Finding hard initial configurations of Rush Hour with Binary Decision Diagrams

Frédéric Servais
Acknowledgment

THIS WORK IS PROVIDED BY THE AUTHOR “AS IS” AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.

IN NO EVENT SHALL THE AUTHOR OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THIS WORK, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THIS BEING SAID, I WANT TO EXPRESS MY GRATITUDE TO EVERYONE WHO DID TAKE ANY KIND OF RESPONSIBILITY FOR THIS WORK, IN ALPHABETICAL ORDER: DAD, ERIC, FREDERIC, IVAN, KOEN, MOM, OLIVIER, SYLVESTRE, THIERRY, ALL THE PEOPLE FROM THE DEPARTMENT D’INFORMATIQUE (ESPECIALLY PIERRE GANTY AND CEDRIC MEUTER) AND LAST BUT NOT LEAST MY PROMOTEUR JEAN-FRANCOIS RASKIN FOR ITS TIME AND ALL HIS INVALUABLE COMMENTS.
## Contents

1 Introduction ................................................. 9
   1.1 The game - Rush Hour ............................... 9
   1.2 The goal - Finding hard initial configurations .......... 10
   1.3 The means - Verification tools ....................... 11
   1.4 The plan ............................................. 12

2 Rush Hour, the game ........................................ 13
   2.1 Complexity classes ................................. 13
   2.2 Generalized Rush Hour .............................. 15
   2.3 Generalized Rush Hour is NP-Hard ................. 16
      2.3.1 Inductive constraints ......................... 17
      2.3.2 basic building blocks .......................... 17
      2.3.3 GRH logic ...................................... 19
      2.3.4 GRH is NP-Hard ................................ 19
   2.4 Rush Hour is in PSpace ............................... 20
2.5 Rush Hour is PSpace-complete ........................................ 21
2.6 Conclusion ................................................................. 22

3 BDDs ........................................................................ 24

3.1 Introduction ................................................................. 24
3.2 Canonicity Lemma ......................................................... 27
3.3 Examples and Maximum size ROBDDs ............................... 28
  3.3.1 Examples ............................................................... 28
  3.3.2 Maximum size ROBDDs ........................................... 31
3.4 Variables ordering ......................................................... 33
3.5 Lower bounds ............................................................... 35
3.6 Operations complexity ................................................... 37
  3.6.1 Reduction ............................................................. 37
  3.6.2 Equivalence .......................................................... 37
  3.6.3 Tautology, Satisfiability, SAT-count. ......................... 38
  3.6.4 Synthesis .............................................................. 38
  3.6.5 Complement .......................................................... 40
  3.6.6 Substitution .......................................................... 41
  3.6.7 Quantification ........................................................ 41
3.7 Conclusion ................................................................. 41
4 Modeling

4.1 NuSMV

4.2 Modeling language

4.2.1 Expressions

4.2.2 VAR and ASSIGN declarations

4.2.3 MODULE declaration

4.2.4 Processes

4.2.5 INIT declaration

4.2.6 TRANS declaration

4.2.7 INVAR declaration

4.3 Model building

4.3.1 Transformation steps

4.3.2 Variables ordering, reordering

4.4 Algorithms

4.4.1 Forward image

4.4.2 Backward image

4.4.3 Reachable states

4.4.4 System diameter

4.4.5 Disjoint transition relation, monolithic

5 First Rush Hour SMV Specification
5.1 Specification description ........................................ 55
  5.1.1 Car module .................................................. 57
  5.1.2 VAR declaration ............................................. 58
  5.1.3 Main’s INVAR declaration .................................. 58
  5.1.4 INIT declaration ............................................ 59
5.2 Specification results and limits ................................. 59
  5.2.1 Empirical results .......................................... 60
  5.2.2 Theoretical limits ......................................... 61
5.3 Conclusion ..................................................... 69

6 Second Rush Hour SMV specification .............................. 71
  6.1 Specification description ....................................... 71
    6.1.1 Primary and Secondary car modules declaration ......... 73
    6.1.2 VAR declaration ......................................... 74
    6.1.3 INIT declaration ........................................ 75
    6.1.4 INVAR declaration ....................................... 75
  6.2 Finding the hardest configuration ............................ 77
  6.3 Finding interesting hard initial configurations ............ 80
    6.3.1 Size-2 Rush Hour ....................................... 81
    6.3.2 Generalized Rush Hour ................................... 85
  6.4 Partitioning .................................................. 86
Chapter 1

Introduction

1.1 The game - Rush Hour

Rush Hour is a commercial one player sliding blocks puzzle. Pieces representing cars and trucks are placed on a 6x6 board. Cars and trucks take two and three board cells respectively. They are placed horizontally or vertically and can only move forward or backward, they cannot leave their line or column. The board is closed except at the end of the third line, this is what we call the exit point. A special car, the red car also called target car, is placed on the same line as the exit point. The goal is to move the cars and trucks around in order to free the target car, i.e. to permit it to leave the board through the exit point.

40 cards describing cars position are provided, these are some initial configurations. They are classified in 4 levels: beginner, intermediate, advanced and expert.

In figure 1.1 the traffic jam is made up of 2 trucks and 3 cars plus the red car.

We call a move a selection of car, direction and amplitude. It is legal to move the selected car in the given direction of the number of cells given by the amplitude, if it does not lead to any collision with another car nor with the board limits. A solution is a sequence of legal moves, such that after applying these moves to the initial configuration the target car is in front of the exit point, ready to leave. Finally we call a step a move of amplitude 1.
A solution in 9 moves is:

1. RED, right, 1
2. E, down, 4
3. RED, left, 1
4. A, left, 2
5. C, up, 3
6. D, left, 4
7. C, down, 3
8. B, down, 3
9. RED, right, 4

1.2 The goal - Finding hard initial configurations

We call initial a configuration that is given as input for being solved. The Rush Hour package provides 40 such initial configurations belonging to 4 hardness categories. The main goal of this work is to find hard initial configurations.

Hard is a lousy, subjective term. Let’s define it.
The first objective property of a solution is its length. A hard configuration should require a lot of moves. But if a solution is made up of many moves, our intuition tells us that it still could be easy. Indeed if at each step there are very few possible moves, it won’t be that difficult to find the way out. On the other hand, if there are too many possible moves maybe a lot of paths lead to the solution. It is not easy to come with a precise conceptualization of this second intuition.

All in all we retain the first property and we call hard initial configuration one that requires a lot of moves or steps (relatively to the other possible configurations).

Once we have found these hard configurations, we will provide the amount of possible moves (reachable states), thus bringing the bread and butter for a qualitative analysis of initial configuration’s hardness.

1.3 The means - Verification tools

Finding hard initial configurations can be done with exhaustive state space analysis. However, doing so, one quickly faces a big hurdle: combinatorial explosion. If one can find some simplifications, like removing useless configurations in order to decrease significantly the state space, then that may lead to some solutions. Samuel Hiard did just that in his ”Recherche automatique de solutions difficiles pour le jeu Rush Hour” [28]. He managed to subdue the state space explosion problem for 6x6 boards, thanks to an impressive optimization work and a thorough analysis of the game, leading to simplifications.

We embarked on a totally different road. The main idea of our work was to try to bypass this explosion problem by using symbolic methods, just like it is done in the field of verification.

Symbolic methods represent sets in some compact way instead of extensively dealing with their elements. Our method to find the hard initial configurations is starting with the set of winning Rush Hour configurations and then iteratively computing, still symbolically, the successive sets of reachable configurations.

We will use what is called binary decision diagrams, BDDs, as data structures that
represent the involved sets. These have indeed some suitable properties and are extensively used in verification problems that deal with state space explosion problem.

Finally we have elected NuSMV as modeling language and verification tool. It will help us define the specification of Rush Hour and translate it into BDDs. Since NuSMV is open source, this environment provides all we need in order to implement our iterative algorithms. all the required low-level routines are already implemented in NuSMV.

1.4 The plan

We start playing, and getting to know, Rush Hour in chapter 2. There we present the results of Flake and Baume [1] showing that Rush Hour, or better a Generalized version of Rush Hour is PSpace-complete. We then outline the consequences and some side results.

In chapter 3 we presents the data structure: binary decision diagrams. After getting to know BDDs, we exhibit some important properties as well as their limitations. Finally we describe the operations available on them and their respective computational complexity.

In chapter 4, we describe the NuSMV’s modeling language. We briefly explain how NuSMV works, how it transforms a specification into BDDs. And finally we describe the low-level routines available, and how they are tied together.

Our first results are presented in the 5th chapter. After the description of our specification, we give the results and try a theoretical explanation of the total failure of this first attempt.

Finally, our second and last attempt is described in the last chapter, concluding our work on mitigated results.
Chapter 2

Rush Hour, the game

In this section we present the Rush Hour game and show that it is theoretically not an easy one. We follow the article of Flake and Blaum showing that Rush Hour is indeed P-Space complete. [1]

2.1 Complexity classes

We now remind some basic notions of complexity theory, for a deeper understanding look into [3].

We call a problem a set of questions which, in turn, are the instances of this problem. "How long is the shortest solution for the above instance of Rush Hour?" is a question, a particular instance of the general optimization problem "find the length of the shortest solution of any Rush Hour problem". A solution to this particular instance is merely a number say "9". A solution to the general optimization problem would be in the form of an algorithm calculating, for any particular instance, the length of the shortest way out.

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem, as a function of the size of the input, using the most efficient algorithm. The space complexity is the size of the memory needed to solve an instance of the problem as a function of the size of the input.
A **decision problem**, is a problem where the answer is YES or NO. "Is there a solution for the above instance of Rush Hour?" is an instance of the decision problem associated with Rush Hour.

Decision problems fall in different complexity classes:

- **P** contains the problem that can be decided by a deterministic turing machine in polynomial time. Proving that a problem is in *P* is merely to give a polynomial algorithm for solving any instance of this problem.

- **NP** is the class of problems that can be solved by a non-deterministic turing machine in polynomial time. To give a polynomial algorithm that check if a solution satisfies an instance of the problem suffice for proving that this problem is in *NP*.

- **PSpace** contains problems that can be solved with a deterministic turing machine using a polynomial space. *PSpace* was proved to be equal to *NP* as a corollary of Savitch’s theorem [2], so providing an algorithm that is polynomial in space and checks, given a solution and a problem instance, if this solution satisfies the instance, is enough for proving that this problem is in *PSpace*.

Dozens of other classes have been defined, but these are the basic, the ones of interest for us here.

We have the following hierarchy:

\[ P \subseteq NP \subseteq PSpace \]

The one million dollars question being to tell if the inclusion are or not equalities.

A **reduction** in complexity theory is the transformation of one problem into another one, in the sense that a solution for the second would lead to a solution to the former.

We say that a decision problem is **Hard** with regard to a complexity class if any problem of this class is reducible in polynomial time to this initial problem, i.e. if there exists a polynomial time algorithm that transform any instance of the second
problem into an instance of the initial problem in such a way that both instances are true or false together.

We say that a decision problem is complete with regard to a complexity class if it belongs to this class and is Hard with regard to this class.

So NP-Complete problems are the most difficult problems in NP, in the sense that an algorithm for solving any NP-complete problem permits to solve any NP problem with the same complexity than the original algorithm. SAT, the boolean satisfiability problem, was the first problem to be shown being NP-Complete [4], proving also that NP-Complete problem do exist. Other NP-Complete problems are the traveling salesman decision problem, the hamiltonian path problem and the vertex cover problem. Proving that a problem is NP-Complete is always done by reduction to an already known NP-Complete problem.

NP-Hard problems are the ones that are as hard as any NP problem. So a NP-Hard problem can be reduced to any NP problem just as in the case of NP-Complete problems, but we do not ask NP-Hard problem to be in NP.

PSpace-complete problems are the hardest problem in PSpace. Any PSpace problem is equivalent to a linear bounded turing machine. A PSpace problem that can be shown to have, for any given such turing machine, an instance that is, in some way, equivalent to this turing machine, is therefore PSpace-complete.

2.2 Generalized Rush Hour

Computational complexity is a function of the size of the input. The size of Rush Hour in its original form is limited as no more than 36 cell and 18 cars need to be encoded. We therefore need to generalize Rush Hour in order to analyze its complexity, i.e. we need to define a general problem containing instances with arbitrary size.

We define the problem Generalized Rush Hour (GRH) as a board with cars and trucks. The board is of arbitrary width and height and an exit point is positioned on an arbitrary extreme board cell. The rules and goal are the Rush Hour ones.
Because we will use polynomial reduction, we are interested in size measure with polynomial precision. We can define the size of the 'input' as the size of the board. Indeed the other information defining the configuration is the position of the cars, but there can be no more cars than half the cells, which are no more than the board size squared.

Other Rush Hour derivative have been defined and studied. J. Tromp and R. Cilibrasi [8] introduced the followings:

- Size-2 Rush Hour: is defined as GRH with only size-2 cars.
- Unit Rush Hour: GRH with size-1 cars
- Walled Rush Hour: GRH enhanced with wall, i.e. fixed block on board cells

We will mainly focus our attention on GRH and Size-2 Rush Hour. We will see that they both have the same theoretical complexity, the latter being simpler in its formulation.

We are now ready to study the complexity of GRH.

### 2.3 Generalized Rush Hour is NP-Hard

G. Flake and E. Baum show that the Rush Hour decision problem [1] is NP-Hard. They proceed by reduction to SAT.

We can map any boolean proposition to an instance of Generalized Rush Hour in polynomial time, in such a way that the boolean proposition is satisfiable if and only if the associated Rush Hour instance has a solution i.e. if there is a sequence of moves such that in the end the red car finds a way out.

The construction relies on several tricks that we detail in the next sections.
2.3.1 Inductive constraints

A packed line refers to a set of cars on the same row or column with no space between each other.

A constrained line is a packed line that can shift by at most two cells.

An anchored line is a packed line with both of its ends constraint by the board border, another anchored line or a constrained line. In the last case, the packed line must touch the constrained line in such a way that it must be impossible moving the constrained line to free the packed line.

Anchored and constrained lines are central in the construction of the building blocks detailed here after.

2.3.2 basic building blocks

In the following constructions anchored lines are represented with grey cars whereas constrained line are with white cars. The bottom-most white car, if one, and the leftmost white car, if one, are called the input. The output are the topmost (if one) and rightmost white cars. We say that an input is closed if the concerned car as moved in the square-center opposite direction. An output is open if the concerned white car has moved in the square center direction.

We present the 9 basic building blocks, they must be thought as part of a big board, the anchored line (grey cars) are in some way blocked by the board borders (see definition above), so they are fixed.

The turn and pass output can open if and only if the input is closed, as figure 2.1 shows.

The intersection upper, respectively right, output can open if and only if the lower, respectively left, input is closed. Any split output can open only if the input is closed. This is shown in figure 2.2.

The conjunction output can open only if both input are closed. The disjunction
output can open if any or both inputs are closed. This is shown in figure 2.3

The switch has no input, its output cannot be both open together. It will be associated to a boolean variable and will represent an assignation. Each output, upper left and right white cars, representing the TRUE and FALSE assignment respectively. So the open output, if one, means the represented variable has been assigned the output corresponding value. This is shown in figure 2.4.

It is important to note that these building blocks can be imbricated in such a way that border anchored line fit together and the input and output will match. We use this point in the following higher level constructions.
Based on these building blocks we can now construct simple logical proposition namely NOT, AND and OR, as is shown in figure 2.6. We use for this the iconic form, shown in figure 2.5, representing the blocks introduced in the previous section.

2.3.4 GRH is NP-Hard

Showing that SAT can be polynomially reduced to GRH decision problem is now straightforward. For each variable in the boolean expression we construct a switch block representing its assignment (TRUE or FALSE), all initialized at '-' . We connect these switches with the GRH logic circuitry (building blocks and logic) according to the boolean expression.
Figure 2.5: Iconics from left to right: pass, turn, split, intersection, empty, disjunction, conjunction, switch.

We then position the target car at the final output, as in figure 2.7, so that it will be freed if and only if the final output opens.

Since all the building blocks and logic circuitry are constant in size and that the corresponding circuit is linear in the size of the boolean expression, the transformation is clearly polynomial in the number of cars. The boolean expression will be satisfiable if and only if there is some path to free the target car. The solution can be ’decompiled’ by reading the assignment of each variable once the target car is freed.

We provide in the figure 2.8 an example for the formula \((\neg A \land B) \lor C\).

### 2.4 Rush Hour is in PSpace

The decision problem of GRH is in \textit{PSpace}. As we mentioned above, to show this we can describe a non deterministic turing machine (NTM) that solve GRH decision problem in polynomial space, indeed \textit{PSpace} is the same as \textit{NPSpace}.

On the tape is the initial configuration. If this configuration is a winning one, the NTM halts. Otherwise the turing machine will guess a move. This move is through the shortest path leading to a winning configuration if such a path exists. The NTM transforms the configuration into the configuration obtained after the guessed move. The NTM repeat this until a winning configuration is found. All this can be done in space that is linear in the number of cars, proving that GRH is in \textit{PSpace}. 
2.5 Rush Hour is PSpace-complete

G. Flake and E. Baum then show that Generalized Rush Hour is PSpace-complete by simulating a bounded turing machine. Their construction is rather technical and of few interest here. However their idea of partitioning the whole board into blocks of constant size using anchored and constrained cars line was reused and even generalized by Hearn and Demaine [5][6] in order to systematize $PSpace – hardness$ result for sliding blocks puzzles. Hearn and Demaine’s construction give rise to a simpler and more elegant proof of the $PSpace – hardness$ of GRH.

They based their model on weighted directed graphs with constraints on the minimum in-flow for vertices. A computation will consist of reversing some edges without breaking the defined constraints. They proved, by reduction to Quantified Boolean
Formulas, that deciding whether it is possible to reverse the direction of a particular edge is PSpace-complete.

Now the central point is that their model of computation requires only 2 blocks (similar to the blocks in Flake and Blaum construction), namely a ”And” and ”Latch”. The ”And” having the same property as above: the output can open if and only if both input are closed. The ”Latch” having the following property: Only one of both input can be open at a time, and switching from one output to the other requires the input to close at least once.

We won’t go into more details here, as it is out of the scope of this work.

### 2.6 Conclusion

Provided that $PSPACE \neq NP – Complete$, Generalized Rush Hour decision problem being PSpace complete means there exists a sequence of instances of increasing size such that the associated shortest solution length is exponential in the size of the instance. Indeed if any such sequence has only polynomial shortest solution length in the size of the associated instance then a Non-deterministic Turing machine could
guess and output the solution in polynomial time, the problem would then be NP-easy which yields a contradiction with the hypothesis.

Flake and Blaum speculated that trucks, cars of length 3, were essential in their construction, but Tromp exhibited all the needed blocks constructed with cars of length 2, proving that size-2 Rush Hour was also PSpace-complete [7].

Tromp and Cilibrasi followed their reasoning and studied Unit Rush Hour [8] they were not able there to prove similar results. However they studied how the worst case solution length grows with the problem size. As we noted above exponential growth rate is mandatory for PSpace-hardness. Tromp and Cilibrasi implemented an exhaustive state space search program in order to find Unit Rush Hour worst-case solution length for different problem sizes. However because of the state explosion problem they restricted there attention to instances with only 1 empty cell. Even with this restriction a 6x6 grid was the computational limit of their exploration, and they found an instance requiring 732 moves, with 35 size-1 cars.

With these limited results they saw an indication that the worst case solution length growth rate was exponential, and therefore this is an indication, very weak, in the direction of Unit Rush Hour being PSpace-complete.

There is no special algorithm for solving a Rush Hour instance. Basically, we have two approaches: breadth-search first or width-search first. The latter sometimes inspired by Dijkstra algorithm [“Rush Hour and Dijkstra’s Algorithm”, Mark Stamp, Brad Engel, McIntosh Ewell, Victor Morrow]. The former being more memory efficient than the latter but not necessarily faster.
Chapter 3

BDDs

Reduced Ordered Binary Decision Diagrams (ROBDD) canonically represent boolean functions, or sets, as direct acyclic graphs. They provide computation of operations like satisfiability, equivalence or tautology with excellent complexity. ROBDDs have been introduced by R. Bryant [11] in 1986 as a restricted form of the Binary Decision Diagrams introduced by Akers [14] in 1978, also known as branching program. They are the basis for powerful symbolic analysis. This section presents these data structures with their properties and limitations.

3.1 Introduction

Binary tree. In the example shown in figure 3.1 the tree represents the boolean function \((x_1 \leftrightarrow y_1) \lor (x_2 \leftrightarrow y_2)\):

Each vertex is associated with a variable, the dotted line denotes the low-branches, i.e. when the value of the associated variable is 0, while the plain lines are for high-branches, the variable value is 1. An assignment is giving a value to each variable. To evaluate the function represented by the tree for a given assignment, one must follow the tree branches according to the variable values, this path leads to a leaf labeled 0 or 1 which is the value of the function for this assignment.

So we can represent a boolean expression by a decision tree, each vertex of the tree
Removing redundant leafs (0 and 1)
Removing non-unique y\_2 nodes

Removing redundant leafs (0 and 1)
Removing non-unique y\_2 nodes
Removing non-unique x\_2 nodes

Figure 3.1: \((x_1 \leftrightarrow y_1) \lor (x_2 \leftrightarrow y_2)\)

Figure 3.2: reduction

is associated with a boolean variable and every path is leading to the truth value of the boolean expression according to the assigned boolean values. Note that in the example the variables are following the same order on every path, this is not mandatory for a binary decision tree.

**Reduction.** In the example from figure 3.1, instead of having 8 leaf nodes representing 0 and 1, we could do the job (of representing the function) with two, one for each value. All the edges pointing to the leafs can be redirected to the corresponding elected one. Moreover the first and third vertices, starting from the left, labeled with variable y\_2 are identical, i.e. their low-branches and high-branches, after the first simplification, point respectively to the same nodes.
The process shown in figure 3.2 is called reduction and leads to a directed acyclic graph that we define now.

**BDD.** A binary decision diagram, is a rooted acyclic graph with one or two terminal nodes of out-degree zero labeled 0 or 1 and a set of variable nodes of out-degree two.

**OBDD.** A BDD is ordered if for any path the sequence of variables associated with the nodes of this path follows a given global order.

**Uniqueness.** A BDD respect the uniqueness property if no two distinct nodes have the same variable name and low and high successor.

**Redundant test.** A BDD node is a redundant test if it has identical low and high successor.

**ROBDD.** An OBDD is called reduced, if it respects the uniqueness property and has no redundant test.

The reduction of a binary decision tree or diagram is the process of applying the following reduction rules iteratively:

- remove non-unique nodes
- remove redundant tests

The reduction terminates when none of these rules can be applied, as the figure 3.3 suggests.
Applying the reduction to the 19 nodes binary tree from figure 3.1, leads to the 8 nodes ROBDD as we saw in figure 3.2.

**Definitions and conventions.** Throughout this chapter we will use the conventions that we define now. \( \mathcal{B} \) is the set of boolean \{0, 1\}. ROBDD will be designated by their root node \( u, v, w \) and \( f_u, f_v, f_w \) are their associated functions. If \( f : \mathcal{B}^n \rightarrow \mathcal{B} : (x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n) \) \( \text{def} \ f_{[x_i=k]} = f(x_1, \ldots, x_{i-1}, k, x_{i+1}, \ldots, x_n) \). An assignment over the set of variables \( X = \{x_1, \ldots, x_n\} \) is a function \( a : X \rightarrow \mathcal{B} \). Assignments are represented with \( a, b \) or \( c \).

### 3.2 Canonicity Lemma

ROBDD have interesting properties, essentially they often provide compact representations of boolean expressions and have efficient algorithms for common logical operations. The central property being that, given a variable ordering, they represent canonically boolean functions.

**Canonicity Lemma.** For any boolean function \( f : \mathcal{B}^n \rightarrow \mathcal{B} : (x_1, \ldots, x_n) \rightarrow f(x_1, \ldots, x_n) \) there is exactly one ROBDD, \( u \), with variable ordering \( x_1 < \ldots < x_n \) such that \( f_u = f \).

**Proof.** The proof is by induction on the number of arguments of \( f \). if \( n = 0 \), the only two boolean functions are constant 0 and constant 1 which are represented by the corresponding leaf. Let’s suppose the lemma holds for \( n \), we must show it is true for \( n + 1 \). Consider the functions \( f_{[x_{n+1}=0]} : \mathcal{B}^n \rightarrow \mathcal{B} = f(x_1, x_2, \ldots, x_n, 0) \) and \( f_{[x_{n+1}=1]} : \mathcal{B}^n \rightarrow \mathcal{B} = f(x_1, x_2, \ldots, x_n, 1) \) they satisfy the following equation (Shannon expansion):

\[
f(x_1, \ldots, x_{n+1}) = (\neg x_{n+1} \rightarrow f_{[x_{n+1}=0]}(x_1, \ldots, x_n)) \land (x_{n+1} \rightarrow f_{[x_{n+1}=1]}(x_1, \ldots, x_n)) \tag{3.1}
\]

Since these two functions have \( n \) variables, by induction hypothesis, there are unique ROBDD node \( u_0 \) and \( u_1 \) such that \( f_{u_0} = f_{[x_{n+1}=0]} \) and \( f_{u_1} = f_{[x_{n+1}=1]} \).
We must consider 2 cases:

- If $u_0 = u_1$ then $f_{[x_{n+1}=0]} = f_{[x_{n+1}=1]} = f$ and $u_0$ is a ROBDD for $f$. If $x_1$ appears in the BDD it must be root, because of the ordering, but we just saw that then its low and high branches would point to the same ROBDD node $u_0$, it is therefore a redundant test and must be removed. $u_0$ is thus the unique ROBDD representing $f$.

- If $u_0 \neq u_1$, let $u$ be a node with label $x_1$, low branch $u_0$ and high branch $u_1$. Per of 3.1 we have $f_u = f$. It is reduced since $u_0 \neq u_1$ and both are reduced. It is also the only ROBDD for this ordering that represents $f$ since such a ROBDD must have $x_1$ as root and must have precisely those low and high branches. ■

3.3 Examples and Maximum size ROBDDs

This section presents some samples of ROBDDs classified by number of variables, with an emphasis on the biggest ROBDDs we can get for each number of variable.

3.3.1 Examples

No variable

There are 2 functions and 2 1-node BDDs: 0 and 1. This is shown in figure 3.4.
1 variable

There are 4 functions, 2 1-node BDDs (see above) and 2 3-nodes. This is shown in figure 3.5.

2 variables

There are 16 functions. 2 1-node, 4 3-nodes, 8 4-nodes, 2 5-nodes.

Maximum size BDDs have 5 nodes. This is shown in figure 3.6.

3 variables

256 functions and maximum size BDDs have 7 nodes. This is shown in figure 3.7.

The 6-nodes ROBDDs are obtained by bypassing a $x_1$ or a $x_2$ node. This is shown in figure 3.8.
Figure 3.7: 3 variables

Figure 3.8: 3 variables, 6-nodes
4 variables

There are 65,536 functions.

Maximum size BDDs have 11 nodes. This is show in figure 3.9

5 variables

There are 4,294,967,296 functions.

Maximum size ROBDDs have 19 nodes. This is shown in figure 3.10

3.3.2 Maximum size ROBDDs

While one, maybe you, can have a lot of fun playing around with ROBDDs, we want to focus on one observation:

Any ROBDDs will have at most 2 terminal nodes 0 and 1, 2 nodes at the $x_1$ level, 12 nodes at the $x_2$ level,... more generally we can show that nodes of level $i$ can be
at most \( l_i = \binom{2}{s_{i-1}} \) where \( s_{i-1} = \sum_{j \leq i-1} l_j \) or \( l_i = s_i \times s_{i-1} \). This comes from the fact that each branch can point to any of the nodes below including the terminal nodes, but all nodes must have different low/high pairs. The sequence \( l_i \) is seriously exponential: 2, 2, 12, 240, 65280, and so on.

However on the other side of the tree, starting from the root we can have at most \( 2^n \) nodes after \( n \) levels, i.e. a complete binary tree. This is illustrated in the maximum size BDD with 5 variables in figure 3.10.

Therefore, maximum BDDs will be formed of a complete binary tree for the \( n - k \) first levels, starting from the root, and a forest of \( 2^{n-k} \) different maximum BDDs of level \( k \). \( k \) must be chosen in such a way that: \( 2^k \leq l_{n-k} \). Since \( l_i \) growth a lot faster than \( 2^n \), maximum size BDDs should grow just like \( 2^n \).

This gives us a better view of BDDs and how it behaves with a growing number of variables: exponentially in the worst cases.
3.4 Variables ordering

We just saw that BDDs size grows exponentially with the number of variables in the worst cases. However the size of a BDD for a given function depends on the chosen variable ordering, and in some cases dramatically. This is illustrated in the following example.

The function \( f_n = (x_1 \land y_1) \lor (x_2 \land y_2) \lor \ldots \lor (x_n \land y_n) \) leads to the following BDDs with respective orderings \( x_1 < x_2 < \ldots < x_n < y_1 < \ldots < y_n \) and \( x_1 < y_1 < x_2 < y_2 < \ldots < x_n < y_n \). This is illustrated in figure 3.11.

![Diagram of BDDs](image)

Figure 3.11: \( f_n = (x_1 \land y_1) \lor (x_2 \land y_2) \lor \ldots \lor (x_n \land y_n) \)

In the first case we get a BDD with \( n + 2 \) nodes. In the second case the BDD has \( 2(2^n - 1) \) nodes. For large value of \( n \), i.e. a large number of variables, this lead to unusable representation of the function.

In the second graph we can see that the structure needs to remember all the \( x_i \) variables before making any decision (pointing to a terminal node). This translates into a complete binary tree of \( 2^n \) nodes for the \( x_i \) variables.

Wegener [21] showed an even stronger result, namely that the fraction of variable orderings \( \pi \) that lead to a polynomial number of nodes in an OBDD for \( f_n \) with respect to all variable orderings is exponentially small. This and other results indicate
that the variable ordering must be chosen carefully.

**Best, better variable ordering.** Friedman and Supowit [19] published an algorithm to compute the best variable ordering for a BDD. But the running time of this algorithm, and of all its derivatives, is exponential in the number of variables.

In fact Bollig, B., Wegener [17] proved, what was before a conjecture, that improving the variable ordering of OBDDs is NP-complete. More precisely they proved that the problem ”Given an ROBDD $G$ representing $f$ and a size bound $s$, does there exist a ROBDD $G'$ with at most $s$ nodes that represents $f$” is NP-complete.

Some NP-complete problems however have efficient algorithms that compute reasonable, and even as good as one want, approximations of the best solution. This is not the case here, Sieling proved that it is also NP-hard to get any approximation [20]: for any $\alpha > 1$, for any function $f$, finding an ordering $\pi$ such that the BDD for $f$ with ordering $\pi$ is greater than the smallest BDD for $f$ by less than a factor $\alpha$ is NP-Hard.

**Heuristics.** Special heuristics were therefore developed, which we can classify in two categories. The first are the one that take advantage of the input structure. Indeed the function represented by the BDD comes from some domain like circuit definition, modeling language, boolean formula, or other. Therefore exploiting the input properties can help find a good ordering.

The second case occurs when we already have a BDD and want to minimize the number of its nodes by improving the variable ordering. We call this process dynamic variable ordering because it is often used while performing some sequence of transformations on BDDs. During these transformations, when some trigger occurs, say a given size has been reached, a variable re-ordering is performed in order to try to get a smaller BDD. The transformations resume when the re-ordering is done.

Dynamic reordering methods include:

- **sift.** Moves each variable, one by one, throughout the order to find an optimal position for that variable, all other being fixed.

- **symmetry sift.** Like sift, but checks if variables that become adjacent during sifting are symmetric. In that case, they are joined in a group.
• **random.** swap two randomly chosen variables by swapping adjacent variables in between, and keeping the best ordering of those involved.

• **annealing.** Based on the simulated annealing.

### 3.5 Lower bounds

We saw that some functions are compactly represented by BDDs while others lead, for some ordering, to an exponentially big BDD. Bryant exhibited in 1992 [13] a family of functions leading to an exponential BDD whatever the ordering is. Moreover he exposed a technique for proving such exponential lower bounds. This is a serious weakness of BDDs.

Essentially, given a variable ordering, we want to construct a set of paths that lead to different nodes. If we have an exponential number of such paths in relation to the size of the input, we will have proven that we have an exponential number of nodes in the BDD.

In the following proof we restrict our set to paths containing the first k variables. We show that any two paths of this set lead to different nodes by exhibiting an assignment for the other n-k variables that complete each path, making the same decisions, in such a way that one leads to 0 and the other to 1. This is only possible if the completions of the paths start at different nodes. We prove this more formally now.

Let X be the set of all variables.

**Balanced partition.** Given a real-valued balance parameter $\omega$ between 0 and 1 we define a balanced partitioning of X as any partitioning of X in two subsets L and R such that $|\omega|X| \leq |X \cap L| \leq [\omega|X|]$.

**L/R-assignements.** we call a left, right, input assignment any function $l : L \to \{0, 1\}$, $r : R \to \{0, 1\}$ respectively. We denotes these assignments L-assignments and R-assignments, respectively.

**Fooling set.** Given a partition (L, R) of X and a function $f$, a fooling set F is a set
of L-assignments such that:

For any \( l, l' \in F, l \neq l' \), there exists a R-assignment with \( f(l, r) \neq f(l', r) \).

**BDD Lower Bound Lemma.** If for every balanced partition (L, R), function \( f \) has a fooling set containing at least \( c^n \) elements, for some \( c > 1 \), then any OBDD for \( f \) must have at least \( \Omega(c^n) \) vertices.

**Proof.** For variable ordering \( \pi : X \rightarrow \{0, ..., |X|\} \), let L be the set of the first elements in X such that L is the left part of a partition. Consider the fooling set for this balanced partition, by hypothesis it has at least \( c^n \) elements. A left input assignment (i.e. an element of the fooling set) defines a path from the root to either a terminal node or to a node labeled with a variable from R. By definition of a fooling set for any \( l, l', l \neq l' \) there is a right input assignment \( r \) such that \( f(l, r) \neq f(l', r) \) but that means that the paths defined by \( l, r \) and \( l', r \) lead to different terminal nodes, so \( l \) and \( l' \) lead to different nodes. Hence there is at least \( c^n \) nodes in the BDD. ■

We can see a fooling set as a set of paths each leading to a different node labeled by a variable in R (or a terminal node), hence the OBDD must at least have a number of nodes equal to the number of paths in the fooling set (the terminating nodes from R).

We note that the balance parameter \( \omega \) may depend on \( n \), the proof stays valid. And if we use a subset of X in the lemma it is still valid.

Returning to the example, shown in figure 3.11, with order \( x_1 < x_2 < ... < x_n < y_1 < ... < y_n \) and partition \( L = \{x_1, ..., x_n\} \) then all 8 possible assignments are forming a fooling set since all branches from these nodes leads to different node labeled with \( y_i \). Of course this is true only for this partition, not all.

**Hidden Weighted Bit function.** Using this techniques Bryant proved that HWB is an example of a function that requires an OBDD of exponential size [13]. He chose this function because it has a representation in VLSI in quadratic space against the input size.
3.6 Operations complexity

We describe the operations available on BDDs, their implementation issues and their complexity.

3.6.1 Reduction

Reduction was described above; it is the application of both reduction rules until none is applicable. These rules are:

- **Non-unique.** Removes nodes that are labeled with the same variable and have the same low branch, and the same high branch.
- **Redundant test.** Removes nodes that have same low and high successor.

To be efficient a reduction algorithm must go bottom-up and level by level. Indeed the application of a reduction rule may create the condition for another reduction to be applicable on an upper node but never on a node of the same level or below. These algorithms runs in \(O(|R|)\) where \(R\) is the involved BDD and \(|R|\) its number of nodes.

3.6.2 Equivalence

In BDD packages, different BDDs will share same nodes, equivalence test is then straightforward if the variable ordering is the same, it then suffices to check that both root pointers lead to the same node, which is done in constant time.

However if BDDs do not share nodes, but variable ordering is still the same, then, thanks to the Canonicity Lemma, we just have to check for isomorphism between both BDDs. This can be done by a parallel depth-first traversal through both BDDs, checking each time we reach a terminal node whether it is indeed the same on both side. This algorithm has a running time of \(O(\min\{|A|, |B|\})\), where \(A\) and \(B\) are the involved BDDs and \(|A|\) its total number of nodes.
If variable ordering is not the same, then things get uglier. A polynomial time algorithm for this problem was published by Fortune, Hopcroft, and Schmidt [16].

### 3.6.3 Tautology, Satisfiability, SAT-count.

**Tautology.** Checking that a ROBDD is always true or always false is a constant time operation. We just have to check it is the terminal node 1 and 0 respectively.

**Satisfiability.** Checking whether a ROBDD is satisfiable is equivalent to checking whether it is not always false. In consequence it can be done in constant time.

Exhibiting a value, assignment, that satisfies the function is also straightforward. Starting from the BDD root we follow a path without ever choosing a branch that directly leads to the terminal 0. If such a path does not exist then the BDD is the terminal node 0. The running time is $O(n)$.

**SAT-count.** The SAT-count algorithm must output the number of assignments that satisfy the function. It is based on the following property. If $N$ paths run through the BDD node $\omega$, then these paths distribute equally among the low and high branches of $\omega$. Let $r$ be the BDD root node, $N(r) = 2^n$, since all paths corresponding to all possible assignments of $n$ binary variables run through the root, and for any node $\omega$ we define $N(\omega) = 0$. Starting from the root and going all downhill from here, level by level, for each node $\omega$ with successors $\omega_0$ and $\omega_1$ we perform: $N(\omega_0) := N(\omega_0) + N(\omega)/2$ and $N(\omega_1) := N(\omega_1) + N(\omega)/2$. If 1 stands for the terminal node 1, then $N(1)$ is the number we are looking for. This algorithm has a running time of $O(|A|)$, where $A$ is the involved BDD.

### 3.6.4 Synthesis

Synthesis permits to apply any binary operation on BDDs. It enables all boolean operations: and, or, xor, implication, and so on. We present the algorithm for a generic binary operation $\otimes$.

We suppose here that both BDDs follow the same variable ordering and are repre-
sented in the same ROBDD, i.e. they share nodes, all 'shareable' nodes. Moreover the result is represented also in the same ROBDD.

The idea is to construct a BDD that 'simulates' a simultaneous topdown traversal of both BDDs (evaluation). The nodes of this new BDD can be thought as corresponding to a pair of nodes \((u_i, v_j)\) one from each input BDD.

The computation starts at the root of both BDDs \(u_0, v_0\). We construct the root node, \(w_0\) for the resulting BDD corresponding to \((u_0, v_0)\). Then for, let’s say, the low-successors \(u_1, v_1\) we construct a low-successor to \(w_0, w_1\). And similarly we construct \(w_2\) as high-successor. We then iterate on all successors. This of course works if both BDDs have there nodes simultaneously with successor with same label. This is rarely the case.

So if on a computation path a variable is bypassed in one BDD and not in the other, then the simulation must 'wait' for the latter BDDs. This nevertheless translates into the creation of a node in the result BDD, just as if a redundant test had been added to the former BDD. This can of course also occur at the root node. An example of synthesis is shown in figure 3.12.

\[
\begin{array}{c}
\text{Figure 3.12: synthesis}(v_1, w_1, or) \\
\end{array}
\]

In order to be efficient we use a uniqueness-table for the resulting BDD which is a hash-table mapping a label and its successor to a node, it is called the unique_table. This means the reduction is applied along the construction. Finally a table of computed nodes is used in order not to compute a node corresponding to a pair \((u, v)\) if already done, it is called the computed_table.

The operation is applied only at terminal nodes. The following terminal nodes \(0 \otimes 0, 0 \otimes 1, 1 \otimes 0\) and \(1 \otimes 1\) corresponding to the pair \((0,0), (0,1), (1,0)\) and \((1,1)\) are
computed when, and if, first encountered in the construction.

```plaintext
Synthesis(u, w, op, result) {
    if ((u op w) is terminal case) then return terminal.
    if (computed_table.contains(u, w)) then return entry.

    Let xk be the top variable of u and w
    if u is labeled by xk then
        let u'0 be the 0-successor of u
    else
        let u'0 := u

    let u'1, w'0, w'1 as u'0
    R0 := synthesis(u'0, w'0)
    R1 := synthesis(u'1, w'1)

    if(R0=R1) the return R0

    if(not unique_table.contains(xk, R0, R1)) then {
        R:= new Node(xk, R0, R1)
        unique_table.insert((xk, R0, R1), R)
    }
    computed_table.insert((u,w), R)
    return R.
}
```

This algorithm has a running time of $O(|A| \times |B|)$, where $A$ and $B$ are the involved BDD.

### 3.6.5 Complement

It is simply done by redirecting all edges pointing to a terminal node to the other terminal node. Implemented with a tree traversal it has a running time of $O(|A|)$. 
3.6.6 Substitution

Constructing a BDD by replacing a variable, $x_i$, with a value, $c$, in another BDD is easy. We redirect all edges leading to $x_i$-nodes to the corresponding $c$-successor. However the resulting BDD is not always reduced, we must then therefore apply the reduction algorithm.

3.6.7 Quantification

Universal and existential quantifications can be obtained by a substitution and a synthesis since we have the following equivalences:

\[
(\forall x_i) \equiv f(x_i=0) \land f(x_i=1) \tag{3.2}
\]
\[
(\exists x_i) \equiv f(x_i=0) \lor f(x_i=1) \tag{3.3}
\]

3.7 Conclusion

In this chapter, we presented reduced ordered binary decision diagrams. We showed how they behave with the number of variables increasing and saw their size is increasing exponentially in the worst case.

We then showed that, while for a fixed ordering and for some functions it is indeed getting bigger quickly (along the number of variables), changing the ordering can help reduce dramatically its size. For some functions, an ordering yields to an exponentially big ROBDD whereas for another ordering the ROBDD’s size is linear in the number of variables. We gave some results suggesting that choosing a good ordering is very important, but it is also a difficult problem (NP-Hard).

However not all functions have this property. We exhibited a result showing that some functions have exponentially big ROBDD representation whatever the ordering is. Doing this we introduced the all important concept of fooling set, which we will use later on.
Finally we described the operations available on these data structures and gave their respective running time. We stress that the running time of the all important synthesis algorithm, which is the generic algorithm for all binary boolean operations, is $O(|A| \times |B|)$. Hence it is crucial to deal with relatively small ROBDDs.

There are of course plenty of BDD-related subjects we didn’t even mention. But we covered all the BDD fields we think are necessary to understand the rest of this work. And, as best as we could, we highlighted the critical points: BDD can get very big but we can manage only relatively small one.
Chapter 4

Modeling

A finite state machine can be described as a 3-uple

\[ M = \langle S, I, R \rangle \]

where \( S \) is the set of states, \( I \in S \) the set of initial states and \( R \subseteq S \times S \) the transition relation.

We suppose states are formed of state variables, in the sense that any state is defined by an assignment of these variables. But on the other side, an assignment of the state variables may or not yield one state of the system, indeed some assignment may represent an invalid state. If \( s_i \in S_i, 1 \leq i \leq n \) are the states variables then we have:

\[ I \subseteq S \subseteq S_1 \times \cdots \times S_n \]

Now, a set of states can be represented by a boolean formula such that a state belongs to this set if and only if the assignment of state variables corresponding to this state makes the formula true. This is called symbolic representations. Moreover, the boolean formula can be encoded in a BDD.
So we can represent $S$, the set of all possible system states, with a boolean formula or a BDD. We can also represent $I$, the set of initial states with a boolean formula or a BDD. Finally the transition relation $R$ can also be expressed as a boolean formula, $B_R$, over two sets of $n$ boolean variables, one for the current states, the other for the next states: $s' = (s'_1, \ldots, s'_n)$ is a possible successor of $s = (s_1, \ldots, s_n)$ if and only if $R(s, s')$ holds if and only if $B_R(s_1, \ldots, s_n, s'_1, \ldots, s'_n)$ is true.

We call **forward image** of a state the set of all its successors. The **backward image** of a state is the set of states of which it is successor. The forward, backward, image of a set of states is the union of the forward, respectively backward, image of all its states:

$$\text{forward_image}(T) = \{ s' | \exists s \in T, R(s, s') \}$$

$$\text{backward_image}(T) = \{ s' | \exists s \in T, R(s', s) \}$$

Since the system is finite, the forward image operator can be applied a finite number of times before reaching a fixed point, this is the set of reachable states (of a state or a set of states):

$$\text{Reachable}(T) = \{ s' | \exists k \in \mathbb{N}, s' \in \text{forward_image}^k(T) \}$$

The states that are reachable in exactly $k$ steps belongs to the $k$-th frontier:

$$\text{frontier}_k(T) = \{ s' | \forall j < k, s' \notin \text{forward_image}^j(T) \land s' \in \text{forward_image}^k(T) \}$$

Finally we define the system diameter as the distance from the set of initial states of the furthest reachable state:

$$\text{diameter}(M) = k \iff (\text{frontier}_{k+1}(I) = \emptyset \land \text{frontier}_k(I) \neq \emptyset)$$
We need to implement four steps:

1. Modeling our problem
2. Transforming our model into BDDs
3. Applying the iteration in order to find the furthest frontier, using the backward or forward image
4. Translating back into the original human understandable form, the corresponding instance

Modeling directly in BDDs form is rather unappealing and certainly inefficient. So we used NuSMV, a verification tool, to help us modeling our problem. We present this tool now.

4.1 NuSMV

NuSMV is an open source re-implementation and extension of the original BDD-based symbolic model checker, CMU SMV, developed at Carnegie Mellon University. It consists in an input modeling language, assertion languages (CTL and LTL) and verification algorithms [23][24][22].

The input language defines the finite state machine, while CTL and LTL enable to describe properties of the system. The verification algorithms check if the defined properties holds for that machine. They are driven by the syntactic structure of the assertions and based on the forward and backward images.

We will use the modeling language and some basic verification algorithms that we both describe quickly in the next sections. We will not use any assertions, nor any algorithm, specifically build for these.
4.2 Modeling language

The modeling input language allows to define synchronous and asynchronous finite state machines (FSM). It is formed of basic expressions and, on a higher level, of several declarations: **VAR, ASSIGN, INIT, MODULE, TRANS, INVAR** and the special purpose **main** module.

4.2.1 Expressions

Expressions are the basic blocks for building specification. They are formed of variables, constants and operators.

Here are some simple expressions.

Boolean and arithmetic operations:

\[
\begin{align*}
    a & \mod 6 \\
    a + b \\
    a + b < c \\
    (a \& b) \rightarrow c \\
    \text{TRUE}
\end{align*}
\]

Assignation expressions:

\[
\begin{align*}
    a & := b + c; \\
    d & := \{b, b+1, b+2\};
\end{align*}
\]

The last assignation is not deterministic. \(d\) can be assigned any of \(b\), \(b+1\) or \(b+2\).

Case expressions:

```
b = 1 : 2;
1 : a;
esac;

The case expression semantic is as follow. For each subexpression, e.g. "b = 1
: 2;", the left-hand part is the condition, the right-hand part is the value of the
subexpression. The case expression returns the right-hand part’s value of the first
subexpression (in top-down order) whose left-hand part evaluates to true.

4.2.2 VAR and ASSIGN declarations

The VAR declaration defines the state variables and their respective types. A state
of the model is an assignment of values to this set of variables compatible with the
types declared.

The ASSIGN declaration defines the initial state and the transition relation. They are
both defined by specifying the initial, respectively next, values of the state variables
via an assignation and the respective keyword init and next.

Example (modulo 6 counter)

MODULE main
  VAR
    value : 0..5
  ASSIGN
    init(value) := 0;
    next(value) := value mod 6;

The FSM defined by this specification has one variable value. The initial state is the
one in which value is 0. The transition relation allows increment modulo 6 of the
value.
4.2.3 MODULE declaration

A module encapsulates some state variables which can be part of a single process jointly, thus evolving in parallel together. A module is instantiated in a VAR declaration of another module. The main module is the root module, all other modules are instantiated as a variable of the main module or of any descendant of the main module.

Example (modulo 6 counter defined via a module)

MODULE main
VAR
  counter : new counter();
MODULE counter
VAR
  value : 0..5;
ASSIGN
  init(value) := 0;
  next(value) := value mod 6;

4.2.4 Processes

A process is a module which is instantiated using the process keyword. All processes evolves by definition asynchronously. It is created by adding the process keyword when instantiating a module in the VAR declaration. The machine executes a step by non deterministically choosing a process and executing its transition.

Example (two asynchronous modulo 6 counters)

MODULE main
VAR
  counter1 : process counter();
  counter2 : process counter();
MODULE counter
VAR
value : 0..5;
ASSIGN
  init(value) := 0;
  next(value) := value mod 6;

The process keyword means the following are valid transitions of the system: (0, 0) → (0, 1) → (0, 2) → (1, 2) → (1, 3)... If the process keyword is removed the only possible transitions are: (0, 0) → (1, 1) → (2, 2)...

4.2.5 INIT declaration

The initial states can be defined via a relation, instead of an assignation.

Example (modulo 6 counter with init relation):

MODULE main
  VAR
    counter : new counter();
  INIT
    counter.value > 3;
MODULE counter
  VAR
    value : 0..5;
  ASSIGN
    next(value) := value mod 6;

The initial states are (4), (5).

4.2.6 TRANS declaration

Like the INIT declaration the TRANS declaration enables one to define the transition relation with a plain relation. Depending on the system, it is sometimes more convenient to use one or the other definition.
Example (modulo 6 counter with TRANS relation):

MODULE main
VAR
    counter : new counter();
MODULE counter
VAR
    value : 0..5;
ASSIGN
    init(value) := 0;
TRANS
    next(value) = (value +1) mod 6;

4.2.7 INVAR declaration

The INVAR declaration enables the definition of the set of normal states or acceptable states, i.e. any states that infringe the relation defined in the INVAR declaration will be discarded.

Example (modulo 6 counter with INVAR relation):

MODULE main
VAR
    counter1 : process counter();
    counter2 : process counter();
INVAR
    counter1.value != counter2.value;
MODULE counter
VAR
    value : 0..5;
ASSIGN
    init(value) := 0;
    next(value) := value mod 6;

Here the INVAR declaration states that any valid transition cannot lead to a states of
the form $(i, i)$ where $0 \leq i \leq 5$. But here the lone initial state was defined as $(0, 0)$ which is not \textsc{Invar} compliant, therefore the set of initial valid states is empty as is the FSM.

### 4.3 Model building

The main task of NuSMV is to build the model as BDDs from the specification. NuSMV constructs four BDDs from a specification:

- Initial states BDD
- Mask BDD
- Transition relation BDD
- Invariant relation BDD

The mask is linked to the encoding. This BDD ensures the encoding does not lead to states that are out of scope. For example if a variable can take value from 1 to 5, it will be encoded on 3 bits. The mask BDD ensures $111_b, 110_b, 101_b$ are discarded as values for this variable. The others BDDs were explained above.

#### 4.3.1 Transformation steps

Since BDDs are build upon boolean relation, the SMV specification has to be transformed into boolean relations. NuSMV does that in several steps.

**Read model.** Parses and checks specification grammar.

**Flatten hierarchy.** Remove all module definitions by replacing all its instantiations of this module by its actual definition prefixing implicated variables by the variable instance name.

**Encode variables.** Substitutes all variables with boolean ones. Booleans stay booleans, integer are encoded in binary form, and so on.
**Build model.** Transforms this binary encoded specification into BDDs.

### 4.3.2 Variables ordering, reordering

There is various heuristics implemented. One can customize the strategies: trigger threshold, reordering heuristic,...

Variables which are grouped together will always remain next to each other in the BDD and in the same order. When dynamic variable re-ordering is carried out, the group of variables are treated as one entity and moved as such.

### 4.4 Algorithms

We present now the algorithms for computing the forward and backward image, the set of reachable states and the system diameter. These are the one we will use to solve our problem. We should note that the main specificities are that they involve the invariant property of the system and are based on the BDD relational product.

#### 4.4.1 Forward image

This method computes the forward image of a set of states $A$ represented by a BDD, i.e. the set of INVAR states which are reachable from one of the INVAR states in $A$ by means of one single machine transition.

The forward image of $A$ is computed as follows:

\[
\begin{align*}
A_1 &:= A \land \text{Invar} \\
A_2 &:= \{ s' | < s, s' >, \ Tr(s, s'), A_1(s) \} \\
&:= \exists s ( Tr(s, s') \land A_1(s) ) \\
A_3 &:= A_2[s/s'] \\
\text{forward image} &:= A_3 \land \text{Invar}
\end{align*}
\]

(*remove invalid states*)

(*relational product*)

(*replace all next variables with current ones*)

(*remove invalid states*)
4.4.2 Backward image

Similarly, this method computes the backward image of a set $A$ of states, i.e. the set of INVAR states from which some of the INVAR states in $S$ is reachable by means of one single machine transition.

The backward image of $A$ is computed as follows:

$$A_1 := A \land \text{Invar} \quad (\text{remove invalid states})$$
$$A_2 := A_2[s'/s] \quad (\text{replace all current variables with next ones})$$
$$A_3 := \{s | s < s', Tr(s, s') \land A_2(s')\} \quad (\text{relational product})$$

$\text{backward\_image} := A_3 \land \text{Invar} \quad (\text{remove invalid states})$

4.4.3 Reachable states

Computes the set of reachable states, i.e. the ones that can be actually reached starting from one of the initial state.

The classical method for computing this set is to successively apply the forward image algorithm, starting from the set of initial states, on the resulting set, and this until a fixed point is reached:

$$R_0 = I$$
$$R_{k+1} = (\exists s(R_k(s) \land T(s, s')))[s/s'] \cup R_k(s)$$

where $R_k$ is the set of states that are reachable in $k$ or fewer step from an initial step.

NuSMV instead makes use of frontiers:

$$F_k(s) = R_k(s) \land \overline{R_{k-1}(s)}$$
$$R_{k+1}(s) = (\exists s(F_k(s) \land T(s, s')))[s/s'] \cup R_k(s)$$
4.4.4 System diameter

The system diameter is computed alongside the reachable states. A counter record
the number of step applied until the computation reaches the fixed point.

4.4.5 Disjoint transition relation, monolithic

As we saw one of the central operation in model checking is the relational product:
$\exists s (T(s, s') \land A(s))$ which computes the set of successors of A. This operation involves
two BDDs, A and T, and T has twice as many variables as A has. Moreover the
relational product has been shown to have exponential worst case complexity [25].
Some tuning have been developed. One is to partitioned the transition relation BDD
into smaller BDD, in such a way that it is possible to used them separately [26].
Chapter 5

First Rush Hour SMV Specification

The first specification we implemented is quite natural and very simple. Because of this simplicity we thought this would lead to small BDDs, thanks to their compactness property. But as we will see this was not the case. The insight of BDDs structure that we presented in the third chapter and especially the fooling set concept enables us to understand better why this model is not suitable here.

This led us to the second specification that yield better results and enable us to gather some interesting facts about Rush Hour that would be rather tedious to obtain with classical methods.

5.1 Specification description

This first specification models cars as independent modules, positioned on a board. Just as in the real game each car can be positioned horizontally or vertically anywhere on the board. We model cars with unique size, 2. Indeed we will see that even this simplification isn’t enough to keep the problem manageable.

Each car is defined by a line and column position, posi and posj, pointing to its leftmost, upper cell and a boolean variable, horizontal, defining its direction, horizontal...
or vertical. The positions (posi, posj) follow the convention of matrix, starting on the upper-left corner (1,1). The figure 5.1 exemplifies the encoding of a possible winning configuration with each car showing its number (id), line and column position and its direction.

![Figure 5.1: configuration encoding following specification 1](image)

Finally, a specification contains a fixed number of cars on a board of fixed size. Hence, in order to search all possibilities, we will have to generate a family of specifications for board with size, $s$, ranging from 4 to 6 and with number of cars, $n$, ranging from 2 to $\lceil s^2/2 \rceil$.

We describe now this specification declaration by declaration. Throughout this chapter we assume the following conventions: $s$ is the board size, $n$ the number of cars, $c_i$ the i-th car and $c_1$ the target car. (It can be found in annex C)
5.1.1 Car module

The car module encapsulates car related variables: $\textit{posi}$, $\textit{posj}$ and $\textit{horizontal}$. $\textit{posi}$ and $\textit{posj}$ are integer between 1 to 6, however if the car is vertical we must have $\textit{posi} \leq 5$, and if the car is horizontal we must have $\textit{posj} \leq 5$, these constraints are enforced by the module’s INVAR declaration.

Its transition relation, declared in ASSIGN, constraints its direction as well as its line or column if it is respectively horizontal or vertical to be constant. Finally it can move one cell left or right, up or down, for horizontal, respectively vertical cars, if the board allows it (enforced by the INVAR declaration). The collision between cars are handled by the main module’s INVAR declaration.

MODULE CAR()

VAR
   posi : 1..6;
   posj : 1..6;
   horizontal : boolean;

ASSIGN
   next(horizontal) := horizontal;
   next(posi) :=
      case
         horizontal : posi;
         posi = 1 : 2;
         posi = 6 : 5;
         1: \{posi - 1, posi + 1\};
      esac;
   next(posj) :=
      case
         !horizontal : posj;
         posj = 1 : 2;
         posj = 6 : 5;
         1: \{posj - 1, posj + 1\};
      esac;
INVAR

\(((\text{posi} + \neg \text{horizontal}) \leq 6) \& ((\text{posj} + \text{horizontal}) \leq 6)\);

5.1.2 VAR declaration

We declare a set of cars that will be placed on the board. They are declared as \textit{process}, since their state variables evolve asynchronously. In the following example we have 3 cars. The target car will be defined in the \texttt{INIT} declaration.

\begin{verbatim}
VAR
c[1]: process CAR();
c[2]: process CAR();
c[3]: process CAR();
\end{verbatim}

5.1.3 Main’s INVAR declaration

While the car’s module \texttt{INVAR} declaration puts constraints on internal variables, the main \texttt{INVAR} declaration handles the restriction that involves several cars. Cars cannot collide, i.e. overlap, at any time. This no-collision constraint is handled by the following \texttt{INVAR} declaration and will be enforced at initial time and after each system transition.

Car $i$ and car $j$ do not overlap if we have the following condition:

\[
\begin{align*}
&(c_i.\text{posi} > (c_j.\text{posi} + \neg c_j.\text{horizontal})) \lor \\
&(c_j.\text{posi} > (c_i.\text{posi} + \neg c_i.\text{horizontal})) \lor \\
&(c_i.\text{posj} > (c_j.\text{posi} + c_j.\text{horizontal})) \lor \\
&(c_j.\text{posj} > (c_i.\text{posj} + c_i.\text{horizontal}))
\end{align*}
\]

Hence we have $n(n - 1)$ no-collision constraints to enforce, one for each pair of cars. Here is an example with 3 cars:
5.1.4 INIT declaration

We want the set of initial states to contain all winning configurations and only these. So we place the target car, $c_1$, at the exit point and let all other cars to be anywhere on the board. Again the collisions between cars are handled through the main module’s INVAR declaration.

INIT

\[
c[1].\text{posi} = 3 \land c[1].\text{posj} = 5 \land c[1].\text{horizontal};
\]

5.2 Specification results and limits

The main issue is that a car can collide with any other, in other words, each car’s position much be checked against all other cars’ position. This interdependency between all variables is what lead to exponentially big BDDs and there is here no exception.
In fact we can get a rough idea on what BDDs’ size is to expect as the number of cars increase, or as the board size increase. And we see that it quickly becomes unmanageable. We present some theoretical views as well as empirical results.

### 5.2.1 Empirical results

We ran our specification for board of size 4, 5 and 6. And for each we started with 2 cars up to the maximum we could. We report the results in the tables below. For each run (board size/number of cars) we have the number of nodes in the INVAR BDD, the number of reachable states and the diameter of the system. These are the relevant information about the size of the system. The input and transition BDDs are, in comparison, small and do not have any explosion property.

For system with board of size 5 and 6, we could only compute the data for up to 5 to 6 cars. For more cars we ran out of memory or out of patience, these systems are out of reach at least using these specific technique and specification.

**board 4x4**

<table>
<thead>
<tr>
<th># cars</th>
<th># nodes in INVAR</th>
<th># reachable states</th>
<th>diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>123</td>
<td>52</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1237</td>
<td>672</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7334</td>
<td>5994</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>24227</td>
<td>33624</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>44209</td>
<td>103440</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>50081</td>
<td>143280</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>30762</td>
<td>60480</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
board 5x5

<table>
<thead>
<tr>
<th># cars</th>
<th># nodes in INVAR</th>
<th># reachable states</th>
<th>diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>233</td>
<td>131</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3918</td>
<td>3640</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>44209</td>
<td>84318</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>321114</td>
<td>1.5935e+06</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1520760</td>
<td>2.3891e+07</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

board 6x6

<table>
<thead>
<tr>
<th># cars</th>
<th># nodes in INVAR</th>
<th># reachable states</th>
<th>diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>368</td>
<td>261</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>9490</td>
<td>12208</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>172583</td>
<td>507834</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>N/A</td>
<td>1.86281e+07</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

5.2.2 Theoretical limits

In our problem, the INVAR declaration leads to a BDD, the Invar BDD, that checks if a configuration is valid, i.e. no two cars collide. This BDD represents the boolean function that is true if the given configuration is valid and false otherwise. We will construct a fooling set for this function. This fooling set yields a lower bound for the Invar BDD and give insight on its behavior when the number of cars increases or when dealing with bigger boards.
Invar Fooling set

We remind (see chapter 3) that given a partition of the set of variables, $S$ into two subsets, $L$ and $R$, a fooling set for the function $f$ is a set of assignments of the variables in $L$ ($L$-assignments), such that for any two of these $L$-assignments, say $a$ and $b$, there is a $R$-assignment, $c$, with $f(a.c) \neq f(b.c)$ where $a.c$ is the assignment over $S$ defined by $a$ for variables in $L$ and by $c$ for variables in $R$. Any BDD for $f$ with a variable ordering such that any variable in $L$ is before any variable of $R$ will have at least as many nodes as the number of members of the fooling set.

If $n$ is the number of cars in the specification, we define $L$ to be the set of all state variables ($\text{posi}$, $\text{posj}$ and $\text{horizontal}$) of $n - 1$ cars. Then a fooling set will be formed of some assignments of all of these variables. So an $L$-assignment is here a configuration of these $n - 1$ first cars. We choose some configurations such that no cars collide, and any two of these configurations are differentiable in the sense that we define now.

Two configurations are said to be differentiable when it is possible to add one more car at a position that is valid (i.e. does not collide with any car) on only one of the two configurations. Of course two configuration obtained by means of a permutation of the cars are not differentiable. But there is more, as the examples below show.

In the first example, shown in figure 5.2, the two configurations are differentiable since placing a car at (4, 4) would collide with the first while not with the second.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure52.png}
\caption{differentiable}
\end{figure}

In this second example, shown in figure 5.3, any added car colliding with a car of the first configuration would collide with a car of the second specification.
In the next example, shown in figure 5.4, we see that two configurations, albeit not having all the same board cells occupied, cannot be differentiated, in the sense defined above. Indeed any added car would collide with a car of the first specification if and only if it would collide in this position with a car of the second configuration. Cell (6,1) is occupied only on the right-hand configuration while cell (6,3) is occupied in the left-hand configuration only.

Finally, we have the following two configurations, shown in figure 5.5, that are differentiable. However the only way to differentiate them, is to collide with car number 5 of the first configuration. No position leads to a collision with the second configuration without leading to a collision with the first. Thus checking for differentiation between two boards is not symmetric.
Invar BDD Lower bound

In summary, provided that the variable ordering keeps each car’s variables together, we can construct a fooling set for the Invar BDD as a subset of the valid configurations with \( n - 1 \) cars, keeping only differentiable ones. Indeed, for any two of these configurations we have an \( R \)-assignments, i.e. the positioning of the last car, that leads in one case to a valid configuration and in the other to an invalid one, because they are by hypothesis differentiable.

We can therefore get a lower estimate of the Invar BDD size. This will be the number of differentiable valid configurations with \( n - 1 \) cars.

To get a quick idea we can use the number of valid configurations, removing the one obtained with the permutation of cars. We therefore keep in our estimate configurations that are not differentiable as in the figures 5.3, 5.4 and 5.5. It is our believe that there are negligible, at least for board with few cars. E.g. for 6x6 boards with 2 cars, there are 1622 valid configurations (without permutations), there are about 50 configurations that must be removed because they are not differentiable from another. Of course with more cars and thus less empty cells our estimate will be very bad. But any way this gives us an idea of what to expect.

One can get these numbers by combinatorial estimates or with a simple counting program. We quickly implemented one with SMV and got these results (still for size-2 Rush Hour):
<table>
<thead>
<tr>
<th>nb cars</th>
<th>nb of valid configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4x4</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>224</td>
</tr>
<tr>
<td>3</td>
<td>1044</td>
</tr>
<tr>
<td>4</td>
<td>2593</td>
</tr>
<tr>
<td>5</td>
<td>3388</td>
</tr>
<tr>
<td>6</td>
<td>2150</td>
</tr>
<tr>
<td>7</td>
<td>552</td>
</tr>
<tr>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

One can get another idea with a combinatorial method. If we take a 5x5 board, the first car can be placed into 40 different positions. Once the first car is placed the second car has 33 to 36 positions available depending on the first car’s position, leading to something near $40 \times 34$ configurations. But we must removed the one obtained with a simple permutation of these two cars, so we have an estimate of $40 \times 34/2 = 680$ valid configurations with 2 cars. The third car has 26 to 32 positions available depending on the positions of the first two cars. This lead, after removing the one obtained from permutation, to an estimation of $680 \times 28/3 = 6347$.

Estimating the average number of positions available for the i-th car is not easy, and it becomes even harder with the number of cars increasing. That is what explain the differences between these estimates and the actual number reported above:

\[
40 \times 34/2 = 680
\]

\[
680 \times 28/3 = 6347
\]

\[
6347 \times 22/4 = 34907
\]

\[
34907 \times 17/5 = 118683
\]

\[
118683 \times 13/6 = 257146
\]
With \( n \) the number of car increasing, more and more configurations with \( n - 1 \) cars will not in any way lead to a valid configuration with \( n \) cars. To understand that, we can think about a 5x5 board with 12 cars, any configuration with 11 cars that isolates 2 free cells (i.e. with no free adjacent cell) will not lead to a valid configuration with 12 cars. This is why the number of configuration starts decreasing when \( n \) approaches its limits: \( s^2/2 \), where \( s \) is the size of the board.

Now we put these estimates of lower bound of the Invar BDDs against the actual results we got from our runs and note that they indeed share the same behavior:

### 4x4 board:

<table>
<thead>
<tr>
<th>nb cars (n)</th>
<th>nb nodes in INVAR</th>
<th>nb of valid configurations with n-1 cars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>123</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>1237</td>
<td>224</td>
</tr>
<tr>
<td>4</td>
<td>7334</td>
<td>1044</td>
</tr>
<tr>
<td>5</td>
<td>24227</td>
<td>2593</td>
</tr>
<tr>
<td>6</td>
<td>44209</td>
<td>3388</td>
</tr>
<tr>
<td>7</td>
<td>50081</td>
<td>2150</td>
</tr>
<tr>
<td>8</td>
<td>30762</td>
<td>552</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>36</td>
</tr>
</tbody>
</table>

### 5x5 board:

<table>
<thead>
<tr>
<th>nb cars (n)</th>
<th>nb nodes in INVAR</th>
<th>nb of valid configurations with n-1 cars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>233</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>3918</td>
<td>686</td>
</tr>
<tr>
<td>4</td>
<td>44209</td>
<td>6632</td>
</tr>
<tr>
<td>5</td>
<td>321114</td>
<td>39979</td>
</tr>
<tr>
<td>6</td>
<td>1520760</td>
<td>157000</td>
</tr>
<tr>
<td>7</td>
<td>N/A</td>
<td>407620</td>
</tr>
<tr>
<td>8</td>
<td>N/A</td>
<td>695848</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>762180</td>
</tr>
</tbody>
</table>
### 6x6 board:

<table>
<thead>
<tr>
<th>nb cars (n)</th>
<th>nb nodes in INVAR</th>
<th>nb of valid configurations with n-1 cars</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>368</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>9490</td>
<td>1622</td>
</tr>
<tr>
<td>4</td>
<td>172583</td>
<td>26160</td>
</tr>
<tr>
<td>5</td>
<td>2153132</td>
<td>280974</td>
</tr>
<tr>
<td>6</td>
<td>N/A</td>
<td>2.12461e+06</td>
</tr>
<tr>
<td>7</td>
<td>N/A</td>
<td>1.16605e+07</td>
</tr>
<tr>
<td>8</td>
<td>N/A</td>
<td>4.72024e+07</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>1.41823e+08</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

We see that the INVAR BDD size and the number of valid configurations of $n - 1$ cars do behave similarly. Moreover, these estimations point us quite precisely where the limits stand with our computational power: at 6 cars for a 5x5 boards and 5 cars for a 6x6 board. And it suggests that tackling the specification with 9 cars on a 6x6 board is not an open option since it will require 100 times more resources than we used for the 6 cars version, the one we could not resolve due to lack of resources. However we do stress that these conclusions are based on an estimation of the differentiable valid configurations, since we do not take into account configurations like the ones in figures 5.3, 5.4 and 5.5.

**Exponential lower bounds?**

We showed how fooling sets give us some insight into why the Invar BDD quickly becomes too big. We now extend our reasoning to show that, under some assumptions, this BDD grows exponentially with the size of the board and the number of cars, i.e. the size of the problem as defined in chapter 1.

Let’s assume that the boolean variables for each car stay grouped together in the BDD variable ordering. These are the \texttt{posi}, \texttt{posj} and \texttt{horizontal} variables. Let $s$, the size of the board, be a multiple of 4 so that we can divide the board into 4x4-blocks of cells. We will then have $(s/4)^2$ 4x4-blocks.
If each car is placed in the middle of one of these blocks, each block getting at most one car, the cars will not collide, we thus get a valid configuration. Because blocks are 4x4 and cars are positioned in the middle of these, two cars cannot be adjacent and therefore, the situations described in figures 5.3, 5.4 and 5.5 cannot occur. Hence two of these configurations will be differentiable if at least one of the 4x4-block is occupied in one of the two configurations and not in the other.

Figure 5.6 illustrates two such differentiable configurations.

If the number of cars is half the number of blocks, we get a lower bound for the number of elements in a fooling set filled with differentiable valid configurations. Indeed, we can construct as many such differentiable configurations as there are ways of choosing $n = s^2/32$ elements in a set of $2n = s^2/16$, where $n$ is the number of cars and $s$ the size of the board:

$$\#L \leq \binom{2n}{n} = \frac{(2n)!}{(n!)^2} \approx 4^n = 4^{s^2/32}$$

Since:

$$\binom{2(n + 1)}{n + 1} = \frac{(2n + 2)(2n + 1)(2n)!}{(n + 1)^2(n!)^2}$$

$$= \frac{(4n^2 + 3n + 2)(2n)!}{n^2 + 2n + 1 (n!)^2}$$
### Table 5.1: reduction of nodes in invar

<table>
<thead>
<tr>
<th>nb cars</th>
<th>nb invar nodes spec1</th>
<th>nb invar nodes spec2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>368</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>9490</td>
<td>1409</td>
</tr>
<tr>
<td>4</td>
<td>172583</td>
<td>34191</td>
</tr>
<tr>
<td>5</td>
<td>2153132</td>
<td>561244</td>
</tr>
<tr>
<td>6</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\[ n \to \infty \quad 4 \times \left( \frac{2n}{n} \right) \]

So we showed that provided that car's variables stay grouped, the Invar BDD will grow exponentially with the size of the board.

**Invar declaration optimization**

In the light of these considerations, we can see that a simple modification can yield some reduction of the size of the invar BDD. Indeed if we add the condition that the target car, the first car, is always horizontal and on the exit row, this reduces its possible positions from 60 to 5 and hence the size of the fooling set by the same factor.

We run both specifications, the original and the modified one, and the results, in table 5.1, partially confirmed our speculation: the size of the BDD is reduced by a factor of 6 and not 12. This difference is explained by the fact that the target car does not permute, the target car stay on the exit row, and two configurations that differ only by the permutation of the cars are not differentiable and thus lead to the same BDD node.

### 5.3 Conclusion

We used our first specification and observed its invar BDD was too big when the number of cars becomes important. We showed with to the fooling set concept, that it seems to be related to the number of valid configurations with \( n - 1 \)-cars, where
\( n \) is the number of cars. We say ‘seem to related’ because we made the assumption that car module variable stay together.

We observed that it become so big that we could not solve completely the size-2 5x5 board. And of course for a 6x6 board it was even worse.

Other attempts to use BDDs for solving some game related problems have been carried out. Michael Baldamus and others, used BDDs to attempt to find winning strategies in american checkers [27]. This is quite similar to what we want to do here. Their results are strikingly similar to ours:

”We conclude two things: First, our optimisations scale up as the 5x5 case becomes solvable if one is patient enough; second, the jump from 4x4 to 5x5 in terms of memory utilisation is so drastic that it is not obvious how one could get beyond 5x5, even if time utilisation were not an issue. The checkers problem thus seems to be a very hard and peculiar one if one tries to solve it with symbolic model checking”.

We are deeply convinced that the issue they encountered are intimately similar to the one we reported in this chapter. We think our reasoning with fooling set concept applied to the function that discriminate valid and invalid configurations is applicable to their problem.

In conclusion this specification is nearly useless to solve our problem. This chapter provide the experimental results as well as theoretic aspects that lead to this conclusion.
Chapter 6

Second Rush Hour SMV specification

Armed with the insight we got into Rush Hour and the BDDs’ theoretical limits from the previous chapters and with our first experimental results, we designed the next specification with two goals in mind: limit the number and size of variables and limit as much as possible the interdependency between variables.

6.1 Specification description

This second specification gives each car a predefined direction and line or column accordingly. There are exactly 2 cars per line and 2 cars per column, called the primary and the secondary car. Each car has a position on its line or column, which is the position of its leftmost, upper cell. Moreover, each car can be on the board or not. Secondary cars can be on the board only if the primary car (the one on the same line or column) is also on the board. This is shown in figure 6.1.

Cars with id 1 to 6 are the primary cars of the columns 1 to 6, they are positioned anywhere on their respective column. The secondary cars on the columns (id 7 to 12 for columns 1 to 6 respectively) are positioned deeper than the primary one, i.e. their positions are greater than the position of the primary car of their respective columns. We get a similar pattern for lines, with primary cars numbered from 13 to 18 and...
secondary cars from 19 to 24.

Finally, we note that unlike the first specification, with one specification we have all the possible combinations of cars. But like in the case if the first specification we must generate one for each board size.

The first advantage of this specification is that it reduces the cars number of variables from 7 bits to 4 bits in the case of the primary cars and to 3 bits in the case of the secondary cars. Indeed, secondary cars can be on the board only if its primary car is onboard and we suppose the secondary car is deeper on the board than its primary one, thus its position ranges between 3 and 5 (2 bits). Even more, the number of actual possible states for cars is reduced from 60 to 6 for the primary cars and to 4 for secondary cars.

The second advantage is that line, respectively column, cars are not interdependent at all except if on the same line, respectively column. In other word, any vertical car has no interdependency with any other vertical car, except the only car on the same column. And this same car will have, but reduced, dependency with horizontal cars.
We said reduced because a vertical car can collide with an horizontal one only if the former is on the line of the latter and the latter is on the column of the former. This reduces greatly the complexity of the interaction between cars.

It can be found in annex D for Size-2 Rush Hour and in annex A for General Rush Hour.

### 6.1.1 Primary and Secondary car modules declaration

The primary and secondary car modules encapsulate the car’s position and its onboard state. `onboard` stay constants with the transitions. While the position `pos` can be increased or decreased provided the board size allows it.

**Primary car**

```plaintext
MODULE CAR1()

VAR
    pos : 1..5;
    onboard: boolean;

ASSIGN
    next(onboard) := onboard;
    next(pos) :=
        case
            !onboard : 1;
            pos = 1 : 2;
            pos = 5 : 4;
            1: {pos - 1, pos + 1};
        esac;
```

**Secondary car**

```plaintext
MODULE CAR2()

VAR
```
pos : 3..5;
onboard: boolean;

ASSIGN
next(onboard) := onboard;
next(pos) :=
case
  !onboard : 3;
  pos = 3 : 4;
  pos = 5 : 4;
  1: {pos - 1, pos + 1};
esac;

6.1.2 VAR declaration

Cars are instantiated following their positions on the board: first cars, 1 to 6 and 13 to 18, are instances of CAR1, while second cars, 7 to 11 and 19 to 24, are instances of CAR2. The facts that there are column cars and line cars, that c[1] and c[7] are on the first column, c[13] and c[19] are on the first line, and so on, will all be specified in the INVAR declaration.

VAR
  c[1]: process CAR1();
  c[7]: process CAR2();
  c[13]: process CAR1();
  c[19]: process CAR2();
  c[2]: process CAR1();
  c[8]: process CAR2();
  c[14]: process CAR1();
  c[20]: process CAR2();
  [ and so on ...]
### 6.1.3 INIT declaration

We specify in the INIT declaration that the target car is at the exit point. The trick being that, on a 6x6 board, the target car is car number 21 if it is on the board else it is car number 15 that must, therefore, always be on the board (because for the 21 to be on board as a secondary car, 15 must also be, being the former’s primary car).

\[
\text{INIT} \\
\quad c[15].\text{onboard} \& \\
\quad (c[21].\text{onboard} \rightarrow (c[21].\text{pos} = 5)) \& \\
\quad (!c[21].\text{onboard} \rightarrow (c[15].\text{pos} = 5));
\]

### 6.1.4 INVAR declaration

The INVAR declaration constrained the system not to have any collision. Moreover it states that secondary cars are indeed secondary, i.e. their position is greater than the position of the primary car of the same line/column. And any secondary car may be onboard only if its primary car is onboard also. Let’s take these conditions one by one.

There are two types of collision. The first type is between cars with same direction on the same line/column. We join this condition with the one stating that secondary cars are indeed secondary. Since cars have size 2:, we must have (for a 6x6 board):

\[
\forall i, 1 \leq i \leq 6 : c_{i+6}.\text{onboard} \rightarrow (c_i.\text{pos} + 1 \leq c_{i+6}.\text{pos})
\]

The second type of collision is between any horizontal car and any vertical car. We have:

\[
\forall i, 1 \leq i \leq 6, 13 \leq j \leq 24 : \\
(c_i.\text{onboard} \land c_j.\text{onboard}) \rightarrow \\
((c_i.\text{pos} \neq j) \lor (c_i.\text{pos} \neq j + 1) \lor
\]
\[(c_j.pos \neq i) \lor (c_j.pos \neq i + 1)\]

Finally, secondary cars can be onboard only if the primary car on the same line/column is onboard:

\[
\forall i, 1 \leq i \leq 5 : c_{i+6}.onboard \rightarrow c_i.onboard
\]
\[
\forall i, 13 \leq i \leq 18 : c_{i+6}.onboard \rightarrow c_i.onboard
\]

We now translate these conditions into an INVAR declaration:

INVAR
\[
(c[7].onboard \rightarrow c[1].onboard) \&
((c[7].onboard \& c[1].onboard) \rightarrow (c[1].posi + 1 < c[7].posi))
\]
\[
(c[19].onboard \rightarrow c[13].onboard) \&
((c[19].onboard \& c[13].onboard) \rightarrow (c[13].posi + 1 < c[19].posi))
\]
\[
(c[1].onboard \rightarrow
\quad ((c[13].onboard \rightarrow
\quad \quad ((c[13].posi > 1) \lor (c[13].posi < 0) \lor
\quad \quad (c[1].posi > 1) \lor (c[1].posi < 0)))
\quad \&
\quad (c[19].onboard \rightarrow
\quad \quad ((c[19].posi > 1) \lor (c[19].posi < 0) \lor
\quad \quad (c[1].posi > 1) \lor (c[1].posi < 0))))
\]
[...And so on...]
6.2 Finding the hardest configuration

We note that for Rush Hour the forward image is equal to the backward image; every move is reversible. In consequence, there is no shortcut and therefore the frontier can be computed as follow:

\[
\begin{align*}
\text{frontier}_0 &= \text{initial states} \\
\text{frontier}_1 &= \text{forward}_0(\text{frontier}_0) \setminus \text{frontier}_0 \\
\text{frontier}_k &= ((\text{forward}_0(\text{frontier}_{k-1})) \setminus \text{frontier}_{k-1}) \setminus \text{frontier}_{k-2}
\end{align*}
\]

**proof.** Let \(\text{frontier}_{k-1}\) be the \(k-1\) frontier. Any state of its forward image cannot be nearer than \(k-2\) from the initial states. If it was nearer, let’s call \(s_1\) this state, then there would be a state \(s_2 \in \text{frontier}_{k-1}\) with \(s_1 = \text{forward}_0(s_2)\). But then, because backward and forward image are equal we have \(s_1 = \text{backward}_0(s_2)\) or \(\text{forward}_0(s_1) = s_2\) thus \(s_2\) is not in the \(k-1\) frontier, since it is nearer than \(k-1\) steps from the initial states. ■

Therefore, the algorithm for computing the hardest configuration is simply the computation of the successive frontiers until we reach the last one. We then pick a state in this last frontier.

We ran our specification on 4x4, 5x5 and 6x6 boards. Unlike the first specification we don’t have a static number of cars on the board (because of the onboard variable). All possible combinations of cars are taken into account, while for the first specification we had to generate a family of specification one for each possible number of cars.

**4x4 board for Size-2 Rush Hour:**

- **Init size** 4
- **Invar size** 413
- **Transition relation size** 388
- **System diameter** 388
The 10 steps configuration found is shown in figure 6.2.

5x5 board for Size-2 Rush Hour

Init size 8

Invar size 5591

Transition relation size 1523

System diameter 34

The 33 steps configuration is shown in figure 6.3.

We resume the results in the following table.
It is striking that there are very few configurations requiring more than 8 steps for the 4x4 board, and more than 23 steps for 5x5 board. We can see (from figure 6.3) that, on a 5x5 board, the 4 configurations that requires 32 steps are derived from the 2 possible moves of the 2 configurations requiring 33 steps. The same phenomenon is visible for the 4x4.

<table>
<thead>
<tr>
<th>distance from winning</th>
<th>nb of configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4x4 board</td>
</tr>
<tr>
<td>2</td>
<td>2208</td>
</tr>
<tr>
<td>3</td>
<td>2122</td>
</tr>
<tr>
<td>4</td>
<td>1464</td>
</tr>
<tr>
<td>5</td>
<td>726</td>
</tr>
<tr>
<td>6</td>
<td>310</td>
</tr>
<tr>
<td>7</td>
<td>150</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>0</td>
</tr>
</tbody>
</table>
Removing those derived configurations would enable us to retain only the more interesting and difficult ones. That’s what we do in the next section. There we also report full result for the 6x6 board for size-2 Rush Hour.

### 6.3 Finding interesting hard initial configurations

While in the previous section we computed one of the hardest configuration, we also reported the size of the frontiers. This showed us that there are few very hard configurations. Now we want to find those few.

We assume that a configuration whose solution takes less steps than another, and is reachable from the second, is easier than the latter.

We consider the set of configurations (states) reachable from any winning configuration. Once a configuration is found that requires the most steps, all the other configurations that are reachable from this one can be safely removed from the set. And we can resume the process of finding the furthest configuration with the remaining configurations in the set.

The algorithm we applied is the following. We recall that forward and backward images are equal for Rush Hour. (see implementation in Annex B)

```plaintext
// F[i] is i-th frontier

F[0] := all_winning_conf;
F[1] := forward_image(F[0])\F[0];
i := 1;
while(F[i] is not empty) {
i++;
F[i] := (forward_image(F[i-1])\F[i-1])\F[i-2];
}

R := empty_set;
while(i>0) {
F[i] := F[i]\R;

```
if(F[i] is empty) i--; 
else {
    conf := pick_one_configuration(F[i]);
    print(conf);
    R := R U reachable(conf);
}

6.3.1 Size-2 Rush Hour

In all the results we observe that there are indeed very few interesting hard configurations.

4x4 Board

Figure 6.2: 10 steps configuration
<table>
<thead>
<tr>
<th>distance from winning</th>
<th>nb of configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>310</td>
</tr>
<tr>
<td>4</td>
<td>726</td>
</tr>
<tr>
<td>3</td>
<td>1464</td>
</tr>
<tr>
<td>2</td>
<td>2122</td>
</tr>
</tbody>
</table>

5x5 Board

Figure 6.3: 33 steps configuration
<table>
<thead>
<tr>
<th>distance from winning</th>
<th>nb of configurations</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Interesting</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>14</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>20</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>34</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>58</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>82</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>154</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>230</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>290</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>328</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>404</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>588</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>932</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>243932</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
Figure 6.4: 65-steps size-2 Rush Hour configuration

6x6 Board

<table>
<thead>
<tr>
<th>distance from winning</th>
<th>nb of configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>63</td>
<td>10</td>
</tr>
<tr>
<td>62</td>
<td>11</td>
</tr>
<tr>
<td>61</td>
<td>12</td>
</tr>
<tr>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>59</td>
<td>21</td>
</tr>
<tr>
<td>58</td>
<td>30</td>
</tr>
<tr>
<td>57</td>
<td>36</td>
</tr>
<tr>
<td>56</td>
<td>37</td>
</tr>
<tr>
<td>55</td>
<td>47</td>
</tr>
<tr>
<td>54</td>
<td>70</td>
</tr>
<tr>
<td>53</td>
<td>107</td>
</tr>
<tr>
<td>52</td>
<td>145</td>
</tr>
<tr>
<td>51</td>
<td>203</td>
</tr>
<tr>
<td>50</td>
<td>269</td>
</tr>
<tr>
<td>49</td>
<td>370</td>
</tr>
<tr>
<td>48</td>
<td>514</td>
</tr>
<tr>
<td>47</td>
<td>707</td>
</tr>
</tbody>
</table>
6.3.2 Generalized Rush Hour

Now that we solved our problem for size-2 Rush Hour we tackle the General Rush Hour case. Trucks on a 4x4 board does not yield any interesting configuration since it takes 3 cells of the column’s 4 (or line).

5x5 Board

This is the hardest Generalized Rush Hour that our method was able to compute with the resources we had.

<table>
<thead>
<tr>
<th>distance from winning</th>
<th>nb of configurations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Interesting</td>
<td>Size-2 Interesting</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>18</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>33</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>40</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>64</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>115</td>
<td>18</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>241</td>
<td>14</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>448</td>
<td>18</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>677</td>
<td>31</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>904</td>
<td>25</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1019</td>
<td>22</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>535518</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>829628</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.15646e+06</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

The main conclusion that we can get from these results is that out of the 156 configurations that require more than 25 steps, only 16 are interesting in the sense defined
above. Even more, the ratio of interesting configurations decreases with the distance going down. Indeed, of the 1019 configurations that requires 19 steps, only 22 are interesting!

We also see, when comparing these results with the one from Size-2 Rush Hour, that the hardest are Size-2 Rush Hour instances shown in figure 6.3 (the second configuration is obtained by removing the car number 12), while the two new hardest configurations requiring 32 steps, do have a truck.

6x6 Board

The required computer resources were too high. But this time it is not because of the Invar BDD. Indeed the successive frontier BDDs are the ones that become big. And we saw in the BDD chapter that the synthesis operation, i.e. the binary boolean operations (and, or,...), have a running time linked to the product of the size of the two involved BDDs. We stressed in chapter 3 that it was therefore crucial not to get too big BDDs. That is what is happening here: operations on big BDDs, hence a long, long, running time. We therefore modified the method in order to reduce the involved BDDs. That is the subject of the last section.

6.4 Partitioning

We saw that we were unable to find the hardest interesting configurations for Generalized Rush Hour on a 6x6 board. But this time it was not because of the Invar BDD, but because of the size of the frontier BDDs. We will partition the problem into smaller independent subproblems, and this hopefully will help.

Let \( g \), a function on the system variables, be an invariant of the transition, i.e it stays equal while the system evolves following its transition. E.g. any onboard variable form such an invariant. We can partition the set of system states, \( S \), into as many subsets as the number of values \( g \) can take. We denote these subsets as \( S_1, S_2, \ldots S_k \). Obviously for any state in \( S_i \), any state reachable from that one will also be in \( S_i \).

We have the following relations:
\[
forward\_image(S) = forward\_image(\bigcup S_i) = \bigcup forward\_image(S_i)
\]
\[
frontier_k(I) = frontier_k(\bigcup I_i) = \bigcup frontier_k(I_i)
\]

So if we have such an invariant and thus a partition we can run our algorithm on each of these partitions and then take the union of the results.

We choose the following invariant: \(\sum_{i<m} c_{ki}.onboard \times 2^i\). It is equivalent as choosing \(m\) cars, then a partition is made of the configurations such that the \(k_i\) car is onboard if and only if the invariant defining this partition has the \(i\)-th bit of its binary representation equal to 1.

In other words we choose some cars and fix their onboard variable. We run all the specifications resulting from the possible different combinations. If we choose a primary and its secondary car there is only 3 possibilities: none is onboard, the primary car is onboard, or both are onboard. With 6 lines and 6 columns each having a primary car and a secondary car, we can partition \(S\) into \(3^{12} = 531441\) partitions. Or choosing 4 pairs of cars we get \(3^4 = 81\) partitions. That’s what we did.

On the implementation level, we just have to add the invariant into the INVAR declaration:

INVAR

\[
\begin{align*}
& c[1].onboard \ & \\
& c[7].onboard \ & \\
& c[13].onboard \ & \\
& !c[19].onboard \ & \\
& c[2].onboard \ & \\
& !c[8].onboard \ & \\
& c[14].onboard \ & \\
& !c[20].onboard \ & 
\end{align*}
\]

[goes on with the invar declaration.]
We implemented that and were indeed able with 82 partitions to compute the desired 
sets: the successive frontiers and the interesting hard configurations for some parti-
tions. We haven’t done it for all partitions, we estimate that with a sequential run it 
would take less than a week on today’s computers to get all the results. Nevertheless, 
we can easily execute the 81 partition related computations in parallel, thus reducing 
the running time with the same factor. Some initial configurations can be found in 
Annex E.
Chapter 7

Conclusion

The primary goal of this work was to find hard initial configurations for the sliding blocks game Rush Hour.

At first sight, Rush Hour seems to be an easy game. However we presented in the second chapter the results from Flake and Baume [1] showing that far, from being easy, Rush Hour is theoretically hard.

To be able to study the theoretic complexity of Rush Hour, we introduced a generalized version of the game, generalized Rush Hour, as well as some derivatives: Size-2 Rush Hour and Unit Rush Hour. Then we outlined the ideas behind the proof of the hardness of this game: ”telling wether an instance of Rush Hour has or not a solution is PSpace-complete”.

Whereas this result may seem only of theoretic interest, it implies that the length of the solution of a Rush Hour instance may increase exponentially in relation with the size of the board and the number of cars, i.e. the size of the problem. Again the constructions involved in the proof of the PSpace-hardness cannot be used to infer that our small 6x6 grid leads to substantially long solutions for some instances. Indeed the proof involved hundreds of cars and boards that overwhelmingly exceed the 6x6 size we dealt with in this work. However, Tromp and Cilibrasi [8] studied Unit Rush Hour behavior on 2x2 board to 6x6 board. They were able to exhibit an instance requiring 732 moves to be solved. Hence, even on small board some Rush Hour instances show a surprising complexity, supporting, on practical grounds, the
Finding hard initial configurations can be done with exhaustive state space analysis. However, doing so, one quickly faces a big hurdle: combinatorial explosion. If one can find some simplifications, like removing useless configurations in order to decrease significantly the state space, then that may lead to some solutions. Samuel Hiard did just that in his "Recherche automatique de solutions difficiles pour le jeu Rush Hour" [28]. He managed to subdue the state space explosion problem for 6x6 boards, thanks to an impressive optimization work and a thorough analysis of the game, leading to simplifications.

We chose a totally different road. The main idea of our work was to try to bypass this explosion problem by using symbolic methods, just like what is done in the field verification.

Symbolic methods represent sets in some compact way instead of extensively dealing with their elements. Starting with the set of winning Rush Hour configurations and then iteratively computing, still symbolically, the successive set of reachable configurations was the method to find the hard initial configurations.

However we quickly crashed into a wall. Our compact representations of states were not that compact. We showed that in our first representation, under some assumptions, the problem was with the BDD that had to separate valid of invalid configurations, i.e. decide if the configuration’s cars collide.

The BDD’s fooling set concept helped us construct a lower bound for the size of the Invar BDD for our 5x5 and 6x6 board problems. This gives us an inner understanding on the reasons why the Invar is getting too big even with a 5x5 board and few cars. This also suggests that the amount of resources needed to bring this computation to a successful conclusion is simply too huge. Moreover, under the same assumptions, we showed that this BDD grows exponentially with the size of the problem.

Our reasoning could apply to other board games where the validity of a configuration depends on the position of some pieces (chess, american checkers, othello). Actually we were impressed by the similitude with the work of Baldamus on american checkers [27]. They tried to use symbolic methods in order to find configurations where one player has a winning strategy, i.e. whatever the second player does, the former always
has an answer leading to victory. They did not manage to deal with 5x5 board in a reasonable time. We are strongly convinced the limitations in their model are of the same nature than the ones we encountered in this work.

With our first specification and the insight we got from applying the fooling set concept to it, we could come with a second specification. This one overcame some weaknesses of the former. It indeed reduced the number of variables per car and the interdependency between variables of two different cars. This enabled us to produce complete results for 4x4, 5x5 and 6x6 board of Size-2 Rush Hour and 5x5 board Generalized Rush Hour.

Thanks to the symbolic nature of our representation we were able to implement an algorithm for finding the interesting configurations (i.e. the ones that are further from the initial states than any other configuration reachable from it), and for each, compute the size of the set of the configurations reachable from that one as well as its distance from the exit (these were indeed two criteria of the hardness in Samuel Hiard work [28].

Other interesting analysis become possible once one has computed the frontier sets. For example, for any frontier computing the number of configurations that have, say, $k$ cars is straightforward. One must simply compute the intersection with the BDD representing the set of all such configurations. With the same method, one can compute, for any frontier, the number of interesting configurations with $k$ trucks. Hence symbolic analysis can in that way bring interesting light on Rush Hour.

However, with its better results, our second specification falls just short of solving the complete General Rush Hour on a 6x6 board. While with some more computing resources a 6x6 complete analysis can be carried through with this exact technique, it is not the case for greater problem like, say a 10x10 board Rush Hour. But we do not rule out the possibility that another specification could bring better results, since we already saw that we got way better results from our second specification against the ones we got with the first.

Finally we were able to deal with General Rush Hour on 6x6 board thanks to the invariant-based system partioning. This method may help solve systems that require big BDDs by dividing these into smaller ones.
In conclusion, the hypothesis that the simple nature of the Rush Hour rules would translate into small manageable BDDs did not seem to be correct in this case. But, we saw in chapter 3 that the function HWB leads to exponentially big BDDs for any ordering while it has been proved that VLSI techniques need spaces of an order of the problem size squared. So different techniques are suitable for different problems, and the failure of one does not always mean the inability of others.

Hence it would be interesting to see how other symbolic techniques, like the ones based on SAT solvers, compare against BDDs for this particular problem.
Annex A

-- ******************************** --
-- GENERAL RUSH HOUR ON A 5x5 BOARD --
-- SPECIFICATION 2 --
-- ******************************** --


MODULE main

VAR

  c[1]: process CAR1();
  c[6]: process CAR2();
  c[11]: process CAR1();
  c[16]: process CAR2();

  c[2]: process CAR1();
  c[7]: process CAR2();
  c[12]: process CAR1();
  c[17]: process CAR2();

  c[3]: process CAR1();
  c[8]: process CAR2();
  c[13]: process CAR1();
  c[18]: process CAR2();

  c[4]: process CAR1();
  c[9]: process CAR2();
c[14]: process CAR1();
c[19]: process CAR2();

 INIT

 c[13].onboard &
 (c[18].onboard -> (c[18].posi = 4)) &
 (!c[18].onboard -> (c[13].posi = 4));

-- ******************************* --

 INVAR

-- ONE CAR PER POSITION MAXIMUM IN INIT AND ANY OTHER STATE--

 (c[6].onboard -> c[1].onboard) &
 ((c[6].onboard & c[1].onboard) -> (c[1].posi + c[1].size - 1 < c[6].posi)) &
 (c[16].onboard -> c[11].onboard) &
 (c[1].posi > 1) | (c[1].posi < (2 - c[1].size) ) ) &

 (c[16].onboard -> (c[16].posi > 1) | (c[16].posi < (2 - c[16].size) ) |
 (c[1].posi > 1) | (c[1].posi < (2 - c[1].size) ) ) &
(c[12].onboard
   ->( (c[12].posi > 1) | (c[12].posi < (2 - c[12].size) ) | (c[1].posi > 2) | (c[1].posi < (3 - c[1].size) ) ) ) &
(c[17].onboard
   ->( (c[17].posi > 1) | (c[17].posi < (2 - c[17].size) ) | (c[1].posi > 2) | (c[1].posi < (3 - c[1].size) )) ) &
(c[13].onboard
   ->( (c[13].posi > 1) | (c[13].posi < (2 - c[13].size) ) | (c[1].posi > 3) | (c[1].posi < (4 - c[1].size) ) ) ) &
(c[18].onboard
   ->( (c[18].posi > 1) | (c[18].posi < (2 - c[18].size) ) | (c[1].posi > 3) | (c[1].posi < (4 - c[1].size) )) ) &
(c[14].onboard
   ->( (c[14].posi > 1) | (c[14].posi < (2 - c[14].size) ) | (c[1].posi > 4) | (c[1].posi < (5 - c[1].size) ) ) ) &
(c[19].onboard
   ->( (c[19].posi > 1) | (c[19].posi < (2 - c[19].size) ) | (c[1].posi > 4) | (c[1].posi < (5 - c[1].size) )) ) &
(c[15].onboard
   ->( (c[15].posi > 1) | (c[15].posi < (2 - c[15].size) ) | (c[1].posi > 5) | (c[1].posi < (6 - c[1].size) ) ) ) &
(c[20].onboard
   ->( (c[20].posi > 1) | (c[20].posi < (2 - c[20].size) ) | (c[1].posi > 5) | (c[1].posi < (6 - c[1].size) )) ) &

[AND SO ON...]
size: 2..3;

ASSIGN
    next(onboard) := onboard;
    next(size) := size;

next(posi) :=
  case
    !onboard : 1;
    posi=1 : 2;
    posi = 4 : 3;
    1: {posi - 1, posi + 1};
  esac;

INVAR
    (!onboard -> ((posi = 1) & (size = 2))) &
    (posi + size <= 6);

MODULE CAR2()

VAR
    posi : 3..4;
    onboard: boolean;
    size: 2..3;

ASSIGN
    next(onboard) := onboard;
    next(size) := size;

next(posi) :=
  case
    !onboard : 3;
    posi=3 : 4;
    posi = 4 : 3;
    1: {posi - 1, posi + 1};
  esac;

INVAR
(!onboard -> ((posi = 3) & (size = 2))) & (posi+size <= 6);

-- ******************************* --

-- **************************** --
Annex B

Implementation of the following algorithm (where F[i] is i-th frontier):

\[ F[0] := \text{all}\_\text{winning}\_\text{conf}; \]
\[ F[1] := \text{forward}\_\text{image}(F[0]) \setminus F[0]; \]
\[ i := 1; \]
\[ \text{while}(F[i] \text{ is not empty}) \{ \]
\[ \quad i++; \]
\[ \quad F[i] := (\text{forward}\_\text{image}(F[i-1]) \setminus F[i-1]) \setminus F[i-2]; \]
\[ \} \]
\[ R := \text{empty\_set}; \]
\[ \text{while}(i>0) \{ \]
\[ \quad F[i] := F[i] \setminus R; \]
\[ \quad \text{if}(F[i] \text{ is empty}) i--; \]
\[ \quad \text{else} \{ \]
\[ \quad \quad \text{conf} := \text{pick}\_\text{one}\_\text{configuration}(F[i]); \]
\[ \quad \quad \text{print}(\text{conf}); \]
\[ \quad \quad R := R \cup \text{reachable}(\text{conf}); \]
\[ \quad \} \]
\[ \} \]
void BddFsm_print_furthest_states_info(const BddFsm_ptr self,
    const boolean print_states,
    FILE* file; FILE* file2)
{
    BddStates initial_states;
    BddInvarStates invars;
    BddStates frontier[150];

    bdd_ptr mask;
    double reached_cardinality;
    double search_space_cardinality;

    mask = BddEnc_get_state_vars_mask_bdd(self->enc);
    invars = BddFsm_get_state_constraints(self);

    /*
     * 0/ Compute frontier[0] and frontier[1]
     */

    // frontier[0]
    initial_states = BddFsm_get_init(self);
    bdd_and_accumulate(self->dd, &initial_states, invars);

    bdd_and_accumulate(self->dd, &initial_states, mask);
    frontier[0] = initial_states;
    bdd_free(self->dd, invars);
    bdd_free(self->dd, mask);

    /*
     * 1/ Compute all frontiers:
     *
     * while(F[i] is not empty) {
     */
i++;
F[i] := (forward_image(F[i-1]) \ F[i-1]) \ F[i-2];
}
*/
int i = 0;
{
  bdd_ptr not_frontier_i_1 = bdd_not(self->dd, frontier[0]);
  bdd_ptr not_frontier_i_2 = bdd_one(self->dd);

  while(!bdd_isnot_zero(self->dd, frontier[i-1])) {
    i++;
    BddStates next_frontier = BddFsm_get_forward_image(self, frontier[i-1]);
    bdd_and_accumulate(self->dd, &next_frontier, not_frontier_i_1);
    bdd_and_accumulate(self->dd, &next_frontier, not_frontier_i_2);
    frontier[i] = next_frontier;

    double nb_frontier_states =
      BddEnc_count_states_of_bdd(Enc_get_bdd_encoding(), frontier[i]);
    fprintf(file2, "%d		%g \\
    
", i, nb_frontier_states);
    bdd_free(self->dd, not_frontier_i_2);
    not_frontier_i_2 = not_frontier_i_1;
    not_frontier_i_1 = bdd_not(self->dd, next_frontier);
  }
  bdd_free(self->dd, not_frontier_i_2);
  bdd_free(self->dd, not_frontier_i_1);
  fprintf(file, "system diameter: %d
", i);
}

/*
  * 2/ print all furthest states
  */
{ 
    bdd_ptr not_reached = bdd_one(self->dd);
    bdd_ptr reachable;
    bdd_ptr not_reachable;
    bdd_ptr choosen;
    BddEnc_ptr enc = Enc_get_bdd_encoding();
    fprintf(file, "system diameter: \%d\n", i);

    int counter=0;
    int conf_count=0;
    while(i>2 && counter < 500) {

        bdd_and_accumulate(self->dd, &frontier[i], not_reached); //frontier[i]\R
        if(bdd_is_zero(self->dd, frontier[i])) {
            fprintf(fp, "\%d\t\t\%d \n\n", i, conf_count);
            conf_count = 0;
            i--;
        } else {
            //choose and print
            choosen = BddEnc_pick_one_state_rand(enc, frontier[i]);
            conf_count++;
            //print
            BddEnc_print_bdd_begin(enc,
                SexpFsm_get_vars_list(Prop_master_get_scalar_sexp_fsm()),
                false);

            printf(file, "===== Configuration at: \%d =====\n", i);
            BddEnc_print_bdd(enc, choosen, nusmv_stdout);
            BddEnc_print_bdd_end(enc);

            // compute reachable and remove from not_reached.
            reachable = compute_reachable(self, choosen, i, file);
            double nb_reachable_states =
                BddEnc_count_states_of_bdd(Enc_get_bdd_encoding(),
                    reachable);
    }
}
fprintf(file,
    "Configuration at %d \t with %g reachable states \n",
i, nb_reahcable_states);

not_reachable = bdd_not(self->dd, reachable);
bdd_and_accumulate(self->dd, &not_reached, not_reachable);

bdd_free(self->dd, reachable);
bdd_free(self->dd, not_reachable);
counter++;
}
}
bdd_free(self->dd, not_reached);
}
}
}

bdd_ptr compute_reachable(const BddFsm_ptr self,
    bdd_ptr input_bdd,
    int distance, FILE* file) {

    bdd_ptr next;
distance = 2*distance;

    while(distance > 0) {
        next = BddFsm_get_backward_image(self,
            BDD_STATES(input_bdd));

        bdd_ptr xor;
xor = bdd_xor(self->dd, input_bdd, next);
bdd_free(self->dd, input_bdd);
input_bdd = next;

        if(bdd_is_zero(self->dd, xor)) {
            bdd_free(self->dd, xor);
break;
} else {
            bdd_free(self->dd, xor);
distance--; 

} }

return next; 

}
Annex C

-- ****************************************** --
--             SIZE-2 RUSH HOUR          --
--       ON A 5x5 BOARD WITH 5 CARS --
--             SPECIFICATION 1        --
-- ****************************************** --

MODULE main

VAR
  c[1]: process CAR();
  c[2]: process CAR();
  c[3]: process CAR();
  c[4]: process CAR();
  c[5]: process CAR();

INIT

  c[1].posi = 3 & c[1].posj=4 & c[1].horizontal;

INVAR

  ( (c[2].posi > (c[1].posi + !c[1].horizontal)) | 
    (c[1].posi > (c[2].posi + !c[2].horizontal)) | 
    (c[2].posj > (c[1].posj + c[1].horizontal)) | 
    (c[1].posj > (c[2].posj + c[2].horizontal)) )&

  ( (c[3].posi > (c[1].posi + !c[1].horizontal)) |
( (c[5].posi > (c[3].posi + !c[3].horizontal)) | 
  (c[3].posi > (c[5].posi + !c[5].horizontal)) | 
  (c[5].posj > (c[3].posj + c[3].horizontal)) | 
  (c[3].posj > (c[5].posj + c[5].horizontal)) ) &

( (c[5].posi > (c[4].posi + !c[4].horizontal)) | 
  (c[4].posi > (c[5].posi + !c[5].horizontal)) | 
  (c[5].posj > (c[4].posj + c[4].horizontal)) | 
  (c[4].posj > (c[5].posj + c[5].horizontal)) );

-- ************************************************************************* --
-- ********** MODULE CAR ********** --
-- ************************************************************************* --

MODULE CAR()

VAR
  posi : 1..5;
  posj : 1..5;
  horizontal : boolean;

ASSIGN
  next(horizontal) := horizontal;

  next(posi) :=
    case
      horizontal : posi;
      posi = 1 : 2;
      posi = 5 : 4;
      1: {posi - 1, posi + 1};
    esac;

  next(posj) :=
    case
!horizontal : posj;
posj = 1 : 2;
posj = 5 : 4;
  1: {posj - 1, posj + 1};
esac;

INVAR
  ((posi + !horizontal) <= 5) & ((posj + horizontal) <= 5);
Annex D

-- ************************************************** --
-- SIZE-2 RUSH HOUR ON A 5x5 BOARD  --
-- SPECIFICATION 2  --
-- ************************************************** --

MODULE main

VAR
    c[1]: process CAR1();
    c[6]: process CAR2();
    c[11]: process CAR1();
    c[16]: process CAR2();

    c[2]: process CAR1();
    c[7]: process CAR2();
    c[12]: process CAR1();
    c[17]: process CAR2();

    c[3]: process CAR1();
    c[8]: process CAR2();
    c[13]: process CAR1();
    c[18]: process CAR2();

    c[4]: process CAR1();
    c[9]: process CAR2();
    c[14]: process CAR1();
    c[15]: process CAR2();
c[19]: process CAR2();

c[5]: process CAR1();
c[10]: process CAR2();
c[15]: process CAR1();
c[20]: process CAR2();

INIT

c[13].onboard &
(c[18].onboard -> (c[18].posi = 4)) &
(!c[18].onboard -> (c[13].posi = 4));

-- ************************************************** --
INVAR

-- ONE CAR PER POSITION MAXIMUM IN INIT AND ANY OTHER STATE--

(c[6].onboard -> c[1].onboard) &
((c[6].onboard & c[1].onboard) -> (c[1].posi + 1 < c[6].posi)) &
(c[16].onboard -> c[11].onboard) &
(c[16].onboard -> (c[16].posi > 1) | (c[16].posi < 0) | (c[1].posi > 1) | (c[1].posi < 0)) &
(c[12].onboard -> (c[12].posi > 1) | (c[12].posi < 0) | (c[1].posi > 2) | (c[1].posi < 1)) &
(c[17].onboard -> (c[17].posi > 1) | (c[17].posi < 0) |
(c[1].posi > 2) | (c[1].posi < 1 )))) &

(c[13].onboard ->( (c[13].posi > 1) | (c[13].posi < 0 ) | (c[1].posi > 3) | (c[1].posi < 2 )) &
(c[18].onboard ->( (c[18].posi > 1) | (c[18].posi < 0 ) | (c[1].posi > 3) | (c[1].posi < 2 ))) &

(c[14].onboard ->( (c[14].posi > 1) | (c[14].posi < 0 ) | (c[1].posi > 4) | (c[1].posi < 3 )) &
(c[19].onboard ->( (c[19].posi > 1) | (c[19].posi < 0 ) | (c[1].posi > 4) | (c[1].posi < 3 ))) &

(c[15].onboard ->( (c[15].posi > 1) | (c[15].posi < 0 ) | (c[1].posi > 5) | (c[1].posi < 4 ))) &
(c[20].onboard ->( (c[20].posi > 1) | (c[20].posi < 0 ) | (c[1].posi > 5) | (c[1].posi < 4 ))) &

1)) &

(c[7].onboard -> c[2].onboard) &
((c[7].onboard & c[2].onboard) -> (c[2].posi + 1 < c[7].posi)) &
(c[17].onboard -> c[12].onboard) &
((c[17].onboard & c[12].onboard) -> (c[12].posi + 1 < c[17].posi)) &
(c[2].onboard ->(
(c[11].posi < 1 ) | (c[2].posi > 1) | (c[2].posi < 0 )))) &
(c[16].onboard ->( (c[16].posi > 2) |
(c[16].posi < 1 ) | (c[2].posi > 1) | (c[2].posi < 0 ))) &

(c[12].onboard ->( (c[12].posi > 2) |
(c[12].posi < 1 ) | (c[2].posi > 2) | (c[2].posi < 1 )))) &
(c[17].onboard ->( (c[17].posi > 2) | (c[17].posi < 1 ) | (c[2].posi > 2) | (c[2].posi < 1 ))) &

(c[13].onboard ->( (c[13].posi > 2) | (c[13].posi < 1 ) | (c[2].posi > 3) | (c[2].posi < 2 )) &
(c[18].onboard ->( (c[18].posi > 2) | (c[18].posi < 1 ) | (c[2].posi > 3) | (c[2].posi < 2 ))) &

(c[14].onboard ->( (c[14].posi > 2) | (c[14].posi < 1 ) | (c[2].posi > 4) | (c[2].posi < 3 )) &
(c[19].onboard ->( (c[19].posi > 2) | (c[19].posi < 1 ) | (c[2].posi > 4) | (c[2].posi < 3 ))) &

(c[15].onboard ->( (c[15].posi > 2) | (c[15].posi < 1 ) | (c[2].posi > 5) | (c[2].posi < 4 )) &
(c[20].onboard ->( (c[20].posi > 2) | (c[20].posi < 1 ) | (c[2].posi > 5) | (c[2].posi < 4 )))

[AND SO ON...]

MODULE CAR1()

VAR
    posi : 1..4;
    onboard: boolean;

ASSIGN
    next(onboard) := onboard;

    next(posi) :=
        case
!onboard : 1;
posi=1 : 2;
posi = 4 : 3;
1: {posi - 1, posi + 1};
esac;

INVAR
!onboard -> posi = 1;

MODULE CAR2()

VAR
posi : 3..4;
onboard: boolean;

ASSIGN
next(onboard) := onboard;

next(posi) :=
case
!onboard : 3;
posi=3 : 4;
posi = 4 : 3;
1: {posi - 1, posi + 1};
esac;

INVAR
!onboard -> posi = 3;

-- ****************************************** --
Annex E

92 steps

Samuel Hiard's configuration:
89 steps

86 steps

76 steps
2RH: 65 steps

2RH: 61 steps

2RH: 56 steps
Bibliography


