl := quantVar : expr

constant := tuple*
tuple := <atom[, atom]*>
arity := positive integer

atom := identifier
relVar := identifier
quantVar := identifier

formula :=
    expr in expr subset              subset
    | some expr                      non-empty
    | one expr                       singleton
    | no expr                        empty
    | not formula                    negation
    | formula and formula            conjunction
    | formula or formula             disjunction
    | all varDecl | formula          universal
    | some varDecl | formula         existential

expr :=
    expr + expr                      union
    | expr & expr                    intersection
    | expr - expr                    difference
    | expr . expr                    join
    | expr -> expr                   product
    | ~expr                          transposes
    | ^expr                          closure
    | {varDecl | formula}            comprehension
    | relVar                         relation
    | quantVar                       quantified variable
```

Figure 6: Kodkod's syntax

to the application of the SAT solver, as depicted in Figure 7. Before to focuses on each step of the Kodkod's architecture, an overview of it will be provided bellow.

Considering that a symmetry represents a permutation of atoms in the problem's universe, that takes models of the problem to other models, and non-models to other non-models (Torlak, 2009), the first step is to detect the symmetries in P (Kodkod problem). Furthermore, P is translated to a Compact Boolean Circuit (CBC(P)) and the Symmetry Breaking Predicate (SBP(P)) is computed. After that, the CBC(P) $\wedge$ SBP(P) are transformed into the Conjunctive Normal Formula, respectively denoted by CNF(P). The analysis process ends when the SAT solver is applied to the CNF(P), yielding a model, if it is satisfiable. For the purpose of clarifying the aforementioned steps, each one will be approached in more detail below.

Going back to the first step, there are many problems that exhibit symmetries. These lead to equivalences or isomorphisms among bindings in the problem's state space, and its problem could be formally defined, as shown below based on (Torlak and Jackson, 2007).

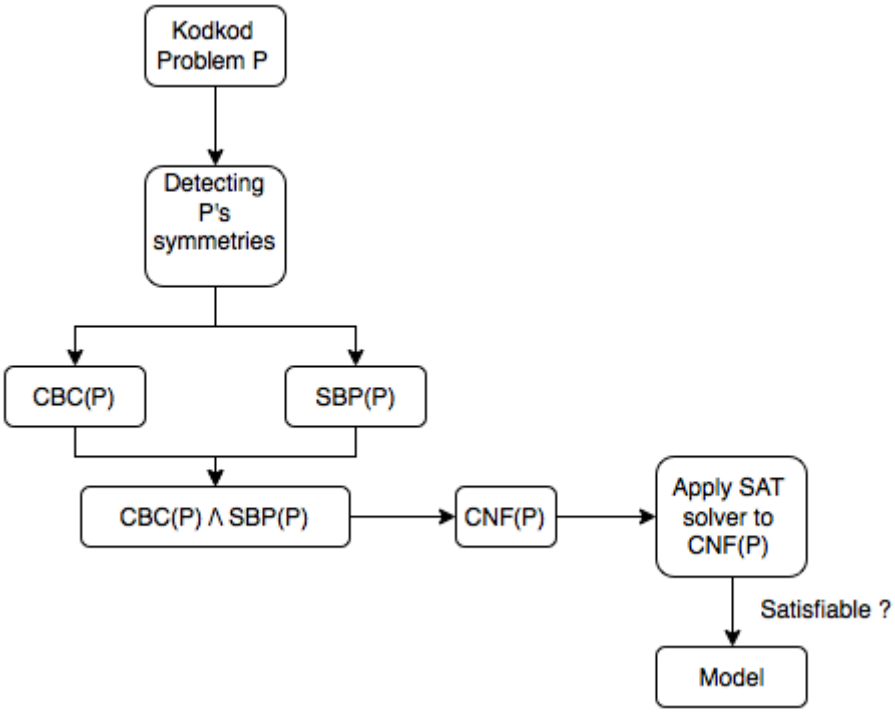


Figure 7: Kodkod architecture

Definition 1. *Let **A** be the problem's universe, **D** a set of declarations over **A**, and **F** a formula over **D**. A permutation l: **A** → **A** is a symmetry of the problem **P** = (**A**,**D**,**F**) if and only if l(**M**) is a model of **P**, written l(**M**) ⊨ **P**, ∀ **M** ⊨ **P**. The models **M** and l(**M**) are said to be isomorphic.*

Symmetry detection aims to identify the atom symmetry classes, for later perform the symmetry, breaking in that set of symmetries. Symmetry breaking, or isomorph elimination, has an important role in Kodkod problems analysis, since it eliminates isomorphic models, thereby reducing the search space. It can be performed by two approaches – static and dynamic. The dynamic approach uses a symmetry-aware model finder on the problem (Zhang, 1994), while the static, is the application of a SAT solver on CNF(P ∧ SBP(P)) (Shlyakhter, 2005). Kodkod takes the static approach to symmetry breaking, since it applies a simple lex-leader symmetry breaking predicate (Shlyakhter, 2001) for the classes that were identified in the symmetry detection. This process contributes to a better performance of many problems.

The Kodkod problem translation into a CBC is performed after symmetry detection. Nonetheless, included in CBC(P) process, the translation from a relational to a boolean formula is also carried out, as shown in (Jackson, 2000). The translation is encoded using matrices composed by boolean values. Given a finite universe of atoms **U**=$\{u_1,...,u_n\}$, to each relation **R** with arity **a**, it is encoded a **a**-dimensional matrix **M** with capacity to $\mathbf{U}^{\mathbf{a}}$ propositional variables, whose entries are defined as follows:

$$\mathbf{M}[i_1,...,i_a] \begin{cases} 1, & \text{if } <u_i,..,u_{i\mathbf{a}}> \in \mathbf{R}_{lw} \\ 0, & \text{otherwise} \\ \mathbf{R}_{\{i\mathbf{a}\}}, & \text{if } <u_i,..,u_{i\mathbf{a}}> \in \mathbf{R}_{up} - \mathbf{R}_{lw} \end{cases}$$

$\mathbf{R}_{lw}$ and $\mathbf{R}_{up}$ represent the lower and upper bounds of a relational constant, associated to a relation $\mathbf{R}$. The tuples that belong to lower-bound are represented by 1s (*true*) in the matrix entries, whereas the tuples that do not belong to upper bound are represented by 0s (*false*) and the tuples that belong to lower bound, and do not belong to upper bound, are unknown and thus are assigned boolean variables that are to be solved.

Besides, the translation from relational formulas to propositional formulas is made by interpreting relational operators as matrix operators (Jackson, 2000).

Furthermore, the CBC is a directed, acyclic graph(V,E), similar to a BDD (Binary Decision Diagrams), being well-suited to represent boolean formulas.

Hence, the next step in the analysis of a Kodkod problem is the translation to the CNF conjoining the CBC with the lex-leader symmetry breaking predicate. The last step consists in applying the SAT solver to CNF(P).

2.3.3 *Translation from Alloy to Kodkod*

The Alloy Analyzer, Alloy's tool, uses the Kodkod model finder as a back-end to analyze a model. Alloy is translated to Kodkod and, in turn, Kodkod translates the correspondent model expressed in relational logic into a corresponding boolean logic formula, and then invokes an off-the-shelf SAT-solver on the boolean formula (Sub-Section 2.3.2). Due to the nature of both languages, the translation is not straightforward. In Alloy, relational variables are divided into signatures (unary relations) and fields (non-unary relations). Signatures form a hierarchy type and are bounded by an integer limit, being these limits relations too. However, in Kodkod all relations are untyped and are interpreted similarly. Relations are bound from both below and above through relational constants. Unlike Alloy, Kodkod has no syntactic sugar (predicates, functions, facts).

The translation process consists in translating the Alloy AST into the Kodkod AST. However, Alloy performs a few simplifications in order to reduce the number of variables in the final CNF formula.

An Alloy model can be represented in Kodkod using the java API that Kodkod provides. To better understand a Kodkod problem, the respective Kodkod code responsible for handling the relation `currentKey` is presented in Figure 3.

Field `currentKey` represents the relation `Room -> keys one -> Time`. The first step in this process is to declare all relations, that in this case are four. `Key`, `Time` and `Room` are unary relations, whereas `currentKey` is a ternary relation. The atoms in the model's universe follow the specification given by an Alloy command,which specified two elements `Room`, five elements `Time` and for all others three elements. Considering the Listing 2.8, a *Universe* is created by all atoms at the atom list

and this one provides a *TupleFactory* for creating constants, represented by *TupleSet*.

```
Relation x3 = Relation.unary("this/Key");
Relation x4 = Relation.unary("this/Time");
Relation x5 = Relation.unary("this/Room");
Relation x11 = Relation.nary("this/Room.currentKey", 3);

List<String> atomlist = Arrays.asList("Key$0","Key$1","Key$2", "Room$0","Room$1", "Time$0",
    "Time$1", "Time$2","Time$3", "Time$4",);
Universe universe = new Universe(atomlist);
TupleFactory factory = universe.factory();
Bounds bounds = new Bounds(universe);
```

Listing 2.8: Relations

A *TupleSet* gathers all atoms that belong to a respective relation, drawn from the universe. Hence, the relation is bounded above by that TupleSet. Both upper and lower bounds are equal when are defined by *Exactly*. Otherwise just the upper bounds will be filled. It is possible that an Alloy command specifies which unary relations have an exact bound, for instance – **`run`**`{} ` **`for`** `3 ` **`but exactly`** `numberOfAtoms nameOfRelation`. This example has the relations `Key` and `Time` bound exactly because they are defined with a total ordering in the Alloy model.

```
TupleSet x3_upper = factory.noneOf(1);
x3_upper.add(factory.tuple("Key$0"));
x3_upper.add(factory.tuple("Key$1"));
x3_upper.add(factory.tuple("Key$2"));
bounds.boundExactly(x3, x3_upper);

TupleSet x4_upper = factory.noneOf(1);
x4_upper.add(factory.tuple("Time$0"));
...
x4_upper.add(factory.tuple("Time$4"));
bounds.boundExactly(x4, x4_upper);

TupleSet x5_upper = factory.noneOf(1);
x5_upper.add(factory.tuple("Room$0"));
x5_upper.add(factory.tuple("Room$1"));
bounds.bound(x5, x5_upper);

TupleSet x11_upper = factory.noneOf(3);
x11_upper.add(factory.tuple("Room$0").product(factory.tuple("Key$0")).product(factory.tuple(
    "Time$0")));
x11_upper.add(factory.tuple("Room$0").product(factory.tuple("Key$0")).product(factory.tuple(
    "Time$1")));
...
bounds.bound(x11, x11_upper);
```

Listing 2.9: Bounds

Upper bounds related to n-ary relations are filled with all possible combinations of atoms, underlying to each relation. In Listing 2.9, the TupleSet $x11_upper$ represents the bounds of the `currentKey` relation. Only two combinations of the thirty are presented – 2 `Room` * 3 `Key` * 5 `Time`.

Finally, as depicted in Listing 2.10, it is required to construct formulas that restrict the acceptable values of the relations. Formula *cur* defines `currentKey` **`in`** `Room -> Key -> Time`, whereas *curOne* specifies **`all`** `t : Time, r : Room |` **`one`** `r.currentKey.t`. Formula fnl is the conjunction of *cur* and *curOne* formulas, and, for that, is the one which is analyzed by solver. The solver might be configured with many options, such as: what is the SAT solver (*DefaultSAT4J, Glucose,* ...) that will analyze the formula and the bounds or whether to symmetry breaking.

```
Formula cur = x11.in(x5.product(x3).product(x4));
Variable t=Variable.unary("t");
Variable r=Variable.unary("r");
Formula curOne = (r.join(x11.join(t))).one();
Formula fnl = cur.and(curOne);

Solver solver = new Solver();
solver.options()...;
Solution sol = solver.solve(fnl,bounds);
```

Listing 2.10: Formula

2.4 LINEAR TEMPORAL LOGIC

The Linear Temporal Logic (LTL) (Huth and Ryan, 2004) is a formal temporal logic to represent any system with modalities that can be qualified in terms of time. Therefore, through this logic, it is possible to encode future behaviour of any system, which is encoded by resorting to LTL temporal operators. Due to the impossibility of LTL dealing with past behaviours, Past Linear Temporal Logic (PLTL) emerged with the purpose of handling past behaviours in reactive systems. In LTL, both future and past behaviours, are seen as a sequence of paths, thus they are seen as a path composed of states. During the remaining chapter, there is an abuse of notation since when there is a mention to LTL operators, it is assumed that the PLTL operators are also mentioned – $PLTL \subseteq LTL$. Despite of encoding temporal systems, it is also possible to verify any system specified by LTL through Bounded Model Checking (BMC), which is convenient for this dissertation since the Kodkod is bounded, thus, the trace's instances is always bounded.

The first BMC technique, to verify models, which are expressed with LTL formulas, was first proposed by Latvala et al. (2005). All these aspects are exploited and developed in the present Section.

2.4.1 *LTL Operators*

LTL aims to represent any system behaviour that evolves in time. To achieve that, any time behaviour is encoded by temporal operators. This way, each temporal operator, according to its meaning, restrains a certain system property as the time evolves.

LTL operators are depicted in Table 1, in which, any one is described in three different ways – textually, symbolically and graphically. In the last one, each red circle corresponds to the current state of the trace (system evolving path), assuming that the trace's start is in the first circle, from the left to the right.

However, the Textual column is an abbreviation of the original temporal operators' name. Therefore, LTL operators are defined as follows: **X** for next, **G** for globally, **F** for finally, **P** for previous, **H** for historically, **O** for once, **U** for until and **R** for release. **P**, **H** and **O** operators are responsible for representing the system behaviour for the past states.

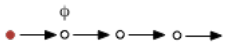
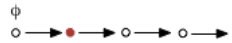
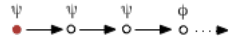
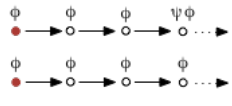
Textual	Symbolic	Diagram
Unary Operators		
$\mathbf{X}\,\phi$	$\bigcirc\,\phi$	
$\mathbf{G}\,\phi$	$\Box\,\phi$	
$\mathbf{F}\,\phi$	$\Diamond\,\phi$	
$\mathbf{P}\,\phi$	$\bigcirc^{-1}\,\phi$	
$\mathbf{H}\,\phi$	$\Box^{-1}\,\phi$	
$\mathbf{O}\,\phi$	$\Diamond^{-1}\,\phi$	
Binary Operators		
$\psi\,\mathbf{U}\,\phi$	$\psi\,\mathcal{U}\,\phi$	
$\psi\,\mathbf{R}\,\phi$	$\psi\,\mathcal{R}\,\phi$	

Table 1: LTL operators

Now, it is time to focus on each temporal operator. The operator **X** means that a certain formula ψ has to hold in the next state from the current one. Unlike **X**, the operator **P** means that ψ has to hold in the previous state from the current one. On the other hand, **G** and the **H** operators are responsible to ensure that a certain formula ψ is valid in all states succeeding the current state. These ones control

the succeeding and the preceding states from the current state – the operator **G** covers all next states and the operator **H** covers all the previous states. The two last unary operators are **F** and **O**, meaning, to a particular formula ψ, ψ eventually has to hold in any state succeeding (**O**) or preceding (**F**) the current state. The **U** operator is binary, so given two particular formulas ψ and ϕ, $\psi\,\mathbf{U}\,\phi$ means that ψ has to hold at least until ϕ, which holds at the current or a future position. The last operator is the **R** and is the more complex one. Given $\psi\,\mathbf{R}\,\phi$, ψ has to be true until and including the point where ϕ first becomes true. If ϕ never becomes true, ψ must remain true forever. Therefore, as above depicted in the table, there is an alternative, either ψ always hold in all states or ψ has to be true until, and including, the point where ϕ first becomes true.

2.4.2 *LTL Bounded Model Checking*

The main idea of Bounded Model Checking consists in search for counter-examples finite search-space. In this case, given a certain temporal formula, the model checker must to verify this formula against the model. This verification is performed over a sequence of future or previous states, or usually called as a simple path. During the verification, the bound is progressively increased, by looking for longer possible counter-examples.

There are two main types of properties that can be specified by using LTL – safety and liveness. The former states that something bad never happens and a certain counter-example might be finite, since just it is necessary to show a state that something bad happens. The latter asserts that something good eventually happens. However, the counter-examples of liveness properties has to be extended to an infinite path and, in this case, it has to satisfy the liveness formula. Hence, a certain counter-example of liveness must show a trace such that something good never happens.

To verify the liveness properties, in (Biere et al., 2003) was introduced the notion of *back-loop*. This way, it is only considered the counterexamples whose prefixes of traces contain a *back-loop* in the last state. More precisely, in liveness properties, the counter-examples are just considered as true, if they hold some infinite execution traces – this is just possible because a finite path can be transformed in an infinite path if there is a *back-loop*.



(a) No loop

(b) (k,l)-loop

Figure 8: The two possible cases for a bounded path

Figure 8b illustrates how to represent a finite path as an infinite path – in this case, although the prefix of the path is finite, if there exists a *back-loop* from the last state of the prefix to any one of the previous states, it is possible to represent infinite paths with a finite number of states. Figure 8a is an example of a prefix path (or trace) without a *back-loop*.

Given a $\mathbf{G}\,\psi$ formula, just prefixes with *back-loop* can represent the *witness* (infinite path in which a certain property holds) of such formula. In particular, the *witness* represents the results of the command *runs* and the counter-examples are the results from the commands *check*. However, in practise, supposing that $\mathbf{G}\,\psi$ is true (ψ holds in all states from $s0$ to sk), if there is not *back-loop* from sk to any state within $[si, sk]$ it is not possible to conclude the *witness* for $\mathbf{G}\,\psi$, since in the state $sk+1$, ψ might not hold. Hence, the $\mathbf{G}$ operator must to have the *back-loop* formula in its expansion – $one\,loop\,\&\,\mathbf{G}\,\psi$.

Besides, according to Biere et al. (2003), a particular LTL formula has to be negated before its solving. The standard LTL semantics is unbounded, thus, there is a duality between the operators $\mathbf{F}$ and $\mathbf{G}$ – $\neg\mathbf{G}\phi \equiv \mathbf{F}\neg\phi$. However, after embedding the LTL semantics into the well known bounded model checking, it is introduced the notion of *back-loop* into the $\mathbf{G}$ semantics, thus, the duality between $\mathbf{G}$ and $\mathbf{F}$ no longer holds – $\neg\mathbf{G}\phi \neq \mathbf{F}\neg\phi$. Given that, all negations must be propagated in order to ensure that unnecessary *infinites* appear in a specific LTL formula. As the $\mathbf{R}$ operator is composed by a $\mathbf{G}$, the aforementioned problem is applied to this operator. As a consequence, likewise the $\mathbf{G}$ and $\mathbf{F}$ operators, the duality of the duality of $\mathbf{R}$ and $\mathbf{U}$ no longer hold. Therefore, to verify a LTL formula, prior to the solving process, it is transformed to Negation Normal Form (Sub-Section 2.4.3).

2.4.3 *Negation Normal Form*

As above mentioned, the procedure of transforming a certain formula into NNF, is necessary. So, this way, this procedure is necessary due to the bounded model checking procedure, namely, in the bounded semantics the duality between the operators *always* ($\mathbf{G}$) and *eventually* ($\mathbf{F}$) no longer holds, according to Biere et al. (2003).

Figure 9 illustrates all the rules to follow with the purpose of translating any LTL formula to its NNF. The well-known NNF transformation rules consists in eliminating both implications and equivalences and then, by repeatedly applying the Morgan's laws (the left side of the image), move negations inwards. Moreover, both existential and universal quantifiers are also expanded as earlier explained. The right side of the image illustrates the NNF rules for the LTL operators. The duality of the LTL operators $\mathbf{G}, \mathbf{F}$ and $\mathbf{R}, \mathbf{U}$ are also depicted bellow. So, the $\mathbf{H}, \mathbf{O}$ operators share the same rule that $\mathbf{G}, \mathbf{F}$. The only difference between them is the direction that the operators take – the first ones represent previous paths from the current position, whilst the second ones represent future paths from the current position (Sub-Section 2.4.1).

Unlike these operators, $\mathbf{X}$ and $\mathbf{P}$ operators have no duality, thus, their NNF rules only consists in negating the formula and keep the same temporal operator. The application of these rules in Alloy are described in Section 4.4.

- $P \rightarrow Q \equiv \neg P \vee Q$
- $\neg(P \rightarrow Q) \equiv P \wedge \neg Q$
- $P \Leftrightarrow Q \equiv (P \vee \neg Q) \wedge (\neg P \vee Q)$
- $\neg(P \Leftrightarrow Q) \equiv (P \wedge \neg Q) \vee (\neg P \wedge Q)$
- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
- $\neg\neg P \equiv P$
- $\neg(\forall x P(x)) \equiv \exists x \neg P(x)$
- $\neg(\exists x P(x)) \equiv \forall x \neg P(x)$
- $\neg[\![\mathbf{G}\phi]\!] \equiv [\![\mathbf{F}\neg\phi]\!]$
- $\neg[\![\mathbf{F}\phi]\!] \equiv [\![\mathbf{G}\neg\phi]\!]$
- $\neg[\![\mathbf{H}\phi]\!] \equiv [\![\mathbf{O}\neg\phi]\!]$
- $\neg[\![\mathbf{O}\phi]\!] \equiv [\![\mathbf{H}\neg\phi]\!]$
- $\neg[\![\mathbf{X}\phi]\!] \equiv [\![\mathbf{X}\neg\phi]\!]$
- $\neg[\![\mathbf{P}\phi]\!] \equiv [\![\mathbf{P}\neg\phi]\!]$
- $\neg[\![\phi\mathbf{R}\psi]\!] \equiv [\![\neg\phi\mathbf{U}\neg\psi]\!]$
- $\neg[\![\phi\mathbf{U}\psi]\!] \equiv [\![\neg\phi\mathbf{R}\neg\psi]\!]$

Figure 9: NNF rules

2.4.4 *Embedding of Temporal Formulas in Alloy*

LTL formulas can be embedded in Alloy with the purpose of designing dynamic models by using temporal operators, instead of using explicit quantification over the states. As a consequence, the modeling becomes easier. To support this, the Alloy has to expand the each temporal operator with its correct semantic.

Figure 10, according to Cunha (2014), illustrates the logical expansion that Alloy has to carry out. So, to each operator there is a logical formula that reflects its meaning (Sub-Section 2.4.1). It is noteworthy that there is a new operator depicted. As a convenience, it is introduced the operator $'$ (prime), which has the same value of **X**, but the only difference is the context that they are used. The operator **X** extends the formula, so it quantifies a certain formula, whereas the operator $'$ extends the expression, thus it can be applied in any expression. More precisely, given a relation r, it is possible to apply the prime operator – r'. On the other hand, it is not possible to apply $\mathbf{X}\,r$.

The nesting of temporal operators is possible, allowing the specification of any temporal operator from the current state. This way, to the dynamic relations are annexed the current state in the context, for instance: supposing that the current context is the `first` and it is applied a certain **X**, so the context is updated to `first.next` and the next temporal operator (in the formula) will quantify over that context.

As discussed in the previous Section, the operator **G** needs to have a *loop*. In the Figure, the `infinite` formula means the *loop* aforementioned, since internally that formula is expanded to `one loop`.

The `next` formula depicted in the expansion rules, represents a total order over the time. Therefore, it is an order between the first and the last state, and this way it is possible to reach the next and previous states.

- $[\![' \phi]\!]_s \equiv$ **some** $s.next$ **and** $[\![\phi]\!]_{s.next}$
- $[\![\mathbf{X}\,\phi]\!]_s \equiv$ **some** $s.next$ **and** $[\![\phi]\!]_{s.next}$
- $[\![\mathbf{P}\,\phi]\!]_s \equiv$ **some** $s.{\sim}next$ **and** $[\![\phi]\!]_{s.{\sim}next}$
- $[\![\mathbf{G}\,\phi]\!]_s \equiv$ **infinite and all** $s' : s.{*}next \mid [\![\phi]\!]_{s'}$
- $[\![\mathbf{F}\,\phi]\!]_s \equiv$ **some** $s' : s.{*}next \mid [\![\phi]\!]_{s'}$
- $[\![\mathbf{H}\,\phi]\!]_s \equiv$ **infinite and all** $s' : s.{*}{\sim}next \mid [\![\phi]\!]_{s'}$
- $[\![\mathbf{O}\,\phi]\!]_s \equiv$ **some** $s' : s.{*}{\sim}next \mid [\![\phi]\!]_{s'}$
- $[\![\phi\,\mathbf{U}\,\psi]\!]_s \equiv$ **some** $s' : s.{*}next \mid [\![\psi]\!]_{s'}$ **and all** $s'' : s.{*}next\ \&\ \hat{}next.s' \mid [\![\phi]\!]_{s''}$
- $[\![\phi\,\mathbf{R}\,\psi]\!]_s \equiv [\![\mathbf{G}\,\psi]\!]_s$ **some** $s' : s.{*}next \mid [\![\phi]\!]_{s'}$ **and all** $s'' : s.{*}next\ \&\ {*}next.s' \mid [\![\psi]\!]_{s''})$

Figure 10: Temporal operators expansion

There are important logical operators that have a crucial role in the formulas – $*$, $\hat{}$ and $\sim$. The first one (reflexive-transitive closure) denotes all states to forward from the current state, the second one (non-reflexive transitive closure) is similar to the reflexive-transitive closure, with the difference that in this case the current state is not included. Finally, the third one (transpose) inverts the order of the elements in some tuple, more precisely, it enables the navigation to the previous states – for instance, the formula $\backsim$`next` denotes the previous state from the current one, whereas the formula $*\backsim$`next` has the same meaning that $*$, but in this case to the previous states. The application of this operator follows the same thinking with any non-reflexive transitive closure operator.

To reach the truly verification, some formula ψ that occurs in a fact or run command should be replaced by $[\![\mathbf{NNF}\,\phi]\!]_{first}$, where the start state is the first, and the formula ψ is transformed to its Negation Normal Form (Sub-Section 2.4.2).

2.5 SAT SOLVING OVERVIEW

The Boolean Satisfiability Problem (SAT) consists in determining, if, for a given boolean formula **F** there is a assignment of of its free boolean variables that make **F** satisfiable. More precisely, it is verified, for the variables of a given boolean formula, if their replacement, by values TRUE or FALSE, the satisfiability (TRUE) of the formula (the formula is satisfiable – SAT) is ensured. The formula is unsatisfiable (UNSAT) when there is not such assignment. This problem is NP-complete (Cook, 1971), which means that is greater than polynomial-time complexity in the worst case. This time can only be changed if NP = P.

SAT solvers are responsible for solving the SAT problems. Many algorithms and solvers have been proposed to solve a SAT, but due to the problem above-mentioned, there is no one that ensures total

correctness and efficiency for all possible input instances. SAT solvers solve a problem based on CNF, since is the generally accepted norm. Then it will be presented a brief description of CNF.

2.5.1 *Conjunctive Normal Form*

SAT solver algorithms use propositional formulas in this form to a matter of efficiency. CNF is, in boolean logic, a formula that is a conjunction of clauses, in turn clauses are disjunction of literals. More precisely, a formula is composed by :

- **Atoms** - propositional variables, for instance: x, y, z.
- **Literals** - either atoms or its negation, for instance : $x, \neg y, \neg z$.
- **Clauses** - disjunctions of some **Clauses**, for instance: $x \vee \neg y \vee \neg z$.

The conjunction of some clauses represents a CNF formula **F**, for instance : $\mathbf{F} = (x \vee \neg y \vee \neg z) \wedge (x \vee \neg y)$. A formula is satisfiable if there is a set of assignments that satisfy all its clauses, in turn, a clause is satisfiable if at least one of its literals are satisfiable. Literals may have two kind of values, either positive or negative. A literal is satisfiable when to a positive literal exists a true (1) assignment, whereas to a negative literal exists a false (0) assignment. If this assignment does not occur, the formula is named unsatisfiable.

2.5.2 *DPLL SAT Solvers*

The Davis–Putnam–Logemann–Loveland (DPLL) algorithm (van Harmelen et al., 2008) is a backtracking-based search algorithm for solving SAT problems. This search in SAT problems can be interpreted as a binary tree, in which each node represents a variable and all nodes bellow are the rest of search-space. This algorithm is the basis of most SAT solvers. Figure 11 illustrates the iterative format of the algorithm. DPLL performs a depth-first search through the space of possible variable assignments, stopping when an assignment that satisfies the formula is found.

The DPLL algorithm takes as input a CNF formula and, either returns UNSAT or, returns SAT with a variable assignment. During the search, a DPLL SAT solver performs the following steps:

- **DecideNextBranch**: This procedure decides a variable to branch the search space. Heuristics, in this procedure, attempt to make a decision in order to chose the variable. Only one decision per level is possible.
- **Deduce**: This procedure applies unit propagation, a method that can simplify a set of clauses. Unit propagation, or unit rule, is based on clauses that are composed by a single literal. If a clause **C** contains a single literal unassigned, every clause containing **C** is removed, being satisfiable if **C** is. Every clause containing $\neg$**C** is deleted, since it does not contribute to the

```
Input :  A CNF formula
Output :  UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ⟵ Deduce
            if status = CONFLICT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backtrack(blevel)
            else if status = SAT then
                Output current assignment stack
                return SAT
            else break
    end
```

Figure 11: DPLL algorithm taken from (van Harmelen et al., 2008)

satisfiability of **C**. This deletion leads to empty clauses yielding the **conflict**. If there are neither conflicts nor free variables returned SAT is returned (satisfiable).

- **AnalyzeConflict**: This operation aims to computing the required backtracking level based on a conflict. Conflicts are exploited to reduce the space to be searched and skip future areas of search space, where the conditions are satisfiable.

- **Backtracking**: Once a conflict is found and analyzed, the SAT solver performs the backtrack, this means that it may returns to the previous level undoing the last decision. If the SAT solver backtracks to level 0, the problem is unsatisfiable, otherwise starts all procedures described above. This perspective of backtrack is called chronological.

2.5.3 *DPLL Optimizations*

A lot of optimizations have been proposed to DPLL SAT solvers, in order to improve its performance. The Conflict-Driven Clause Learning (CDCL) algorithm is an optimization of DPLL, since it has procedures such as clause learning and non-chronological backtracking.

Clause learning is the procedure of learning from conflicts what happened during a SAT solver search. Most important DPLL optimization is related with learning from conflicts. When a conflict is found, this kind of SAT solvers must be capable of identifying what caused the conflict, generating and learning a clause that describes the conflict. The procedure aims to not analyze the same conflict twice, thereby limiting the search-space.

Another feature of CDCL SAT solvers is their backtracking, since unlike DPLL, CDCL uses non-chronological backtracking. This approach to backtrack is capable of undoing several decisions (backtrack to more than one level), unlike chronological that can only undo the last decision (backtrack to

one level). When a conflict is found, a conflict cause is produced that determines the origin of the conflict. Finally, the clause is learned and the backtracking is performed.

Besides these, Two-Watched Literals and restarts are also two important DPLL optimizations, however, less important.

3

STATE OF THE ART

The purpose of the State of the Art is to explore concepts and ideas relevant to the dissertation's goal. Thus, concepts such as parallelization techniques as well as some parallel SAT solvers, Software Product Lines (SPLs) verification using dynamic algorithms and two techniques of parallel bounded verification will be presented.

In particular, the parallel SAT solvers are presented because Kodkod is built over SAT solvers and one of this dissertation's goals is to implement a parallel version to verify Kodkod models. SPLs are explored since they have a particular common point with the systems considered in this research work (dynamic system with rich configurations) – each selection of features is a configuration, thus, a certain SPL represents multiple configurations. Finally, parallel bounded verification is exploited since this dissertation aims to apply a parallel verification on Electrum.

Such topics will be described in detail in the remainder chapter, which is structured as follows: parallelization techniques as well as some parallel SAT solvers are approached in Section 3.1, SPLs verification using dynamic algorithms in Section 3.2 and finally, the parallel bounded verification in Section 3.3.

3.1 SAT SOLVING PARALLELIZATION STRATEGIES

The computer processing power has evolved, as a consequence of the arising of multi-core CPUs, leading to the appearance of the first parallel SAT solvers. The first SAT solvers were sequential, thereby hindering the solving of more complex problems due to the time that it takes. Hence, the arising of parallel SAT solvers has been widely addressed and explored, being the motive of competition between solvers [1], in which the minimum detail makes a big difference in the performance.

This kind of SAT solvers present some challenges, such as workload balance and synchronization. Synchronization is required to take full advantage of multiprocessing, in order to avoid idle processes. There are three classes of parallel solving (Marques et al., 2012) – cooperative, competitive and collaborative, considering that the first two are the most used and explored. Cooperative strategies split the search-space to be explored by each computational unit (either a core, a processor or computer)

1 See more about SAT competitions at http://www.satcompetition.org/

in order to find a solution. Each computational unit, with a partial view of the formula, explores its search-space applying different sequential SAT solvers with the same heuristics. Competitive strategies do not split the search-space, where each computational unit exploit the same formula. Different strategies, like heuristics, are applied over each computational unit. The solving process stops when some computational unit gets one solution. The least used, but nowadays the most explored, is the collaborative strategy. This strategy combines both sequential and parallel approaches, by using a master-slave architecture. It handles a SAT problem sequentially using a master process. To aid the sequential task, the slaves deduce some information in parallel.

Parallel algorithms, using cooperative strategies, are also based on CDCL SAT solvers, which enable clause sharing. This process enables the improvement of the performance, since when a conflict happens, one clause is learned and shared to other computational units, thus avoiding further conflicts.

The communication between computational units, in this kind of systems, is crucial and required. There are two kinds of communication (Hölldobler et al., 2011) – shared-memory and distributed-memory. In the former, there exists only one address space common to all process, whereas in the latter, the components are connected by a communication network through which they communicate by passing messages. Generally, the shared-memory architecture is faster than distributed-memory, since there is no communication between computational units and the access to memory is faster than a network communication. The distributed-memory architecture needs more computational power than shared-memory architecture.

3.1.1 *Cooperative Solving*

Cooperative parallel SAT solvers, also known as search space splitting, attempt to split the search-space into different parts. Hence, a part of the search-space is given and handled by every processing unit (either a core, a processor or computer). Due to the nature of this approach it is also named as “divide-and-conquer”. Since any processing unit exploits one search-space, the workload balance between them is very important in order to have no idle computational units.

Most of the SAT solvers with cooperative strategy, use the *master-slave* approach to address the workload balance. The most popular and used search-space splitting technique is guiding paths. However, there are also other approaches such as scattering functions (Shimizu et al., 2006) and XOR partitioning (Plaza et al., 2008). During the solving process, to avoid idle processes, the distribution of sub-formulas is performed dynamically. There are many ways to achieve this, Dynamic Work Stealing and Task Farm being examples of that.

In the following sub-section, the search-space technique guiding paths will be described, as well as two ways to dynamically distribute sub-formulas.

3.1.1.1 *Search-space splitting using guiding paths*

Guiding paths (Zhang et al., 1996) are the most popular strategy to perform the search-space splitting – a technique that, as the search process evolves, keeps a list of variables that have been assigned until the current state of the search process. For each propositional variable (literal) two boolean values are recorded – the corresponding assigned value (TRUE or FALSE) and another boolean value (TRUE or FALSE) that shows if both assignment values have been tried on the variable. In case both assignment values have been tried, the propositional variable is called **closed**, otherwise they are called **open**. These open variables represent subspaces that have not been explored yet. In guiding paths, when a set with **n** variables is chosen, there are $2^{\mathbf{n}}$ possible guiding paths.

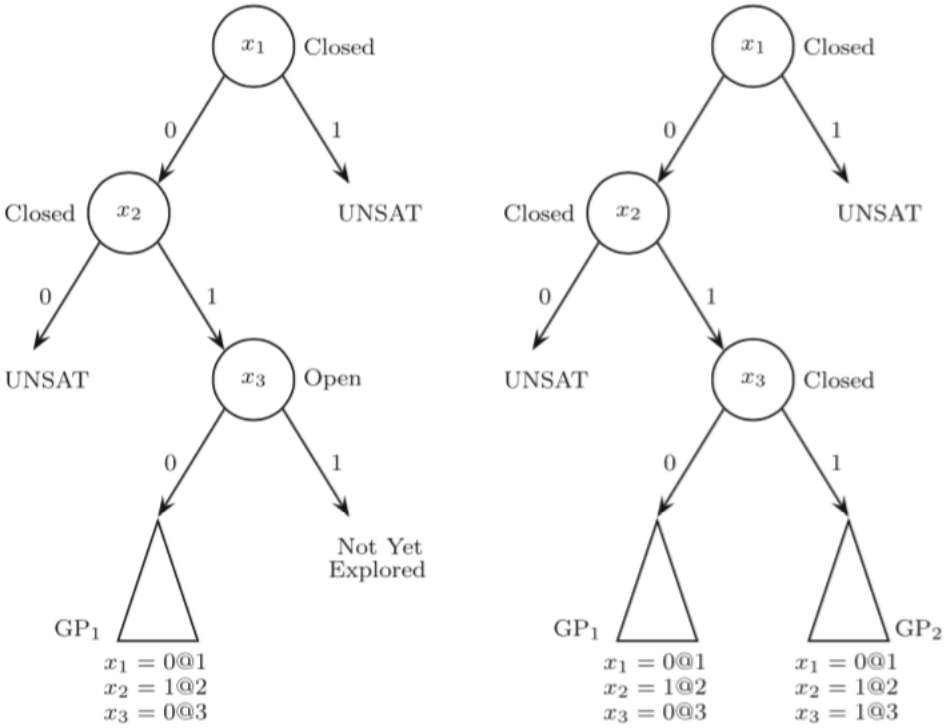


Figure 12: Search space splitting example using guiding paths taken from (Martins et al., 2012)

Since the SAT problems can be interpreted as a binary tree, Figure 12 illustrates the use of guiding paths to represent the search-space. For each tree, one process is assigned, while the left-side shows the current search state of the first process, the right-side shows the creation of disjoint subspaces (which can be searched in parallel) through the open variables in guiding paths. The guiding path GP_1, at third level of the tree, is expressed as follows : $GP_1 = [(\neg x_1, \mathbf{closed}), (x_2, \mathbf{closed}), (\neg x_3, \mathbf{open})]$. Following the guiding path, it is easy to realize that the variable x_1 $(\mathbf{closed})$, at level zero, is assigned the value **0** (left branch), at level two the variable x_2 $(\mathbf{closed})$ is assigned the value **1** (right branch). Finally, at level three, the current value of the variable x_3 is **0** (left branch) and is **open** because the formula has not been tested yet where x_3 is set to **1** (right branch).

The process responsible for GP_2 begins the search on the subspace defined as follows: $GP_2 = [(\neg x_1, \textbf{closed}), (x_2, \textbf{closed}), (x_3, \textbf{closed})]$. Once search starts, the first process replace the variable x_3 status to closed – $[(\neg x_1, \textbf{closed}), (x_2, \textbf{closed}), (\neg x_3, \textbf{closed})]$. The flags, that represents whether both possible values were assigned (**1** or **0**), were replaced in the guiding path, above presented, by **open** and **closed** respectively.

The mechanism explained statically distributes the search-space, which enables the inactivity of many processes leading a decrease in performance. For each computational unit one search-space is assigned, which stops after the analysis of that space. Consequently, this leads to a weak workload balance.

There are search-spaces more difficult to analyze than others, hence a dynamic distribution is required. Two examples of that are described below.

3.1.1.2 *Dynamic work stealing*

Dynamic work stealing (Jurkowiak et al., 2001) is a strategy to address workload balance problems. A static distribution of a problem can lead to some processes becoming idle, while another process is still working. In this approach, the problem is distributed over the processes and when some process has finished its role, it steals work from others processes that are still working.

Most of the SAT solvers use master-slave architecture to communicate. This architecture has one master process that controls one or more slaves, in turn slaves are responsible to analyze sub-formulas. The Master is responsible for ensuring workload balance between slaves – receiving and sending information for all its slaves. When a slave finishes the analysis of its sub-formula, it communicates with the master in order to receive more sub-formulas to analyze. Once the notification is sent, the master process selects the slave that has a smaller guiding path and splits its search-space to give the unexplored part to the slave that was idle. However, there is no relation between the size of the guiding path and the size of area to search. More precisely, choosing the shortest guiding path does not entail to choose the smallest unexplored area of search.

Fast resolution of a sub-formula, by a process, can lead to an inefficient CPU time management, since the slaves are constantly requesting more work to the master. Thereby, when the master receives a request for more work, it interrupts the slaves that are working, in order to get more unexplored search-spaces, to send work to idle slaves. Hence, there are some slaves which can be idle for a long time.

The problem above described can be minimized, by creating a queue that contains unexplored guiding paths. Periodically, the queue is filled by slaves that have the smallest guiding path, providing the possibility of the master process to not interrupt the slaves that are working. When the master process receives some request, before interrupting some slave, it verifies if there exists some guiding path on the queue. Only in case there are none will the master interrupt the slave that has the smallest guiding path.

3.1.1.3 *Task Farm*

Task Farm (Gil et al., 2009) is another approach to dynamically distribute sub-formulas, in order to ensure workload balance. This method is implemented over a master-slave communication architecture. In this scheme, the master ensures the workload between slaves, by distributing one search-space per slave. When the slave finishes its analysis, it notifies the master in order to get more work. Likewise Sub-Section 3.1.4.3, the slaves receive the search-space to exploit, in form of guiding-paths. The search-space splitting can be performed either by decision heuristics or manually by the developer. In this strategy, when a slave ends its work it requests the master for more work – work that never was executed by any slave. This way, Task Farm does not steal work from a slave that is working, unlike Sub-Section 3.1.4.3 – in such strategy when a slave ends its work, it requests the for more work and the master steals from any slave which is working.

3.1.2 *Competitive Solving*

Competitive (Martins et al., 2012) SAT solvers, also known as portfolios, handle a SAT problem using the same formula for each computational unit, but with different strategies. Each computational unit applies different strategies to solve the problem, using, for that, different SAT solvers or different search parameters. These search parameters, used by SAT solvers, can vary between restarts, decision heuristics, polarity, and learning strategies. A SAT problem is solved in parallel by computational units, like a competition, finishing when one of them ends. Even though it is competitive, this strategy may apply cooperative strategies, in some cases changing information between itself. Learned clauses are part of the information that can be exchanged, in order to achieve a better performance. Although there is information exchanged between the computational units, the competitive strategy is easily distinguishable from the cooperative strategy, approached previously, due to the search space analyzed by them.

3.1.3 *Collaborative Solving*

This kind of approach combines sequential and parallel strategies, by performing in most cases the master-slave architecture. A SAT problem is analyzed sequentially by the master process and, the slaves obtain information to aid the sequential solving. In fact, the main collaborative SAT solver, the MultiThreaded SAT Solver (MTSS) (Vander-Swalmen et al., 2009), introduces two new concepts – *rich* and *poor* threads. The former performs as the master process, by executing the search sequentially. The latter are the slaves. The *poor* thread aims to aid the rich thread in the solving, by yielding logical or heuristic information to simplify its task.

This solving strategy is less used than cooperative or competitive, due to its implementation complexity.

3.1.4 *Parallel SAT Solvers*

Since the dissertation's main goal is to implement a parallel bounded model checker, the study of different parallel SAT solvers is required. Both the complexity of a SAT problem and the computer power evolution lead to the appearance of many SAT solvers using several approaches of SAT solving. This section presents several parallel SAT solvers that use different strategies to perform SAT solving. Both communication and scheduling SAT solving strategies are in focus below.

3.1.4.1 *cmcSAT*

cmcSAt (Marques et al., 2012) is a cooperative SAT solver, though based on the competitive strategy (also known as portfolios), with shared memory paradigm. The search-space is analyzed by eight threads based on the sequential solver MiniSAT (Eén and Sörensson, 2003). Each thread is configured in order to do not exploit paths, on its search-space, that are exploited by other threads. In order to achieve a better performance, threads cooperate between themselves, sharing information through the clause sharing procedure. Thus, this strategy is classified as cooperative. Although being a cooperative SAT solver, it is based on a competitive approach since all threads are launched with the same search space but with different configurations.

To guarantee that each thread exploits divergent paths, cmcSAT solver uses decision heuristics where each variable has a priority associated in order to exist more control about its search-space. Heuristics are responsible for choosing the next variable to be assigned, as well as its value. Nonetheless, in this SAT solver, the choice of the next assigned variable is also conditioned by its priority. VSIDS heuristic, one of the most used by SAT solvers, is also used in this solver.

Another feature underlying this SAT solver is clause sharing. This feature speeds up the search procedure, since it avoids conflicts that occurred earlier, by exchanging, between threads, the respective learnt clauses resulting of a conflict. To avoid the overhead of copying large clauses, the size of shared clauses is bounded by some value.

3.1.4.2 *PMSat*

PMSat (Gil et al., 2009) is the parallel version of MiniSAT (Eén and Sörensson, 2003) – a sequential SAT solver. Thereby, PMSat is a cooperative SAT solver, that adopts a distributed-memory paradigm, by using a Message Passing Interface (MPI).

Master-slave hierarchy is also used here, to divide the workload balance, which is ensured by Task Farm, that is, an approach to dynamically distribute sub-formulas. Specifically, the master is responsible for splitting the formula into sub-formulas and provide each sub-formula to its slaves. Once a slave ends, it notifies the master in order to receive more work, if there exists. Following the present idea, there exist two strategies to perform the splitting of formulas, either choosing the variables that occur on the biggest clauses or the most frequent variables.

As well as master-slave, clause sharing is also used in this cooperative SAT solver, though completely handled by the master. Slaves are responsible to send two kinds of information to the master: the learnt clauses and the results from sub-formula analysis. Following that, these two informations are sent together. Another role of the master is to keep all shared clauses sent by the slaves, and later resend them to the slaves, again. This process requires the master to do some processing, in order to not send clauses that already are on the slaves. This can produce an overhead, leading to a loss of performance, since it is a substantial quantity of information to handle and process. One way to avoid this, is limiting both number and size of the clauses that are shared, being this an optional procedure.

3.1.4.3 *ManySAT*

ManySAT (Hamadi et al., 2009) is a competitive (or portfolio) parallel SAT solver with shared-memory paradigm. ManySAT 1.0 was the first portfolio SAT solver with a multicore architecture, just working with four cores. Each core applies a lot of strategies to assist the solving, such as restart policies, heuristics, polarity choice algorithms and clause sharing.

The solving instances (core), by using sequential algorithms, communicate between themselves to a better performance. However, the major drawback of this solver is the number of instances that can be executed in parallel, which are only four. Each core performs the same search-space analysis, but with different strategies.

One of ManySAT's features is clause sharing, where the information is exchanged between solver instances. A clause may be shared in case it does not exceed a limit, defined until eight literals, in order to improve the performance.

3.1.4.4 *Plingeling*

Plingeling (Biere, 2010) is the multithreaded version of the sequential SAT solver Lingeling (Biere, 2010). This SAT solver is associated with the competitive class. Unlike ManySAT or cmcSAT, the number of threads is imposed by the user, there being no limit in the number of threads.

For each thread a separated SAT solver instance is generated, based on many factors such as the choice of random number seed, the effort allowed in different pre-processing algorithms and the decision heuristic. To sum up, there exist two crucial points about Plingeling. The first one is that shared clauses are limited to an unitary clause, and the second one is that when a thread ends its work, all the others end too.

3.1.4.5 *GridSAT*

GridSAT (Chrabakh and Wolski, 2003) was the first Grid SAT solver – which aim to acquire and release resources during the solving, as a way of improving solver power. This solver is the parallel version of sequential SAT solver zChaff (Mahajan et al., 2004). It uses a distributed-memory paradigm, and the workload distribution is achieved by dynamic divide-and-conquer.

Master-slave architecture is contained at GridSAT, being possible to dynamically increase the slaves. The master process is responsible for distributing the work over its slaves. At the beginning of the problem, the solving starts with just one slave. When this slave verifies that it is not capable of solving alone the problem, it sends a request to the master in order to create another slave. The slave requests the master when is running out of memory or has a sub-problem that is too big. In addition to sub-formulas, learnt clauses are also sent to the master, thereby enabling the clause sharing. There are also some important features to speed up the solving, such as intelligent backtracking and clause reduction. The major drawback is the number of slaves that it starts the solver procedure, which is just one, hence, the performance drops.

3.1.4.6 *SAT4J 2.1*

SAT4J 2.1 (Martins et al., 2010) is a hybrid parallel SAT solver, which combines both competitive and cooperative architectures, and extends the sequential SAT solver SAT4J 2.1.

In this specific solver, the solving process starts with a portfolio approach. During the initial process, each thread, through its heuristics, analyzes the variables more actively. Then, it is exploited with cooperative method, where the search-space is split based on the variables obtained earlier. Each search-space eventually reaches to its limit, causing the usage, again, of portfolio approach.

Besides shared clauses, there are many strategies to improve the performance of solving such as restart, polarity and learning simplification.

3.1.4.7 *JaCk-SAT*

JaCk-SAT (Singer and Monnet, 2007) is a cooperative SAT solver, that applies a technique named problem splitting. This parallel architecture cuts the variable set V into two subsets $V1$ and $V2$ of about equal size.

Let $\mathbf{SP} = (V, C)$ be the initial SAT problem (CNF formula), where V is the set of variables in the formula and C the set of clauses. Performing the decomposition, two sub-formulas $\mathbf{SP1} = (V1, C1)$ and $\mathbf{SP2} = (V2, C2)$ are obtained, and a residual clause-set $C3$ that is composed by the clauses that have variables in $V1$ and $V2$. The clause $C1$ only has variables inside the set $V1$, whereas $C2$ only has variables inside the set $V2$.

1. $Decompose(SP, SP1, SP2, V_{js}, C3)$
2. $SearchAllSol(SP1, SP2, S1, S2)$(executed in parallel)
3. $Join(S1, S2, V_{js}, S)$
4. $CheckSat(S, C3)$

Figure 13: JaCk-SAT algorithm taken from (Singer and Monnet, 2007)

Figure 13 illustrates the main steps taken by the algorithm. $Decompose$ is responsible for splitting the initial formula SP in two sub-sets $sp1$ and $sp2$.$V_{js} = V1 \cap V2$ and represents $Join$ variable-set. $SearchAllSol$ searches in parallel all solutions of disjoint sub formulas. Then the $Join$ is performed, where the solutions $S1$ and $S2$ are joined into a global solution S. Finally $CheckSat$ is carried out, where the global solution is checked with the residual clause $C3$.

$Join$ and $Check$ steps are performed in polynomial time over the number of solutions, but exponential time in terms of n. Thus, the performance is one of the major drawbacks of this method.

3.2 VERIFICATION OF SPLS USING DYNAMIC ALGORITHMS

Software product lines (SPLs) (Clements and Northrop, 2001), or software product line development, are a set of software systems that have particular features in common. SPLs are emerging, since they enable the optimization of processes in software engineering, by making software from reusable parts. Since this software engineering paradigm is increasingly applied in critical systems, both system modeling and system verification are required to guarantee total correctness in the system. The verification of SPLs is related to this dissertation's proposal, because an SPL represents multiple configurations of a certain system, the reason why it is approached as State of Art.

Recently, a novel technique, that consists in applying symbolic algorithms, was proposed to establish the system verification, by examining a set of states for each step. Each different system that composes an SPL is called a *product*, and a *feature model* (Schobbens et al., 2006) is normally built to represent its commonalities and differences. Unlike single systems, SPL model checking requires the verification of all products on its system, in order to verify whether any violates its specification. To express all products in a single model and make their verification, it was proposed in (Classen, 2010) and (Classen et al., 2010) a formalism that expresses the system behaviour and a new model checking algorithm. This formalism is called Featured Transition Systems (FTS) – a transition system where the transitions are labelled with features. Concretely, an FTS is a tuple $\langle S, Act, trans, I, AP, L, d, G \rangle$, in which each element is defined as follows:

- S is a set of states
- act is a set of actions
- $trans$ is a set of transitions between states and actions (act)
- I is the set of initial states
- AP is a set of atomic propositions
- L is a labelling function
- d is a feature model (N, px). N represents the set of features and px is set of products.

- G is a total function, labelling each transition with a feature expression.

Despite of FTS formalism, new kind of model checking algorithms was also proposed that outperform the enumerative approach. These algorithms, also known as FTS algorithms, exploit the FTS's structure in order to avoid an exponential number of verifications. Even though these algorithms improve the performance of SPLs, they rely on an explicit enumeration of the state space, in other words, the progress is made one state at a time – an exhaustive enumeration of the set of products. Furthermore, an FTS construction is not easy or friendly, thus the creation of a high-level language is required in order to be more used at industrial level.

In (Classen et al., 2014) an extension to handle such disadvantages was proposed by implementing symbolic algorithms, changing the fSMV language – a feature-oriented extension of SMV introduced in (Plath and Ryan, 2001) –, as well as its model checker, and finally it is proved the expressiveness equivalence between FTS and fSMV. In order to make the SPLs modeling process user-friendly, the fSMV was explored and a bi-directional translation between FTS and fSMV proposed.

Symbolic model checking of FTS uses Computation Tree Logic (CTL) to express products' properties of an SPL. To achieve that, an extension of CTL was created – the fCTL logic. This one allows the verification of temporal properties of a set of products, since CTL is extended with feature quantifiers. From this, symbolic algorithms were implemented for checking an FTS against an fCTL formula. Previous problems like enumerate and visit system states one by one, are solved with this kind of algorithms.

3.2.1 *fSMV Language and its Model-Checker*

SPLs are described in the SMV language, as the base system, in which a product is built by adding features in a certain order. The fSMV language, proposed in (Plath and Ryan, 2001), is an extension of the SMV language. According to (Classen et al., 2014), the fSMV language suffered an extension of its formal semantic, as well as its model checker, which was extended from NuSMV to fNuSMV.

An fNuSMV model consists of a set of variable declarations and assignments. The former defines the state space, whereas the latter defines the transition relation. To better understand that, a typical example of SMV will be presented, since this one is the base of fSMV language.

```
MODULE main
VAR
    request: boolean;
    state: idle, busy;
ASSIGN
    init(state) := idle;
    next(state) := case state = idle & request: busy;
                        true: {idle, busy};
                   esac;
```

Figure 14: SMV model taken from (Classen et al., 2014)

Figure 14 illustrates a controller that is either idle or busy, treating a request. The model is divided into three keywords – **MODULE**, **VAR** and **ASSIGN** – that represent its different sections. The keyword **MODULE** indicates the model's name, whereas the keyword **VAR** represents the section where the variables are declared. In this case there are two variables – `request` and `state`. The first one is a boolean that handles its requests, whether the second one represents the state, more precisely, whether it is idle or busy. On the other hand, the section that defines the value of the variables is defined by the keyword **ASSIGN**. In this section, `init` represents the system state value at its begin, which is idle. The assignment `next` defines the transition relation, by proceeding as a conditional statement. If the controller is idle and if exists a request, the controller will be busy and handle it. The expression {`idle, busy`} represents a non-deterministic choice between two values. Although it is not depicted on the example, the keyword **SPEC** defines the specifications, where it is possible to declare the model's property by using CTL.

In fNuSMV, the transition system is never built explicitly, as depicted in Figure 14, since it builds symbolic transition relation, which is a boolean function with two copies: one for the first state and another for the last, to each variable. An fSMV model is a base system and a list of features that are specified based on SMV language. This way, a feature declaration could be divided into the three following parts.

- **REQUIRE** has the function to define variables that the feature needs.
- **INTRODUCE** defines the new variables or new specifications in the system.
- **CHANGE** defines the changes performed in the existing variables. There are two kinds of changing, defined below.
  - **IMPOSE** a new definition of an existing variable.
  - **TREAT** existing variables differently.

A feature acts on base system, in order to change its behaviour. This leads to the creation of a new base system, by combining the specification of the feature and the base system. The fSMV language, as previously said, might be translated into FTS, as well as from FTS to fSMV. Due to fSMV's high level, the translation from FTS to fSMV is easier.

For a certain system **b** and a feature list **f1**, ..., **fn**, the composition $\mathbf{b} \odot \mathbf{f1} \odot ... \odot \mathbf{fn}$ represents a symbolic FTS. This process is achieved by a lifting technique, in which is added a boolean variable to each feature, and each changing that occurs in the feature is kept by feature variable. Moreover, from this composition system, and considering a base system and a feature, a new system is originated.

The fNuSMV tool and the composition process, effected by a script in php via command-line, are stand-alone (there is no communication between them).

In fNuSMV the features are independently specified, and a product is created by composition. The major difference between fNuSMV and NuSMV consists in implementations of a boolean function over the features variables, which represent the products that a certain property holds. This boolean

function is only possible because there is enough information to note what are the products that violate or satisfy a property. Hence, this is the main extension of NuSMV, once, this one executes the standard CTL model, checking algorithm.

3.2.2 *Results Analysis*

To compare this work with the previous one – the verification using symbolic algorithms against the verification approach that uses an exhaustive enumeration of the set of products –, a base system was specified (lift system), added some features, and finally verified the time spent in each one. The comparison was performed between fNuSMV – that uses symbolic algorithms –, and NuSMV – that applies an exhaustive enumeration of the products, using the normal CTL algorithm. fNuSMV tool outperforms the NuSMV tool in most of the features (in some cases it is faster about 400 seconds). Noteworthy, the model-checker paradigm has changed, however, the verification performance of SPLs might be considerable improved, as refereed above.

3.3 PARALLEL BOUNDED VERIFICATION

Parallel verification of Alloy models has been increasly researched, since their analysis, performed by Alloy Analyzer, is bounded. This means, that the analysis of some operation is bounded by a scope, more precisely, the search space is bounded. Hence, there exist certain assertions that need larger scopes (larger search space) in order to be possible to check them. On the other hand, the scope increase cause an exponential growth in the analysis time leading, in certain cases, to the exhaustion of machine resources such as the memory, and this way, the parallelization of Alloy models is required. Nonetheless, the temporal bounded verification imposes a bound equal to the max temporal trace length, however, the temporal semantics allows to have infinite temporal traces.

3.3.1 *TranScoping Technique*

The most usual strategy associated to parallelization of Alloy models is cooperative (Sub-Section 3.1.1), where different search spaces are solved in parallel by different computational units (either a core, a processor or computer). These search spaces (CNF formulas) are achieved by splitting the problem's formula into sub-formulas in which, a correct split might mean a considerable improvement of the performance.

Alloy models are translated into a CNF formula through its tool (Alloy Analyzer), combining the FOL formula and the bounds, and then, it is solved using an off-the-shelf SAT solver. The main challenge associated with cooperative strategies is the capacity of correctly distributing the original formula into sub-formulas, with the purpose of solving them in parallel. Hence, to reach this goal, it was proposed in (Rosner et al., 2013a) a technique, named *tranScoping*, that consists in exploiting the

original formula to small scopes, in order to achieve the best distribution to solve it in larger scopes. Therefore, various splitters – responsible for the splitting of the formulas – were exploited with two obligatory characteristics – *tranScopability* and *predictability*. The first one addresses the capability of a splitter to extract information generated from smaller to larger scopes – let **x** be the information retrieved from the analysis of a problem to a scope **i**, following the *tranScopability*, the **x** has to be used to accomplish the analysis of problems to a scope **j**, where $\mathbf{j} > \mathbf{i}$ –, whereas the second one ensures that the best splitter to a scope **j** remains the best splitter to smaller scopes than **j**.

Two kinds of splitters were proposed, the *VSIDS splitter* and the *"Field" Splitter*, wherein the second one gathers a set of splitters, since it corresponds to the split of the *fields* presented in the Alloy model. Both proposals of splitting will be presented on the next two subsections, followed by a discussion on how to choose the best splitter.

3.3.1.1 *VSIDS*

VSIDS is a decision heuristic used in most SAT solvers, which aims to select the next variable to be assigned in the solving process. To achieve that, a list that contains the frequency of each literal used is kept, where each literal has a counter that is set to zero at the beginning of solving process. When a clause is added (by learning) to the clause database, the counters associated to each literal inside the clause are incremented. The decision process is achieved by choosing the unsigned literal with higher ranking on the list of literals.

The major drawback associated to this method is the fact of being performed sequentially, thereby limiting the splitting of formulas to larger scopes. As the sub-formulas analysis is carried out based on the variables presented in the ranking, to enable the analysis of larger scopes it was also proposed the analysis during five or ten seconds, in order to manipulate the ranking produced after that time.

3.3.1.2 *The "Field" Splitting*

Field splitting is another proposal to split a formula into sub-formulas. Any relation in Alloy, or field, in the translation process into its propositional formula is represented by a matrix, which contains propositional variables and the dimension is given by the scope, designated in Alloy.

Given a *signature* **A** – `sig A {field: B}` – and a *check* command – `check assertion for n but 3 A, 2 B` –, in case of existing counter-examples, **A** signature has $\{\mathbf{A_1}, \mathbf{A_2}, \mathbf{A_3}\}$ as domain, whereas **B**'s domain is composed by $\{\mathbf{B_1}, \mathbf{B_2}\}$. The resulting matrix that corresponds to the representation of this field has dimension 3(**A**) * 2 (**B**), defined as follows (considering **P** as a proposition variable, $\mathbf{n} \in \{1,2,3\}$ and $\mathbf{i} \in \{1,2\}$):

$$M_{n,i} = \begin{bmatrix} P\mathbf{A}_0\mathbf{,B}_0 & P\mathbf{A}_0\mathbf{,B}_1 \\ P\mathbf{A}_1\mathbf{,B}_0 & P\mathbf{A}_1\mathbf{,B}_1 \\ P\mathbf{A}_2\mathbf{,B}_0 & P\mathbf{A}_2\mathbf{,B}_2 \end{bmatrix}$$

A propositional variable P$\mathbf{A}_n$,$\mathbf{B}_i$ is true if (A_n,B_i) belongs to the binary relation *field*, since $field \subseteq (\{\mathbf{A_1}, \mathbf{A_2}, \mathbf{A_3}\} \times \{\mathbf{B_1}, \mathbf{B_2}\})$. To achieve the splitting goal, respecting the complexity solving, it is required to select the propositional variables from matrix – by selecting the propositional variables from the bottom-right entry until top-left. More precisely, based on the matrix above depicted, the variables selection respects the following order:

1. P$\mathbf{A}_2$,$\mathbf{B}_2$
2. P$\mathbf{A}_2$,$\mathbf{B}_0$
3. ...
4. P$\mathbf{A}_0$,$\mathbf{B}_1$
5. P$\mathbf{A}_0$,$\mathbf{B}_0$

Considering a list of propositional variables, such as above enumerated, the splitting of a formula into sub-formulas is based on this distribution. Obviously, given an Alloy model, all fields are split and the best way to distribute them is chosen based on some factors, which will be discussed below.

3.3.1.3 *Selecting the Right Splitter*

The proposal of this approach is to find the best splitter to perform the analysis of larger scopes, based on information extrapolated from the analysis of smaller scopes. Therefore, any splitter provides a set of information related to its performance in order to choose the best one. This information is defined by the following macros (considering **S** as the *splitter* and **l** as the *scope* of any problem) :

- $\mathbf{NUM}_{S,l}$ is the number of sub-problems which the splitter has partitioned
- $\mathbf{MAX}_{S,l}$ is the maximum time that takes to analyse a sub-problem
- $\mathbf{AVG}_{S,l}$ is the average time to analyse all sub-problems
- $\mathbf{SUM}_{S,l}$ represents the time to analyse all sub-problems
- $\mathbf{DEV}_{S,l}$ represents the standard deviation of the time analysis of all sub-problems
- $\mathbf{MED}_{S,l}$ is the median of the time analysis of all sub-problems

The most important parameter, presented on a splitter technique, is the **MAX**. In most of the cases, the splitter is not the best one to perform in larger scopes, if this parameter closely approximate of the time required to solve the original formula.

SUM is another important parameter, since it determines the total time spent in solving all sub-problems, thus the higher the **SUM** that the splitter presents, the less likely it is to be chosen as the best splitter.

Below, it is shown an example of choosing the best splitter by analyzing small scopes, in particular to a certain assertion with `Identifier`'s scope set to 6 and based on the *signature* in Figure 15, it is produced the Table 2 , which is sorted by ascending order of **MAX**, since it is the most important parameter. Each row corresponds to a different splitter, such that each `D.field` (abv. of `Domain.field`) represents the exploration of propositional variables, produced recurring to the respective matrix that represents a field. Based on these results, the analysis of larger scopes, by performing the splitting through `D.srcBinding`, leads to better performances. More precisely, for instance to scope 10, the sequential analysis takes more than 15 days, while the parallel, by performing `D.srcBinding` splitting, takes 1053.48 seconds (17.558 minutes), showing a huge difference between both approaches.

```
sig Domain {
   routing:space → endpoints
   dstBinding:Identifier → Identifier
   srcBinding:Identifier → Identifier,
   AdstBinding:Identifier → Identifier,
   BdstBinding:Identifier → Identifier
}
```

Figure 15: An excerpt of an Alloy model to illustrate the tranScoping technique

Splitter	NUM	MAX	AVG	SUM	DEV	MED
D.srcBinding	77	0.08	0.02	1.45	0.02	0.01
D.BdstBinding	77	0.09	0.02	1.50	0.02	0.01
D.dstBinding	192	0.18	0.04	7.67	0.03	0.03
D.routing	102	0.21	0.02	1.98	0.03	0.01
VSIDS	228	0.49	0.01	3.55	0.05	0.00
D.AdstBinding	192	1.22	0.05	0.78	0.10	0.02

Table 2: Parameters retrieved from the splitting analysis

Although these parameters provide a greater knowledge about each splitter and the parameter **MAX** indicates, in most of cases, the best splitter, there is no exact way to choose the most appropriate splitter, since it would require the manual analysis of a set of parameters.

To sum up, this approach improves the performance of analysis time of Alloy models. On the other hand, it contains two main challenges in order to achieve it – the first one is to ensure that the best splitter to smaller scopes is the best way to exploit the model to larger scopes, whereas the second one is the capability of choosing the best splitter based on splitter's parameters (**NUM**, **MAX**...).

3.3.2 *Ranger Technique*

Ranger is a technique, introduced in (Rosner et al., 2013b), to parallelize the analysis of Alloy models. This approach establishes a linear ordering on the state space of all candidate solutions that would be analyzed by Alloy Analyzer. According to this linear order and based on the restrictions of an

Alloy model, it is possible to create ranges of candidate solutions with the purpose of solving them in parallel, and this way, splitting the original problem into independent sub-problems.

The state space is represented by a *vecSpec*, where a respective state or *configuration*, is represented as a concrete entry in the vecSpec. A range is composed by a set of configurations, where each one represents a possible solution to the problem. To obtain all that information, the Ranger technique has to interface with the Alloy Analyzer to get the binary relations' list, and also with Kodkod in order to produce the vecSpec. Moreover, it obtains the CNF file resulting from the model's translation, since the restrictions are injected directly at clause level to each range. To better clarify, the main phases of this technique are presented below.

3.3.2.1 *Establishing a linear ordering over Configurations*

The Alloy Analyzer translates an Alloy model into Kodkod, where an uniform naming for the atoms is produced, based on the scope defined in the Alloy model. For instance, to this Alloy command `check assertion for n but exactly 5 Node`, it is produced a naming table defined as follows.

Sig	**scope**	**naming**
Node	5	Node$0,...,Node$4

Table 3: Naming table produced by Kodkod

Based on that naming Table 3, the information contained on the list of binary relations and others taken from Kodkod, such as a copy of the atom universe and the details about the appearing of the atoms in the relation's domain and range, a vector specification is built – known as vecSpec.

Figure 17 illustrates the vecSpec built based on the Alloy model depicted in the Figure 16, which is an excerpt of a model to binary trees, on its left. A vecSpec describes the state space and is used to retrieve the state space of configurations. It consists in a mapping between $AtomNames \times RelNames$ and $P(AtomNames)$, for instance, based on the vecSpec depicted below, given $Object\$0 \times root$ the result of computing $Object\$0.root$ is the set $\{null\$0, Node\$0, ..., Node\$4\}$. Observing the $vecSpec$, it is easily understandable that a configuration represents the procedure of choosing some concrete value for each $vecSpec$'s entry, taken from the result of applying the aforementioned map to each entry.

```
one sig null {}
abstract sig Object {}
sig BinTree extends Object { root :  Node
+ null }
sig Node extends Object { left, right :
Node + null }
...
check assertion for 0 but 1 BinTree,
exactly 5 Node
```

Figure 16: An excerpt of an Alloy model to modeling binary trees

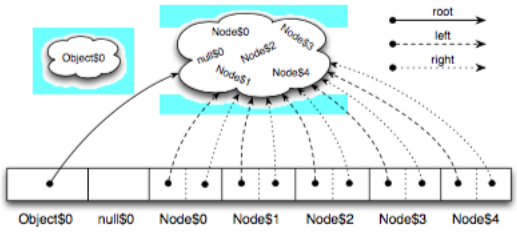


Figure 17: Graphical representation of a vecSpec taken from (Rosner et al., 2013b)

This technique takes the strict linear ordering, which Kodkod imposes for all atoms, denoting it by $<_K$, and extends it to a lexicographical ordering between configurations, expressed by $<_{LC}$. This way, Ranger has a total control over configurations, enabling the procedure of *nextConfig*.

3.3.2.2 *Range Partition*

The basis of this technique is splitting the state space into non-overlapping intervals, named as *ranges*, where a correct splitting is required in order that all ranges have the same number of configurations.

Configurations are considered partition points, behaving as boundaries between ranges. In particular, to a certain range $\mathbf{R} = [C1, C2]$ (with $C1 <_{LC} C2$) and a positive number $\mathbf{n}$ of splits, this technique is capable of producing $\mathbf{n}$ configurations as partition points from $\mathbf{R}$ – given $\mathbf{n}$ partition points, a list $PP_1, .., PP_{n-1}$ of configurations is drawn from $\mathbf{R}$ with the purpose of achieving the following set of ranges $\{[C1, PP_1], [nextConf(PP_1), PP_2], ..., [nextConf(PP_{n-1}), C2]\}$, where $nextConfig(C_i)$ is the next configuration after C_i, based on the linear ordering $<_{LC}$.

To establish such splitting, there exist three main algorithms. The first one, given two configurations and the vector specification, returns the partition point between them. The second one exploits the linear ordering of the state space, since it can ensure the next configuration based on the linear ordering $<_{LC}$. Considering such algorithm, given a configuration Cf_1 and the vector specification, it returns a configuration Cf_2 such that $Cf_1 <_{LC} Cf_2$. The last one is the main algorithm, since it retrieves all ranges with the purpose of solving each one in parallel. It retrieves a list of ranges based on the vector specification, the number of partitioning points and the global range (between the first and last configuration). In this procedure are invoked the first two algorithms one time by each iteration, leading to the production of two ranges at a time. More precisely, for each iteration, the first and second algorithms are responsible for splitting one range into two – in particular $[C1, C2]$ is split into the ranges $[C1, C]$ and $[nextConf(C), C2]$

3.3.2.3 *Parallel Range Solving*

This technique performs its parallel analysis following the master-slave architecture, by using a cluster of computers – each computer represents a slave, which executes each sub-problem sequentially.

Kodkod has a technique, called symmetry-breaking, that eliminates the isomorphic models of the problem, leading to reduce the search-space. Thus, the symmetry-breaking has a direct impact on the Ranger's performance, since it reduces the number of configurations. The algorithm that performs the analysis of a sub-problem is defined as follows: given a range and the vector specification, it identifies what are the responsible clauses, drawn from CNF file, to perform the searching between the range. Once the clauses are identified, they are solved using an off-the-shelf SAT solver.

There are two approaches to accomplish the range procedure during the parallel analysis – one is the *flat range partitioning* and another is the *recursive range partitioning*. The former consists in determining a large enough value of **n** ranges in order of being solved in parallel by the available workers. This way, a worker is required to perform the full range into **n** ranges using algorithm *rangePartitioning*. The latter is an approach to handle the main drawback of the aforementioned technique – this one accomplishes the range partitioning using a static way. The recursive technique is dynamic, since it identifies the location of nontrivial sub-ranges. In particular, the worker that is performing the analysis of the oldest active sub-problem, splits its work with the purpose of distributing it, to other workers.

4

KODKOD TEMPORAL EXTENSION

The Electrum framework aims to ease the specification and verification of dynamic models – models that hold mutable variables. To achieve that, its language provides temporal operators, which make the division of the static from temporal part explicit to the developer. This temporal extension to the Kodkod constraint solver is necessary in order to provide a clearer embedding of Electrum into Kodkod. As this language (Electrum) supports the temporal modeling, it needs a back-end that support it as well. Moreover, as the optimization of the verification procedure is an important goal, this extension performs the first step of such new verification procedure – the division of the dynamic from the static properties.

The new Electrum's bounded model-checker, that will be presented in Chapter 5, translates directly its models to Kodkod, thereby avoiding the translation to Alloy as middle step. This will allow the clarification of Electrum's semantics. Both the syntax and the semantic of these temporal operators follow standard LTL (Huth and Ryan, 2004). Although the LTL semantics is unbounded, with the purpose of achieving the bounded model checking, it is performed the embedding proposed in (Biere et al., 2003), as already presented in Section 2.4.

Hence, it is required to perform some steps such as, convert the original formula into its NNF, the translation of temporal formulas into the standard FOL and finally, the splitting of static from the temporal part of the Kodkod model – such splitting is a crucial step for the purpose of applying the new parallelization procedure approached in Chapter 6. Note that, the necessity of translating a certain formula into its NNF is related to the bounded model checking procedure, in particular, for the bounded semantics the duality between the operators *always* (**G**) and *eventually* (**F**) no longer holds, according to Biere et al. (2003). Furthermore, as the standard Kodkod's AST does not support the LTL operators, the first step is to extend it and obviously, to achieve the temporal solving process, both the variable relations and the temporal operators must be expanded internally into the standard Kodkod.

The procedures aforementioned are described in the remaining chapter, starting with a description of the Kodkod's AST extension in Section 4.1, then, an architecture description in Section 4.2, the bounds expansion in Section 4.3, the NNF conversion in Section 4.4, the temporal operators translation in Section 4.5 and finally the splitting of the temporal from the static part of a certain problem is approached in Section 4.6.

It is noteworthy that, despite of being used as the Electrum back-end, this temporal extension is properly expressive and flexible (like the standard Kodkod) to be used directly by the users.

4.1 TEMPORAL KODKOD LANGUAGE

The Kodkod does not support the typical LTL operators, thus, its AST was extended to do it. This extension was conceptualized with the purpose of establishing the natural difference that exists between the LTL operators, more precisely, between the binary ones and the unary ones, but also, the difference between the operators which apply to formulas and the ones that apply to expressions.

For the purpose of Kodkod supporting the LTL operators, both formulas and expressions were extended. The expression's extension, the `TempExpression` visitor, supports primed expressions. To deal with the LTL operators that apply to Formulas, two visitors were created – the `UnaryTempFormula` and the `BinaryTempFormula`. The former handles the unary ones, the well known LTL operators composed by $\{\mathbf{X}, \mathbf{G}, \mathbf{F}, \mathbf{P}, \mathbf{H}, \mathbf{O}\}$. On the other hand, the latter represents the binary ones – the **U** and the **R**.

In addition to operators support, the typical Kodkod relations were also extended to allow a new type – the temporal relations. This way, the visitor `VarRelation` was created to support this kind of relations into regular Kodkod. It is noteworthy that, `VarRelations` are relations whose its value can variate in the time, contrary to the normal relations, which are considered statics. Furthermore, in the translation process (described in this chapter), to this kind of relation is indexed, as range, the relation `Time`, turning it into a mutable relation.

4.2 ARCHITECTURE

As mentioned in Sub-Section 2.3.3, a certain Kodkod problem is composed by a *formula* (that represents the constraints over the relations), an *universe* (which constitutes the atoms of the problem), and by the *bounds* that are formed by tuples of atoms (drawn from the universe) to each relation presented on the model.

Figure 18 depicts the temporal extension architecture. Specifically, it illustrates the process of translating an LTL temporal formula into the standard Kodkod FOL.

The first step of this process consists in applying the `NNF(F)`, which represents the conversion of the formula `F` to its Negative Normal Form. After that, in this process, new relations are created for each `VarRelation` in `F`, but with, one more value of arity (but keeping the name), for the purpose of indexing relation `Time`. Furthermore, it is carried out the translation of `F` into the standard Kodkod FOL – the expansion of the temporal operators following the rules previously mentioned. Finally, it is carried out the splitting of the temporal from the static component – this step is required due to the parallelization procedure present on the new Electrum back-end, which will be exploited in detail in Chapter 6.

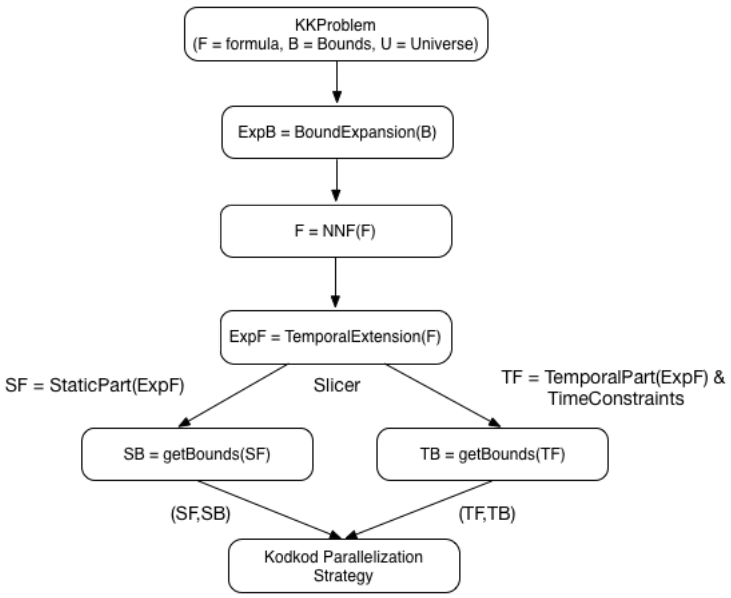


Figure 18: Kodkod temporal extension architecture

The static part of `ExpF` are the formulas that have no `VarRelations`, on the other hand, the temporal part of `ExpF` represents all formulas that have at least one `VarRelation`. After finishing the slicing procedure, two new kinds of bounds is created – the `SB` (Static Bounds) and the `TB` (Temporal Bounds). These bounds are built based on the relations that appear in each formula.

Moreover, it is noteworthy that the formula `TF` has associated the `TimeConstraints`, as depicted in Figure 18. These constraints are required to achieve the bounded verification according to Biere et al. (2003), previously discussed in Section 2.4. The relations created to establish these constraints are the following: `init`, `last`, `Time`, `next`, `loop` and `nextt`. Obviously, the first four relations, which are related to the time, are the most important of this procedure.

In particular, the first two relations are the first and the last state, assuming that there is a total order over all states/times. Moreover, the relation `Time` is considered as the state, whereas relation `next` represents the total order created from `init` until `end`. Obviously, the aforementioned relations are responsible for simulating the evolving of some problem in the time. Only the `next` relation is binary, since it corresponds to the product between two times and the remainder relations are unary. The creation of such relation is illustrated below, provided by the Kodkod implementation.

```
Relation Time = Relation.unary("Time");
Relation init = Relation.unary("init");
Relation end = Relation.unary("end");
Relation next = Relation.binary("next");
Formula totalOrder = next.totalOrder(Time,init,end)
```

Listing 4.1: `TimeConstraints` relations

The `loop` and the `nextt` relations are necessary to carry out the verification of finite-state systems, according to Biere et al. (2003), to the temporal operator `always` – detailed explanation in Section 2.4.2. This way, the implementation of the required `loop` is depicted below, implemented in Kodkod. Both `loop` and `nextt` relation are binary. The first one is the relation between the last state and some previous state, whereas the second one is the union between the `next` (showed above) and the `loop` with the purpose of, in all states, being possible to exist a `loop`. As a requirement to expand the operator **G**, the relation `infinite` is declared as `loop.one()`.

```
Relation nextt = Relation.binary("nextt");
Relation loop = Relation.nary("loop", 2);
Formula loopDecl =  loop.partialFunction(end,Time);
Expression nextDecl = next.union(loop);
Formula nextFnct = nextt.eq(nextDecl);
Formula allStuff = Formula.and(loopDecl,  nextFnct);
Formula infinite = loop.one();
```

Listing 4.2: `TimeConstraints` formulas

4.3 BOUNDS EXPANSION

The bounds expansion is the first step of this temporal extension. In this procedure new bounds are created, where to the bounds of variable relations is appended the relation `Time` and the time relations (mentioned in previous section) are bounded. Moreover, to the universe's bound is added **n** new atoms, where **n** is the maximum number of the trace's length received as input. Hence, the bounds of static relations should remain unchanged.

On the other hand, to append the `Time` relation to the variable relations, increase its arity is a requirement. Therefore, the creation of new variable relations is necessary, where the relation name is kept but the arity is increased. Obviously, the old variable relations are replaced by this new one. Beside that, it is built a map between the old and the new variable relations in order to, in the translation of the temporal operators (Section 4.5), the old variable relations being replaced by the new ones.

To clarify this procedure, a brief example will be presented. Considering the `originalBounds` the original bounds, `numberOfTimes` the maximum number of the trace's length and both received as input:

```
originalBounds.universe = {a1,a2,b1,b2}
originalBounds = {var A : [a1,a2], B : [b1,b2]}
numberOfTimes = 3
```

The `originalBounds` have two relations, the variable relation `A` and the static relation `B`. Moreover, each one have a set of atoms drawn from the `originalBounds.universe`. The first step is to create new bounds (`newBounds`), but also to create new `numberOfTimes` atoms with the aim of add them to its universe:

```
newBounds.universe = originalBounds.universe + {t1,t2,t3}
```

This way, the new bounds have on its universe the atoms responsible for the bounding of the `Time`. Once the universe is filled with temporal atoms, it is time to create new variable relations.

```
VarRelation V = VarRelation.unary("A")
```

The variable relation `V` is created to replace the old relation `A`. This new one keeps the name, but now is binary since `A` was unary. Thereafter, the relations are bounded.

```
Relation Time = Relation.unary("Time")
newBounds = {Time: [t1,t2,t3], var A : [a1,a2].product([t1,t1,t3], B : [b1,b2]}
```

This illustration example just only mentions the bounding of the relation `Time`, but beyond this one, there are more five temporal relations (`init`, `last`, `next`, `loop`, `nextt`) that are bounded in the procedure, as referred in the previous section. Following the same thought, the relation `Time` is filled by the atoms earlier created and added to the `newBounds.universe`. On the other hand, to the variable relation `V` is appended the relation `Time`, creating a cartesian product between the atoms of `V` and the atoms of `Time` (`product` function represents the it). From this results six possible combinations – `[a1,t1],...,[a2,t3]`. Finally, the bounds of the relation `B` are kept, since it is static.

4.4 NEGATION NORMAL FORM TRANSFORMATION

The NNF transformation, as mentioned in Section 2.4.2, is a required step in this temporal extension, to ensure a correct translation of the LTL operators into FOL.

A certain formula is in NNF, if the negations only occur in front of atomic propositions. The Figure 9 illustrates the rules to transformer a formula to its NNF. This procedure is achieved by applying visitors over the original formula. Therefore, the formula is analysed from the root of the formula until the its leafs, by traveling the respective structure of its AST (Abstract Syntax Tree).

Each formula is changed in the descending exploration of the analysis – the negations are added to formulas (if necessary based on the rules) and the formula operators are also changed. Nonetheless, in the ascending exploration, the verification of the unchanged negations of the original formula is carried out and verified if they are removed or not.

From Figure 9, the identification of the responsible visitors of transforming any formula to its NNF is quite simple, being them – `NotFormula` $(\neg\phi)$, `BinaryFormula` $(\phi\{\wedge|\vee|\rightarrow|\Leftrightarrow\}\psi)$, `BinaryTempFormula` $(U|R)$, `UnaryTempFormula` $(X|G|F)$, `QuantifiedFormula` $(\forall|\exists)$ and `NaryFormula` $(\phi\{\wedge|\vee\}\psi\{\wedge|\vee\}\gamma...)$.

The Figure 19 depicts an excerpt of the algorithm responsible of transforming a formula into its NNF. This way, just for illustration, the presented visitors are only three – `NotFormula` (line 2), `BinaryFormula` (line 8), and `UnaryTempFormula` (line 14). In order to deal with the current context on the transformation of a certain formula, a FIFO (First In First Out) is used to keep, in any visitor, its current context with booleans, where if the current context is `True` means that the

Input :A Formula ϕ **Output** : $NNF(\phi)$

```
1: while(thereAreFormulasToVisit)
2:     visit(NotFormula φ)
3:         pushBoolean(True)
4:         f = φ.formula().accept(this)
5:         if( getBoolean() ) popBoolean(); return f.not()
6:         else               popBoolean(); return f
7:
8:     visit(BinaryFormula φ)
9:         if( getBoolean() ∧ φ.op() == AND)
10:            setBoolean(False)
11:            return φ.left().not().accept(this).compose(OR, φ.right().not().accept(this));
12:        ...
13:
14:    visit(UnaryTempFormula φ)
15:        if( getBoolean() ∧  φ.tempOperator() == ALWAYS )
16:            setBoolean(False)
17:            return φ.formula().not().compose(EVENTUALLY).accept(this);
18:        ...
19:    ...
```

Figure 19: Excerpt of the NNF algorithm

current formula in analysis is negated, otherwise the current formula in analysis is not. Therefore, this auxiliary structure performs a crucial role in the transformation procedure. Moreover, there exist four main operations with the purpose of dealing with the auxiliary structure, defined as follows:

- `pushBoolean(Bool)` - adds to the head of the list a boolean value.
- `setBoolean(Bool)` - sets the head of the list
- `popBoolean()` - removes the head of the list
- `getBoolean()` - returns the head of the list

Given a certain formula $\neg\phi$, that is negated, the verification process starts in the visitor `NotFormula` (line 2), where the variable `True` is added to context. Thereafter, it is applied the visitor in the formula ϕ (line 4). In case of ϕ being, for instance, a binary formula which `AND` is the operator, the current value of the context is changed to `False` (line 10) and a new formula is returned (line 11) with the new binary operator. As the operator has changed, both right and left sub-formulas are negated, according to 8. The aforementioned step is the descending phase, on the other hand, the ascending phase starts when the exploration of ϕ reaches the leafs of the AST. In this ascending phase, as the context of the first negation is false, ϕ is returned (line 6) not negated. All visitors follow this strategy, as for example the `UnaryTempFormula` (line 14).

It is noteworthy that, the formulas $\neg\gamma$, where $\gamma \equiv \neg\phi$ are directly handled in the visitor `NotFormula` (line 2). Here is made a verification to validate if the γ is `instanceof` of `NotFormula`, in case of true it is returned the formula γ.

Obviously, the context works properly due to the equal number of calls of two important functions – `pushBoolean` and `popBoolean`. For each push, there exist always a pop, reason why, the context is always updated.

4.5 TEMPORAL OPERATORS TRANSLATION

As already mentioned before, one of the Electrum back-end procedure is to expand the temporal operators for explicit quantifiers over the states in a trace/path. This expansion follows the rules referred in Sub-Section 2.4.4.

Figure 20 illustrates the two of the most important visitors associated to this translation process – the `UnaryTempFormula` and the `Relation` visitors. In the first one the formulas that contain unary operators are translated, whereas in the second all relations presented in a certain model are addressed, with special focus in the variable relations. The remainder visitors not illustrated in the example (`BinaryTempFormula` and `TempExpression`) follow the same strategy.

Input : A Formula ϕ
Output : The formula ϕ translated according to Sub-Section 2.4.4

```
1:  while(thereAreFormulasToVisit)
2:      visit(UnaryTempFormula φ)
3:          pushOperator(φ.op())
4:          pushVariable()
5:          ψ = φ.formula().accept(this)
6:          α = getQuantifier(getOperator(), ψ, maxPostDepth)
7:          popVariable()
8:          popOperator()
9:          return α
10:
11:     visit(Relation r)
12:         if( r instanceof VarRelation )
13:             return getRelation(r).join(getVariable())
14:         else return r
15:     ...
```

Figure 20: Temporal translation algorithm

Hence, to support this mechanism were created two control structures – a FIFO that contains the current variable/expression and another that contains the current temporal operator. The first one is required to know, in the translation process, the current variable or expression quantified over the temporal operator and compose it with any variable relation that appear in the formula. For each temporal formula or expression is created a variable and bounded/quantified according to the temporal operator. On the other hand, the second one is necessary to know what is the current temporal operator, to then, expand it according to its translation's definition (Sub-Section 2.4.4). In this case, a FIFO is necessary to handle the cases in which there are a nesting of temporal operators (temporal operators contain temporal operators).

These control structures are accompanied by the following auxiliary functions to manipulate them.

- `pushVariable() | pushOperator(operator)`
- `popVariable() | popOperator()`
- `getVariable() | getOperator()`

The function `pushVariable()` adds a variable/expression to the FIFO, by respecting some constraints. If there are no variables in the variables control structure, the relation `init` (first state on the total order over the time) is added to the FIFO, otherwise it creates a certain variable `tx`, where the `x` is incremented where the function is invoked. Moreover, there are more restrictions, for instance, if the current operator is the `x`, the expression `getVariable().join(next)` – the next state from the current state – is added to the FIFO. It is noteworthy that, in Kodkod, the function `join()` is the well-known relational operator dot join or composition. Obviously, the remaining two functions, responsible to handle this list, have the function of removing the head of the list (`popVariable()`) and to return the head of the list (`getVariable()`). It is easily denoted that the functions which handle the operator control structure are analogous to the variable ones, with the exception of the `pushOperator(operator)` function, since the operator received as input is just added to the list.

This way, supposing that there exists a certain unary temporal formula ϕ ready to be expanded. The first step of such analysis is in the visitor `UnaryTempFormula` (line 2). After that, the temporal operator is added to the operators list (line 3) and a variable, that is quantified over the time, is created (line 4). The next two steps are to analyze the formula ϕ (line 5) and to translate the unary LTL operator to FOL following the rules presented in Sub-Section 2.4.4 (line 6). To achieve that, the function `getQuantifier` receives as input the current operator (`getOperator`), the resulting formula (ψ) of exploring ϕ and the number of primed expressions (" ' ") that were verified in the analysis of ϕ. Thus, in case of, for instance, the `maxPostDepth` is two (`exp.post().post()`), the formula must have which force the existence of two next states from the current one – `some t.nextt.nextt`. Obviously, the `maxPostDepth` is handled in the visitor responsible for the temporal expressions : the visitor `TempExpression`. Finally, the expansion of ϕ ends when the current variable and operator are removed from the list, `popVariable` (line 7) and `popOperator` (line 8) respectively.

Furthermore, mentioning another case not shown in the figure, such as in the case of analyzing a formula with the operator **U**. In this case, before to explore its sub-formulas, the function `pushVariable()` is twice invoked since there are at least two different quantifications. Following the same thought, there are also two invocations of the function `popVariable()` at the end of expansion.

Focusing now on the relations treatment, obviously it is made in the visitor `Relation` (line 11). In this visitor is performed the composition of the variable relations with the current temporal expression/variable. In this case, the composition of a certain unary relation r, is accomplished if r is a

variable relation (`VarRelation` – line 12). Nonetheless, such composition is achieved with the aid of the function `getRelation` (line 13) – function that, given *r*, returns its expanded variable relation (relation with higher arity, but keeping the same relation's name), created in the bounds expansion in Section 4.3. Note that, to the variable relation a composition (*join*) with the current time instant (basically a variable that quantifies the current time instant) is performed. Therefore, the temporal variable is appended to the expanded relation of *r*, that is binary and this way there is no any arity error.

For the purpose of illustrating in detail how this procedure works, an minimal example will be explained. Such example, which is presented below, is composed of: a simple Kodkod variable `v`, two relations – one variable (`varRel`) and one static (`staticRel`) and a formula `f`. However, the fifth line, the last one, represents the pretty printer (`f.toString()`) of `f`, giving a better understanding of `f`.

Concretely, `f` represents a quantification of `v`, which is of type `staticRel` , over a temporal formula. This temporal formula has the `always` operator, and the operator of its sub-formula is the `eventually`. Inside of such sub-formula, there is a primed expression - `varRel'`.

```
Variable v = Variable.unary("v");
Relation staticRel = Relation.unary("staticRel");
VarRelation varRel  = VarRelation.unary("varRel");
Formula f = v.in(varRel.post()).eventually().always().forAll(v.oneOf(staticRel));
//(all v: one staticRel | always(eventually((v in varRel'))))
```

Above is depicted a formula resulting of expanding the formula `f` into the standard FOL, considering the well-known rules previously approached.

```
//(all v: one staticRel |
        (one loop ∧ (all t0: one (init . *next) | (some t1: one (t0 . *next) |
            (some (t1 . next) ∧ (v in (varRel . (t1 . next))))))))
```

The formula above depicted is produced after crossing the respective AST of `f`. Such crossing starts by analysing the visitor `QuantifiedFormula` – this visitor approaches the quantifications over a certain formula. After that, the interior formula is handled. This way, the next visitor to be visited is the `UnaryTempFormula` and the **`always`** operator is expanded following its meaning, thus, it is is created such expression – `(one loop ∧ (all t0: one (init . *next) |...` . Following that, the next sub-formula is analysed by visiting again the visitor `UnaryTempFormula`, since **`eventually`** operator is analysed there. In such expansion the current quantified variable is `t0` (created in the first expansion), thus, a new variable `t1` must be created that quantifies over `t0.*next`. Hence, considering the meaning of the **`eventually`** operator, it is expanded to the following expression `(some t1: one (t0 . *next) | ...` . Note that, when a variable relation is analysed, it is necessary to perform the *join* with the current instant time. Moreover, to the variable relation (`varRel`) in the formula that are quantified be the **`eventually`** a primed expression is applied, thus, the relation is expanded as follows `(varRel. (t1 . next))`. If there was not a

primed expression, the expansion would be `(varRel . t1)`. After analysing the leafs of the AST, it is performed the backtrack and the formula above depicted is formed.

4.6 SLICING PROCEDURE

The original formula slicing is a crucial step, since the parallelization process, which will be handled in Chapter 6, works based on two original formula components – the temporal and static component. This way, the original formula slicing is at top level formula conjunctions. A particular sub-formula is considered temporal if contains any `VarRelation`, otherwise it is considered static.

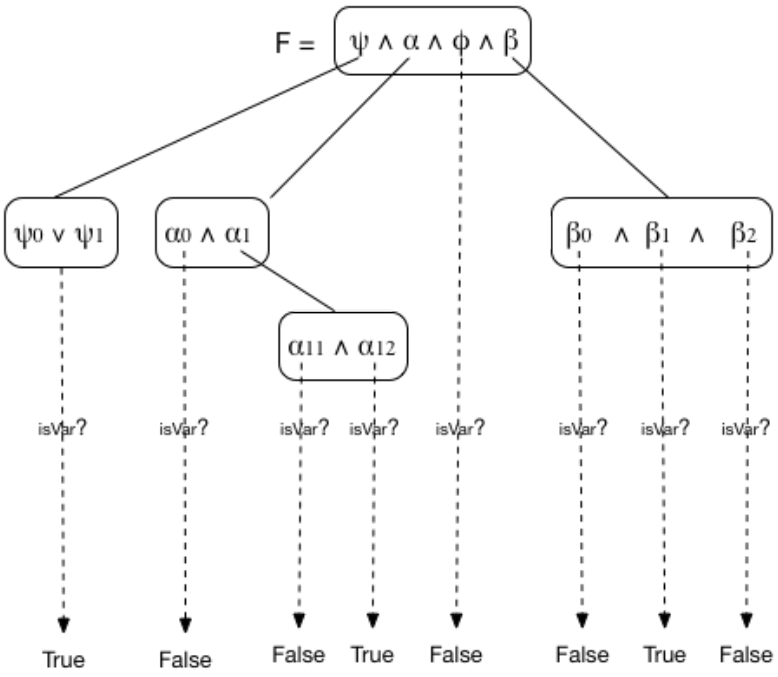


Figure 21: Slicing strategy

Figure 21 describes the formula `F` composition at conjunctions level, as well as its AST. Given a certain formula, the temporal relations verification is performed when there are no conjunctions to expand. As depicted at the Figure, when a formula is a conjunction it is expanded, otherwise it is carried out to the temporal relation verifications. It is noteworthy that, in the figure is presented the formula ψ expansion, just to illustrate that, formulas with different operators from conjunctions are analysed with the purpose of verifying any temporal relation apparition. In the AST depicted below, the boolean value `True` denotes that the formula contains at least a `VarRelation` – considered temporal formula – , otherwise the formula have no any `VarRelation` and it is considered static.

The output of this procedure consists in two formulas – a static and a dynamic –, and each one is composed by the conjunction of all formulas, according to the `VarRelation` presence. This way,

the application slicing output is a temporal formula `T` and a static formula `S`. The former is composed by $\{\psi \wedge \alpha_{12} \wedge \beta_1\}$ and the latter by $\{\alpha_0 \wedge \alpha_{11} \wedge \phi \wedge \beta_0 \wedge \beta_2\}$.

In this process, the algorithm is omitted due to its low complexity. The key of this algorithm is the expansion of conjunctions, thus, such analysis is made in two visitors – the visitor `BinaryFormula` ($\phi \wedge \psi$) and `NaryFormula` ($\phi \wedge \psi \wedge \gamma$...), when the operator is $\wedge$.

5

ELECTRUM BOUNDED MODEL-CHECKER

The Electrum specification language is supported by two model-checking techniques, one bounded and one unbounded. The former[1] is based on the Alloy Analyzer, by using the Kodkod as back-end to verify its models. The latter[2] is built over the nuXmv (Cavada et al., 2014) – a symbolic model checker. One of this dissertation's goals is to improve the Electrum's bounded model-checker, thus, this chapter just will focus on such technique.

Electrum's bounded model-checker (BMC) is built over the Alloy's tool, known as Alloy Analyzer. Likewise Alloy Analyzer, the Electrum tool is a self-contained executable, which also includes the Kodkod model finder as well as a variety of SAT solvers. The first version of the Electrum's BMC checks the models iteratively, and before that, translate them to Alloy, which in turn invokes the Kodkod to solve them. Such version presents some limitations – some features of the Alloy Analyzer that are not implemented, the translation into Kodkod is pretty complex and has several unnecessary steps.

This way, the developing of a new BMC for the Electrum aims to implement some features of Alloy Analyzer that the first Electrum BMC does not offer and translate the models directly to Kodkod, thereby avoiding the Alloy translation as middle step. Hence, to support that, it was built a Kodkod temporal extension (previously approached in Chapter 4), in order to support the Electrum's language (temporal operators and variable signatures and fields). Therefore, to achieve that it is necessary to establish the translation's semantics from Electrum into Kodkod. Besides, the new BMC implements some additional features, which the first BMC does not implement, such as: a native feature to iterate over the states on the Visualizer, an Evaluator that handle the temporal solution not using the type time. Moreover, allied to that, the new parallel model checking technique that will be presented in Chapter 6 can also be selected as a solver on the Electrum interface, giving this way the opportunity for the user choose the solver that it wishes. Furthermore, this new tool has, in the sample models part, some well known temporal problems – such as the *Firewire*, the *Hotel Room System*, the *Lift System*, the *Ring Election* and the *Span Tree* – in order to get familiar the user with this language. Both BMC versions are based on the Alloy's tool, thus, it was necessary to extend the tool in some parts, such as: to extend both the grammar and the parser with the purpose of recognizing the new language, and

1 Available at https://github.com/haslab/electrum
2 Available at https://nuxmv.fbk.eu/

extending the Alloy's AST in order to be supported the temporal operators, as well as both variable signatures and fields.

All these topics will be thoroughly explored in the remaining chapter. This way, this chapter is composed as follows: Section 5.1 describes in detail the Electrum language, the translation from Electrum to Kodkod is handled in Section 5.2, a description of the atoms' renaming performed by this tool is provided in Section 5.3 and finally, in Section 5.4 is discussed how the visualization of counter-examples is achieved, and other proposals of visualization are discussed.

5.1 ELECTRUM SPECIFICATION FRAMEWORK

Electrum is a formal specification language inspired in Alloy and TLA, that combines the best of them. From the Alloy, it takes the structural concept, the syntax and the logic based on relations, on the other hand, from TLA it takes the capability of freely defining predicates with primed variables. Hence, it allows to specify both static and dynamic properties – the first ones are typically expressed through first-order-logic, whereas the second ones are expressed by using temporal logic. Its framework supports the verifications of these properties, enabling the verification of systems with rich configurations – systems whose state space is characterized by rich structural properties (the initial state does not have a concrete valuation) whose evolution must satisfy certain temporal properties.

With the purpose of specifying dynamic systems, the Electrum's syntax must to support temporal operators, as well as the supporting of primed variables. The previous topics will be explained in detail in this Section. Moreover, to establish the contributions of this dissertation on the Electrum's BMC, a comparison between the first and the new BMC will be also presented.

5.1.1 *Electrum Language*

As previously referred, Electrum is inspired in two of the most popular specification languages nowadays – Alloy and TLA. Since the Alloy has an easy and friendly syntax that supports transitive closure operations and both universal and existential quantification, the Electrum's syntax is an extension of it. In order to support the features provided by TLA (capacity of specifying models by using LTL as well as primed expressions), the Alloy's syntax was extended to achieve it.

Figure 22 illustrates the Electrum's syntax. It is easy to realize this syntax is an extension of the Alloy's syntax (Figure 1) and to clarify the differences between them, the new syntax introduced to support the Electrum language is underlined in Figure 22.

Following that, a signature declaration (`sigDecl`) and a field (`field`) may be tagged as variable (`var`), meaning that their valuation may evolve in time. Hence, both signatures and fields (considering also the ones that do not have a variable signature as domain) that are not variable, are considered as static, meaning they remain constant as the time evolves.

```
alloySpec ::= [moduleDecl] import* specification*
moduleDecl ::= module qualName [[name,+]]
import ::= open qualName [[qualName,+]] [as name]
specification ::= sigDecl | Constraints | assertDecl | cmdDecl
Constraints ::= factDecl | predDecl | funDecl
sigDecl ::= [var] [abstract] [mult] sig name,+ [sigExt] {field,*}[block]
sigExt ::= extends qualName | in qualName [+ qualName]*
mult ::= lone | some | one
field ::= [var] [disj] name,+ : [disj] expr
factDecl ::= fact [name] block
predDecl ::= pred [qualName .] name [paraDecls] block
funDecl ::= fun [qualName.] name [paraDecls] : expr { expr }
paraDecls ::= (field,*) | [field,*]
assertDecl ::= assert [name] block
cmdDecl ::= [name :] [run | check] [qualName | block] [scope]
scope ::= for number [but typescope,+] | for typescope,+
typescope ::= [exactly] number qualName
expr ::= const | qualName | @name | this
| unOp expr | expr binOp expr | expr arrowOp expr
| expr [ expr,* ]
| expr [! | not] compareOp expr
| expr (⟹ | implies) expr else expr
| let letDecl,+ blockOrBar
| quant field,+ blockOrBar
| { field,+ blockOrBar }
| ( expr ) | block
|  expr'
const ::= [-] number | none | univ | iden
unOp ::= ! | not | no | mult | set | # | ∽ | * | ^ | always | eventually |  once |
    historically | after | previous
binOp ::= ∨ | or | ∧ | and | ⟺ | iff | ⟹ | implies | &| + | - | ++ | <: | :> | . |
    until | release
arrowOp ::= [mult | set ] -> [mult | set ]
compareOp ::= in | = | < | > | =< | >=
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | bar expr
bar ::= |
quant ::= all | no | sum | mult
qualName ::= [this/ ] (name /)* name
```

Figure 22: Electrum's syntax

Besides that, the Electrum language, inspired by TLA, supports primed expressions (the expression's value in the next time/state) that might be applied in an expression (`expr`), by tagging it with ' (prime). Clearly, an expression not tagged with ' is considered a non-primed expression (the expression's value in the current time/state).

Furthermore, for the purpose of the Electrum language supporting the classical temporal logic, the temporal operators were added to its syntax. These ones may be divided in two classes – the binary and the unary operators. The first ones (**`until, release`**) were added to the binary operator class (`binOp`) in the syntax, whereas the second ones (**`always, eventually, once, historically, after, previous`**), obviously, were added to the unary operator class (`unOp`). The meaning of each one was explained in detail in Sub-Section 2.4.1. The remaining syntax follows the Alloy's syntax and its logical expressiveness.

5.1.2 *An Electrum Example*

The Electrum language is well-suited to modeling dynamic systems – systems that evolve over the time. This way, the Figure 23 illustrates the hotel room system under Electrum. However, in Section 2.1 there exists an introduction of this problem, as well as in Figure 3 the respective Alloy modeling of this problem. Comparing the two modeling approaches, it is easy to note that the Electrum is more appropriated to modeling this kind of systems, due to the classical temporal logic, but also due to the capability of declaring signatures and fields as variable.

```
open util/ordering[Key] as ko
sig Key {}
sig Room {
    keys: set Key,
    var currentKey: one keys }

one sig FD {
    var lastKey: Room -> lone Key,
    var occupant: Room -> Guest }

sig Guest {
    var gKeys: Key }

fact DisjointKeySets {
    keys in Room lone -> Key }

fun nextKey[k: Key, ks: set Key]:set Key{}

pred init {
    no Guest.gKeys
    no FD.occupant
    all r: Room | FD.lastKey[r] = r.
    currentKey }

fact traces {
  init
  always {
    some g: Guest, r: Room, k: Key |
      entry [g, r, k] or checkin [g, r, k]
      or checkout [g]
  }}
```

```
pred entry[g: Guest, r: Room, k: Key] {
    k in g.gKeys
    k = r.currentKey or k = nextKey[r.
    currentKey, r.keys] r.currentKey' = k
    all rr: Room - r | rr.currentKey' = rr.
    currentKey gKeys' = gKeys
    FD.lastKey' = FD.lastKey
    FD.occupant' = FD.occupant
}


pred noFrontDeskChange {..}
pred noRoomChangeExcept [rs:set Room]{..}
pred noGuestChangeExcept [gs:set Guest] {..}
pred checkout [g:Guest]{..}
pred checkin [g: Guest, r: Room, k: Key] {..}
pred NoIntervening {..}


assert NoBadEntry {
    always | all r: Room, g: Guest, k: Key |
    entry[g, r, k] and some FD.occupant[r] ⟹
     g in FD.occupant[r]
}


check NoBadEntry for 3 but 2 Room, 2 Guest
```

Figure 23: Hotel room system under Electrum

Contrary to Alloy, in this example there is no mention to the signature `Time`, making this modeling problem less error-prone and verbose. Therefore, both variable signatures and fields are tagged as variable (**`var`**), that in this case are : `currentKey`, `lastKey`, `occupant` and `gKeys`. In Alloy, to declare such dynamic structures, it is necessary add the signature `Time` as range of the each field.

These variable fields, in the remaining model specification, are constrained through temporal operators. For instance, the fact `traces` is comprised of a predicate `init` and a constraint that must be valid for all states in the systems, due to the temporal operator `always`. This last one constraint ensures that for all states, there is always some "movement" in the hotel – either an `entry`, a `checkin`

or a checkout of the guests. Nonetheless, before that, in order to initialize the initial state of the system, the predicate `init` is applied. As there is no any temporal operator over the `init`, the variable fields are constrained just over the initial state. Thus, in the initial state no guest has `(no Guest.gKeys)`, the front-desk (hotel) has no occupant `(no FD.occupant)` and finally, there exists a synchronization between the last key of the front-desk and the current key of all rooms `(all r:Room | FD.lastKey[r] = r.currentKey)`.

Moreover, it is easy to verify that modeling this kind of system by using Electrum, the predicates do not receive as argument the typical two states, with the purpose of evolving to the next state. This way, as the Electrum language provides the operator ', the frame conditions specification or any kind of reference to the next state become easier.

Unlike Alloy, in Electrum the time's scope is part of the preferences panel on the Electrum's interface, thereby avoiding the usual Alloy's scoping – on the verification command. It is noteworthy that, the scope is set in the preferences panel just in the new version of the Electrum's bounded model checker since in the first version of it, the scope was set on the verification command as well – this new version of the Electrum's bounded model checker is part of this dissertation' main goals and is detailed discussed in the remainder chapter.

Summing up, the task of modeling dynamic systems by using Electrum makes easier the user's work, since has less concern with the idiom and on the other hand, has more focus on the properties that it actually wishes to verify.

5.1.3 *First Electrum Bounded Model-Checker*

The first version of the Electrum's BMC translates its models to Alloy and this one verifies them. Moreover, before of translating a certain model into Alloy, the model is expanded into its Negative Normal Form for the purpose of guaranteeing a correct translation of the temporal operators – NNF expansion explained in detail in Sub-Section 2.4.3. Nonetheless, the translation into Alloy and the expansion into NNF are performed simultaneously. Obviously, the translation into Alloy respects the bounded model checking procedure (Sub-Section 2.4), for instance, the implementations of the Loop and the expansion into NNF.

In the translation from Electrum to Alloy, in order to identify whether the signatures or fields are variable, each one has a boolean variable that identifies it. Thus, both variable signatures and fields as well as the non variable ones share the same visitors in the Alloy's AST. Contrary to these ones, for the purpose of supporting the temporal operators (always, eventually, historically, once, after, previous), a visitor was created to identify them – the visitor `ExprTemp`. Nonetheless, the binary temporal operators are not supported in this BMC version.

This way, a certain Electrum model is translated into Alloy by crossing the Alloy's AST, and performing simultaneously the expansion of the model into to its NNF, the expansion of the variable signatures and fields into the respectively signatures and fields with one value of arity, as well as the

expansion of the temporal operators. Hence, the first step of this procedure is to create a static signature `Time`, in order to create, in the expansion of the temporal operators, variables of the type `Time` and appending them as range in the variable signatures and fields. Moreover, it is necessary to create all the required constraints in Alloy, that infer, for instance, the order over the time, as well as, the Loop with the purpose of adding in the expansion of the operator `always` – this steps are described in detail at Chapter 4, but implemented in Kodkod. Since in Electrum, both signatures and fields tagged with `var` are considered variable, so, in the translation into Alloy are created new signatures and new fields, with their arity with one more value in order to append as range the time variables, on the temporal operators' expansion. Such expansion happens at the visitor `ExprTemp`, where each operator is translated into FOL, by following the rules of Sub-Section 2.4.4.

The checking procedure of this BMC version is iterative, where the Alloy model resulting of the translation from Electrum is checked by increasing trace sizes up to the scope on Time specified in the verification command. Nonetheless, the checking procedure stops if when a counter-example is found, to a specified scope on time. Hence, if the checking for all scopes on Time is unsatisfiable, the problem is unsatisfiable. More precisely, given a certain model considering the follow command:

```
check cmd {} for 3 but 10 Time
```

Considering it and supposing that is consistent (there is not counter-example), the problem will be checked ten times, therefore, executing each time for a exactly scope i, with $i \in [0,9]$.

For instance, for the hotel room system problem, described above, in Sub-Section 5.1.2, the solving of the model to the check command `NoBadEntry` with scope 10 stops at the end of 5 verifications. It happens because, the checking of the Alloy model specification, resulting of translating the Electrum model, for a scope of 5 time instants is satisfiable (the check is not consistent) and returns a counter-example. Moreover, it is easy to note that, the solving procedure with the scope defined to exactly i time instants, $i \in [0,3]$, is unsatisfiable (the check command is consistent). This BMC gives to the user a opportunity to iterate of all possible counter-examples that broke the specified properties of the model.

5.1.4 *New Electrum Bounded Model-Checker*

The new version of the Electrum's BMC is an important goal of this dissertation. This way, the building of a new BMC aims to make a direct translation from Electrum to Kodkod, avoiding the translation to Alloy as middle step, making the translation procedure easier and faster. Besides that, with the purpose of simplifying the Kodkod models resulting from the translation, the semantics's translation of signatures, namely the hierarchy and the typing of them, was modified (will be explained in detail in Section 5.2). Furthermore, this new Electrum's back-end has a novel method of partitioning and checking the Kodkod models in parallel (will be explained in detail in Chapter 6). As previously referred, the first version of the Electrum's BMC, the verification procedure is performed iteratively by increasing trace sizes up to the scope on time, and this way, it stops when a counter-example is

found or the trace size has reached to the end. In case of returning a counter-example, the user just can visualize the trace since the initial time until the time for which the verification procedure has returned a counter-example – this might be considered as a drawback. This drawback is solved in the new BMC, since Kodkod returns a temporal instance – a counter-example where the trace is composed by all times –, enabling this way the user visualize the complete trace.

Besides, the new Electrum's BMC provides a new feature on the Visualizer, to iterate over the trace. Such feature is a new combo-box on the Visualizer that is independent of the typical projection. Obviously, to achieve the visualization of a certain temporal instance a dynamic renaming must be made since the atoms of a certain Kodkod solution might belong to different signatures at different time instants. Allied to this, on Evaluator the user can insert predicates over the counter-example depicted on the Visualizer, allowing this way the insertion of predicates over the complete trace. The first version of BMC also allowed this, but due to its nature, the type `Time` was explicitly declared on the predicates. These contributions will be discussed on remainder Chapter.

Moreover, in order to become the translation from Electrum to Kodkod straightforward, the Kodkod has to support all requirements of the Electrum. Therefore, it has to support the declaration of both variable signatures and variable fields as well as the temporal operators. Besides that, it must to follow the bounded model checking procedure (detailed in Section 2.4). As previously referred, all this steps are covered by the Kodkod temporal extension, thoroughly detailed in Chapter 4.

However, in this BMC version, contrary to first one, there exists supporting of the binary temporal operators (**`until, release`**) and the past temporal operators (**`historically, once, previous`**), both at the back-end level (Kodkod supports it) and at the front-end level. Hence, the Alloy's AST and both the grammar and the lexer were extended with the purpose of supporting them.

Moreover, now, the time trace length is set on the Electrum' preferences interface, since in this version the time is not handled by the Electrum. Besides that, in the solving preferences on the Electrum's interface there exists a option to choose the novel procedure to check models in parallel.

5.1.4.1 *Architecture*

The new Electrum's BMC provides the normal features like Alloy Analyzer. Contrary to Alloy Analyzer, this new Electrum's BMC provides two additional options on its interface – the capability of defining the max trace length as well as the parallel solver.

Figure 24 depicts a basic running architecture of this tool. Thus, given a certain Electrum model, when the user executes a command, the first step is to parse the model and, verify if there are some syntax or typing errors, the model is loaded into the Electrum's AST. Thereafter, it is carried out the translation from Electrum to Kodkod by crossing the visitors present on the AST, previously built.

Besides, the translation into Kodkod implies to create the bounds for both signatures and fields and to translate the signatures and the fields into Kodkod relations. Moreover, all the constraints in the Electrum model are translated into a Kodkod formula as well as the typing and the multiplicity of both signatures and fields.

Once the translation was achieved, the Kodkod invokes off-the-shelf SAT solvers for the purpose of verifying the respective Kodkod model, resultant from the Electrum model, based on certain options such as Symmetry-breaking, Skolem-depth, among others. When Kodkod finishes the solving of such model, the output and the stats are broadcasted for the Electrum tool.

In this case, if the result is satisfiable (there is, at least, a instance which satisfies the model) a Kodkod instance that breaks the specified properties of the model specification is also sent, otherwise the result is unsatisfiable and there is not any counter-example that satisfies the model specification. Besides, in case of there exists some counter-example, the user can visualize it on the Electrum's Visualizer, as well as iterate over all possible counter-examples.

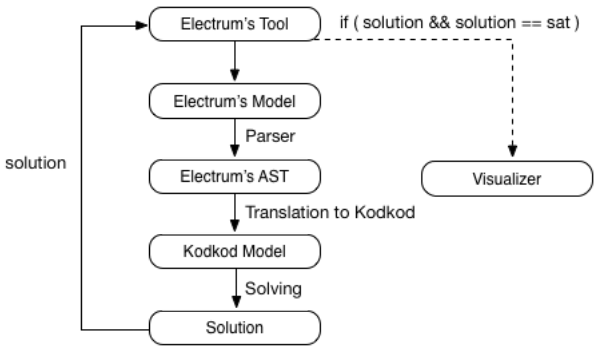


Figure 24: Electrum's BMC architecture

Moreover, the Electrum BMC Visualizer also provides the evaluator, a feature that allows the user to insert predicates over the depicted counter-example, providing to user a better perception of the problems on the specification.

Besides that, the typical features of both Visualizer and Analyzer are kept. On Analyzer are kept for instance the capability of returning the Kodkod model, the CNF file custom and the font on editor. In respect of Visualizer it is possible, for instance, to apply the *magic layout* feature on the counter-examples, customize the counter-examples theme and visualize a printing of the a map between both signatures and fields and their atoms in the counter-example, among others.

5.2 TRANSLATION SEMANTICS FROM ELECTRUM TO KODKOD

The new Electrum BMC translates its models directly into Kodkod, thus, it is necessary to establish a correct translation's semantics in order to convey the right meaning of the Electrum model in Kodkod. Such translation's semantics is inspired in (Hodkinson et al., 2002), being this way simplified the translation of both signatures and fields into Kodkod, namely in the typing and the multiplicity of them. The remaining components of the model do not suffer any change on the translation's semantics, since the translation of both constraints and relational expressions is relatively straightforward and follows that of Alloy. Moreover, the translation of temporal operators is also straightforward due to

the Kodkod temporal extension where such operators are supported. Hence, this semantics in this new BMC version is simple and more understandable than the first version. While the signatures and the fields, in the translation procedure, are expanded by iterating a list of signatures, and for each signature its fields are analyzed, the expansion of the temporal operators is achieved traversing the visitors that represent them – `BinaryExprTemp` and `ExprTemp`.

5.2.1 *Signatures and Fields*

As previously referred, the translation's semantics from Electrum to Kodkod was simplified, but keeping the same capability of expressing both signatures and fields as well as the constraints in Kodkod, in order to guarantee a correct verification. This way, signatures and fields are represented in Kodkod by relations. A signature is represented by a relation with arity one and the fields by relations with their arity and typing presented in the Electrum model. Until now, there are no differences between Alloy, but the big one is in the creation of signature and fields tagged as variable. Therefore, as the Kodkod temporal extension provides the capability of supporting variable relations, such signatures and fields are directly encoded into this kind of relations – `VarRelation`. In order to explain this better, it will be considered the following Electrum model and simulated its translation into Kodkod:

```
abstract sig A { r: some A }
var sig B,C extends A {}
```

Figure 25: Electrum model example

Given the small Electrum model depicted above, their signatures and fields will be transformed in relations in the translation into Kodkod, as depicted following:

```
VarRelation B = VarRelation.unary("B");
VarRelation C = VarRelation.unary("C");
Relation A = Relation.unary("A");
Relation R = Relation.nary("r", 2);
```

Note that, as previously referred, the variable signatures were encoded into the new type of relations – `VarRelation` – and its arity is one. Moreover, the field `R` has arity two (`r: A -> A`), as declared in Electrum model. Once the relations are created, it is necessary to specify in Kodkod the constraints that ensure the correct typing of such signatures and fields and their multiplicity. Therefore, the constrains responsible for specifying these signatures and fields, at Kodkod level, are:

```
always (A = B + C and no B & C)
always r in A -> A
always all a: A | some a.r
```

The temporal operator **always** is depicted in all four formulas depicted above, since it is required to ensure the typing and the multiplicity of each signature or fields for all states in the trace. Since the signature A is not abstract, it is comprised only of the elements of its sub-signatures, so A = B + C, besides, the intersection of its sub-signatures cannot happen, being the formula that demonstrates it, defined as follows: **no** B & C. The formulas that express the not intersection between sub-signatures have to be made for each pair of sub-signatures, more precisely, supposing the signatures Z,B,C extending the signature A, in this case, the Kodkod formula responsible for expressing this hierarchy would be : A = B + C + Z **and no** B & C **and no** B & Z **and no** C & Z.

The next step is to define the constraints that express the typing and the multiplicity of both signatures and fields, where the typing is typically represented by the constraint **always** r **in** A -> A. Obviously, this expression is possible because there exists a matching of arity between the field r (2) and the sum of arity of the signatures A (1). Besides, the multiplicity is expressed by the Kodkod formula **always all** a: A | **some** a.r, meaning, for all states there exists a field r with A as domain and a A as range, with multiplicity **some**.

This semantics, projected in (Macedo et al., 2016), has suffered an optimization in the translation of abstract signatures into Kodkod, in order to speed up the checking procedure, since there are less variables for solving. More precisely, as the signature A is abstract (it is comprised only of the elements of its sub-signatures), the creation of the Kodkod relation A is unnecessary. Even though changing the semantics, the expressiveness is kept with the purpose of transmitting the right meaning of an abstract signature into Kodkod. This way, the optimization is proposed as follows:

```
always no B & C
always r in (B + C) -> (B + C)
always all a: (B + C) | some a.r
```

This way it is correctly ensured an equivalence between the constraints defined above and the ones proposed in the first instance. Note that, now the signature A takes the union of B and C (its sub-signatures). Hence, the assignment A = B + X is omitted, since the the value o A is implied as B + C. However, it is noteworthy refer that any constraint expressed in the translation of signatures or fields into Kodkod must have the operator **always**, in order to keep the same constraints about the declaration of both signatures and fields for all states of the trace. For instance, given the following signature declaration in Electrum:

```
sig S in A {}
```

Therefore, the declaration of this signature implies a new constraint in the translation into Kodkod: **always** s **in** A in order to ensure for all states that S $\subseteq$ A.

5.2.1.1 *A brief comparison with semantics of the first BMC*

As the first BMC version translates its Electrum models into Alloy, and then, this one performs the respective translation into Kodkod, it will be presented a brief comparison between the Kodkod gen-

erated in the first BMC version and the new one. The example approached to illustrate the differences between the two semantics is Figure 25, the one that was approached in the last section to illustrate the new translation's semantics. This comparison serves to show how the new translation's semantics is easier to understand as well. Therefore, in the previous BMC, to the Electrum model defined in Figure 25 would be created six relations (the additional relations responsible for performing the bounded model-checking are not depicted – `loop`, `next`,...), formed as follows:

```
Relation Time = Relation.unary("Time");
Relation B = Relation.nary("B");
Relation C = Relation.nary("C");
Relation BVar = Relation.nary("BVar",2);
Relation CVar = Relation.nary("CVar",2);
Relation R = Relation.nary("r", 2);
```

Obviously, the relation `Time` is explicitly created since created, in the translation into Alloy, the signature Time, besides, for each variable signature is necessary to create two relations – one where is declared the signature with arity one, and another one with the domain defined by the first one and the range defined by the relation `Time`, thereby ensuring a relation between the signature with Time. It is easy to verify the simplicity of the new BMC semantics, mainly, due to the capability of creating variable relations provided by the Kodkod temporal extension.

Moreover, the constrains responsible for establishing the typing and the multiplicity in Kodkod, produced by the previous BMC version, are depicted as follows:

```
 no (B & C)
(all this: B | (this . BVar) in Time)
(BVar. univ) in B

(all this: C | (this . CVar) in Time)
(CVar . univ) in this/C

(all this: B + C | some (this . r) ∧ (this . r) in (B + C))
(r . univ) in (B + C)

(all this: B + C |
      all s: Time |
            (none + (BVar . s) + (CVar . s)) in (B + C) ∧
          (B + C) in (none + (BVar. s) + (CVar . s))
 )
```

Obviously, the specification of these constraints is far of the typical manner which the new Electrum's BMC applies. Comparing the two specification for the same model, the new BMC has a bigger simplicity. Note that, in the Kodkod specification depicted above is necessary two extra constraints in order to guarantee that the variable relations have the `Time` as range. Moreover, the typing constraints are performed in two steps, by declaring two constraints – one where a certain field that is composed (join operation) with its range has to be contained in its domain, and another one where the field is

composed with its domain and has to be contained in the range. The former is declared as follows `(r . univ) in (B + C)`, but, to a better understand the expression `r.univ` (reduces the arity of `r` by deleting the range) in this context, supposing the field `r` with arity three, the typing constraint in this case would be `(r . univ).univ in (B + C)`. The latter is typical formed over a quantification over the domain defined as follows `all this: B+C | (this . r) in (B + C)`. Relative to the multiplicity constraints, here, they are defined like in the new Electrum's BMC.

5.2.2 *Temporal Operators*

The translation of the temporal operators from Electrum to Kodkod is straightforward, since the Kodkod temporal extension supports them. Figure 26 illustrates the visitors responsible for the translation of such operators. In the visitor `ExprTemp` (line 3) are translated the unary temporal operators, whereas in the `BinaryExprTemp` (line 7) are translated the binary ones.

Input : Temporal Expression encoded in Electrum
Output : Temporal Expression encoded in Kodkod

```
1: while(thereAreExpressionsToVisit)
2:     ...
3:     visit(ExprTemp φ)
4:         case ALWAYS: return (φ.sub.accept(this)).always();
5:         case EVENTUALLY: return (φ.sub.accept(this)).eventually();
6:         ...
7:     visit(BinaryExprTemp φ)
8:         case UNTIL: return (φ.left.accept(this)).until((φ.right.accept(this)))
9:         case RELEASE: return (φ.left.accept(this)).release((φ.right.accept(this)))
```

Figure 26: Electrum temporal operators' translation

Supposing an Electrum expression that has the operator **`always`** will be expanded, so, the analysis starts at line 4. Before to return a Kodkod formula resulting from this operator, the expression inside the operator is explored (`sub`). Once returned the Kodkod formula resultant from the exploration of the expression inside the operator, is returned a Kodkod formula, where the operator **`always`** is applied to the sub-formula presented in the operator. In case of being the operator **`eventually`**, the procedure is analog. Note that, it was omitted four temporal operators at the visitor that represent the unary ones – **`historically, once, previous, next`**.

In case of being analysed a binary temporal operator, the procedure is similar to the unary one. Thus, the only difference is that, two inside expressions are analysed, rather than one. The expression on the left is denoted by the method `left`, whereas the one on the right is denoted by the `right`. Thereafter, it is returned the Kodkod formula with the respective binary temporal operator – **`until, release`**.

5.3 SOLUTION RENAMING

As the Electrum BMC is built over the Alloy's tool, it is necessary to adapt various important procedures to achieve a proper functioning of this new tool – one of them is the renaming that the Alloy performs. Just in order to contextualize, it will be described how the Alloy performs such procedure and what is its purpose. Hence, it is explained how to adapt such feature to the Electrum and what changes that were performed to achieve a proper behaviour. More precisely, the Electrum needs to adapt such procedure since the atoms of a certain Kodkod solution might belong to different signatures at different time instants.

Whenever a command is executed and the solving result is satisfiable (there exists some counter-example), the atoms presented on the solutions are renamed before being depicted in Visualizer. Such procedure aims to rename the atoms returned by Kodkod, by appending new ids (identifiers), respecting a total order, to them and, in some specific cases change the original name of them. As previously referred, it will be explained a typical Alloy example, depicted below in Figure 27, to illustrate this procedure.

```
abstract sig A { r: some A }
sig B,C extends A {}
run {}
```

Figure 27: Alloy model example

The renaming procedure has the purpose of exploring all the signatures and their sub-signatures presented at a certain model. As the fields are composed of signatures, they are not explored in this procedure – the atoms belonging to a certain field are properly renamed when a signature is renamed.

Given the Alloy model depicted above, it is easy to verify that given a typical command `run` the Kodkod will return various instances/solutions that satisfy the model. Moreover, the scope applied to this model is three, the predefined scope when the user does not specify the scope. Hence, as the signature `A` is abstract, the Kodkod relation resulting of `A` is not created, however, all sub-signatures of `A` are composed of `A`'s. Therefore, both `B` and `C` are composed of the following bounds depicted below in Listing 5.1. On the right of bounds is illustrated a possible solution returned by Kodkod to satisfy this specification – Listing 5.2.

```
B = [[A$0],[A$1],[A$2]]
C = [[A$0],[A$1],[A$2]]
r = [[A$_,A$_],..]
```

Listing 5.1: Original bounds following Alloy

```
B = [[A$0]]
C = [[A$1]]
r = [[A$_,A$_],..]
```

Listing 5.2: Solution returned by Kodkod in Alloy

Consequently, the renaming procedure renames the atoms of both signatures, `B` and `C`, with the purpose of they have the same name of its signature instead of `A`. Following this thought, the atom presented on the signature `B` is renamed to `B$0`, whilst the atom presented on `C` is renamed to `C$0`.

Moreover, supposing that the signature `B` also have the atom `A$2`, this one would be renamed to `B$1` – the atoms are always renamed with a total order over the ids. Besides, during this procedure, in order to a correct representation of the atoms presented on the fields, it is built a mapping between the original atom's name and its respective renaming. Moreover, the renaming happens for all signatures, for instance, given a model just with a signature `A'` and a simple command `run` with a predefined scope where, after to solve it, the instance returned from Kodkod contains just an atom `A'$2`. Given this, the renaming performed by Electrum transforms the original atom to `A'$0`, thereby ensuring that in the visualization procedure the ids have a total order from zero.

Once the Alloy renaming explored and explained, it is time to explain how apply such procedure for a temporal instance on the Electrum's tool. Nevertheless, in the Electrum the variable signatures have a set of atoms per each time. Therefore, the renaming is done dynamically `n` (where `n` = max trace length.) times per model, one time per each specific time/state. Given the sample Electrum model defined in Figure 25 (in this example the signatures `B` and `C` are variable) with the max trace length equal to ten, it will be explained how the temporal renaming is achieved.

```
B(var) = [[A$0],[A$1],[A$2]]
C(var) = [[A$0],[A$1],[A$2]]
r = [[A$_,A$_],..]
```

Listing 5.3: Original bounds following Electrum

```
B = [[A$0,Time$0], ... ,[A$1,Time$9]]
C = [[A$1,Time$0], ... ,[A$0,Time$9]]
r = [[A$_,A$_],..]
```

Listing 5.4: Solution returned by Kodkod in Electrum

As depicted in Listing 5.4, both signatures and fields are tagged as variable, when a temporal instance is returned from Kodkod, the `Time` relation is appended as range on them. Note that, through Kodkod evaluator, it is possible to return a certain configuration of a specific time instance, thereby avoiding the exhausting searching of a temporal instance just to get it.

Besides, the main purpose of the renaming procedure is to rename the atoms of a certain solution with the purpose of writing it into an xml file and then the Visualizer reads such xml file. Hence, without focusing on the Visualizer, it is easy to observe that is required perform a dynamic renaming for all time instants as it evolves, when a counter-example is being depicted on the Visualizer. To achieve that, it is required to establish an factor order for the atoms' id, more precisely, as previously referred, the Alloy applies always an increscent id from zero for a set of atoms belonging to a specific signature. If such strategy is applied to Electrum renaming, for the first and last state, the result would be:

```
Time$0 :: B = [[B$0]]; C = [[C$0]]
Time$9 :: B = [[B$0]]; C = [[C$0]]
```

Hence, the Electrum Visualizer would show the same instance, because there is not any criteria to choose the id for the atoms. Thus, this way, to address this problem, it was established a criteria to choose a specific id for a certain atom. Such criteria is the position of the original atom in the problem's universe, ensuring this way to user visualize the trace evolution as the time evolves. More precisely, if the problem's universe is composed as follows `U = [A$0,A$1,A$2]`, and applying the criteria previously referred, the result of renaming the atoms at time instant zero is `B = [[B$0]], C = [[C$1]]`, since $index(A\$0) = 0$ and $index(A\$1) = 1$. Following the same procedure, at the time instant nine, the result of applying the rename is `B = [[B$0]], C = [[C$1]]`. Obviously, if the renaming is preformed following this strategy, the identification of the first and last time instant is easily identified.

Besides, the criteria of choosing the atoms' id would be different, but less elegant and clean. Thus, the other option is to assign the original atom's id to the renamed atom. To achieve that, it is necessary make a parser of the original atom from the character `$`, and get the id.

5.4 VISUALIZER

Likewise Alloy, the Electrum's BMC provides a Visualizer capable of showing counter-examples that broke the specified properties on the model. This Visualizer uses the same mechanism that Alloy (Sub-Section 2.1.3), reading an xml file with a certain instance to depict it visually. In Alloy, the temporal models are often analysed with a projection over the time signature. Contrary to Alloy, in Electrum, the time is handled internally, thus, there is not the type time to project, so, the typical projection is unfeasible to depict a specific counter-example. Moreover, in Electrum, the projection over the time must be implicit and not optional for the user, since the purpose of this framework is to verify temporal specification. Therefore, such Visualizer has a native combo-box where the user can iterate over all specific time instances of some counter-example.

Figure 28 shows the Electrum's BMC Visualizer showing a counter-example of the hotel room system example under Electrum (Figure 23), in time instant 1. Like the figure shows, on the top of the Visualizer there is a combo-box with n possible time instants (n is the number of time instants that the Kodkod has returned), which gives to the user, the possibility of iterating over all possible time instants, respecting the total order over the time. Besides, the Visualizer simulates the loop property (a feature of bounded model-checking), in turn, is internally created and returned by Kodkod, which means that the user can see such feature simulated visually. Therefore, when the user is iterating over the time in a certain example, if it reaches the back loop index (or back loop time instant) the Visualizer jumps back to the initial loop index. More precisely, given a certain counter-example where the max trace length is set to 10 and the loop relation is filed by the pair [8,4] (4 is the initial loop index and 8 is the back loop index), the combo-box is filled by 8 time instants, which is exactly the back loop index. Following that, when the user is iterating growingly over the time, once it reaches

the time instant 8, the next solution to be depicted is the one in the time instant 4. Once it reaches the time instant 4 the cycle starts again.

As mentioned before, the time is internally handled by the Kodkod and there is not the type time, so, the typical projection is infeasible. To address this issue, the Electrum creates a set of xml files, where each one is a specific solution in a certain time instant and, whenever the user iterates in the time, an xml file is read for such specific time instant.

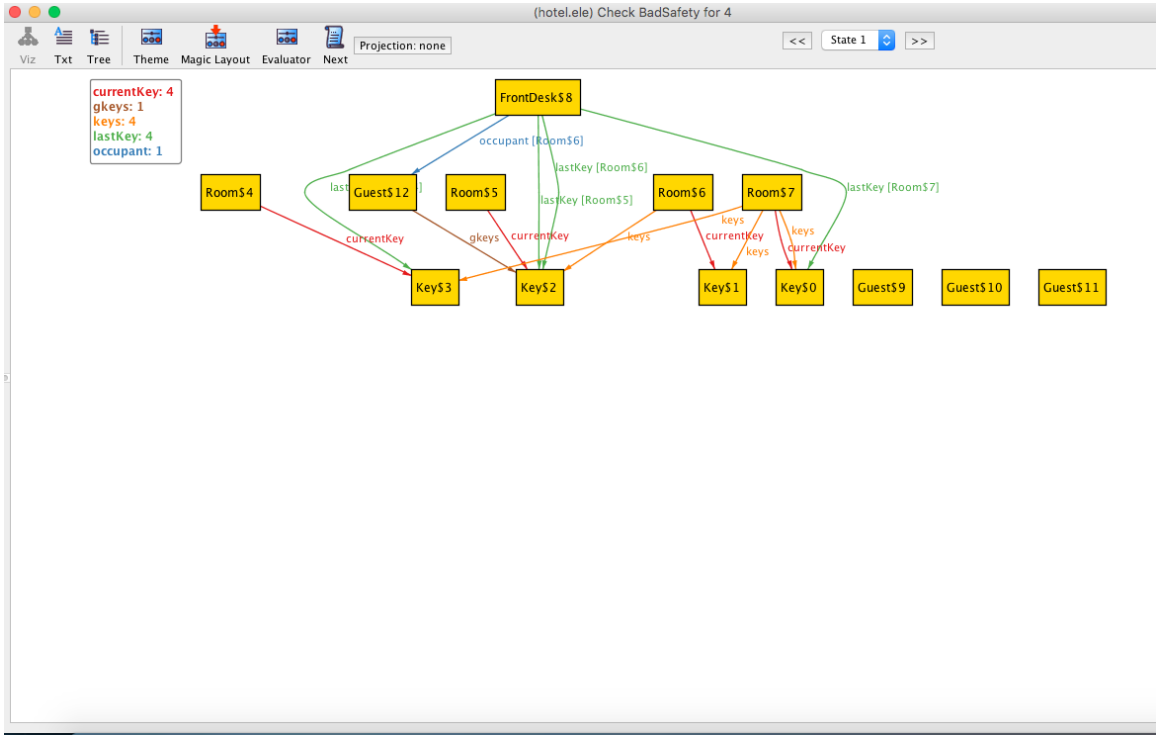


Figure 28: Hotel room system counter-example under Electrum

Therefore, when the user checks some Electrum model, in case of being satisfiable, the Kodkod returns a specific counter-example. Given such counter-example and a specific time instant, through the Kodkod's evaluator, a solution for such time instant is built, thereafter the renaming procedure is performed and finally, an xml file is written with the solution renamed. Hence, this procedure is performed for all time instants defined by the user. Note that, these xml files have three additional attributes: the max length trace, the initial loop state and the back loop state with the purpose of creating the combo-box with the right number of states as well as the loop animation.

As mentioned in the previous section, the renaming procedure is carried out for each time instant before of writing each xml file in order to ensure which the user can see as the system evolves over the time. As a consequence, in the figure there are not equal atom' ids due to the criteria chosen for the renaming. For instance, the atom $Guest\$9$ is the atom $Guest\$0$ on the solution returned by Kodkod, but as the $index(Guest\$0)$ is 9, the atom is renamed to $Guest\$9$.

When the user asks a next instance, if there is any, the procedure mentioned above is performed again and the xml files are rewritten. Moreover, to read a certain xml for a certain time instant, the xml file's name is customized for each one as follows $pathToTemporaryFolder/confState\mathbf{i}.xml$, where **i** is a certain time instant.

Even though the procedure of generating the xml files generally is immediate, for large trace lengths might take considerable time. Thus, a thread is used to do this in order to not influence the solving time depicted at Electrum interface.

5.5 EVALUATOR

The Evaluator is a feature presented on the Alloy Analyzer, which allows for the user to make predicates over the counter-example depicted on the Visualizer. When the user opens the Evaluator window, the first step that Alloy does is to make the parser of the model in the editor, which gave rise to the counter-example. Once the parser performed, an Alloy's AST is filled with the Alloy model. Thereafter, the xml file that contains the counter-example is read and both the signatures and the fields are exactly filled by the corresponding atoms in the xml file. This way, it is ensured that, any expression typed by the user will not have atoms that are not depicted on the Visualizer.

Hence, this procedure does not work in Electrum since an xml file is generated for each time instant. A proposal for this issue would be to read all xml files and create a temporal solution, however, such solution is infeasible due to the unknown number of xml files and this way, a high computational power would be required. To address this issue, the ideal solution would be to generate an xml file with all atoms present in the temporal solution. Following this strategy, recurring to a proper data structure, as the xml files are being written, the non-repeated atoms are being gathered with the purpose of writing a global xml file with non-repeated atoms. More precisely, given a simple Electrum model just with a variable signature A whose its solution returned by the Kodkod is $[[A\$0, Time\$0], [A\$0, Time\$1], [A\$1, Time\$2]]$ and obviously with the max trace length is set to 3.

In this case, three xml files will be written and , at the same time, the non-repeated atoms of A are being gathered. Therefore, the xml file resulting from the gathering of all atoms is comprised as follows:

```
<alloy>
<∈tance command="run{}">

<sig label="univ" ID="2" builtin="yes"></sig>

<sig label="A" ID="3" parentID="2">
   <atom label="A$0"/>
   <atom label="A$1"/>
</sig>
</∈tance>
</alloy>
```

Thus, as the signature A at time instant 0 is composed by the atom $A\$0$ and at time instant 1 and 2 is composed by the atom $A\$1$, the signature is, in the global xml file, composed by these two ones. Hence, if the user writes the specific expression $univ$ on the Evaluator, the result is $\{A\$0, A\$1\}$. This procedure works this way as well for the fields, where the non-repeated tuples are being gathered as the xml files are being generated.

6

PARALLELIZATION STRATEGY

Verification tools of high-level formal specification languages typically handle a specification in an opaque manner. Such procedures usually *amalgamate* all the constraints in a single monolithic verification task, which often encumbers the procedure when the specification is comprised by several complex declarative constraints. Considering this issue, for a considerable set of specifications, we argue that there is a notion of "*configuration*" – a partial solution that can be extracted and explored independently. Making an analogy, a set of configurations in a problem can be, for instance, the set of all network topologies over which a distributed algorithm is expected to be verified.

A new procedure for verifying Electrum models in parallel is an important part of this dissertation. This parallelization technique can be applied to any Kodkod model, but the first step of it must be achieved manually – the splitting of the model in two disjoint sub-models. However, the application of such procedure on temporal models is a way of automating it. Therefore, this splitting can be automatically calculated for temporal models since there exists an explicit division on them – the static from the dynamic properties.

Therefore, this parallelization technique aims to decompose a specification into different configurations and subsequently parallelizing the analysis procedure. Such decomposition is performed in the static part – from the static part, the candidate non-symmetric configurations are generated, which are then integrated into the dynamic part and launched in parallel. To achieve that, an effective implementation of such technique is provided as an extension to the Kodkod constraint solver. Nonetheless, before implementing this technique as an extension to the Kodkod constraint solver, to assess the potential of decomposing problems, a pilot study was performed on a set of formal specification frameworks – B (ProB), TLA+ (TLC), Alloy and Kodkod. However, due to the purpose of this dissertation, such study does not deserve emphasis, thus, it is not presented.

The previous topics will be explored in detail in the remainder chapter, which is structured as follows: in Section 6.1 is showed how a Kodkod model finding problem might be decomposed, the Kodkod constraint solver extension is described in Section 6.2, in Section 6.3 is illustrated such decomposition considering a Kodkod problem and finally, the final results are presented in Section 6.4.

6.1 DECOMPOSED MODEL FINDING

First of all, model finding aims to find model instances that satisfy certain constraints. Such instances are basically bindings for a set of free variables $\mathcal{V}$ such that certain formulas ϕ, defined over $\mathcal{V}$, hold. As a consequence, the search space for these variables is restricted by the well-known upper- and lower-bounds of Kodkod.

Definition 2. A Kodkod model finding problem P is a tuple $\langle \mathcal{U}, l, u, \phi \rangle$ where $\mathcal{U}$ is a universe of elements, $l, u : \mathcal{V} \to \mathcal{U}$ assign to each variable $v \in \mathcal{V}$ lower- and upper-bounds, respectively, with $l(v) \subseteq u(v)$, and ϕ is a set of formulas over $\mathcal{V}$ variables. A binding $b : \mathcal{V} \to \mathcal{U}$ is a solution of P if ϕ holds and $b(v) \subseteq u(v) \backslash l(v)$ for every $v \in \mathcal{V}$.

The definition depicted above illustrates a Kodkod problem, as well as how a certain solution of it is built. Besides, it is considered that the selection of the tuple's elements of a certain Kodkod problem P is denoted as $\mathcal{U}_p$, l_p, u_p and ϕ_p, respectively. Note that, if there is no satisfying solution, an empty binding $\perp : \emptyset \to \mathcal{U}$ is returned. Furthermore, considering b, it is possible to return a next solution for the problem specification by negating b and adding it to the iterated problem's constraints.

The decomposed model finding is the crucial part of this parallelization strategy. Therefore, a model finding problem P can be be decomposed since its variables bounds can be used to provide partial solutions to the model finding procedures. Considering a subset of variables $\mathcal{V}_p$, with $\mathcal{V}_p \subseteq \mathcal{V}$, a certain binding $b : \mathcal{V}_p \to \mathcal{U}$ is a partial solution of another problem if it is within the bounds defined for those variables and the formulas defined exclusively over $\mathcal{V}_p$ variables hold.

Definition 3. A binding $b : \mathcal{V}_p \to \mathcal{U}$ is partial solution of a model finding problem $\langle \mathcal{U}, l, u, \phi \rangle$ if $\mathcal{V}_p \subseteq \mathcal{V}$ for $l, u : \mathcal{V} \to \mathcal{U}$, and it is a solution of the model finding problem $\langle \mathcal{U}, l|_{\mathcal{V}_p}, u|_{\mathcal{V}_p}, \phi|_{\mathcal{V}_p} \rangle$.

The previous definition describes the notion of partial solution of a model finding problem. Note that, $l|_{\mathcal{V}_p}, u|_{\mathcal{V}_p}$ and $\phi|_{\mathcal{V}_p}$ define the bounds and the formulas, respectively, referred exclusively for the subset of variables $\mathcal{V}_p$. Besides, this definition assumes the empty binding $\emptyset$ is always a partial solution of every model finding problem.

However, to achieve this parallelization strategy, the partial solutions must be integrated into the bounds of regular problems to speed up the model finding procedure. This way, the following definition defines such embedding, considering the $\oplus$ operator the overriding of mappings.

Definition 4. A partial solution $b : \mathcal{V}_p \to \mathcal{U}$ can be integrated into a model finding problem $P = \langle \mathcal{U}, l, u, \phi \rangle$ as $\langle \mathcal{U}, l \oplus b, u \oplus b, \phi \rangle$, denoted by $P \oplus b$.

After a partial solution being integrated into the remainder problem, it is solved. For better illustrating how a problem can be solved using this strategy, a logical syntax was created to identify it, showed in the following definition.

Definition 5. A decomposed model finding problem P_p is a tuple $\langle \mathcal{U}, \mathcal{V}_p, \psi, l, u, \phi \rangle$ such that:

- $P_0 = \langle \mathcal{U}, l, u, \psi \wedge \phi \rangle$ is the *amalgamated* model finding problem
- $P{\downarrow} = \langle \mathcal{U}, l|_{\mathcal{V}_p}, u|_{\mathcal{V}_p}, \psi \rangle$ is the *partial* model finding problem
- $P{\uparrow} = \langle \mathcal{U}, l, u, \phi \rangle$ is the *remainder* model finding problem

A binding $b : \mathcal{V} \to \mathcal{U}$ is a solution of P_p if it is a solution of the amalgamated problem P_0.

Thus, after these definitions and logical syntax defined, it is time to do an overview of the parallelization technique considering them. The first step is to split a subset of variables $\mathcal{V}_p$, with $\mathcal{V}_p \subseteq \mathcal{V}$, and the global formula into two sub-formulas, wherein one of them may only refer variables of $\mathcal{V}_p$. With that, a decomposed model finding problem P_p is achieved since it is a tuple $\langle \mathcal{U}, \mathcal{V}_p, \psi, l, u, \phi \rangle$. This way, it is easy to split the problem into two ones – the partial ($P{\downarrow}$) and the remainder ($P{\uparrow}$). The former is comprised of the original universe, a formula ψ (disjoint from the $P{\uparrow}$'s formula – ϕ) and the bounds ($l|_{\mathcal{V}_p}$ and $u|_{\mathcal{V}_p}$), which refer exclusively variables of $\mathcal{V}_p$. Besides, ψ may only refer to variables bounded by its bounds. The latter has the original universe, the original bounds and a formula ϕ (disjoint of $P{\downarrow}$'s formula). Obviously, the amalgamated problem (P_0), considering that it has the original bounds and universe, its formula is the conjunction of ψ and ϕ. Furthermore, from $P{\downarrow}$ are generated the partial solutions, or configurations, and then each one, following Definition 4, is integrated into $P{\uparrow}$. From this results the creation of several integrated problems (with (integrated problems) = (partial solutions)) that that can be independently solved either sequentially or in parallel.

6.2 DECOMPOSED KODKOD

Making a quick mention to the pilot study, such that the integrated problems were solved sequentially, but in this extension built to the Kodkod constraint solver they are launched in parallel as the configurations are calculated.

The extension built to Kodkod constraint solver for implementing this strategy has the purpose of solving a decomposed problem P_p. Thus, as previously referred, from the partial problem $P{\downarrow}$ the non-symmetric configurations are generated, then each one is integrated into the remainder problem $P{\uparrow}$ and finally the integrated problems $P{\uparrow}\oplus_{pi}$ are launched in parallel. When one of these problems terminates and is satisfiable, with the purpose of the user inspecting such solution, it is pushed into a blocking queue. In the meantime, if one of the other integrated problem was found to be satisfiable, such solution is readily available. For a properly management of the integrated problems, they are managed by an executor with a FIFO queue. If any of these integrated problems is found to be satisfiable, such solution is returned to the user. Note that, when the user requests another solution succeeding s_{ik}, the system iterates the integrated problem $P{\uparrow}\oplus_{pi}$ recurring to the capability of the Kodkod to iterate for next solutions.

As shown in the pilot study, there is a potential success for satisfiable problems, since a solution is returned as soon as an integrated problem is satisfiable, so performance gains are expected. In

contrast, the potential for unsatisfiable problems is unclear, since every integrated problem must be analyzed. To address this issue, a *hybrid* strategy is proposed. In this strategy, the parallel solving of the integrated problems is paired with a thread solving the amalgamated problem. While in the parallel mode are used four thread to perform such procedure, however, in hybrid mode the parallel model is performed by three threads and one thread is used to solve the amalgamated problem. In this mode there is a distinguished thread with higher priority running the amalgamated problem P_0 – if such thread terminates every integrated problem is also terminated. Following this technique, in the best case, a satisfiable integrated problem will finish before the amalgamated one. In contrast, in the worst case, the amalgamated problem will finish before the integrated ones. However, due to cache interference, performance will be slightly worse than running the amalgamated problem independently.

6.3 DECOMPOSITION FOLLOWING AN EXAMPLE

In order to better illustrate how this technique works, in this section, a practical example (the hotel room system example) of this technique will be approached. Obviously, this technique is applied to Kodkod after a certain Electrum model being translated into it.

Considering the hotel room systems example, an appropriate splitting would be from the static and dynamic properties. Following that, as Electrum is translated into the temporal Kodkod, by Section 4.6 this decomposition is automated. However, just to illustrate concretely what is the static component of the hotel system example, the Figure 29 shows a partial model in Electrum.

As the figure depicts, the static component is the one that does not contain variable relations. This way, the formula of $P\downarrow$ is the respective Kodkod formula resulting from the translation of `Room: keys in Room lone -> Key` into it. Following the same idea, the formula of $P\uparrow$ is the Kodkod formula resulting from the translation of the remainder Electrum model into it.

```
open util/ordering[Key] as ko
sig Key {}
sig Room { keys: set Key }
one sig FrontDesk {}
sig Guest {}
fact DisjointKeySets { Room<:keys in Room lone-> Key }
```

Figure 29: $P\downarrow$ of the hotel room system example

Considering the partial problem, five Kodkod relations are created in the translation into Kodkod – `Key, Room, keys, FrontDesk` and `Guest`. Hence, these relations are bound by a set of variables following a specific scope defined in the amalgamated problem (Electrum model). After splitting the model, the next step it to generate the candidate configurations of $P\downarrow$ recurring to the Kodkod *solveAll* method, that given a formula and variable bounds a solution iterator is returned, thereby enabling the iteration over the configurations. Moreover, the *solve* method, considering a for-

mula and the variable bounds as well, a simple solution is returned. In Listing 6.1 and Listing 6.2, two simple configurations are depicted, resulting from the solving of $P\downarrow$. Configurations are partial solutions of the amalgamated model (P_0), as well as solutions of $P\downarrow$. Each one represents a specific distribution of a set of variables for each relation. The generation of such configurations is optimized by the Kodkod feature – the symmetry breaking mechanism – thus, just the non-symmetric configurations are generated. Note that, all possible configurations, the `FrontDesk` will be always filled be the variables `F0` with multiplicity `one` following its meaning in the Electrum model.

```
Key=[[K0],[K1],[K2],[K3]],
Ord.First=[[K0]],
Ord.Last=[[K3]],
Ord.Next=[[K0,K1],[K1,K2],[K2,K3]],
Guest=[],
Room=[[R3]],
keys=[[R3,K0],[R3,K1],[R3,K2],[R3,K3]]
FrontDesk=[[F0]]
```

Listing 6.1: First configuration

```
Key=[[K0],[K1],[K2],[K3]],
Ord.First=[[K0]],
Ord.Last=[[K3]],
Ord.Next=[[K0,K1],[K1,K2],[K2,K3]],
Guest=[[G1],[G2],[G3]],
Room=[[R2],[R3]],
keys=[[R2, K1], [R2,K2],[R2,K3],[R3,K0]]
FrontDesk=[[FrontDesk0]]
```

Listing 6.2: Second configuration

Once the candidate configurations have been generated, each one is being integrated into $P\uparrow$. The native support of the Kodkod for partial instances allows such integrations into the remainder problem.

Listing 6.3 illustrates the bounds of the integration of the Listing 6.1 or Listing 6.2 into $P\uparrow$. For each integrated problem, for the relations presented on $P\downarrow$ the search space is restricted by a set of variables presented in the configuration that gave rise, recurring to the Kodkod method `boundExactly` – this procedure is identified by $P\uparrow\oplus p$. For instance, given a certain integrated problem P_i, **`Conf`**$_i$**`.Key.solution`** is the set of variables of the relation `Key` on the configuration i. The remaining relations are properly bounded by a set of variables that remain unchanged for all integrated problems. Note that, `fd` is a TupleSet that represents the `FrontDesk` relation (just with a Tuple with a simple variable `F0`), `rb` is a TupleSet with the `Room` variables, `kb` with the `Key` variables and `gb` with the `Guest` variables.

The solving of each integrated problem is achieved recurring to the Kodkod method *solve*, considering the formula of $P\uparrow$ and the bounds depicted in 6.3.

```
bounds.bound(lastKey, fd.product(rb).product(kb));
bounds.bound(occupant, fd.product(rb).product(gb));
bounds.bound(currentKey, rb.product(kb));
bounds.bound(gkeys, gb.product(kb));
bounds.boundExactly(Key,Conf_i.Key.solution)
bounds.boundExactly(ordering/Ord.First,Conf_i.ordering/Ord.First.solution)
bounds.boundExactly(ordering/Ord.Last,Conf_i.ordering/Ord.Last.solution)
bounds.boundExactly(ordering/Ord.Next,Conf_i.ordering/Ord.Next.solution)
bounds.boundExactly(Guest,Conf_i.Guest.solution)
bounds.boundExactly(Room,Conf_i.Room.solution)
bounds.boundExactly(keys,Conf_i.keys.solution)
bounds.boundExactly(FrontDesk,Conf_i.FrontDesk.solution)
```

Listing 6.3: The bounds of an integrated problem P_i

6.4 EMPIRICAL EVALUATION

This section accesses the performance of the novel parallelization technique. Considering six examples – Dijkstra (Dijk), Handshake (Hand), Ring Election (Ring), Hotel Room System (Hotel), Red-Black Trees (RBT) and Span tree (Span) – the results of solving them by applying the new solving technique are explored here. Note that, two of these examples are not temporal models (the Handshake and the Red-Black Trees), thus, for these kind of examples, the obtainment of the partial problem has to be achieved manually, however intuitively, since there is no any criteria to do that, contrary to the temporal ones.

Performance results are divided for satisfiable and unsatisfiable problems for convenience since the expectation of good results is different for each case. However, the problems presented in satisfiable problems are the same that in unsatisfiable ones, so, the only thing that changes between them is the predicate over the model. These results are provided as follows: given a specification **Model** with a certain specific input $\mathbf{n}$, the number of configurations $p_{\#}$ generated by $P{\downarrow}$ and the satisfiability ratio $\mathbf{p}_{\%}$ are generated. In particular, the satisfiability ratio is the ratio of those solutions that are satisfiable configurations, more precisely, if there exists a partial problem that generates two configurations, or partial solutions, and the two integrated problems which are created from them are satisfiable, in this case the ratio is 1.00. Supposing that just one integrated problem was satisfiable, the ration would be 0.5. Moreover, the time results, in seconds, of solving any model are presented for three modes – the amalgamated mode ($\mathbf{T_0}$), the parallel mode ($\mathbf{T_p}$) and the hybrid model (T_h). Besides, the last column of the tables results is filled by the fractionated gain between the best and worst solving time. Note that, the timeout (TO) is set to 10 000 seconds and the bold values represent the best time to solve its specification.

Table 4 summarizes the results for the satisfiable problems. In these ones, as the pilot study predicts, for most of the problems (Dijk(1), Hand(1), Hotel(1) and RBT(1)) both the parallel and the hybrid modes outperform the amalgamated. Such outperforming is also applied to the problems with a large number of configurations and reduced satisfiability ratio – RBT(1).

The best speedup reached is for the Handshake example, where the gains are extremely large. For instance, at $\mathbf{n} = 14$ both the parallel and the hybrid modes are faster 80.9 seconds (the gain is 102.5 times) to solve the same specification. The generation of one configuration is the main factor for this outperforming, since there is only one integrated problem to solve. At $\mathbf{n} = 15$, as the number of configurations is zero, it takes 0.1 seconds to solve the problem. However, in this case, the amalgamated approach approximates its speedup with the parallel and the hybrid approaches.

In contrast, for Span (2) the speedups are less significant, where the gain goes up to 4 times. However, there are two examples (Ring (1) and Span (1)) where the amalgamated mode outperforms the two other ones. Such outperforming goes up to 1.3 seconds, which is almost insignificant. Besides, these problems do not take more than 5 seconds (amalgamated and hybrid modes), thus, the gain is marginal.

Model	n	$p_{\#}$	$(p_{\%})$	T_0	T_p	T_h	G
Dijk(1)	25	26	(0.92)	38.2	3.9	**3.6**	10.5
Dijk(1)	26	27	(0.93)	37.3	4.0	**3.8**	9.9
Dijk(1)	27	28	(0.93)	25.3	4.4	**4.1**	6.2
Dijk(1)	28	29	(0.93)	30.4	4.8	**4.3**	7.0
Dijk(1)	29	30	(0.93)	43.9	5.0	**4.6**	9.5
Dijk(1)	30	31	(0.94)	28.1	5.3	**5.0**	5.7
Hand(1)	14	1	(1.00)	81.7	**0.8**	**0.8**	102.5
Hand(1)	15	0	(1.00)	2.0	**0.1**	**0.1**	15.6
Hand(1)	16	1	(1.00)	1496.9	**9.7**	10.1	148.1
Hand(1)	17	0	(1.00)	40.6	**0.1**	0.2	266.6
Hand(1)	18	1	(1.00)	TO	**4.7**	5.0	$+\infty$
Hand(1)	19	0	(1.00)	2724.8	**0.2**	**0.2**	15570.1
Hand(1)	20	1	(1.00)	TO	**609.2**	744.7	$+\infty$
Hotel(1)	7	7016	(0.55)	44.8	2.0	**1.9**	23.4
Hotel(1)	8	12833	(0.60)	66.8	**2.5**	2.6	25.7
Hotel(1)	9	211470	(>0.6)	217.6	3.0	**2.7**	81.1
Hotel(1)	10	>999999	(>0.6)	227.8	3.2	**3.1**	73.2
Hotel(1)	11	>999999	(>0.6)	1105.3	**3.8**	**3.8**	290.2
Hotel(1)	12	>999999	(>0.6)	185.3	**4.4**	**4.4**	42.5
RBT(1)	9	4862	(0.01)	1.6	**0.4**	0.5	3.2
RBT(1)	10	16796	(0.00)	47.3	**3.1**	3.7	12.4
RBT(1)	11	58786	(0.00)	385.2	**3.8**	4.3	88.0
RBT(1)	12	208012	(0.00)	3914.0	**0.8**	0.9	4505.1
RBT(1)	13	>999999	(0.00)	TO	**12.0**	14.2	$+\infty$
Ring(1)	6	415	(0.01)	**2.7**	67.4	4.3	0.6
Ring(1)	7	2372	(0.08)	**1.1**	2.4	2.1	0.5
Ring(1)	8	16072	(>0.1)	**1.9**	190.9	4.0	0.5
Ring(1)	9	125673	(>0.1)	1.5	**1.1**	**1.1**	1.3
Ring(1)	10	>999999	(>0.1)	**1.5**	1.9.4	2.3	0.7
Ring(1)	11	>999999	(>0.1)	**2.4**	61.2	4.9	0.5
Ring(1)	12	>999999	(>0.1)	**3.3**	19.4	5.3	0.6
Span(1)	13	>999999	(1.00)	**0.9**	1.5	1.4	0.6
Span(1)	14	>999999	(1.00)	**1.1**	1.7	1.7	0.6
Span(1)	15	>999999	(1.00)	**1.2**	2.3	1.4	0.5
Span(1)	16	>999999	(1.00)	**2.0**	2.7	2.7	0.7
Span(1)	13	>999999	(1.00)	2.8	2.6	**1.4**	1.0
Span(1)	14	>999999	(1.00)	15.3	3.4	**3.3**	4.6
Span(1)	15	>999999	(1.00)	9.2	**3.8**	**3.8**	2.4
Span(1)	16	>999999	(1.00)	6.4	**4.6**	5.0	1.3

Table 4: Satisfiable models evaluation

Table 5 summarizes the results for the satisfiable problems. As predicted in the pilot study, for unsatisfiable problems the parallel approach is unclear, in which its execution is often outperformed

by the amalgamated execution. Hence, a hybrid approach was proposed to compensate the losses. Considering this, as the results indicate, he amalgamated approach is never more than 2 times faster than the hybrid approach. However, for the most cases that the amalgamated execution outperforms the hybrid strategy, the results are balanced by the results of their satisfiable version for such example. More precisely, this means that the losses for unsatisfiable problems are often overcome with gains for satisfiable problems, reaching a gain in average.

Curiously, for RBT(2) and Ring(2), both the parallel and the hybrid approaches outperform the amalgamated approach. Considering the former example, the speedup ranges from 1.1 up to 7.2 times, whereas for the latter, the speedup ranges from 14.1 up to 601.3 times. Note that, the execution these two problems times out for larger $\mathbf{n}$ values, however the execution for the two other approaches terminates before of it.

After analysing these results, it is easy to denote the gain of this strategy is not easy to predict because depends on:

- the performance for the amalgamated problem P_0
- the performance for the partial problems $P{\downarrow}$
- the performance for the integrated problems $P{\uparrow}{\oplus}p$
- the number of configurations generated by $P{\downarrow}$
- the ratio of those that can be extended to complete solutions

Now, it is time to analyse the advantages and the disadvantages associated with this technique. Basically the advantages are related to a better performance for the satisfiable problems and the creation of a hybrid mode to handle the unsatisfiable ones in order to balance the performance. Nonetheless, the manual decomposition of the models and the key ideas not be new are the aspects that represent disadvantages of this technique. The similar approach is the Ranger technique, previously explored in Sub-Section 3.3.2. However, the manual decomposition is a point addressed on this thesis, since there is a natural splitting on temporal models – dynamic and static properties (slicing implementation described in Section 4.6). Following that, the configurations are generated from the static component presented on the specification model, then for each one, an integrated problem consists in joining it with the dynamic properties.

Model	n	$p_{\#}$	$(p_{\%})$	T_0	T_p	T_h	G
Dijk(2)	25	26	(0.00)	**19.8**	52.9	27.7	0.7
Dijk(2)	26	27	(0.00)	**17.9**	67.5	26.2	0.7
Dijk(2)	27	28	(0.00)	**23.4**	80.4	35.6	0.7
Dijk(2)	28	29	(0.00)	**24.3**	94.8	35.1	0.7
Dijk(2)	29	30	(0.00)	**30.9**	110.0	45.3	0.7
Dijk(2)	30	31	(0.00)	**31.6**	128.6	46.9	0.7
Hand(2)	12	1	(0.00)	**6.0**	13.8	6.2	1.0
Hand(2)	13	0	(0.00)	0.2	**0.1**	**0.1**	2.0
Hand(2)	14	1	(0.00)	**127.3**	1060.3	145.1	0.9
Hand(2)	15	0	(0.00)	2.5	**0.1**	**0.1**	18.7
Hand(2)	16	1	(0.00)	**2504.2**	TO	3361.0	0.7
Hotel(2)	4	75	(0.00)	**6.4**	11.7	11.5	0.6
Hotel(2)	5	312	(0.00)	**66.8**	96.7	129.3	0.5
Hotel(2)	6	1421	(0.00)	**234.5**	1095.4	458.9	0.5
Hotel(2)	7	7016	(0.00)	**776.1**	TO	1434.7	0.5
Hotel(2)	8	12833	(0.00)	**2023.6**	TO	3508.6	0.6
RBT(2)	9	4862	(0.00)	7.4	**6.1**	7.0	1.1
RBT(2)	10	16796	(0.00)	77.7	**23.5**	26.5	2.9
RBT(2)	11	58786	(0.00)	821.4	**96.2**	113.6	7.2
RBT(2)	12	208012	(0.00)	TO	**496.7**	623.3	$+\infty$
Ring(2)	4	24	(0.00)	1.1	**0.6**	0.8	14.1
Ring(2)	5	89	(0.00)	4478.6	**6.1**	7.4	601.3
Ring(2)	6	415	(0.00)	TO	**235.6**	316.4	$+\infty$
Ring(3)	5	89	(0.00)	**1.6**	2.3	2.5	0.6
Ring(3)	6	415	(0.00)	**4.8**	12.1	7.9	0.6
Ring(3)	7	2372	(0.00)	**14.5**	103.8	23.8	0.6
Ring(3)	8	16072	(0.00)	**75.9**	1087.5	133.1	0.6
Span(1)	5	58	(0.00)	**0.2**	1.2	0.3	0.7
Span(1)	6	457	(0.00)	**0.6**	13.2	1.0	0.6
Span(1)	7	5777	(0.00)	**4.5**	662.4	8.3	0.5
Span(2)	5	5	(0.00)	**0.4**	3.2	0.7	0.6
Span(2)	6	6	(0.00)	**0.6**	30.1	1.1	0.5
Span(2)	7	7	(0.00)	**1.8**	817.5	4.0	0.5

Table 5: Unsatisfiable models evaluation

7

CONCLUSIONS AND FUTURE WORK

The Electrum language is well-suited to verify dynamic systems with rich configurations. However, its BMC has limitations, thus, this dissertation aims to overcome them. Particularly, it focuses on improving the analysis procedure of Electrum models, namely, the definition of its semantics through a translation into Kodkod as well as a novel procedure of verifying Electrum models in parallel. To achieve these goals, a temporal extension to the Kodkod constraint solver was developed, which is crucial to achieve such goals.

The goals initially proposed in the dissertation plan have been achieved. Particularly, one of this research's goals consisted in implementing a parallel version of Electrum's BMC. However, to verify whether the parallel version of this tool pays off, a pilot study was performed to assess its potential, where the splitting of the models was achieved manually. Nonetheless, dynamic systems with rich configurations have a natural splitting – the dynamic and static properties. Hence, an automatic splitting was easy to perform at back-end level. However, this parallelization strategy can also be applied to the verification of system purely static, where there is no such natural division.

The first version of the Electrum's BMC tool performed the models' analysis by translating each Electrum model to Alloy. At this point, the models were analyzed by translating them into Kodkod, which in turn it analyzes each problem by using an off-the-shelf SAT solver. Thus, to skip a layer of complexity, another goal of this Master's Thesis was to translate directly the Electrum models into Kodkod. Nonetheless, Kodkod does not support the temporal component presented in Electrum. This way, a temporal extension to the Kodkod constraint solver was carried out with the purpose of clarifying its semantics. Despite of supporting the temporal operators and the variable signatures/fields of Electrum, it performs the automated splitting of the models with the purpose of applying the parallelization strategy and it implements the classical temporal bounded model-checking constraints such as: the transformation into the negative normal form, the loop constraint, the order over the time, among others previously explained. Note that, the same way that Electrum is an extension of Alloy, the Electrum back-end is an extension of the Alloy back-end.

Furthermore, the integration of the Kodkod temporal extension into the Electrum's BMC tool was not straightforward in the part of analyzing the solutions from Kodkod. In particular, the visualization of the instances in the Electrum's BMC Visualizer. To achieve it, the Visualizer was extended to

handle temporal models, by creating a specific feature to iterate over the states of the temporal trace. Besides, to iterate over the trace, the temporal instance is distributed for a set of xml files.

Concluding, this dissertation presents the following contributions:

- a temporal extension to the Kodkod constraint solver
- a new Electrum's BMC that translates its models directly into Kodkod and back into Electrum's BMC Visualizer
- a new verification strategy that performs the analysis the Electrum models in parallel
- a comparison between the old (sequential) and the new (parallel) solving procedures

This dissertation has some topics that can be improved in the future, concretely, to improve the parallelization strategy as well as the Electrum's BMC tool. Such topics are in concrete defined as follows:

- the creation of a feature that performs an automatic division of any Kodkod model with the purpose of automatically applying the parallelization strategy not only in temporal problems
- the parallelization strategy has a multithread architecture, which leads to exist a cache interference in the solving. A solution would be to perform it with a multiprocess architecture
- the integration of the unbounded Electrum model-checker, built over SMV, into the tool developed in this thesis
- the creation of a new Electrum's BMC Visualizer, since the current one is an adaption ad hoc adaptation of the Alloy Visualizer

BIBLIOGRAPHY

Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 131–146, 2010. doi: 10.1007/978-3-642-14052-5_11. URL http://dx.doi.org/10.1007/978-3-642-14052-5_11.

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 334–342, 2014. doi: 10.1007/978-3-319-08867-9_22. URL http://dx.doi.org/10.1007/978-3-319-08867-9_22.

Wahid Chrabakh and Richard Wolski. Gridsat: A chaff-based distributed SAT solver for the grid. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*, page 37, 2003. doi: 10.1145/1048935.1050188. URL http://doi.acm.org/10.1145/1048935.1050188.

Andreas Classen. Modelling with fts: a collection of illustrative examples. *PReCISE Research Center, University of Namur, Namur, Belgium, Tech. Rep. P-CS-TR SPLMC-00000001*, 2010.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 335–344, 2010. doi: 10.1145/1806799.1806850. URL http://doi.acm.org/10.1145/1806799.1806850.

Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439, 2014. doi: 10.1016/j.scico.2013.09.019. URL http://dx.doi.org/10.1016/j.scico.2013.09.019.

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

Alcino Cunha. Bounded model checking of temporal formulas with alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 303–308, 2014. doi: 10.1007/978-3-662-43652-3_29. URL http://dx.doi.org/10.1007/978-3-662-43652-3_29.

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120, 2006. doi: 10.1145/1146238.1146251. URL http://doi.acm.org/10.1145/1146238.1146251.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SATModular verification of code with 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003. doi: 10.1007/978-3-540-24605-3_37. URL http://dx.doi.org/10.1007/978-3-540-24605-3_37.

Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. Dynalloy: upgrading Alloy with actions. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 442–451, 2005. doi: 10.1145/1062455.1062535. URL http://doi.acm.org/10.1145/1062455.1062535.

Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36, 2010. doi: 10.1145/1831708.1831712. URL http://doi.acm.org/10.1145/1831708.1831712.

Luís Gil, Paulo F. Flores, and Luís Miguel Silveira. PMSat: a parallel version of MiniSAT. *JSAT*, 6 (1-3):71–98, 2009. URL http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_5_Gil.pdf.

Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4): 245–262, 2009. URL http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf.

Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable and undecidable fragments of first-order branching temporal logics. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 393–402, 2002. doi: 10.1109/LICS.2002.1029847. URL http://dx.doi.org/10.1109/LICS.2002.1029847.

Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Julian Stecklina, and Peter Steinke. A short overview on modern parallel SAT-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011.

Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000.

Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN 978-0-262-10114-1. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.

Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing Satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001. doi: 10.1016/S1571-0653(04)00321-X. URL http://dx.doi.org/10.1016/S1571-0653(04)00321-X.

Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. doi: 10.1145/177492.177726. URL http://doi.acm.org/10.1145/177492.177726.

Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

Leslie Lamport. The PlusCal algorithm language. In *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, pages 36–60, 2009. doi: 10.1007/978-3-642-03466-4_2. URL http://dx.doi.org/10.1007/978-3-642-03466-4_2.

Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple is better: Efficient bounded model checking for past LTL. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, pages 380–395, 2005. doi: 10.1007/978-3-540-30579-8_25. URL http://dx.doi.org/10.1007/978-3-540-30579-8_25.

Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *FSE 2016*. ACM, 2016. to appear.

Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pages 360–375, 2004. doi: 10.1007/11527695_27. URL http://dx.doi.org/10.1007/11527695_27.

Ricardo Marques, Luís Gil Silva, Paulo F. Flores, and Luís Miguel Silveira. cmcSAT - A cooperative multicore SAT Solver. 2012.

Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Improving search space splitting for parallel SAT solving. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 336–343, 2010. doi: 10.1109/ICTAI.2010.56. URL http://dx.doi.org/10.1109/ICTAI.2010.56.

Ruben Martins, Vasco M. Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012. doi: 10.1007/s10601-012-9121-3. URL http://dx.doi.org/10.1007/s10601-012-9121-3.

Joseph P. Near and Daniel Jackson. An imperative extension to Alloy. In *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 118–131, 2010. doi: 10.1007/978-3-642-11811-1_10. URL http://dx.doi.org/10.1007/978-3-642-11811-1_10.

Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010, San Jose, CA, USA, November 7-12, 2010*, 2010. URL https://www.usenix.org/conference/lisa10/margrave-tool-firewall-analysis.

Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001. doi: 10.1016/S0167-6423(00)00018-6. URL http://dx.doi.org/10.1016/S0167-6423(00)00018-6.

S Plaza, I Markov, and Valeria Bertacco. Low-latency SAT solving on multicore processors with priority scheduling and xor partitioning. In *International workshop on logic and synthesis*, 2008.

Nicolás Rosner, Carlos Gustavo López Pombo, Nazareno Aguirre, Ali Jaoua, Ali Mili, and Marcelo F. Frias. Parallel bounded verification of Alloy models by transcoping. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, pages 88–107, 2013a. doi: 10.1007/978-3-642-54108-7_5. URL http://dx.doi.org/10.1007/978-3-642-54108-7_5.

Nicolás Rosner, Junaid Haroon Siddiqui, Nazareno Aguirre, Sarfraz Khurshid, and Marcelo F. Frias. Ranger: Parallel analysis of Alloy models by range partitioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA,*

November 11-15, 2013, pages 147–157, 2013b. doi: 10.1109/ASE.2013.6693075. URL http://dx.doi.org/10.1109/ASE.2013.6693075.

Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*, pages 136–145, 2006. doi: 10.1109/RE.2006.23. URL http://dx.doi.org/10.1109/RE.2006.23.

Shohei Shimizu, Patrik O. Hoyer, Aapo Hyvärinen, and Antti J. Kerminen. A linear non-gaussian acyclic model for causal discovery. *Journal of Machine Learning Research*, 7:2003–2030, 2006. URL http://www.jmlr.org/papers/v7/shimizu06a.html.

Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Electronic Notes in Discrete Mathematics*, 9:19–35, 2001. doi: 10.1016/S1571-0653(04)00311-7. URL http://dx.doi.org/10.1016/S1571-0653(04)00311-7.

Ilya Shlyakhter. *Declarative symbolic pure-logic model checking*. PhD thesis, Massachusetts Institute of Technology, 2005.

Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*, pages 249–258, 2007. doi: 10.1007/978-3-540-68111-3_27. URL http://dx.doi.org/10.1007/978-3-540-68111-3_27.

Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.

Emina Torlak and Greg Dennis. Kodkod for Alloy users. In *First ACM Alloy Workshop, Portland, Oregon*, 2006.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 632–647, 2007. doi: 10.1007/978-3-540-71209-1_49. URL http://dx.doi.org/10.1007/978-3-540-71209-1_49.

Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, editors. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier, 2008. ISBN 978-0-444-52211-5. URL http://www.sciencedirect.com/science/bookseries/15746526/3.

Pascal Vander-Swalmen, Gilles Dequen, and Michaël Krajecki. A collaborative approach for multi-threaded SAT solving. *International Journal of Parallel Programming*, 37(3):324–342, 2009. doi: 10.1007/s10766-009-0097-6. URL http://dx.doi.org/10.1007/s10766-009-0097-6.

Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4):543–560, 1996. doi: 10.1006/jsco.1996.0030. URL http://dx.doi.org/10.1006/jsco.1996.0030.

Jain Zhang. *The generation and applications of finite models*. PhD thesis, PhD thesis, Institute of Software, Academia Sinica, Beijing, 1994.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.







