University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master's Degree Course in Computer Science

Symbolic Soundness Verification of Data Petri Net

Supervisor
Prof. **Davide Bresolin**
Università di Padova

Master Candidate:
Nicola Ferrari
Università di Padova

A.A. 2022-2023

My university career was certainly not wonderful, indeed I would say rather tormented and when you get to the end you begin to take stock of what this experience has left you.
Surely it taught me that maximum effort is not synonymous with guaranteed success or good results, but it is also important to accept bad results and try and try again until success is achieved and if I have gotten this far is certainly thanks to this spirit of determination.
One thing for sure is that I've tried to put passion into everything I've done, even when the environment didn't seem favourable.
This thesis represented for me a good conclusion of this path, working with a programming language that I really appreciate and being able to learn many things about a world that was unknown to me.
Now, however, the time has come for thanks: To my family, to my friends, to the ITI of Ferrara and above all to my grandfather, thank you.
Special thanks to my supervisor for following me and helping me in writing this thesis.
Thanks also go to my high school teachers for the valuable knowledge and skills they provided me on the Java programming language and the graphical interfaces that allowed me to speed up the work of this project.
The end of a journey is always the dawn of a new beginning.

Nicola Ferrari

Abstract

Data Petri have important applications especially within concurrent and discrete-event systems. To faithful represent a system, a DPN must respect the Soundness properties, i.e. those properties that guarantee the proper functioning of the system. In a system a series of events can occur in a certain order and to be correct we must be sure that no possible combination of events can lead to inconsistent results.

Data Petri Nets are an extension of standard Petri Nets work, with data and concepts of constraint programming. While in basic petri nets the transitions are limited to consuming tokens from the incoming place and adding them to the outgoing place, in the DPN a constraint will be represented in the transition and the task of our program is to verify whether this constraint will always be respected, will never be respected or whether in some cases it will be respected and in others not.

In logic programming, constraints can be of many types, but in this thesis we will only deal with logical constraints that use one of the six relational operators ($=, \neq, >, <, \geq, \leq$) which will relate a variable to a constant, or a variable to another variable. The domain of these variables is the set of real numbers $\mathbb{R}$ or the set of boolean values, true and false. After defining the field of application it is essential to find a way to verify the soundness of the system and to do this we proceed as recently proposed by Felli et al. [1]: with the construction of the Constraint Graph. Through the implementation of the algorithms that are detailed in this thesis it is possible to verify the soundness of a DPN, knowing that DPN and Constraint Graph are related to each other through the *obs*-simulation, so if the Constraint Graph is data-aware sound, then the DPN is also data-aware sound.

Indice

Part I

Petri Nets

Introduction

Soundness of Data Petri Net

Petri nets are created to represent a discrete distributed system. A variant of the classic Petri nets is represented by the Petri nets extended with data. This variant introduces the use of data where transitions are governed by guards, i.e. conditions that express a series of constraints on the values of variables. The conditions we consider the variable - constant case or the variable - variable case. In this work both types of constraints will be treated and each variable can be read or written. In the case of a variable in writing all the previous constraints will be removed, while in reading it will verify that the new constraint added keeps the set of constraints satisfiable. Verification of the Soundness property in DPN are an infinite state systems, and thus cannot be verified by enumeration of the states. For this reason, a possible solution is to generate the Constraint Graph for which the Soundness properties will have an outcome equivalent to those of the DPN, but it is a finite structure [1]. This document shows the various steps that have been followed to implement of the Constraint Graph, starting first from a theoretical analysis of the classic Petri Nets and then from the subsequent extension with data. The three properties that guarantee the Data-Aware Soundness and the implementation of the algorithms to generate the Constraint Graph will be exposed later. The starting point is the Paper Soundness Verification of Decision-Aware Process Models with Variable-to-Variable conditions (Marco Montali, Massimiliano De Leoni and Paolo Felli) in which the theoretical elements for the implementation of this work are explained. Soundness verification algorithm is implemented with the Java programming language and structured with a graphical interface so as to be able to view the DPNs and the subsequent transformation into a Constraint Graph. Some operations required the use of constraint handlers to compute the satisfiability of constraints or the extension of constraints with new implied constraints (the saturation operation). This was possible thanks to two types of Solver: the Constraint Solver and SMT Solver. Initially the project was implemented with the Constraint Solver, but then, given the strong inefficiency on large DPNs, we switched to SMT Solvers which are more efficient. Choco Solver was used as CS, while Z3 was used as SMT. An interesting aspect was comparing the time taken by these two types of Solver for medium-large sized examples. Finally, for the generation of test examples, the ProM software was used which is able to process files in .pnml format, i.e. a file expressed in XML language which allows you to define petri nets. In the last part the part relating to the generation of both random and non-random DPN is managed and is tested on an increasing number of nodes. Large case tests also consider the size of the constraint sets and the number of variables in them.

CHAPTER 1

Introduction to standard Petri nets

In this chapter, standard Petri nets are introduced and then extended with data. The functioning of the Data Petri Net includes all the rules foreseen by the Petri nets, for this reason the first step will be to know well all the rules that characterize a Petri net in order to then be able to study its extensions. The objective of this chapter is to be a summary of all the preliminary concepts need for the implementation of the project.

1.1 Petri Nets

The applications and theory within Petri nets began with C.A. Petri in the early 1960s from which they take their name. Since then this area has been greatly developed both in theory and in applications, particularly for its usefulness in identifying basic aspects of concurrent systems from both a mathematical and conceptual point of view. In the context of Distributed Systems, Petri Nets are often used to describe the structure of a distributed system as a graph. The graph represented is bipartite, i.e. composed of two types of nodes: Places and Transitions, and each node of one type can only have nodes of the other type connected by arcs. An example of Petri Net is shown in Figure 1.1.

The Places are graphically represented by white circles, while the transitions by rectangles. Each Place can have Tokens inside it, represented by a black circle inside the Places.

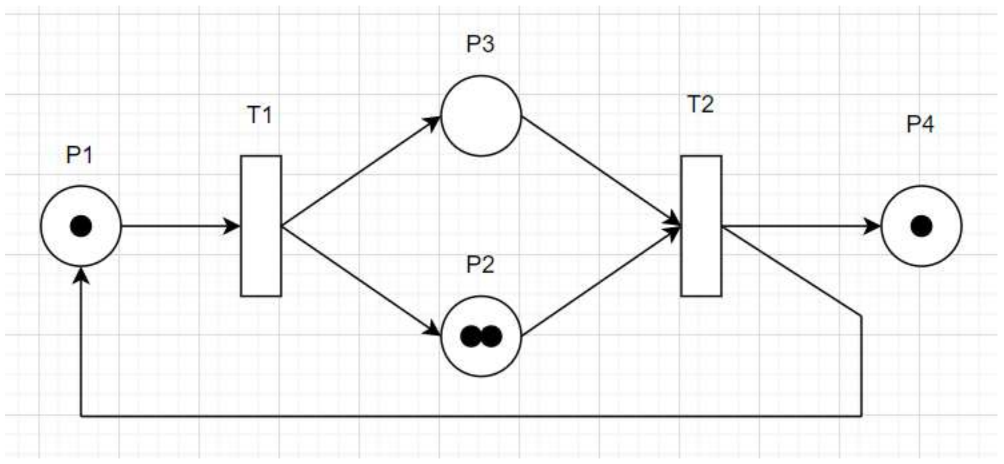


Figure 1.1: An example of Petri Net

1.2 Formal Definition of Petri Net

A Petri Net is defined by a triple (P,T,F) as Rozenberg et al. shows in [5] where

- **P** : Places of the Petri Net, represented by a circle;
- **T** : Transition of the Petri Net, represented by a Rectangle;
- **F** : Arcs of the Petri Net, they can connect a Transition to a Place or viceversa and graphically represented by arrows. Formally it can be represented by the notation $F \subseteq (P \times T) \cup (T \times P)$.

and the following properties are respected:

- for every $t \in T$ there exists $p,q \in P$ such that $(p,t),(t,q) \in F$
- for every $t \in T$ and $p,q \in P$, if $(p,t),(t,q) \in F$, then $p \neq q$.

We define the set X as the elements of the Petri Net and formally $X = P \cup T$. a Net N = (P,T,F) can in fact be seen as a directed graph G_N: the nodes of G_N are the elements of X and there is an edge from two elements x to y iff $(x,y) \in F$. Thus, the reflexive and transitive closure F* and the transitive closure F^+ indicate the paths in G_N: $(x, y) \in F^*$ iff there is a (possibly empty) directed path from x to y, and $(x, y) \in F^+$ iff there is a nonempty directed path from x to y.

The sets of places, transitions and arcs of a Petri Net N are defined respectively by S_N, T_N and F_N. In a Petri Net there are also the sets of pre-set ($^\bullet x$) and post-set ($x^\bullet$). Pre-set is defined by $^\bullet x = \{y \in N \mid (y,x) \in F_N$, while post-set is defined by $x^\bullet = \{y \in N \mid (x,y) \in F_N\}$. In Figure 1.1 the pre-set of T2 is given by p2,p3, while the post set of P3 is T2. The other properties that are obtained are properties that derive from the properties of graphs such as the concept of directed path or strongly connected component. A directed path of a net is a sequence of elements (x_0 ... x_k) where the starting element is x_0 and the final element is x_k, formally a sequence of elements satisfying $x_i \in x^\bullet{}_{i-1}$ for each i ($1 \leq i \leq k$). If an element is isolated, then $^\bullet x = x^\bullet = \emptyset$. In Figure 1.1 the sequence P1 - T1 - P3 - T2 - P4 is a path with starting element P1 and final element P4. An undirected path is a non-empty sequence (x_0 ... x_k) of elements satisfying $x_i \in {}^\bullet x_{i-1} \cup x\, ^\bullet{}_{i-1}$ for each i ($1 \leq i \leq k$). The starting element is x_0 and the final element is x_k. A Petri Net is strongly connected if there exists a directed path for each two elements of the net, while is weakly connected if there exists an undirected path for each two elements of the net. A weakly connected net is strongly connected if and only if for each directed arc (x,y) there is a directed path leading from y to x.

1.3 Transition Firing and Marking

The states of a Petri Net are defined by the concept of marking. State changes occur within the network which are caused by the occurrences of transitions. A marking of a network N is a function that maps each place to the number of tokens in the place. Formally the function is defined as follows, $m: S_N \rightarrow \mathbb{N}$ where $\mathbb{N} = \{0,1,2,3, ...\}$. The null marking is the marking that maps each place to the value 0. From the marking it is possible to define the concept of Transition Firing: a transition (t) is enabled by a marking m if m marks all the places in $^\bullet t$. Its occurrence transforms m into the marking m′, which is defined for each place s by m′(s) =

$$\begin{cases} m(s) - 1 \text{ if } s \in {}^\bullet t - t^\bullet \\ m(s) + 1 \text{ if } s \in t^\bullet -^\bullet t \\ m(s) \quad \text{otherwise} \end{cases}$$

A marking is called dead if enables no transition of N. An example of dead marking is the null marking since a transition always requires at least one token from incoming places.

Given m, marking of a Petri Net, a finite sequence of transitions (t_1, ... , t_k) is called a finite sequence of occurrences enabled by m if there are markings (m_1 , m_2, ... , m_k) such that:

$$m \overset{t_1}{\rightarrow} m_1 \overset{t_2}{\rightarrow} ... \overset{t_k}{\rightarrow} m_k$$

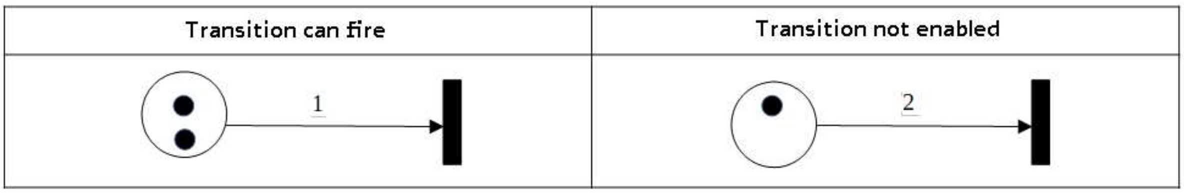


Figure 1.2: An Example of Transition Firing

A sequence of transitions is simplified with σ = t_1, t_2, ... t_k. A marking m′ is said to be reachable from a marking m if there is a finite sequence m $\xrightarrow{\sigma}$ m′. A Petri Net can also be unlimited, i.e. admitting an infinite sequence of transition occurrences. From this definition we define the following properties:

- If there is a finite occurrence sequence (σ) which originates in m and destination in m′ and m′ enables an occurrence sequence (σ)′ then σ σ' is also enabled to m.
- if there is an occurrence sequence σ that goes from m to m′ and from l to l′ then m′(s) - m(s) = l′(s) - l(s) for each place s.
- An infinite transition sequence is enabled for marking m if and only if every finite prefix of the sequence is enabled for m.
- if m and l are markings satisfying m(s) $\geq$ l(s) for every place s, every occurrence sequence enabled in m is also enabled in m.

In each Petri Net there is an initial and final marking, usually represented graphically by the number of tokens present in each place respectively in an initial and final configuration. An occurrence sequence of a marked net must be enabled by the initial marking and thus the reachable marking set is determined by the set of markings reachable by the initial marking. An example of Transition Firing is shown in Figure 1.2 .

1.4 Boundary of a Petri Net

A Place of a petri net is said to be k-limited if in each reachable marking it contains at most k tokens, so in this case the net is said to be bounded. In formal terms, a Petri Net is said to be bounded with bound b if m(s) $\leq$ b for each reachable marking m, i.e. if all its places are b-bounded. If k = 1 then the network is defined as binary or safe, in fact each place can have one token or none. A binary network generally works like a finite state automaton. In case the network is not k-limited then it is said to be unlimited.

An example of an unlimited Petri Net is shown in Figure 1.3 .

1.5 Construction of the reachability set of a Petri Net

The reachability properties of a Petri Net can be analyzed using the reachability tree. This data structure is built using a depth-first logic in which the root node represents the initial marking and at each level k the markings reachable in k steps are represented. Markings that no longer enable transitions or that have already been visited are called leaf nodes. When a leaf is reached, a backtracking operation is possibly performed in case there are still markings to visit. If the Petri net is unbounded then the reachability tree is infinite and therefore the algorithm that generates it will not have a termination. So to solve this problem we can use the Coverage Tree data structure, used to represent unlimited petri nets. Instead of assigning a different token value for each Place, you can check in which Place there is an unlimited number of tokens and replace the number with a variable (usually ω) in this way it is possible to algorithmically construct the tree without an infinite loop.

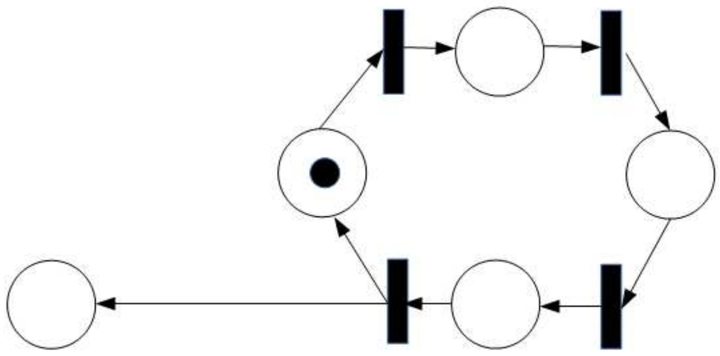


Figure 1.3: "Example of Unlimited Petri Net

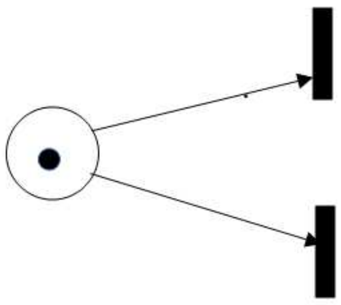


Figure 1.4: Example of Conflict

1.6 Particular structures of Petri nets

- Conflict: Case in which from a Place there are two outgoing arcs towards two transitions, but only one of the two can trigger. In this case the classic decision criterion is to choose the transition in a non-deterministic way (Figure 1.4)
- Concurrency + Synchronization: Case in which two transitions which have no input point in common are both enabled and are followed by output points which are also input points for a common transition. (Figure 1.5)
- Confusion: Case where two competing transitions and both conflicting with other third transitions (Figure 1.6)

1.7 Soundness of a Petri Net

To guarantee the Soundness of a Petri Net it is necessary to respect the following properties:

- **Unboundness** : there is no bound on the number of tokens that a place can hold, in particular a Finite Marked Petri Net is bounded if and only and there is a bound b such that all its places are b-bounded and therefore this property fails for infinite nets and you can't verify the soundness .
- **Deadlocks and Livelocks** : there's a situation when you cannot reach the end (Example in Figure 1.7), so each live marked net must have at at least one transition firing.
- **Improper Termination** : if there's a token in the output place, then there are no tokens in other places (Example in Figure 1.8)

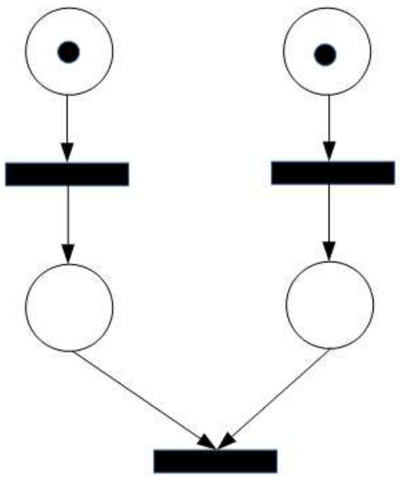


Figure 1.5: Example of Concurrency + Synchronization

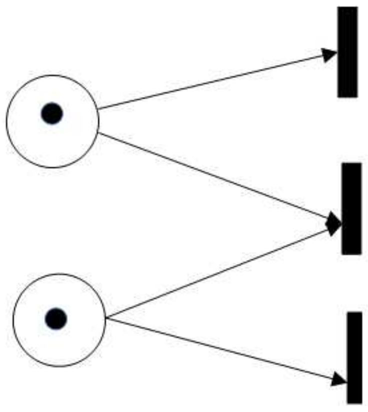


Figure 1.6: "Example of Confusion"

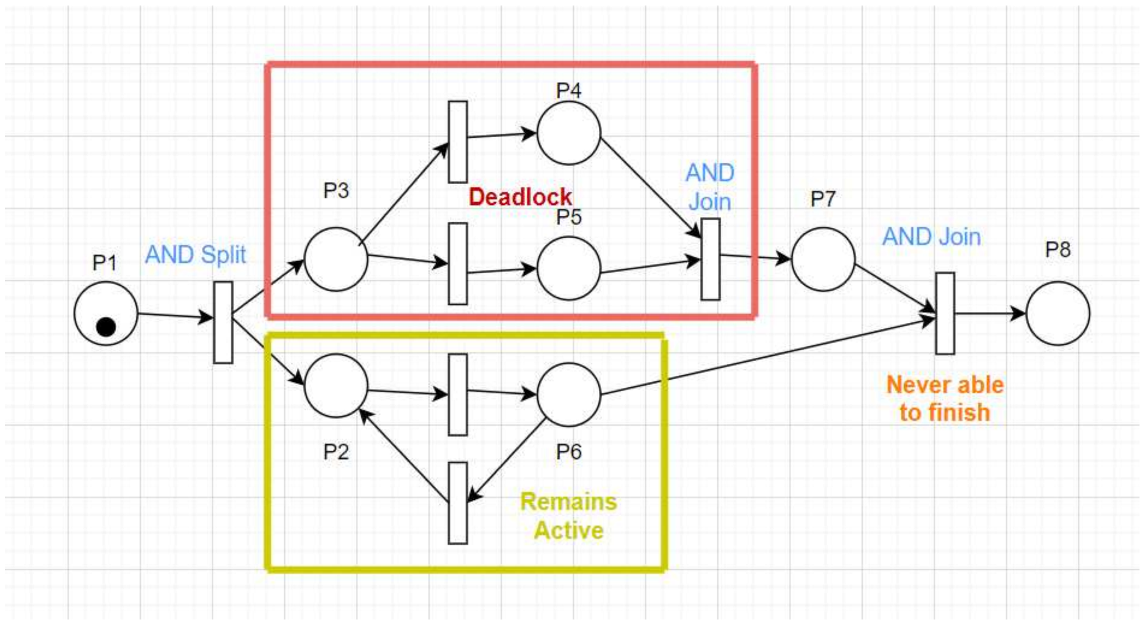


Figure 1.7: This figure shows an example of a deadlock that makes it impossible for the entire network to arrive at the final place.

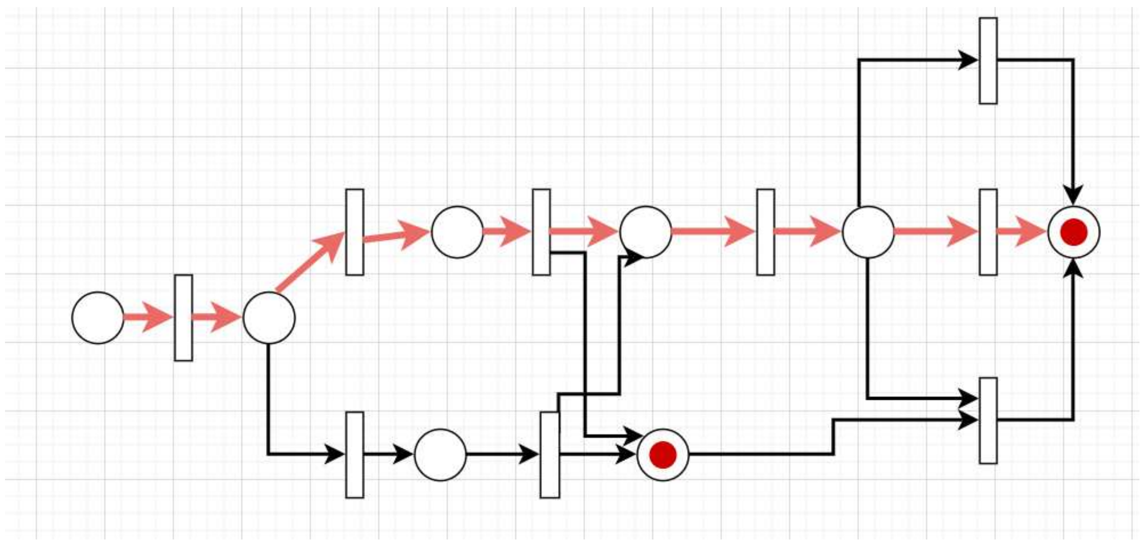


Figure 1.8: This image shows that a token is present in the destination place, but at the same time another token is present in another internal place.

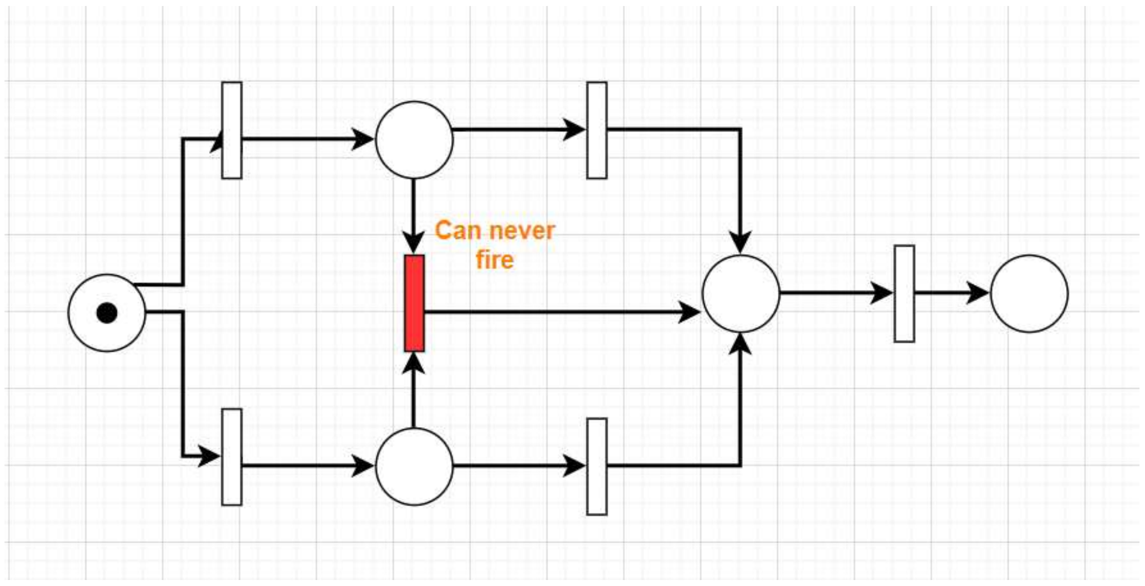


Figure 1.9: The figure shows an example of a dead transition, in fact in this case the transitions a and b cannot fire at the same time and consequently the two places entering the transition f cannot simultaneously have the token that allows it to fire.

- **Dead Transitions** : the presence of transitions that can never be activated (Example in Figure 1.9), so every marking is reachable from initial marking.

1.8 Examples of Petri Nets

In this section we will analyze some interesting examples of Petri nets that allow us to understand how they work, especially when the net is characterized by a graph in which different situations need to be managed.

1.8.1 Producer-Consumer problem

For each transition the local states with one edge entering the transition, i.e. the input places, are replaced by local states with edges leaving the transition, i.e. the exit places. The system therefore consists of three elements:

- The Producer: It must buffer the units that the system must produce
- The Buffer: This element stores the drives it receives from the manufacturer, but only one drive can be present within the buffer at any given time
- The Consumer: Fetch units from the buffer.

Very important to underline the fact that there is no direct communication between producer and consumer, but they must cooperate asynchronously via the buffer. In fact, the producer has the task of filling the buffer, the consumer of emptying it.

The producer can be in the two local states pl1 or pl2. In the local state pl1 can perform the transition p and enter state pl2 consequently moving the token from one state to another. It can then synchronize with the buffer through the shared transition sr. The producer returns to state Pl1 and the buffer place contains a token, indicating that the buffer is full. The shared transition can flush the buffer and the consumer moves to its local state with2. Then the consumer can transition ec2 (i.e. consume the unit) and return to its local state with1. Meanwhile the producer could have transitioned pr independently and could have transitioned sr as soon as the buffer was empty.

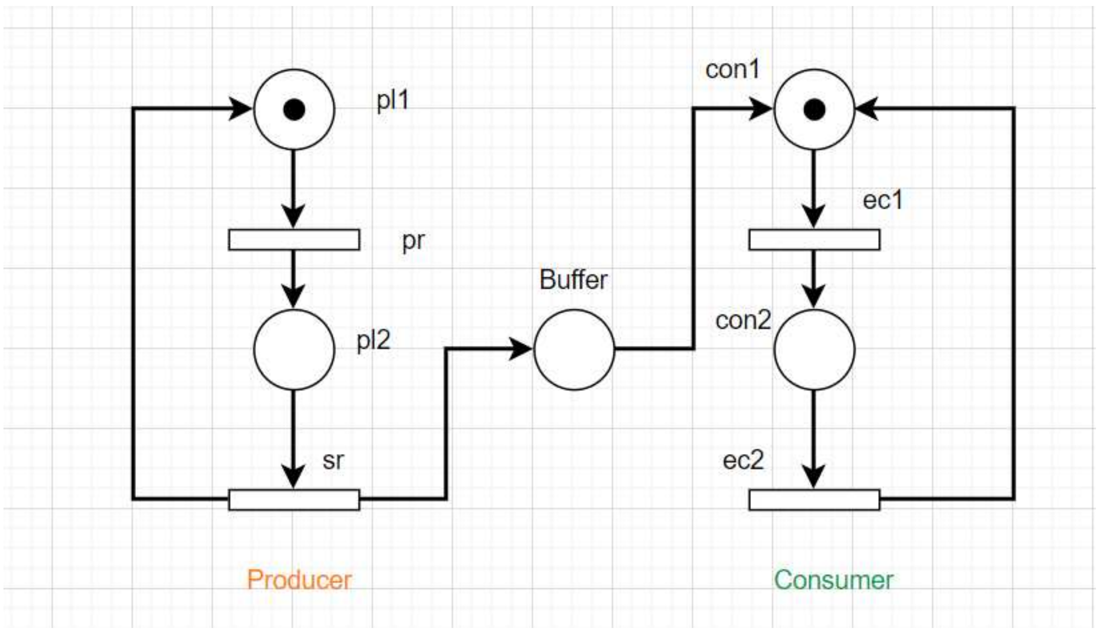


Figure 1.10: Producer - Consumer Example

1.8.2 Mutual Exclusion Problem

The system consists of three components:

- C1 (First component);
- C2 (Second component);
- P (Shared Resource).

At any moment, only one component can use the shared resource and this resource also represents the only point of communication between the two components, which in fact never communicate directly. To schedule access to the resource it is necessary to verify if the shared resource is available by checking if there is a token in place P. In mutual exclusion the shared resource needs checks to prevent some other process from trying to use a resource that is currently being used by another process. The critical section of component 1 is represented by the coml location and that of the component 2 by the com2 location. The non-critical part of the component is represented by the positions ri (the remainder) and wi (expected) with i which can have the value 1 or 2. Thus, component i has local states ci, ri, and wi. I can perform ini, outi and di actions (enter critical section, exit critical section and an action outside the critical section). To enter or exit the critical section it must synchronize with the authorization component, which has local states p (resource is not used), cl (resource is used by component 1) and c2 (resource is used by component 1), and can perform the actions ini and out , i = 1,2. When component i has obtained authority and is done using the resource, it returns authority to position p through transition outi.

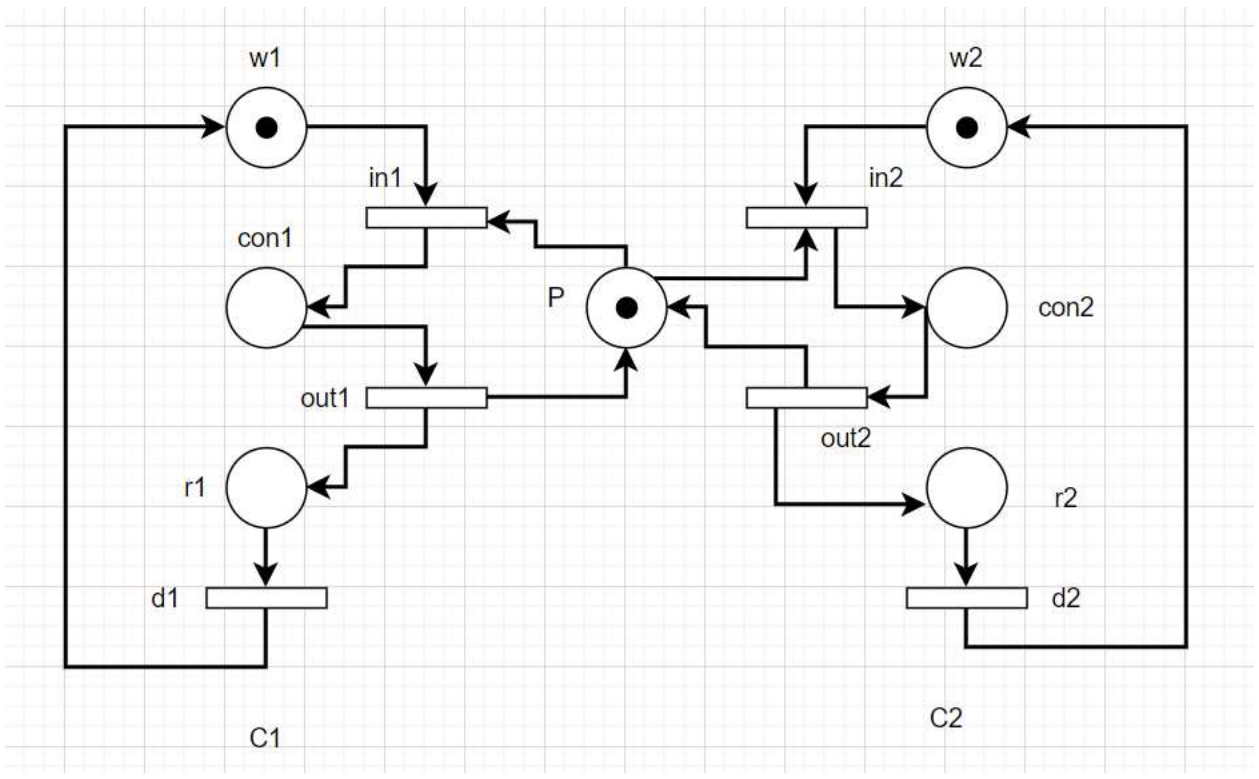


Figure 1.11: Mutual Exclusion Example

CHAPTER 2

Data Petri Net

After having analyzed the general rules of petri nets we move on to the study of its extension with data. The data is represented inside variables and the transitions contain guards where two elements are compared through a relational operator. Guards can be either variable-constant constraints or variable-variable constraints. This means that a transition to be able to trigger will not only need to have at least a certain number of tokens, but also its guard must be true.

2.1 From Petri Nets to Data Petri Nets

After having introduced the standard definition of petri net it is now essential to define the concept of Data Petri Net and all the associated definitions. In DPN the concept of guard becomes fundamental, which is associated with conditions that are conjunctions or disjunctions of atoms of the form variable-operator-constant or variable-operator-variable. Variables take values from Domains given by pairs D = <ΔD,∑D> where ΔD is a set of possible values and ∑D is a set of binary predicates on ΔD .

Domain examples are given by:

- DR = <R, {<,>,= }>
- DZ = <Z, {<,>,=}>
- Dbool = <{true,false}, {=}>
- Dstring = <S, {=}>

Also all these binary predicates are closed under negation, so for each predicate its negation is included (for example {=} includes also {$\neq$}) and it is indicated with the symbol $\oplus$). The Domain is defined for each variable. A set of variables V is expected, where Vw indicates a variable that is written, while Vr indicates a variable that is read. Each variable is associated with a type and the constraints are created using at least one variable.

Given a set of variables V, a constraint is the largest set containing:

- $V_D \oplus \Delta_D \iff v \in (V^r \cup V^w)$ and $\oplus \in \sum D$;
- $V_1 D \oplus V_2 D \iff v1 \in (V^r \cup V^w)$, $v2 \in V^r$ and $\oplus \in \sum D$.

So the conditions predict an operator between two variables or between a variable and a constant that must have the same domain. The variables must undergo an assignment operation indicated by the function $\beta : (V^r \text{ U } V^w) \rightarrow u$. Given a variable assignment β and a guard $\Phi = (v \oplus x)$, where v represents the value of a variable and x a variable or a constant, it is said that Φ evaluates to true when variables are substituted for β, written $\Phi[\beta]$ = true . A State Variable Assignment (SV) is a function α : V -> u which assigns values to each variable $v \in V$ with the restriction $\alpha(VD) \in \Delta D$.

2.2 Formal Definition

Given these premises it is now possible to give a formal definition of Data Petri Net (DPN) which is an extension of the traditional Petri nets with add-ons. While a traditional Petri net is defined by three parameters (P,T,F), a DPN (N) is defined as N = <P,T,F,V,dom,αI, read, write, guard>, where:

- P: finite set of Place;
- T: finite set of Transition;
- F: the set of arcs that connect Place-Transition or vice versa;
- V: finite set of Variables;
- dom: function assigning a domain D to each v $\in$ V.
- αI : initial SV assignment;
- read: T $\longrightarrow 2^V$, set of variables read from a transition;
- write: T $\longrightarrow 2^V$, set of variables written by a transition;
- guard : T $\longrightarrow \Phi$(V) returns a guard associated with the transition.

Therefore a Data Petri Net provides an initial marking (M_I) from which the system begins its process and evolves until it reaches a specific final marking M_f. read(t) and write(t) respectively indicate the set of variables in V^r and V^w that are mentioned in the guard of transition t.

2.2.1 Semantics Execution

To define the transitions from one state to the next, it is essential to define the semantic execution of the system. Given a DPN N, the set of possible states of N is formed by all pairs (M,α) where:

- M $\in$ B(P)[1] is the marking of the Petri Net;
- α is a SV assignment.

As in traditional petri nets, a transition must have the conditions to be able to trigger and in order to reach the final marking it is essential that until the final marking is reached there is always at least one transition that can be triggered, thus updating the marking. At each transition that is triggered, an update of variables and a pair (t, β) where t is a transition and β represents the update of the variables occurs.

2.3 Legal Transition Firing

A DPN N evolves from state (M, α) to state (M', α') with transition firing (t, β) which has guard(t) = $\Phi \iff$

- $\beta(V^r) = \alpha(v)$, if v $\in$ read(t), variable assignment β assigns values to α like a read variable;
- The new SV assignment α' is like α but updated as β and processed as:

$$\alpha'(v)=\begin{cases}\alpha'(v)\ \textbf{if}\ v\notin write(t)\\ \beta(V^W)\ \textbf{otherwise}\end{cases}$$

- β is valid, called $\Phi\beta$ = true and the guard is satisfied when the value is assigned according to β.
- Each input of t contains at least one token (M(p)>0) for some p $\in$ P and (p,t) $\in$ F

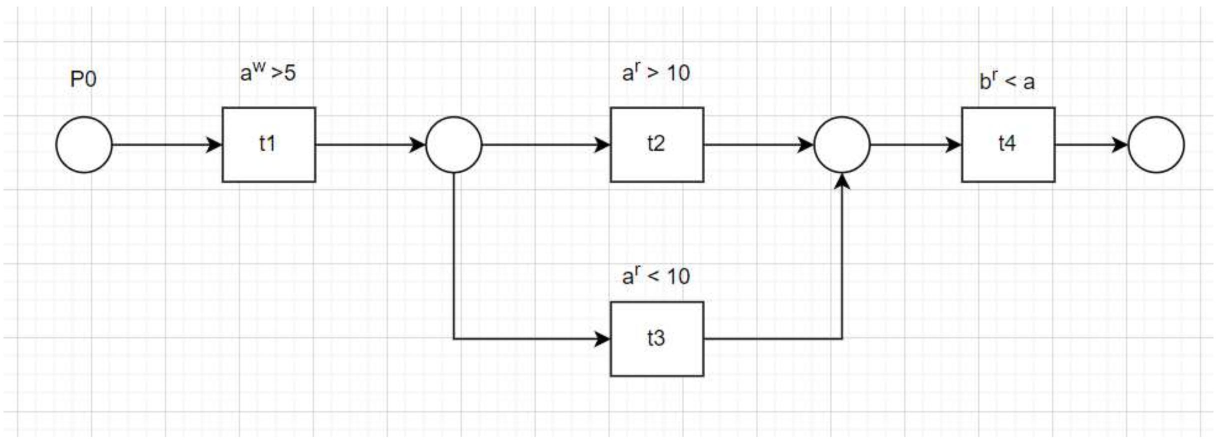


Figure 2.1: An example of DPN

- The new tagging is elaborated according to the semantic executions of the Petri net, denoted by M [t>M′.

A legal transition is defined by writing (M, α) $\rightarrow$ (M′, α') through the pair (t, β). In order to reach a final marking it may be necessary to perform more than one transition and therefore this concept is also extended to a sequence of transitions called traces ((t′,β'), . . . , (t^n, β^n)) and this allows traversing the different reachable markings ((M^0, α^0), . . . (M^n,α^n). A DPN can have many traces and a trace set is defined as the set of all possible traces of a DPN. An example of DPN is shown in Figure 2.1. This DPN contains two variables a and b. Places are represented by circles, while transitions by rectangles. Above the transitions in curly brackets are the guards, i.e. the constraint to be evaluated to trigger or not the transition. The constraint consists of the elements described above, and both variable-constant constraints and variable-variable constraints are represented in the example. In this example, the tokens are not shown, for the simple fact that it is a very standard example where there is a token in the place p_0, while in all the others there are 0. therefore the initial numbering will be [p_0 = 1,p_1 = 0,p_2 = 0,p_3 = 0]. At this moment there are not yet the tools to be able to understand if the DPN respects the data-aware soundness properties, however imagining that all the transitions can be performed, the subsequent markings will become in the order:

- Marking: [p_0 = 1,p_1 = 0,p_2 = 0,p_3 = 0] (Initial Marking);
- Marking: [p_0 = 0,p_1 = 1,p_2 = 0,p_3 = 0];
- Marking: [p_0 = 0,p_1 = 0,p_2 = 1,p_3 = 0];
- Marking: [p_0 = 0,p_1 = 0,p_2 = 0,p_3 = 1] (Final Marking).

2.4 Data-Aware Soundness

In order to evaluate the soundness of a DPN it is essential to verify compliance with the Data-Aware Soundness properties, in order to quantify the achieved markings and the SV assignments for the case variables. Given a DPN N where we write (M,α) $\rightarrow^*$ (M′,α'), if there is a sequence of legal transitions leading from (M,α) to (M′,α'). Given two markings M′ and M″ of N, then we establish that M″ > M′ $\iff$ $\exists$p $\in$ P | M″(p) $\geq$ M′(p) e $\exists$p $\in$ P | M″(p) > M′(p). A DPN with initial marking M_I and final marking M_f is said to be data-aware sound if and only if the following properties are verified:

- $\forall$(M, α) $\in$ Reach_N . $\exists$ α' . (M, α) $\rightarrow^*$ (M_f, α');
- $\forall$(M, α) $\in$ Reach_N . M *ge* M_f $\implies$ (M = M_f);
- $\forall$t $\in$ T. $\exists M_1$,M_2, α1, α2, β , (M1,α1) $\in$ Reach_N e (M_1, α_1) $\rightarrow$ (M_2, α_2) through (t,β).

Where Reach_N represents the set of reachable states starting from the initial markup M_I and the initial variable assignments α_I. The first property implies that to verify soundness a final marking must always be reached. The second property implies that a final marking is always reached such that there are no tokens in the network that are not in the output place. For the verification of the second property, the concept of compatibility between two markings is used, which occurs when a marking always has in all places a number of tokens $\geq$ or $\leq$ of the other. Two markings are comparable only if they are also compatible and if the $\geq$ condition is true, then the analyzed marking must be equivalent to the final one. The third condition instead verifies the absence of dead transitions, i.e. if there is no way that a transition can be triggered. This can happen as in traditional petri nets whereby the number of tokens present in the place can never be $\geq$ of the tokens required by the transition, or if, for example, a set of constraints make a certain constraint unsatisfiable in any case. For example, if a transition occurs with the constraint $a^{(w)} > 5$, it means that the variable a is written and all subsequent reading constraints on this variable must take into account that a cannot be ≤ 5. In fact, if later I encounter a transition with the constraint $a^{(r)} < 2$, this transition can never be triggered, because in any case a can't be less than 2.

2.5 Problems verifying the Soundness of a DPN

Checking the Soundness of a DPN can be problematic since it is undecidable. DPNs cannot be directly algorithmically analyzed due to the presence of data and their respective updates, in fact they involve a state space that can have infinite states and this occurs in particular when the case data are updated using arithmetic operators.

If the DPN provided only variable-constant cases it would be possible to analyze the Soundness through the transformation of the DPN into a CPN (Colored Petri Net), but this work is carried out on more general forms which also include variable-variable cases. For this reason the technique to be adopted is different and instead of a CPN the Constraint Graph is used which is an abstraction of the original process, at least from the point of view of Soundness. Being in finite states, the Constraint Graph allows to analyze the three properties of Soundness without running into the problem of infinite states. For this reason, the goal of this work is to create software that, given a DPN, is able to transform it into a Constraint Graph and then verify its three Soundness properties.

2.6 Constraint Graph of a DPN

Given a DPN, M is defined as the set of markings of N and M_I the initial marking. C_v denotes the set of possible constraints in V. The Gragh Constraint CG_N is then defined as a tuple where (S,S_0,A) are:

- $S \subseteq M \times 2C_v$ is a set that includes all the nodes of the Constraint Graph;
- $S_0 = (M_I,C_0) \in S$ is the initial node where the initial constraints given by set C_0 are given by $C_0 = \bigcup v \in V \{v = \alpha_I(v)\}$;
- $A \subset S \times (T \cup \tau_T) \times S$, is the arc set which is defined by mutual induction as described below.

So S represents the node set of the Constraint Graph, where each node is composed of a marking of N and a constraint set representing its constraints. S0 instead is an element of S which represents the initial node of the CG, in which it has an initial marking and a constraint set in which it inserts the constraints foreseen by the initial variable assignments. Finally A represents the set of arcs which in formal terms is defined with:

A transition $((M,C),t,(M',C'))$ is in A $\iff$:

- $M \mid t> M'$;
- $C' = C \oplus \mathrm{guard}(t)$ is satisfiable.

A transition $((M,C), \tau_T, (M',C'))$ è in A $\iff$

- $\mathrm{write}(t) = \emptyset$;

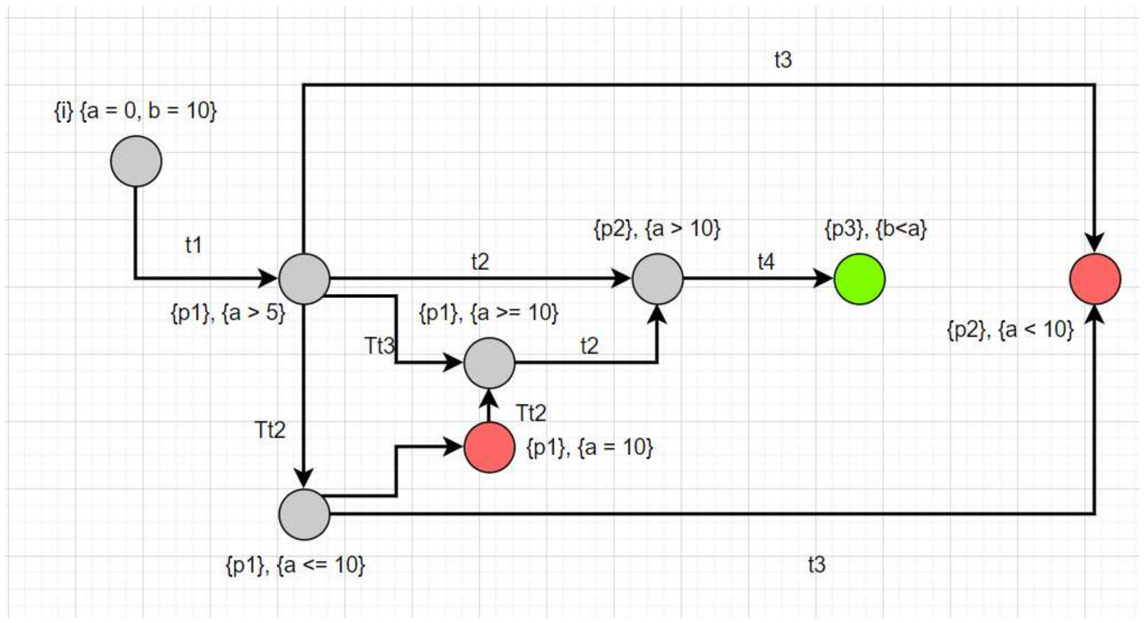


Figure 2.2: This example shows a Constraint Graph example, in this case you can see that it is a graph with only one node type.

- ∃M′ s.t. M[t>M′;
- C″ = C ⊕ ¬guard(t) is satisfiable.

An example of Constraint Graph of the DPN in figure 2.1 is shown in figure 2.2

2.7 Types of Transition

There can be two types of transitions in a Constraint Graph. The first type of transition provides that given a node (M,C), a new node (M′,C′) can be reached via a transition t ∈ T of the DPN $\iff$ A((M,C) → (M′,C′)) via the transition t. The conditions are as follows:

- M′ is the marking according to the transition t from M due to the underlying petri net;
- The Constraint Set C′ obtained by adding the guard of t to the current set C of the first algorithm that will be analyzed later is satisfiable. In this case after the triggering of a transition its guard must be true and compatible with the new SV assignment.

The first type of transition is also defined within the DPN, however there is a second type of transition that must be explicitly managed in the Constraint Graph: Silent Transitions. A guard always expects a true or false branching, so the CG has to handle both the true case (first type of transition) and the false case (second type of transition or silent transition). Given a node (M,C), a new node (M′, C′) can also be reached through a series of Silent Transitions $\tau_T \iff$ A((M,C), tau_T , (M′,C′)) denoted as (M,C) → (M′,C′) through the transition τ_T. In this case it requires:

- Transition t is not writing a variable;
- t can be triggered given the M marking of the original DPN;
- The constraint Set C″ obtained by adding the negation of the guard is satisfiable.

This case simulates the case-based reasoning that is required to consider every possible SV assignment that is produced in the original DPN after a transition is triggered. Intuitively an edge labeled with τ_T is intended to model all SV assignments consistent with the current Constraint set in which the guard of t is not true. Constraint solvers are also used in the work, in particular the two technologies of Constraint Solver and SMT Solver.

2.8 The obs-Simulation Relation

The obs-simulation relation connects the reachability graph of a DPN <V,E> and its Constraint Graph <S,S0,A>. We say that a relation R $\subseteq$ S x V

- is a *obs-simulation* of the DPN from the Constraint Graph if and only if (s,(M,α)) $\in$ R implies that for some (M,α) $\overset{t,\beta}{\rightarrow}$ (M′,α′) then there exists a single-step trace fragment σ with obs(σ) = t and s $\overset{\sigma}{\rightarrow}$ s′ such that (s′,(M′,α′)) $\in$ R.
- is a *obs-simulation* of the Constraint Graph from the DPN if and only if for each single-pass trace fragment σ e s $\overset{\sigma}{\rightarrow}$ s′ there exists (M, α) $\overset{t,\beta}{\rightarrow}$ (M′,α′) with obs(σ) = t such that (s′,(M′, $alpha'$)) $\in$ R.

A node of the Constraint Graph obs-simulates a state of the DPN if there is an obs-simulation relationship R of the DPN by the Constraint Graph, such that they are included in the relationship. We also say that the Constraint Graph obs-simulates the DPN if S0 obs-simulates (MI, αI). Same thing goes for the opposite. As proved in [1] The important lemma that we can derive from this relation is that the Constraint Graph obs-simulates the DPN. The result of this lemma implies that the constraint graph can reproduce any possible execution of the reachability graph, and therefore of the DPN. This gives us a very formal and precise characterization of the ability of the constraint graph, which is finite state, to take into account the possible infinite executions of the DPN. This does not imply that any property that is true in the DPN is also true in the Constraint Graph or vice versa. However, the three Data-Aware Soundness properties return the same result in the DPN and the Constraint Graph. From the previous result we derive the theorem that if a Constraint Graph is data-aware sound, then the corresponding DPN is also data-aware sound. The difference is that the Constraint Graph is finite-state and therefore it is always possible to verify the data-aware soundness properties. So we can say that the Constraint Graph is a correct abstraction of the DPN regarding the verification of the Soundness. From this we then arrive at the goal of the project: to automatically implement the DPN Constraint Graph and verify the soundness properties to then tell the user whether the DPN is data-aware sound or not.

2.9 Algorithms to Implement

In this section we will analyze the algorithms to be implemented within the project and their pseudocodes to be used as a starting point.

2.9.1 Algorithm 1: Construction of the new Constraint Set

The operation of adding a constraint to the Constraint Set is represented in the form C′ = C $\oplus$ c and the algorithm to obtain it is the following:

```
    Algorithm 1: Procedure for computing C′ = C $\oplus$ c
1.    if c = ($v^r \odot x$) then
2.       C′ $\leftarrow$ C′ $\cup$ {($v \odot x$)}
3.    if c = ($v^w \odot x$) then
4.       C′ $\leftarrow$ C′ $\cup$ {($v^w \odot x$}
5.       C′ $\leftarrow$ saturate(C′)
6.      foreach c = ($v^r \odot y$) or c = ($y \odot v^r$) in C′ do
7.         C′ $\leftarrow$ C′ $\setminus$ {c}
7.      foreach ($v^w \odot z$) in C′ do
8.         C′ $\leftarrow$ C′ $\setminus$ {$v^w \odot z$}
9.         if $z \neq v^r$ then
10.           C′ $\leftarrow$ C′ $\cup$ {($v^r \odot z$)}
11.    return saturate(C′)
```

The algorithm receives as input a constraint set C and the constraint of the guard c and the new constraint set is obtained by adding the constraint c to set C. We consider *x,y,z* as read constants or variables ($\in V^r$). Theoretically, it also requires a saturation operation which, given a set of input constraints, returns the set itself with the addition of all those constraints which are logical implications of the constraint set. This is done using the variables and constants present in the constraint set and combining them with all possible relation operators. On a practical level, in the implementation with SMT Solver the second saturation operation is removed and replaced with a more efficient mechanism that allows to obtain the same result.

Bearing in mind that the symbol $\odot$ includes any relation operator ($=,\neq,>,<,\geq,\leq$), line 1 verifies that the constraint has the first variable in reading and in that case it simply adds the constraint c to the constraint set C. The more complicated case is the one on line 3 where it is verified that the constraint has the variable v in writing. In this case the new constraint with the variable being written is added to the new constraint set and is also extended to all the constraints applied through saturation. When a write constraint is inserted, the variable is initialized again to a new value and therefore basically performs a reset operation. This translates into a loop that iterates through the entire constraint set and eliminates all the constraints that contain that variable being read. The next loop also releases the constraints with the write variable but then re-insert them with the read variable, except that the second operand is not the variable itself. Whether the variable v is read or written, the result is returned after applying the saturation function.

2.9.2 Algorithm 2: Construction of the Constraint Graph

This is the algorithm to be implemented. As input, a DPN N, an Initial Marking M_I is received, while as regards the output, two results are returned:

- A Boolean value indicating whether the Constraint Graph is data-aware sound or not.
- A set S containing all nodes of the Constraint Graph
- a Set A containing all edges of the Constraint Graph

In the previous chapters, the constituent elements of the Constraint Graph have been presented, from nodes to arcs, up to the markings. Now the goal of this algorithm is to build the Constraint Graph and verify its Soundness properties. All this by exploiting the algorithm for building a new constraint set which was explained in the previous chapter and which is represented in the pseudocode with the symbol $\oplus$. The algorithm starts by creating an initial node s_0 which among its elements takes the initial marking M_I and the Constraint Set made up of all the initial State Assignments. The first node is thus added to the set S and consequently each Constraint Graph will be composed of at least one node. The set of Arcs, on the other hand, is initialized to and will be updated with the new arcs inserted. The latter set might not even contain elements at the end of the algorithm, in fact if there is a transition with a guard that always has false as a result, no new nodes are created and consequently no new arcs are created. In fact, the transition would always be false, while its Silent Transition would always be true, but this implies that the guard of the Silent Transition is a logical implication of the initial node's Constraint Set, consequently it would be equivalent and does not create a copy of the node. The last part of the initial state concerns the initialization of the set L, which contains a series of nodes of the constraint graph. In the initial case this set will contain only the element s_0 . Subsequently the first while loop begins which at each iteration will extract a node from L and will continue until the set L is completely emptied.

```
    Algorithm 2: Data-Aware Soundness-checking procedure

Input: A DPN N = <P,T,F,V,dom,$\alpha_I$, read,write, guard> and an initial marking
$M_I$ for N.
Output : true if N is data-aware sound, false otherwise

1.     $C_0 \leftarrow \bigcup_{v \in V} \{v = \alpha_I(v)\}$
2.     $s_0 \leftarrow <M_I, C_0>$
3.     $S \leftarrow \{s_0\}$
4.     $A \leftarrow \emptyset$
5.     $L \leftarrow \{s_0\}$
```

```
    while L $\neq \emptyset$ do
      (M,C) $\leftarrow$ pick(L)
      L $\leftarrow$ L $\setminus$ {(M,C}
      foreach t $\in$ T s.t M $\xrightarrow{t}$ M' do
        C' $\leftarrow$ C $\oplus$ guard(t)
        C'' $\leftarrow$ C
        if write(t) = $\emptyset$ then
          C'' $\leftarrow$ C'' $\oplus$ $\neg$guard(t)
        if satisfiable(C') then
          if $\exists$ ($\bar{M}$,$\bar{C}$) $\in$ S s.t M' > $\bar{M}$ $\wedge$ C' = $\bar{C}$ then
            return false
          S $\leftarrow$ S U {(M',C')}
          A $\leftarrow$ A U {<(M,C),t,(M',C')>}
          L $\leftarrow$ L U {(M',C')}
        if satisfiable(C'') $\wedge$ C $\neq$ C'' then
          S $\leftarrow$ S U {(M,C'')}
          A $\leftarrow$ A U {<(M,C), $\tau_t$, (M,C'')>}
          L $\leftarrow$ L U {(M,C'')}
    return analyzeConstraintGraph(<S,$s_0$,A>)
```

2.10 Solver and Constraint Programming

Constraint programming refers to a programming paradigm where the relationships between variables are represented in the form of constraints. A Constraint Satisfaction Problem is defined by:

- A set of variables V = {X_1,X_2, ... , X_n}
- A Domain for each variable D = {D_1,D_2, ... , D_n}
- A set of constraints on these variables

The constraint is defined as a relationship between variables that defines a subset of the Cartesian product of the domains D_1xD_2x...xD_n Finding the solution to a constraint satisfaction problem means finding an assignment of values to variables consistent with the constraints. A constraint satisfaction problem can be represented with the Constraint Graph and in fact this representation has been used to verify the Soundness properties of a DPN. Some fundamental elements of constraint programming and which will be present implicitly or explicitly are the following:

- **C**onstraint Store: A collection of the constraints of the problem which will also be the basis of the implementation of all the main algorithms for the construction of the Constraint Graph;
- **S**atisfiability: a collection of constraints is satisfiable (consistent) if there is (at least) one solution. In the program there will be a specific satisfiable() function which, given a Constraint Store as input, will tell if the set of constraints is satisfiable or not. This function uses Solvers which, as we will see later, can be of various types.
- **C**onstraint Propagation: Inference of new constraints starting from the given ones. This part in the implementation of the program will be represented implicitly: in fact initially there was the saturation function which, given a set of constraints, extends it with all the constraints that are a logical consequence of that set. However the saturation function implemented with a naive algorithm has created many problems of inefficiency and so it was decided to use a mechanism through SMT Solver whereby given two sets of different constraints it is verified whether one is a logical implication of the other and vice versa with the end result of greatly improving the efficiency of the program.

2.10.1 Constraint Solver

The Constraint Solver is a solver of constraints, defined by Constraint Programming. Constraint Programming is a combinatorial problem solving paradigm that relies on many techniques. Users declare constraints on feasible solutions for a set of decision variables. In addition to constraints, users specify the method for solving these constraints, and the latter can be based on methods such as historical backtracking and constraint propagation. There are many constraint solvers, Choco Solver is used in this work. Choco is an open source java library dedicated to constraint programming where the user defines his problem declaratively by specifying the set of constraints that must be satisfied in each solution. The problem is then solved with constraint filtering algorithms with a search mechanism.

2.10.2 SMT Solver

SMT is the acronym of Satisfiability Modulo Theories. It involves determining whether a mathematical formula is satisfiable, in particular by generalizing the Boolean satisfiability problem (SAT) to more complex formulas involving real integers or more complex data structures. Expressions are interpreted within a certain formal theory in first-order logic with equality. SMT solvers such as the Z3 have been used as building blocks of many applications in computer science, including automated theorem proving, program analysis, and software testing. The SMT problem is typically NP-Hard.

Part II

Building the Constraint Graph

CHAPTER 3

Building a Data Petri Net

This second part of the thesis describes in detail how the program has been implemented. The programming language used is Java with the object paradigm. For the realization of the software it is necessary to define the elements for the construction of the Data Petri Net. The various constraints, the Places, the Transitions, the arcs and all the constituent elements of a Data Petri Net will then be defined. Once the DPN has been created, it will be translated into the Constraint Graph.

3.1 Place

The first constituent element of a DPN is the Place, represented by a special class. As we know from the previous chapters, a Petri net is a bipartite graph, i.e. defined by two types of nodes, one of these two types is the Place. the Place class inside contains a name that is represented as a string and a certain number of tokens. Graphically the Place is represented as an unfilled circle, while its tokens represented by small circles filled with black inside. In the practical implementation of Place, a further boolean attribute has been added: the *finalPlace* attribute indicates whether a given place is part of the final marking or not. A Place is final if it represents one of the places that the system must reach to complete its process.

3.2 Transition

The second element of the bipartite graph of a petri net is given by the transition, which requires a certain number of tokens from incoming Places to be triggered and which generates a certain number of tokens from outgoing Places. To represent this information, the following attributes were created:

- A list of edges entering the transition;
- A list of edges leaving the transition;
- A constraint of the specific *MyConstraint* class which will represent the guard associated with the transition;
- a *type* string that has the value *W* if the first variable is writing or *R* if the first variable is reading;
- A *cost* variable that defines how many tokens the transition needs to fire;
- A *addTokens* variable that defines how many tokens should be generated in outgoing places.

A transition is represented graphically by a rectangle and labeled with the guard that must be true in order to be able to shoot. The Transition class contains a method that checks if the transition can be triggered. Both Places and Transitions are subclasses of an Object class, in this way it is possible to create data structures of Objects and enclose both Place and Transition.

3.3 Arc

After implementing the two types of nodes that can exist in a DPN, the edges must be defined in order to connect them. In a Petri net an arc must necessarily connect two nodes of different types, therefore it is not possible to have a Place connected to another Place and the same thing is true between two transitions. So the attributes that the Arc class has are:

- The Place associated with the arc;
- The Transition associated with the arc;
- The Direction of the arc which can be Place-Transition, therefore with source the Place and destination the Transition or Transition-Place with source the Transition and destination the Place.

3.4 Constraints

Another fundamental element of a Data Petri Net concerns the constraints. For the constraints, a special class has been created containing three attributes:

- The first operand which must necessarily be a variable and which is of type String.
- The relational operator that binds the first item to the second. Also of type String and can have six values: $\{=,\neq,>,<,\geq,\leq\}$
- An integer flag that takes the value 0 if the first operand is for reading (stored in the constant READV) and the value 1 if the first operand is for writing (stored in the constant WRITEV);

Within this class there is also a negate() method that allows you to obtain the negation of that constraint. The possible combinations are the following:

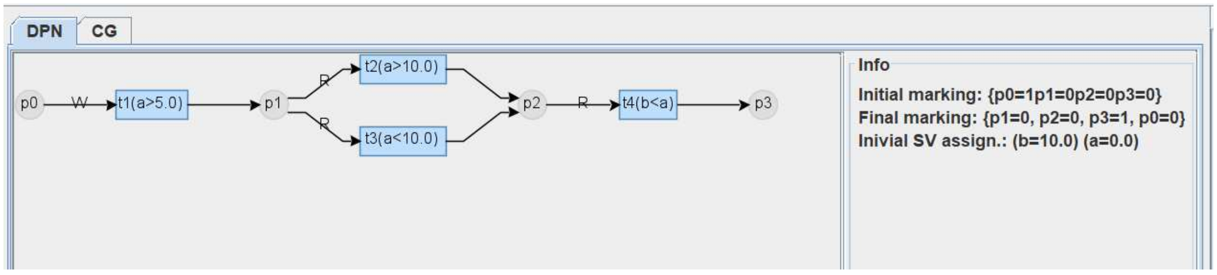
- The operator $\neq$ becomes =;
- The operator = becomes $\neq$;
- The > operator becomes $\leq$;
- The operator < becomes $\geq$;
- The operator $\leq$ becomes >;
- The operator $\geq$ becomes <.

So far we have only handled two of the three constituent elements of a constraint: the first operand and the operator. As regards the management of the second operand, the distinction of a variable-constant or variable-variable constraint must be made. To do this, two subclasses of the constraint class are defined.

3.4.1 Variable-Constant Constraints

If the second operand is a constant, a special subclass is created with an attribute of type double. The value of this attribute is returned by the *getConstant()* method. Therefore, to define a constraint of this type, an instance of this subclass will be created directly by specifying the name of the variable, the operator and the constant. The operator and first variable will be defined in the superclass, while the real value in the subclass. For example to define a$^{(w)}$ > 5 we execute:

```
MyConstraintOneV ov = new MyConstraintOneV("a",5,">");
ov.setRW(MyConstraint.WRITEV);
```



Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Figure 3.1: The DPN generated by the software and corresponding to the DPN in Figure 3.1

3.4.2 Variable-Variable Constraints

If the second operand is a variable, a special subclass is created with an attribute of type String. The value of this attribute is returned by the *getSecondString()* method. Therefore, to define a constraint of this type, an instance of this subclass will be created directly by specifying the name of the two variables and the operator. The operator and the first variable will be defined in the superclass, while the second variable in the subclass. Note that the second variable is always read. For example, to define $a^{(w)} > b(^{r})$ you can use the following code:

```
MyConstraintTwoV tv = new MyConstraintTwoV("a","b",">");
ov.setRW(MyConstraint.WRITEV);
```

3.5 Graphic representation of the DPN

Through the Java programming language it was possible to represent a DPN in graphical form. in particular the *swing* and *awt* libraries for the part relating to the graphics of the program and the *mxgraph* library for the part relating to the construction of the graph.

As we said in previous chapters, the program has the requirement not to have variables with undefined value, so all the variables present in the DPN must be initialized with some constraints. Taking the generated DPN as an example in Figure 2.1. In this specific case it is assumed that the variables are initialized with a = 0 and b = 10 and the output generated by the program is shown in Figure 3.1 . The program window shows a tabbed menu where it is possible to view the DPN or the Constraint Graph abbreviated as CG. Places are represented by circles, while transitions by rectangles. Arcs, on the other hand, are represented by arrows connecting place to transition or vice versa. In the right pane, the initial markup is shown specifying the initial tokens for each place, the final markup and the initial variable assignments.

CHAPTER 4

The elements of a Constraint Graph

The implementation of the DPN was described in the previous chapter, however the tools that allow transforming the DPN into a Constraint Graph and thus verifying the data-aware soundness properties have not yet been described. In this and in the next chapters we deal with these points and in particular in this chapter we define the structure of a Constraint Graph by analyzing all its constituent elements. Once these elements have been described, their graphic display is shown within the program.

4.1 Marking

One of the constituent elements of the node of a Constraint Graph is the marking, already mentioned in the chapter on DPNs. Marking has also been defined within the program by implementing the *Marking* class.
The marking is given by a vector that associates a certain number of tokens to each Place.
There are two attributes in this class:

- A HashMap of type <Place, Integer> which is the data structure that associates a certain number of tokens to each Place;
- A set of Places which contains Places with tokens greater than 0 and which is used for various control operations and for graphical display of the nodes of the Constraint Graph.

The Constructor just initializes the attributes as empty. While to add an element within the HashMap the addElement method is used. For the rest, the various Get and Set methods are inserted.

4.2 Node

Every graph is made up of nodes and so this is also true for the Constraint Graph. At the implementation level, the *CGNode* class has been created.
The CGNode class has 3 main attributes:

- A constraint set V containing the constraints associated with the node;
- A tag M containing the tag of the node;
- A String name containing the name of the node

It foresees two constructors who, given a set of constraints V and a marking M, create the new node. The difference between the two constructors is that you can specify the guard or not.
Otherwise all the Get and Set methods of the attributes are present.

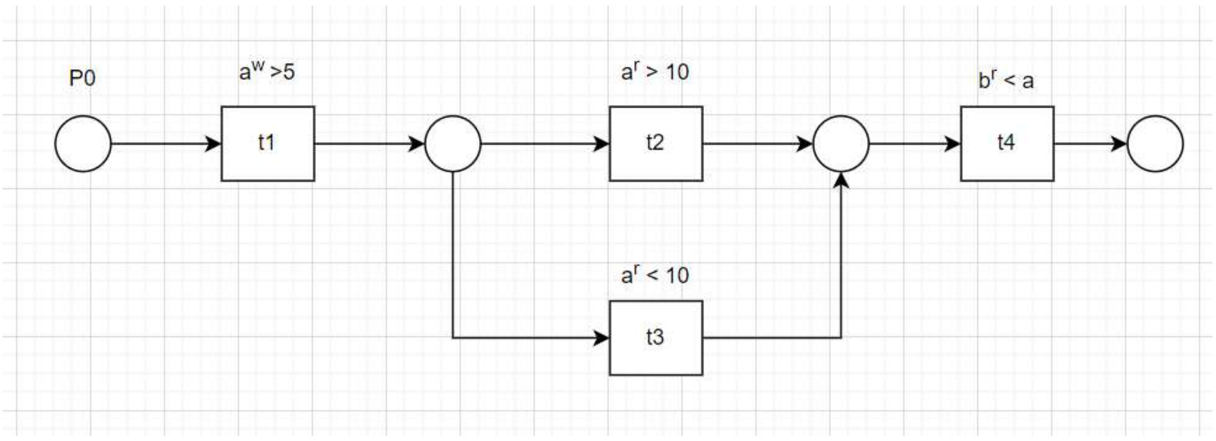


Figure 4.1: An example of DPN

4.3 Arcs

A graph consists of a set of nodes that are interconnected by edges and the same is true for the Constraint Graph. Arcs have been implemented within the *CGArc* class.

The attributes of the class are as follows:

- Source : represents the source CGNode;
- Transition t: represents the transition between the two nodes;
- Destination: represents the destination CGNode.

The constructor method simply takes values for each of the attributes as parameters and creates the object. For the rest, the various Get and Set methods are inserted.

4.4 Graphic representation of the Constraint Graph

In the last chapter, the following DPN was illustrated, bearing in mind that the variables are initialized with a = 0 and b = 10. and results in the following Constraint Graph shown in Figure 4.2.

The Constraint Graph also represents the Silent Transitions, i.e. the negated guards of the transitions. In the figure, however, it can be seen that the silent transition of $b<a$, i.e. $b \geq a$, is not represented. To understand why, we need to keep in mind the following cases on transitions:

- Case 1: The guard of the transition is always true, in this case a single node is created in the constraint graph which represents the case in which the guard is *true*;
- Case 2: The guard of the transition is always false, in this case no new node is created in the Constraint Graph, this because the *true* case can never occur and at the same time the guard constraint in the *case false* is already an implication of the current node, so if a new node were created, it would simply be equivalent to the one that generated it;
- Case 3: The guard of the transition can be either true or false, in this case two new nodes are created, one for the transition and the other for the silent transition.

Only a subset of all node constraints are represented in the curly braces for ease of visualization and this is also the reason why inside CGNode I created the *guard* attribute. For example, all the nodes of this Constraint Graph have within them the constraint b = 10 which is not represented within the curly braces with the sole exception of the initial node. One difference between the Constraint Graph and the DPN is that while the DPN is a bipartite graph for which there are two types of nodes, in the Constraint Graph there is only one type of node.

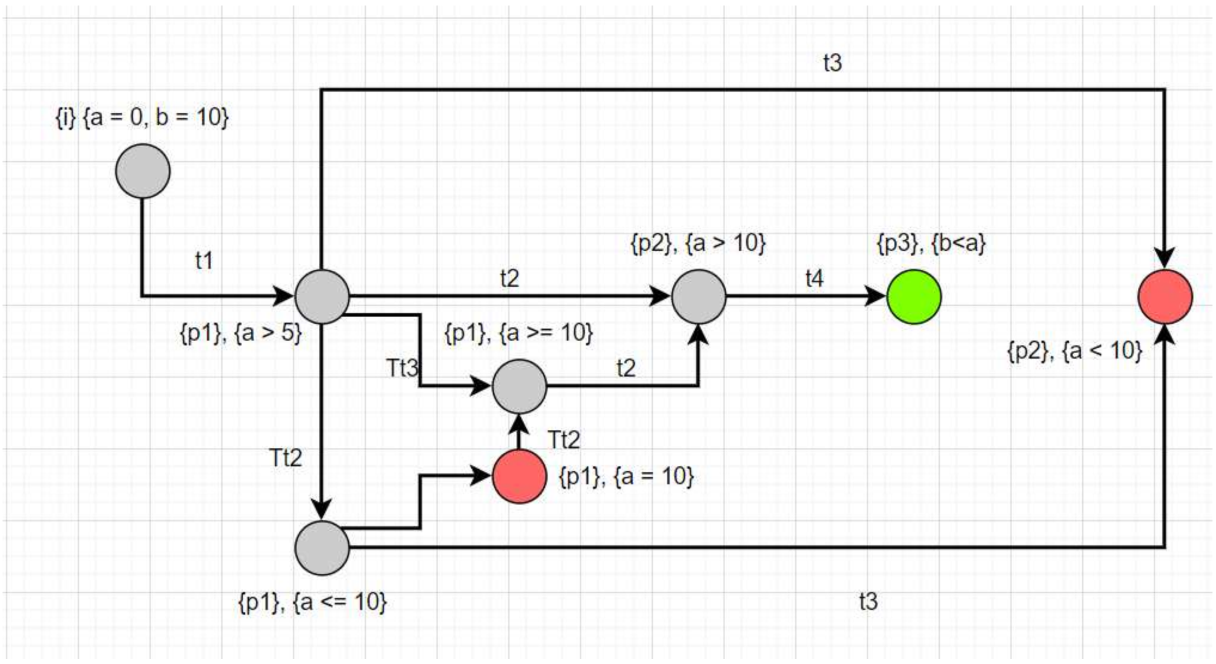
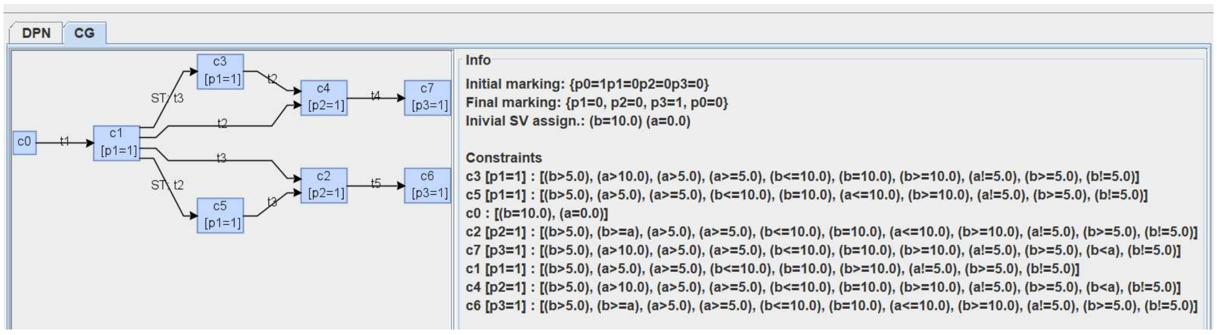


Figure 4.2: The Constraint Graph corresponding to the DPN in Figure 4.1



```
Info
Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Constraints
c3 [p1=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c5 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c0 : [(b=10.0), (a=0.0)]
c2 [p2=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c7 [p3=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c1 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c4 [p2=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c6 [p3=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
```

Figure 4.3: The Constraint Graph generated by the software corresponding to the Constraint Graph in Figure 4.2

This illustration also allows us to know if the DPN respects the data-aware soundness properties and as can be seen from the two crossed out nodes the answer is no. In fact there are two nodes that cannot reach a final state; one where the marking is {P1} and $a = 10$ and one where the marking is {P2} and $a < 10$ since being $b = 10$, the transition with guard $b < a$ can never occur. This DPN therefore violates the data-aware soundness property 1 which dictates reaching a node with final marking from any node of the Constraint Graph. However, in the next chapters we will describe in detail how to build the Constraint Graph and how to verify algorithmically that the three data-aware soundness properties are respected.

In the implementation of the program the java classes used for the graphical display are the same ones used for the DPN, with the difference that in this case a single type of node has been used. For completeness, all the various nodes of the Constraint Graph have been inserted in the panel on the right with the set of all their constraints attached, while inside the rectangle only the guard of the transition that led to the creation of the node is represented.

CHAPTER 5

SMT Solver

In the Software SMT Solver is used to perform operations on constraints, in particular it will be used for the operations of saturation, satisfiability and equivalence checking of two nodes. Microsoft's Z3 is used as SMT Solver and the interaction takes place through pipes with the shell. This Chapter will describe what an SMT Solver is, what Z3 is and how the software can interact with it.

5.1 Introduction to SMT Solvers

SMT is the acronym of **satisfiability modulo theories** and has the objective of determining whether a Boolean formula is satisfiable. This type of technology generalizes the Boolean satisfiability problem (SAT) to more complex formulas involving data structures and primitive data such as real numbers. SMT Solvers are tools that aim to solve the SMT problem for specific logical theories. In the implemented program Z3 was used as SMT Solver which had numerous important applications, especially in the following areas:

- Program analysis;
- Software Test;
- Automated Theorem Proving;
- Checking programs

5.2 Introduction to Z3

Z3 is an efficient SMT Solver for symbolic logic developed by Microsoft as Leonardo de Moura et al. shown in [4]. Each component of this Solver implements specialized algorithms which together form the software as a whole. Z3 has numerous applications in software verification, theorem proving, and program analysis. Z3 supports arithmetic, fixed-size bit vectors, extensional arrays, data types, uninterpreted functions, and quantifiers. Z3 is implemented in C++ language and its architecture is composed of the following elements:

- **Simplifier**: the input formulas are processed through an efficient, but incomplete simplification. The simplifier uses the standard rules of algebraic reduction and performs contextual simplification as it identifies definitions within a context and reduces the remaining formulas using the definition;
- **Compiler** : The simplified abstract syntax tree of the formula is converted into a different data structure including a set of clauses.

- **Congruence Closure Core**: receives truth assignments to atoms from the SAT Solver. Atoms can be specific atomic equations and formulas such as arithmetic inequalities. This propagation occurs through a data structure called E-graph where its nodes can indicate one or more theoretical solvers. When two nodes join, the set of solver references are joined, and their union is propagated as an equality to the theory solvers at the intersection of the two solver reference sets.
- **SAT Solver**: This solver integrates standard search pruning methods such as two-check literals for efficient Boolean constraint propagation using conflict clauses, a phase cache to drive case splits, and performs backtracking not chronological.
- **Theory Combination**: The traditional methods for combining theoretical solvers rely on the solvers' ability to obtain all the implied equalities or processing steps that introduce added literals into the search space.
- **Deleting Clauses**: Instantiating the quantifier has the side effect of producing new clauses containing new atoms in the search space. Z3 garbage collects clauses with its own atoms and terms that are not useful in concluding the ramifications. Conflict clauses, and the literals used in them, are not deleted, however, so instantiations of quantifiers that were useful in producing conflicts are kept as a side effect.
- **Theory Solvers**: Z3 uses a linear arithmetic solver. Array theory uses lazy instantiation of the array axioms.
- **Quantifier instantiation using E-matching**: Z3 uses new algorithms that identify matches on E-graphs incrementally and efficiently.
- **Model Generation:** Z3 has the ability to produce models as part of the output. Models assign values to constants in the input and generate partial function graphs for function predicates and symbols.
- **Relevancy Propagation**: Solvers based on DPLL(T) assign a potentially boolean value to all atoms appearing in a target. In practice, many of these atoms are of no interest, and Z3 ignores these atoms for expensive theories, such as bit vectors, and inference rules, such as instantiating quantifiers.

The set of all these components form the architecture shown in Figure 5.1:

5.3 Z3 operations for implementation

Z3 is a very complex solver that allows you to perform a large number of operations and going into it in detail is not the goal of this thesis. Only a few operations need to be used in this document. First you need to specify the logic to be used in our software, among the main types that Z3 makes available you can find:

- **QF_LRA** : quantifier-free linear real arithmetic
- **QF_LIA** : quantifier-free linear integer arithmetic
- **QF_RDL** : quantifier-free real difference logic
- **QF_IDL** : quantifier-free integer difference logic

QF_RDL logic is used in the implementation since it is powerful enough to represent the variable-to-constant and the variabile-to-variable constraints needed to create the constraint graph.
First, it is essential to know how to declare variables of both real and boolean types

```
(declare fun RealVar() Real)
(declare-const BoolVar Bool)
```

The second step is to define constraints via *assert*.

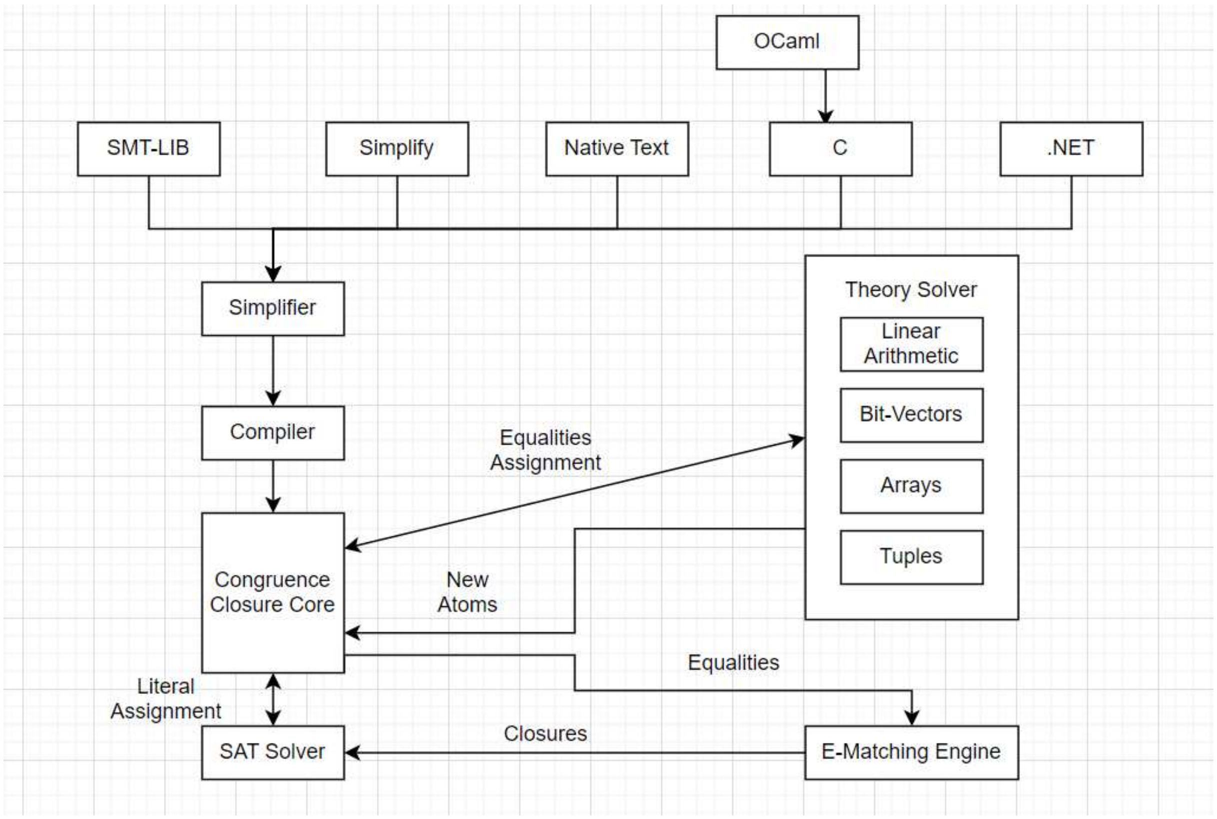


Figure 5.1: The Architecture of Z3

```
(assert (op (- var1 var2) value))
```

In this case, instead of op, one of the six relational operators must be inserted and the operation written in this way verifies the following equation:

var1 - var2 op value

Once all the constraints represented as differences of values have been specified, it is possible to check whether the set of constraints is satisfiable or not through the following command:

```
(check-sat)
```

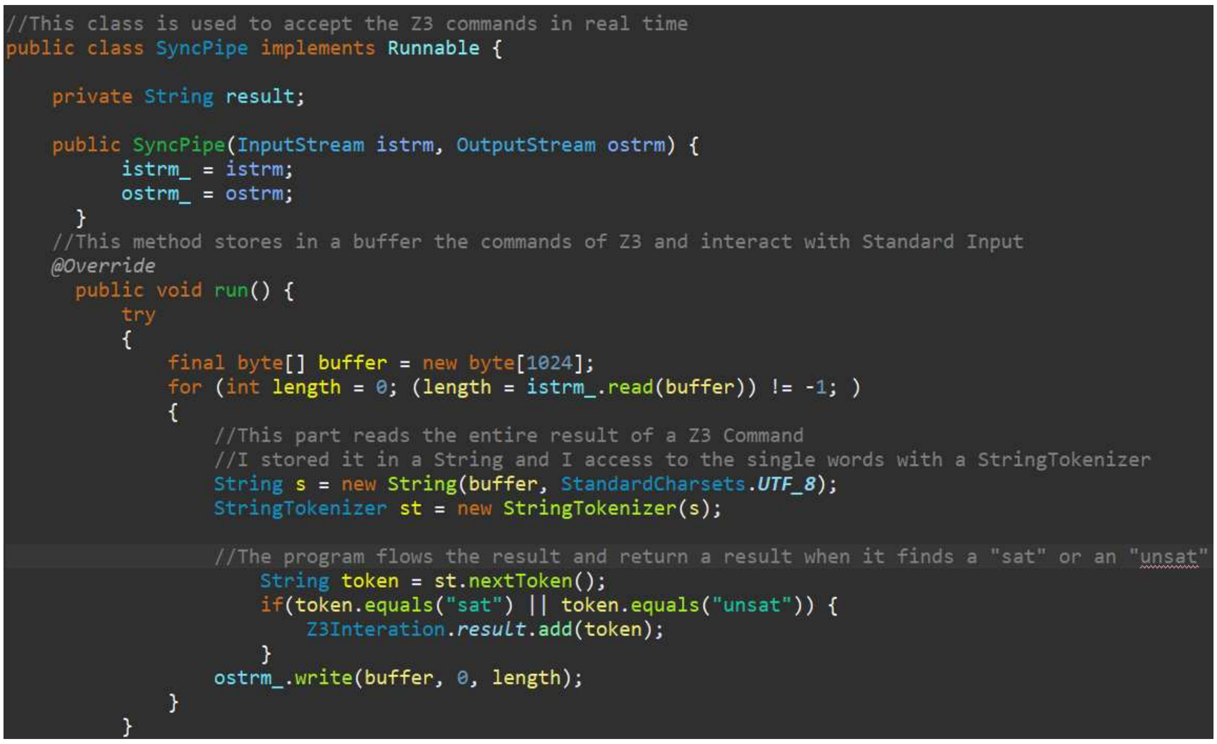
the following results can be returned:

- **sat** : if the constraint set is satisfiable;
- **unsat** : if the constraint set is unsatisfiable.

In some algorithms such as saturation, an incremental approach may be needed, i.e. the possibility of dynamically inserting and removing certain constraints. This can be done for example with the following commands

```
(push)
(assert ...)
...
(check-sat)
(pop)
```

The *push* operation adds a level on the stack from which one or more constraints can be pushed via *assert* Through *check-sat* it is possible to verify that the set of constraints with the addition of the



```
//This class is used to accept the Z3 commands in real time
public class SyncPipe implements Runnable {

    private String result;

    public SyncPipe(InputStream istrm, OutputStream ostrm) {
          istrm_ = istrm;
          ostrm_ = ostrm;
      }
    //This method stores in a buffer the commands of Z3 and interact with Standard Input
    @Override
      public void run() {
          try
          {
              final byte[] buffer = new byte[1024];
              for (int length = 0; (length = istrm_.read(buffer)) != -1; )
              {
                  //This part reads the entire result of a Z3 Command
                  //I stored it in a String and I access to the single words with a StringTokenizer
                  String s = new String(buffer, StandardCharsets.UTF_8);
                  StringTokenizer st = new StringTokenizer(s);

                  //The program flows the result and return a result when it finds a "sat" or an "unsat"
                      String token = st.nextToken();
                      if(token.equals("sat") || token.equals("unsat")) {
                          Z3Interation.result.add(token);
                      }
                  ostrm_.write(buffer, 0, length);
              }
          }
```

Figure 5.2: The implementation of the Pipe to interact with Z3

new constraints is satisfiable or not and finally with *pop* the last inserted level of the stack is removed, thus returning to the set of starting constraints.

Finally another useful application for the implementation of this software is to verify that two sets of constraints are logical implication of each other (C1 $\iff$ C2) .

If the following statement is not satisfiable then the two sets of constraints are equivalent.

```
(assert (not (= C1 C2))
```

5.4 Interaction with Z3

Before describing the saturation algorithm it is essential to describe how the interaction between Z3 and the program written in Java takes place. Z3 is SMT Solver used for our application.

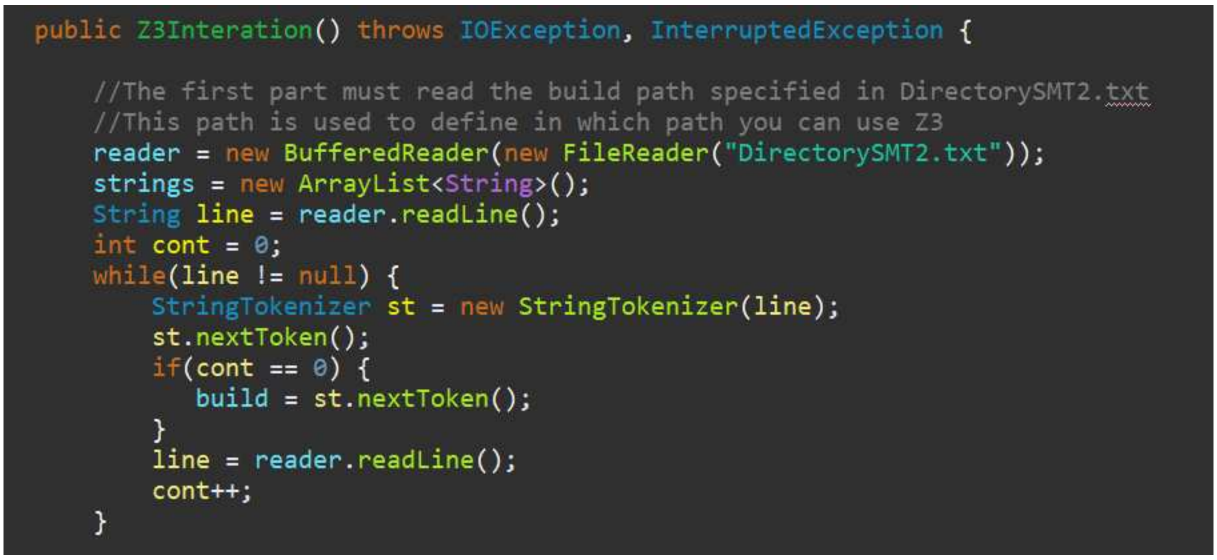
There are two approaches to interacting with Z3 through Java:

- Pipe interaction, ie using Z3 in another process and creating a communication channel between the Constraint Graph construction software process and the process using Z3;
- Integration of Z3 within the software through specific APIs of the programming language.

For the implementation of the project, the first approach was used, ie a communication channel was built between the command line shell and our software.

To do this, the *SyncPipe* class was created. This class implements the pipe to create communication between the program and the command shell. It implements a Thread where in its run() method it reads everything that is written by Z3 and once it finds a "sat" or "unsat" it updates its result. Figure 5.2 shows the implementation of the Run method of the SyncPipe class where you can create a byte buffer where you write and read the commands to be transmitted on the command line to run Z3.

After creating this class it is necessary to create the class that uses the pipe to create an interaction between the program and Z3. At the implementation level, the *Z3Interation* class has been defined as shown in Figure 5.3 and 5.4. This class allows the program to interact with Z3, to do this it is necessary to configure the Z3 build path in the DirectorySMT2.txt file starting from the folder with the project



```
public Z3Interation() throws IOException, InterruptedException {

    //The first part must read the build path specified in DirectorySMT2.txt
    //This path is used to define in which path you can use Z3
    reader = new BufferedReader(new FileReader("DirectorySMT2.txt"));
    strings = new ArrayList<String>();
    String line = reader.readLine();
    int cont = 0;
    while(line != null) {
        StringTokenizer st = new StringTokenizer(line);
        st.nextToken();
        if(cont == 0) {
           build = st.nextToken();
        }
        line = reader.readLine();
        cont++;
    }
```

Figure 5.3: The first part of the Interation with Z3 through Pipe.

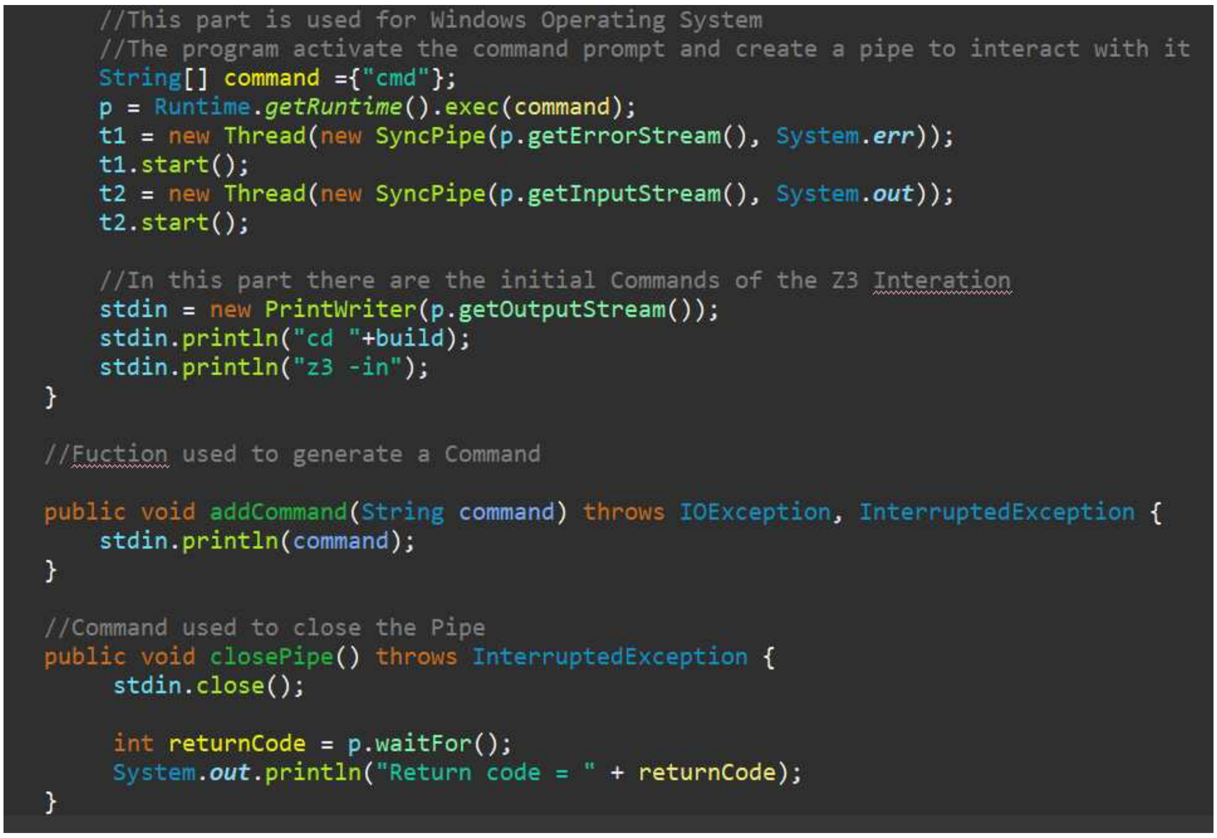
name. Once configured, the file is opened and the path is read. The basic version of this program is implemented for Windows operating systems, in fact it activates a Process class object to which it sends the cmd command to open the command prompt. However it is also possible to extend to the Linux operating system by replacing

```
String[] command ={"cmd"};
```

with:

```
String[] command ={"/bin/bash"};
```

Subsequently, thanks to the SyncPipe class, a pipe is created to guarantee the interaction between the command prompt and the program. From the command prompt, go to the build folder and start sending the various commands to run on the Z3. The outputs of the commands will be stored in p.getOutputStream and are directed to the standard input and the final result is stored inside the global variable result. In the rest of the class there are the various Get and Set methods.



```
        //This part is used for Windows Operating System
        //The program activate the command prompt and create a pipe to interact with it
        String[] command ={"cmd"};
        p = Runtime.getRuntime().exec(command);
        t1 = new Thread(new SyncPipe(p.getErrorStream(), System.err));
        t1.start();
        t2 = new Thread(new SyncPipe(p.getInputStream(), System.out));
        t2.start();

        //In this part there are the initial Commands of the Z3 Interation
        stdin = new PrintWriter(p.getOutputStream());
        stdin.println("cd "+build);
        stdin.println("z3 -in");
    }

    //Fuction used to generate a Command

    public void addCommand(String command) throws IOException, InterruptedException {
        stdin.println(command);
    }

    //Command used to close the Pipe
    public void closePipe() throws InterruptedException {
         stdin.close();

         int returnCode = p.waitFor();
         System.out.println("Return code = " + returnCode);
    }
```

Figure 5.4: The second part of the Interation with Z3 through Pipe.

CHAPTER 6

Construction of the new Constraint Set

The program to be developed essentially involves the implementation of two algorithms, one to update the set of constraints and another for the actual construction of the Constraint Graph with control functions to verify the Soundness properties. The first algorithm is used by the second to update the constraint set of the new node to be created. In the next paragraphs we will proceed with the analysis of these two algorithms and their implementation. The results produced by the program must then be subsequently tested to validate their correctness and identify any errors. This chapter analyzes the process of building a new Constraint Set which will then define the node of the Constraint Graph.

6.1 Saturation Algorithm

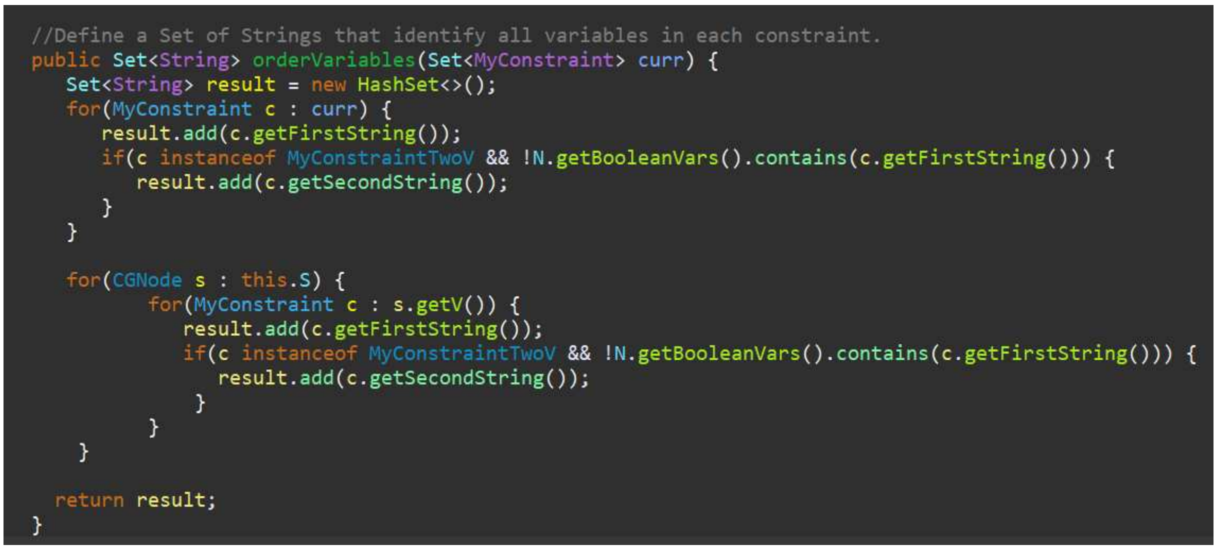
A fundamental building block in the creation of the software is the saturation algorithm which, given a set of constraints, is able to detect which constraints are the implication of that particular set. For example, if within the current constraint set contains a > 10, then a > 5 is a constraint that is implied. In the Saturation algorithm, 3 sets are taken into consideration:

- V : Set of Variables;
- Cost : Set of Constants;
- Op : Set of Operators.

The set of operators includes the 6 relational operators >, < , $\geq$, $\leq$, =, $\neq$. As a first step, all possible constraints are computed by combining each variable-variable and variable-constant pair for each possible operator. Each constraint is verified if it is an implication of the Constraint Set (C) by verifying the following property:
$C \implies v \iff C \cup \neg v$ is unsatisfiable.
In case the input Constraint Set is not satisfiable, the set itself is returned. In this work, the Saturation algorithm required the use of a series of external mechanisms in order to manage the constraints:

- Constraint Solver: A set of classes which provide methods which allow to verify the satisfiability of a set of constraints, in particular Choco Solver has been used;
- SMT Solver: another technique to be able to manage a set of constraints. It turns out that for the type of constraints that must be managed in this work it is the most efficient solution.

Both versions have been implemented in the program. When using the Constraint Solver, saturation has two roles in the construction of the constraint graph : to compute the set $C \oplus c$ (Algorithm 1) and to check if two constraint sets are equivalent in Algorithm 2. As the experimental evaluation will show in chapter 8,9 and 10, saturation is the most computationally intensive part of the program. For this



```
//Define a Set of Strings that identify all variables in each constraint.
public Set<String> orderVariables(Set<MyConstraint> curr) {
   Set<String> result = new HashSet<>();
   for(MyConstraint c : curr) {
      result.add(c.getFirstString());
      if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
         result.add(c.getSecondString());
      }
   }

   for(CGNode s : this.S) {
         for(MyConstraint c : s.getV()) {
            result.add(c.getFirstString());
            if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
               result.add(c.getSecondString());
             }
         }
    }

  return result;
}
```

Figure 6.1: The Algorithm to select all Variables from a Constraint Set

reason we have found a way with SMT Solver to verify that two sets of constraints are implication of each other without resorting to saturation and thus increasing the efficiency of the program.

Some functions common to both saturation algorithms will be described below, bearing in mind however that the saturation algorithm from Constraint Solver has several limitations compared to that from SMT Solver.

6.1.1 The selection of variables

In the implementation of the software the variables are represented as strings and in both cases it is useful to have a set containing all the variables without repetitions, for this reason the function *orderVariables(Constraint Set)* has been implemented, which returns in output a Set of strings with the names of all the variables.

The implemented code is in Figure 6.1:

This method allows you to extract from a set of constraints, the set of all the strings that identify variables within that set. The first string of a constraint is always added, while in the second case only if the constraint is of the variable-variable type and if the variable is not a boolean, i.e. contained within a Petrinet string set obtained via getBooleanVars(). In fact, if the variable is boolean, the type of constraint is variable-variable, but in reality the second string can assume 3 values:

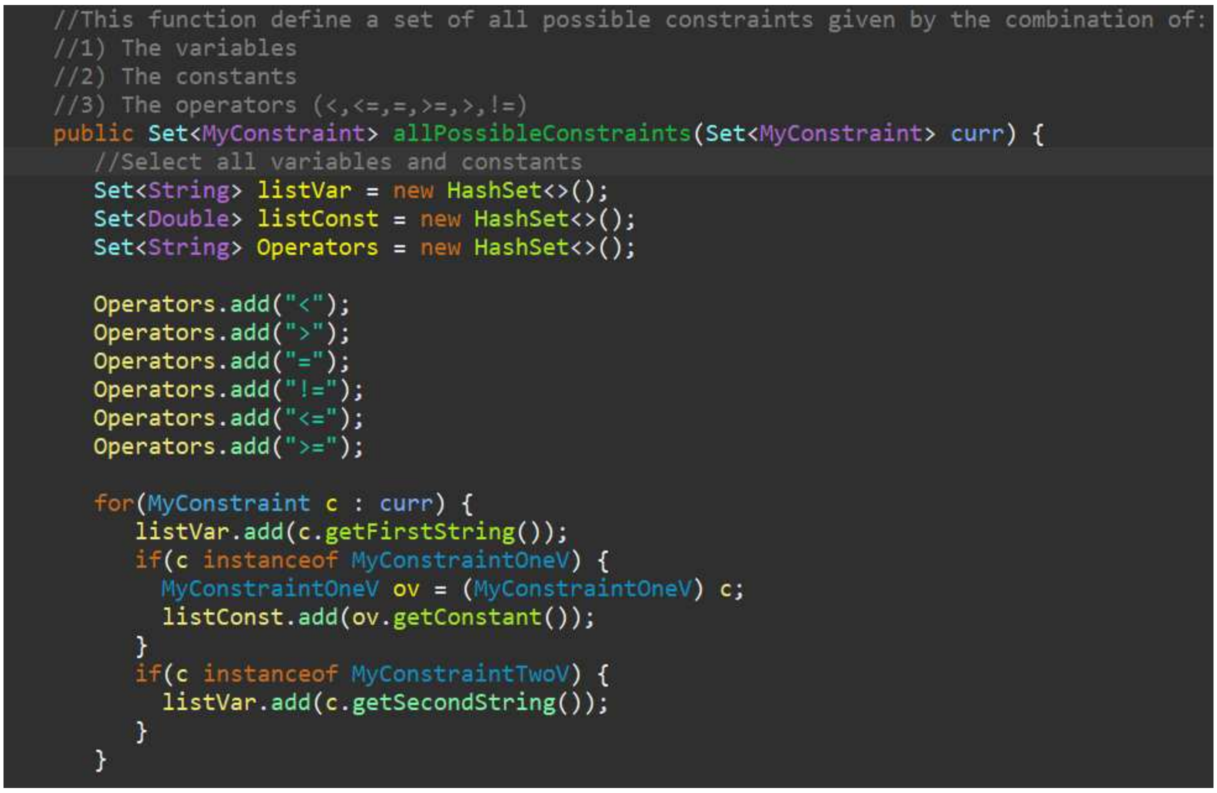
- “” : Undefined value
- “true” : True value
- “false” : False value

They are three strings, but they are not variables, so they should be excluded from adding to the final set. This algorithm is used to extract the Set of Strings of all the variables present within the set of input constraints.

6.1.2 All possible constraints of a Constraint Set

The saturation algorithm is based on verifying that every possible constraint obtained from the combination of the six relation operators with the set of variables and constants present is an implication of the Constraint Set. To do this, the *allPossibleConstraint(Constraint Set)* function was implemented which takes a Constraint Set as input and returns the Set of all possible constraints, as shown in Figure 6.2.

The first part of the code defines the three sets to consider:



```
//This function define a set of all possible constraints given by the combination of:
//1) The variables
//2) The constants
//3) The operators (<,<=,=,>=,>,!=)
public Set<MyConstraint> allPossibleConstraints(Set<MyConstraint> curr) {
   //Select all variables and constants
   Set<String> listVar = new HashSet<>();
   Set<Double> listConst = new HashSet<>();
   Set<String> Operators = new HashSet<>();

   Operators.add("<");
   Operators.add(">");
   Operators.add("=");
   Operators.add("!=");
   Operators.add("<=");
   Operators.add(">=");

   for(MyConstraint c : curr) {
      listVar.add(c.getFirstString());
      if(c instanceof MyConstraintOneV) {
        MyConstraintOneV ov = (MyConstraintOneV) c;
        listConst.add(ov.getConstant());
      }
      if(c instanceof MyConstraintTwoV) {
        listVar.add(c.getSecondString());
      }
   }
```

Figure 6.2: The first part of algorithm to select all possible Constraints given variables and constants

- The set of variables (Strings);
- the set of constants (double);
- the set of operators (strings).

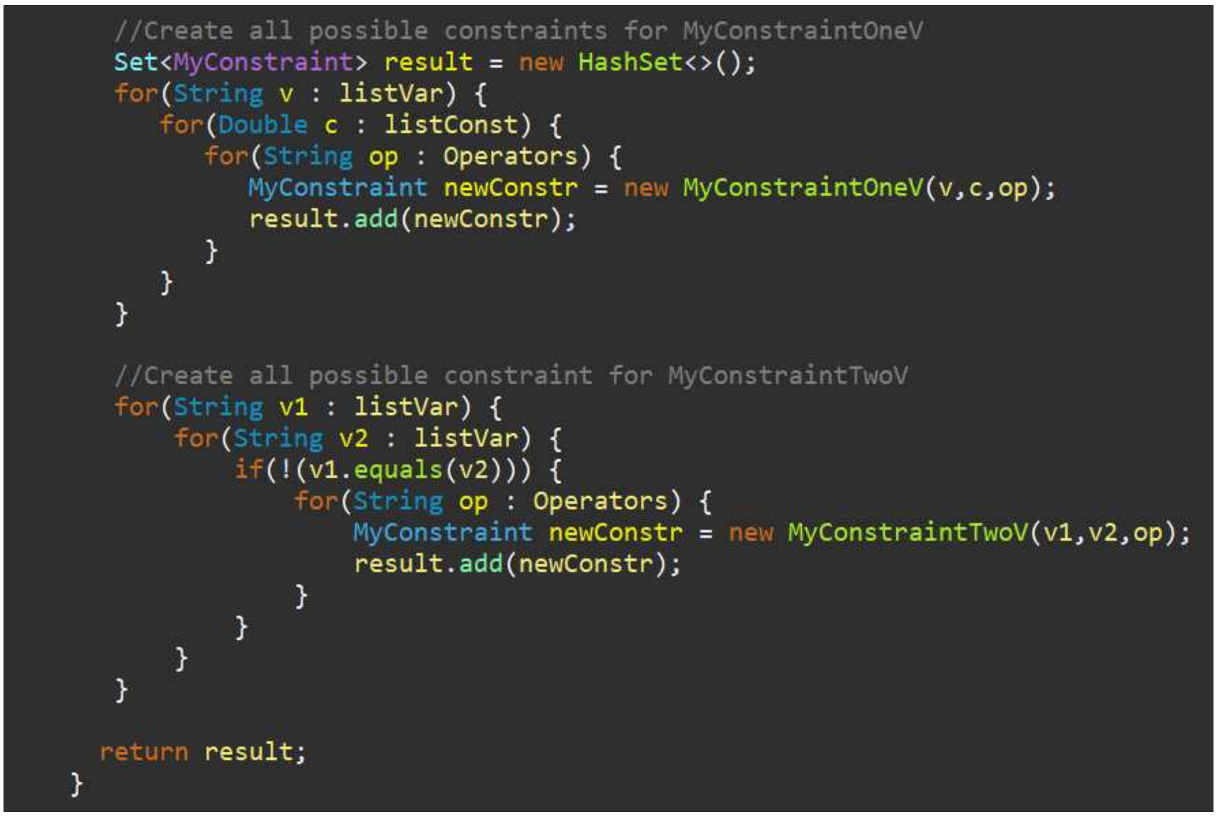
The three lists are then initialized. The six relational operators are added to the operator list. Subsequently a cycle is made distinguishing between variable-constant constraints and variable-constraints. In any case, the first variable is added to the set of variables, while if the second operand is a constant it is added to the set of constants, if instead it is a variable it is also added to the set of variables. The reason why an object of class *Set* and not *List* was used is to avoid storing the same string more than once.

Subsequently, all the possible combinations of constraints are performed: we start from the variable-constant constraints and then move on to the variable-variable constraints. For variable-variable constraints, you check that you don't enter a constraint that combines a variable with itself. All new constraints are added to the *result* constraint set and output.

6.1.3 The Limitations of Saturation with Choco Solver

The first implementation of the Saturation algorithm used the Constraint Solver Choco Solver. An important premise to make is that this saturation algorithm has some more limitations than the one implemented with SMT Solver, which instead represents the definitive solution. These limitations concern the following points:

- The difficulty of managing real numbers which forced a conversion of double variables into integers, using the IntVar class;
- The heavy inefficiency of Constraint Solver compared to SMT Solver, which is why we switched from Constraint Solver to SMT Solver.



```
        //Create all possible constraints for MyConstraintOneV
        Set<MyConstraint> result = new HashSet<>();
        for(String v : listVar) {
           for(Double c : listConst) {
              for(String op : Operators) {
                 MyConstraint newConstr = new MyConstraintOneV(v,c,op);
                 result.add(newConstr);
              }
           }
        }

        //Create all possible constraint for MyConstraintTwoV
        for(String v1 : listVar) {
            for(String v2 : listVar) {
                if(!(v1.equals(v2))) {
                    for(String op : Operators) {
                        MyConstraint newConstr = new MyConstraintTwoV(v1,v2,op);
                        result.add(newConstr);
                    }
                }
            }
        }

       return result;
    }
```

Figure 6.3: The second part of algorithm to select all possible Constraints given variables and constants

For this reason this saturation algorithm is actually incomplete, however it allows you to compare the technology of the Constraint Solver and that of the SMT Solver and understand which of the two is more advantageous than the other, at least for the types of constraints that are managed in this program. In fact, using saturation with Choco Solver on an example that generated 69 nodes took about 10 minutes to compute the entire Constraint Graph, while with Z3 it took only 40 seconds.

Furthermore with Z3 it is possible to work with variables of real type and thus solving the other limitation of Choco Solver.

6.1.4 Implementing Saturation with Z3

Using the incremental commands seen in chapter 5.2 and the subsequent interaction with Z3 seen in chapter 5.3 it is possible to implement the saturation algorithm with Z3 efficiently and also taking into account the real numbers. To implement the saturation algorithm it is necessary to define the constraints of the current set in Z3 by declaring all the variables inside it. The variable "wildcard" set to 0 is also specified, so it is possible to represent constraints like *y = 10* as *(assert (= (-y wildcard) 0))*, i.e y-wilcard = 0.

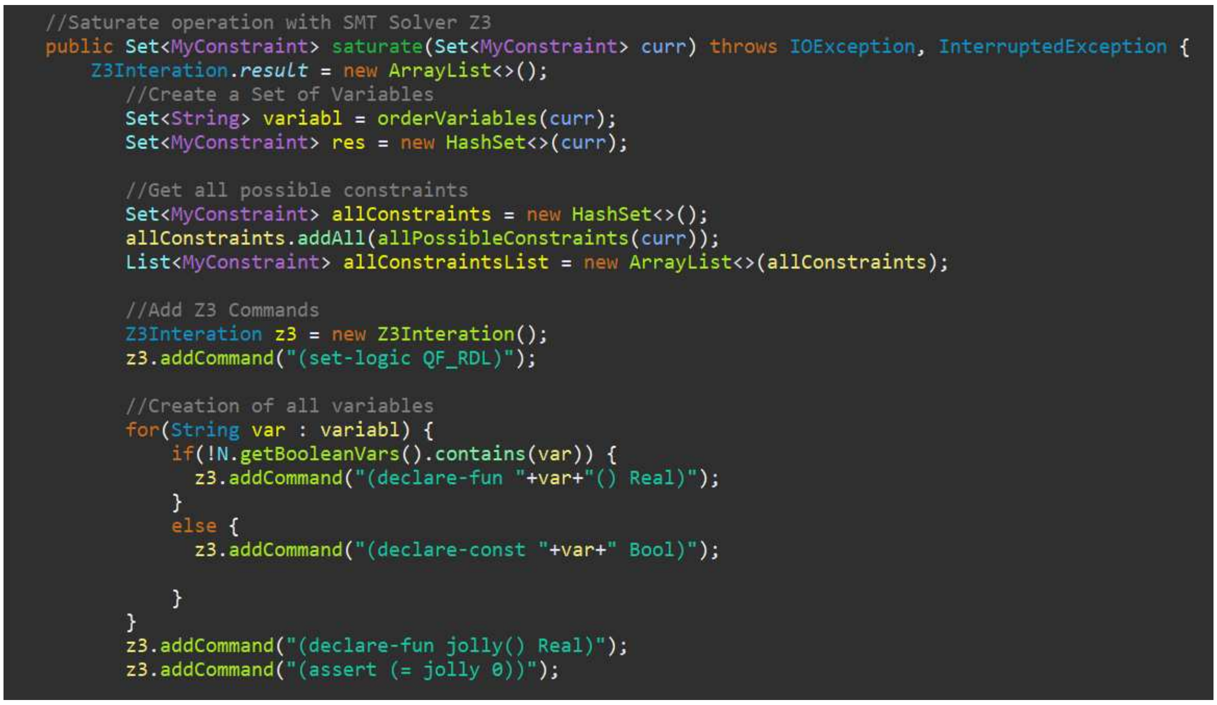
The next part is to enter the constraints of the current Constraint Set in variable-constant form

Immediately afterwards the constraints of the current Constraint Set are entered in the form variable-variable. Furthermore, the constraint is added in the else branch if the constraint concerns a boolean variable.

Once all the current constraints of the Constraint Set have been added, all the possible constraints, previously calculated with the function seen in Chapter 5.3.2, are added and the saturation is implemented by verifying with (push) and (pop) that each individual constraint is a logical implication of the current Constraint Set. So in the formula it occurs:

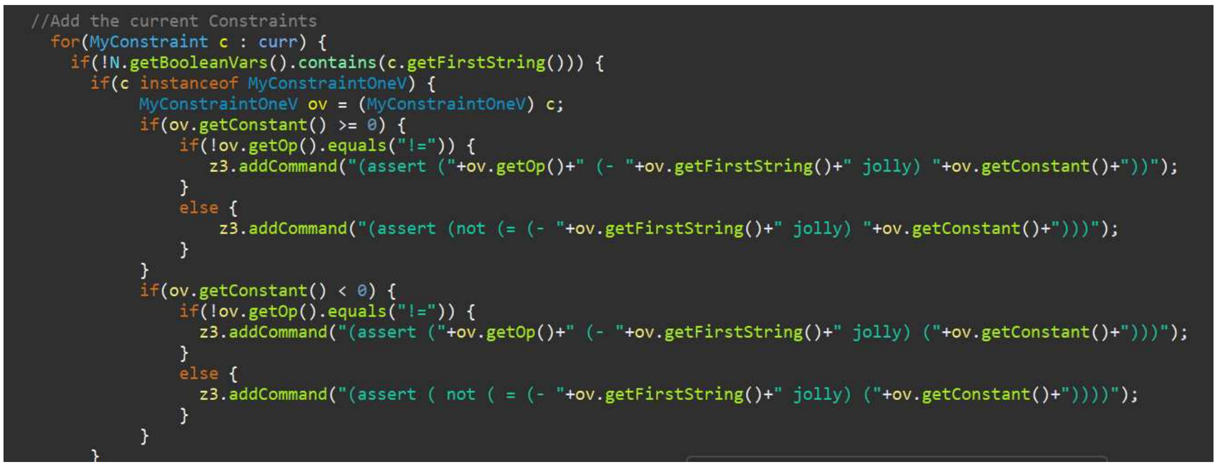
$$C \cup \neg c \textit{ is not satisfiable}$$

where C is the current Constraint Set and c is the constraint to add and check if it is a logical implication of C.



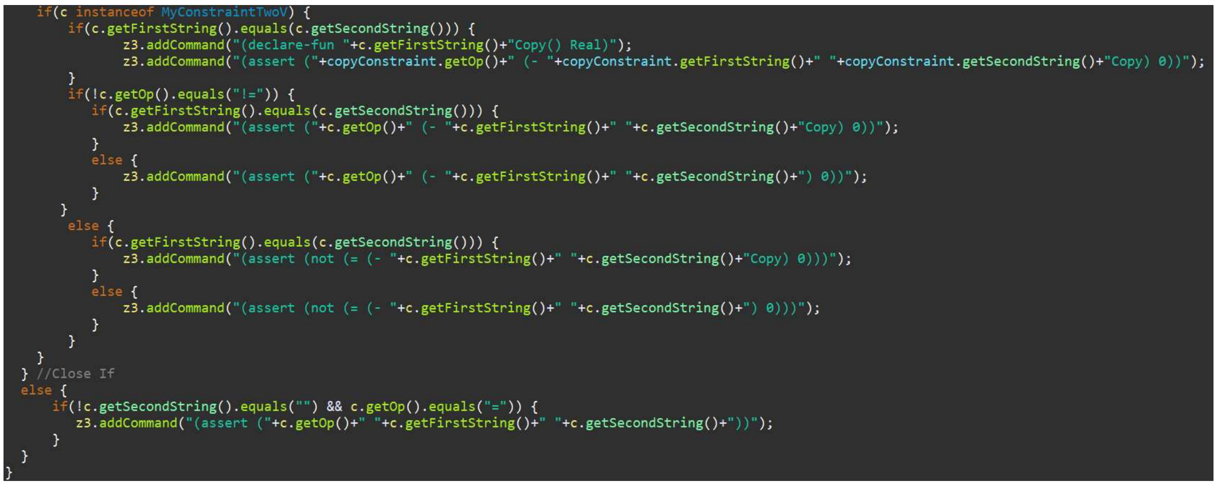
```
//Saturate operation with SMT Solver Z3
public Set<MyConstraint> saturate(Set<MyConstraint> curr) throws IOException, InterruptedException {
    Z3Interation.result = new ArrayList<>();
       //Create a Set of Variables
       Set<String> variabl = orderVariables(curr);
       Set<MyConstraint> res = new HashSet<>(curr);

       //Get all possible constraints
       Set<MyConstraint> allConstraints = new HashSet<>();
       allConstraints.addAll(allPossibleConstraints(curr));
       List<MyConstraint> allConstraintsList = new ArrayList<>(allConstraints);

       //Add Z3 Commands
       Z3Interation z3 = new Z3Interation();
       z3.addCommand("(set-logic QF_RDL)");

       //Creation of all variables
       for(String var : variabl) {
           if(!N.getBooleanVars().contains(var)) {
             z3.addCommand("(declare-fun "+var+"() Real)");
           }
           else {
             z3.addCommand("(declare-const "+var+" Bool)");

           }
       }
       z3.addCommand("(declare-fun jolly() Real)");
       z3.addCommand("(assert (= jolly 0))");
```

Figure 6.4: The first Part of Saturation Algorithm



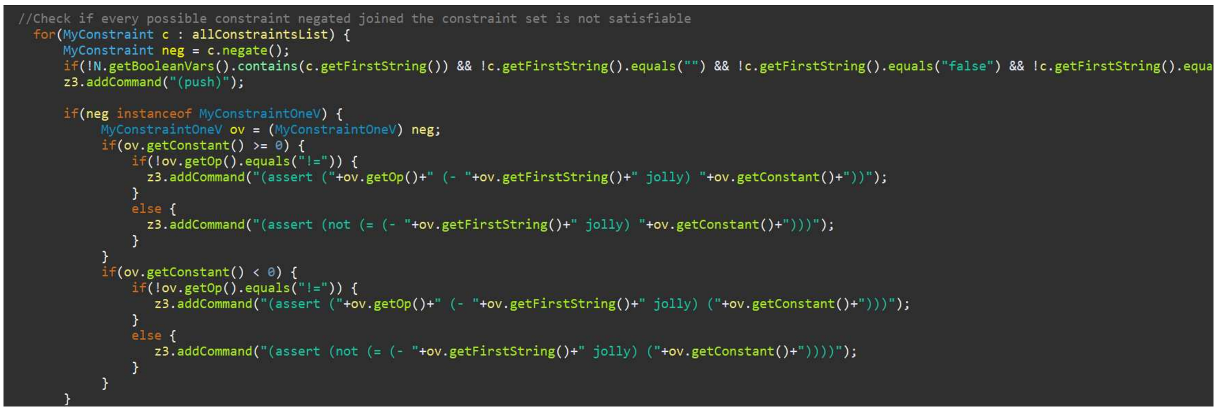
```
//Add the current Constraints
  for(MyConstraint c : curr) {
    if(!N.getBooleanVars().contains(c.getFirstString())) {
      if(c instanceof MyConstraintOneV) {
           MyConstraintOneV ov = (MyConstraintOneV) c;
           if(ov.getConstant() >= 0) {
               if(!ov.getOp().equals("!=")) {
                  z3.addCommand("(assert ("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+"))");
               }
               else {
                   z3.addCommand("(assert (not (= (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+")))");
               }
           }
           if(ov.getConstant() < 0) {
               if(!ov.getOp().equals("!=")) {
                 z3.addCommand("(assert ("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+")))");
               }
               else {
                 z3.addCommand("(assert ( not ( = (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+"))))");
               }
           }
      }
```

Figure 6.5: The second Part of Saturation Algorithm



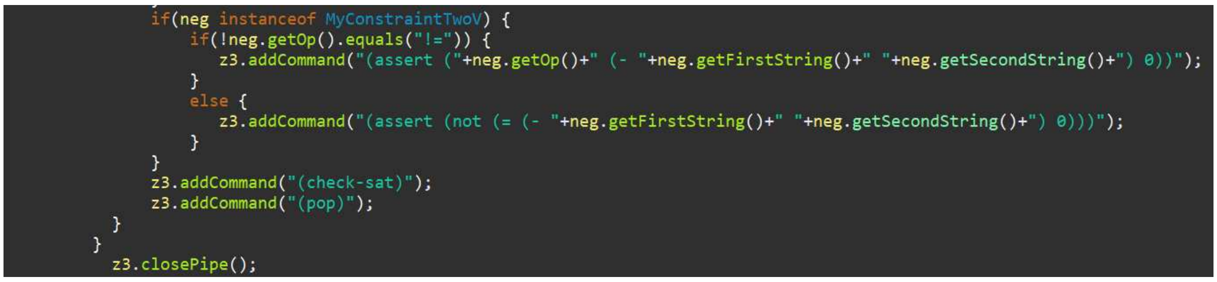
```
    if(c instanceof MyConstraintTwoV) {
        if(c.getFirstString().equals(c.getSecondString())) {
               z3.addCommand("(declare-fun "+c.getFirstString()+"Copy() Real)");
               z3.addCommand("(assert ("+copyConstraint.getOp()+" (- "+copyConstraint.getFirstString()+" "+copyConstraint.getSecondString()+"Copy) 0))");
        }
        if(!c.getOp().equals("!=")) {
           if(c.getFirstString().equals(c.getSecondString())) {
               z3.addCommand("(assert ("+c.getOp()+" (- "+c.getFirstString()+" "+c.getSecondString()+"Copy) 0))");
           }
           else {
               z3.addCommand("(assert ("+c.getOp()+" (- "+c.getFirstString()+" "+c.getSecondString()+") 0))");
           }
       }
        else {
           if(c.getFirstString().equals(c.getSecondString())) {
               z3.addCommand("(assert (not (= (- "+c.getFirstString()+" "+c.getSecondString()+"Copy) 0)))");
           }
           else {
               z3.addCommand("(assert (not (= (- "+c.getFirstString()+" "+c.getSecondString()+") 0)))");
           }
        }
    }
  } //Close If
  else {
      if(!c.getSecondString().equals("") && c.getOp().equals("=")) {
         z3.addCommand("(assert ("+c.getOp()+" "+c.getFirstString()+" "+c.getSecondString()+"))");
      }
  }
}
```

Figure 6.6: The third Part of Saturation Algorithm



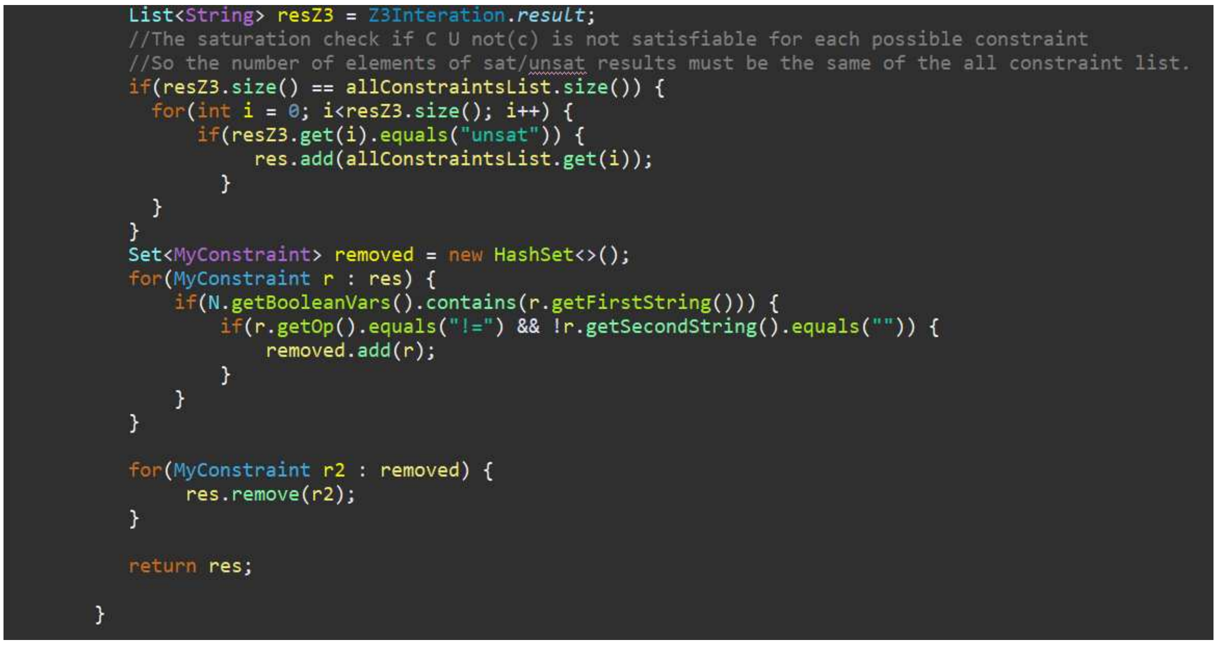
```
 //Check if every possible constraint negated joined the constraint set is not satisfiable
   for(MyConstraint c : allConstraintsList) {
       MyConstraint neg = c.negate();
       if(!N.getBooleanVars().contains(c.getFirstString()) && !c.getFirstString().equals("") && !c.getFirstString().equals("false") && !c.getFirstString().equa
       z3.addCommand("(push)");

       if(neg instanceof MyConstraintOneV) {
            MyConstraintOneV ov = (MyConstraintOneV) neg;
            if(ov.getConstant() >= 0) {
                if(!ov.getOp().equals("!=")) {
                  z3.addCommand("(assert ("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+"))");
                }
                else {
                  z3.addCommand("(assert (not (= (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+")))");
                }
            }
            if(ov.getConstant() < 0) {
                if(!ov.getOp().equals("!=")) {
                   z3.addCommand("(assert ("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+")))");
                }
                else {
                   z3.addCommand("(assert (not (= (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+"))))");
                }
            }
       }
```

Figure 6.7: The fourth Part of Saturation Algorithm



```
            if(neg instanceof MyConstraintTwoV) {
                if(!neg.getOp().equals("!=")) {
                   z3.addCommand("(assert ("+neg.getOp()+" (- "+neg.getFirstString()+" "+neg.getSecondString()+") 0))");
                }
                else {
                   z3.addCommand("(assert (not (= (- "+neg.getFirstString()+" "+neg.getSecondString()+") 0)))");
                }
            }
            z3.addCommand("(check-sat)");
            z3.addCommand("(pop)");
        }
    }
      z3.closePipe();
```

Figure 6.8: The fifth Part of Saturation Algorithm



```
        List<String> resZ3 = Z3Interation.result;
        //The saturation check if C U not(c) is not satisfiable for each possible constraint
        //So the number of elements of sat/unsat results must be the same of the all constraint list.
        if(resZ3.size() == allConstraintsList.size()) {
          for(int i = 0; i<resZ3.size(); i++) {
              if(resZ3.get(i).equals("unsat")) {
                  res.add(allConstraintsList.get(i));
              }
          }
        }
        Set<MyConstraint> removed = new HashSet<>();
        for(MyConstraint r : res) {
            if(N.getBooleanVars().contains(r.getFirstString())) {
                if(r.getOp().equals("!=") && !r.getSecondString().equals("")) {
                    removed.add(r);
                }
            }
        }

        for(MyConstraint r2 : removed) {
             res.remove(r2);
        }

        return res;

    }
```

Figure 6.9: The sixth Part of Saturation Algorithm

First of all, it is checked whether the new constraint to be inserted is of the type variable-constant or variable-variable. Then the *(check-sat)* command is executed to return *sat* or *unsat* and finally remove the last constraint with *(pop)*.

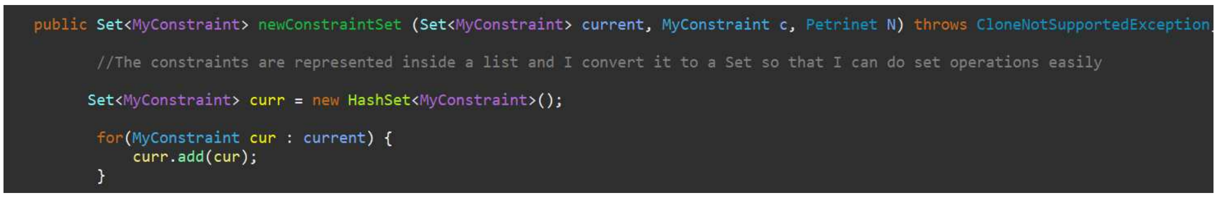
Once all the constraints have been inserted, the pipe is closed with the *closePipe()* method.

Finally, inside the *Z3Interation* class, a list of strings is stored that takes into account all the results, i.e. it will be a collection of *sat* or *unsat* for each constraint evaluated. Check that the number of elements of this list is the same as the set of all possible constraints, since otherwise it would mean that the algorithm has not stored the result of each constraint. In the final Constraint Set *res* all those constraints that resulted in *unsat* are added Finally, the constraints from the *res* set which concern a boolean variable and which use the *!=* operator (i.e. $\neq$) are removed since for simplicity of representation it was decided to represent all the constraints of a boolean variable with the = operator, in fact if for example *var != false* then it is true that var = true.

6.2 The New Constraint Set Construction Algorithm

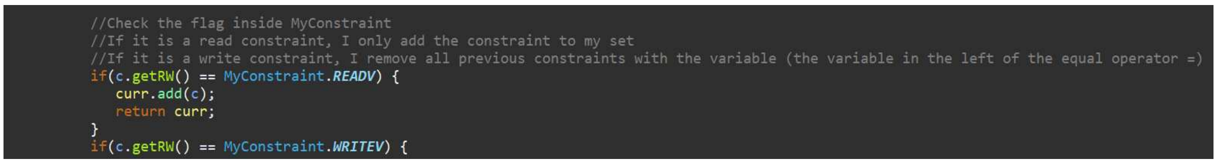
Now there are all the elements to implement the algorithm for the construction of the new Constraint Set defined in Chapter 2. In the practical implementation the following solution was implemented.

The algorithm takes 3 parameters as input: the DPN N, the Set of Constraint current and the transition guard c. To avoid problems of passing by address, the values present in the Current Set



```
public Set<MyConstraint> newConstraintSet (Set<MyConstraint> current, MyConstraint c, Petrinet N) throws CloneNotSupportedException

        //The constraints are represented inside a list and I convert it to a Set so that I can do set operations easily

       Set<MyConstraint> curr = new HashSet<MyConstraint>();

        for(MyConstraint cur : current) {
            curr.add(cur);
        }
```

Figure 6.10: The first part of the New Constraint Set Algorithm



```
        //Check the flag inside MyConstraint
        //If it is a read constraint, I only add the constraint to my set
        //If it is a write constraint, I remove all previous constraints with the variable (the variable in the left of the equal operator =)
        if(c.getRW() == MyConstraint.READV) {
           curr.add(c);
           return curr;
        }
        if(c.getRW() == MyConstraint.WRITEV) {
```

Figure 6.11: The second part of the New Constraint Set Algorithm

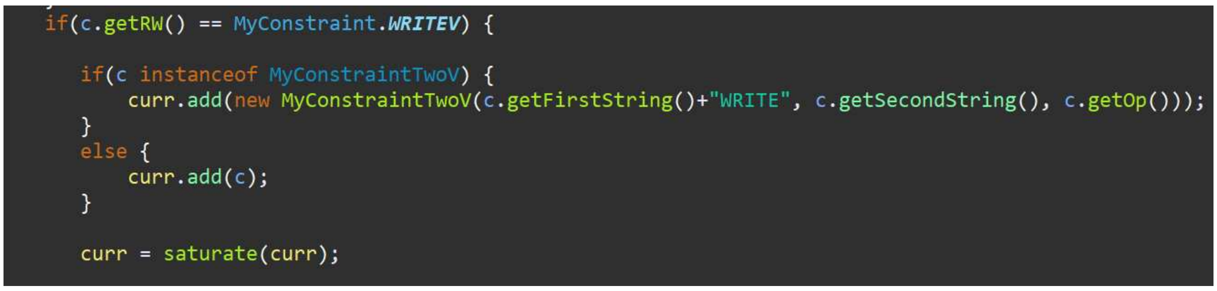
have been copied within the curr set and all subsequent operations are performed on this specific Set. The next part provides for the two cases for which the constraint can be read or written. In the case of reading, the constraint is simply added to Set curr and the resulting set of constraints returned. The Saturation operation has been omitted, since with Z3 it is possible to verify if two nodes of the constraint graph are equivalent in a more efficient way.

In the case of the constraint in writing, the constraint c is inserted inside the Set curr. A new constraint is created where the writing variable is rendered adding *WRITE* at the end of the name and the control variable is set to true. Then the Saturation operation is carried out.

Once the saturation has been performed, the new set of constraints is updated in the following basic steps:

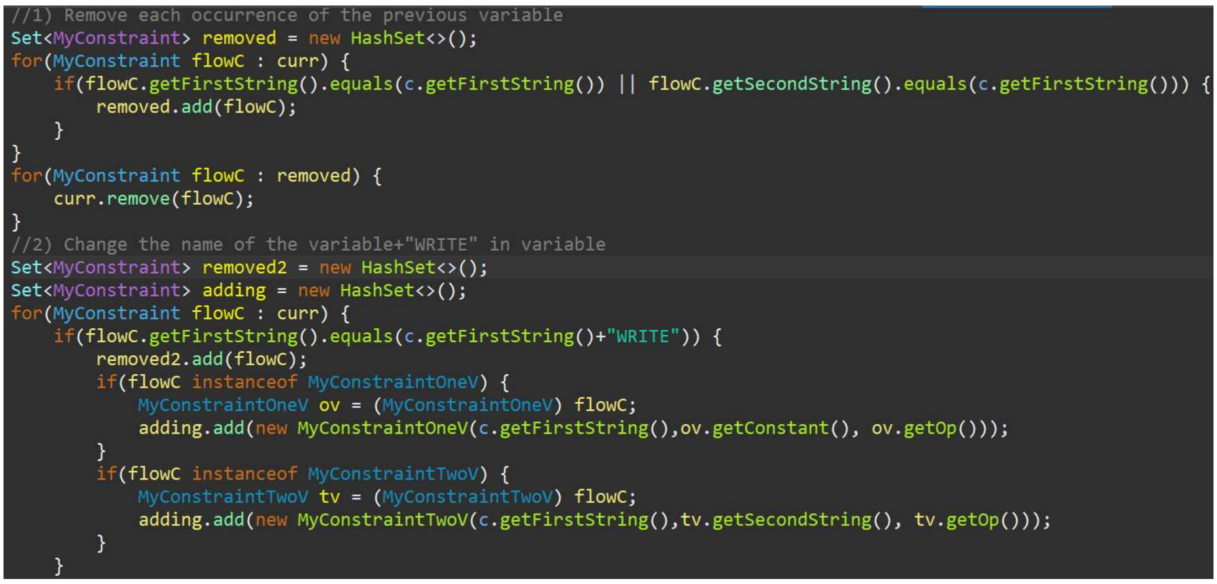
- 1) All constraints containing the previous variable are eliminated;
- 2) The constraints left with the variable with the addition of *WRITE* are renamed to the original variable name.

Finally, all the constraints to be removed are removed and the new constraints are added with the renamed variable.



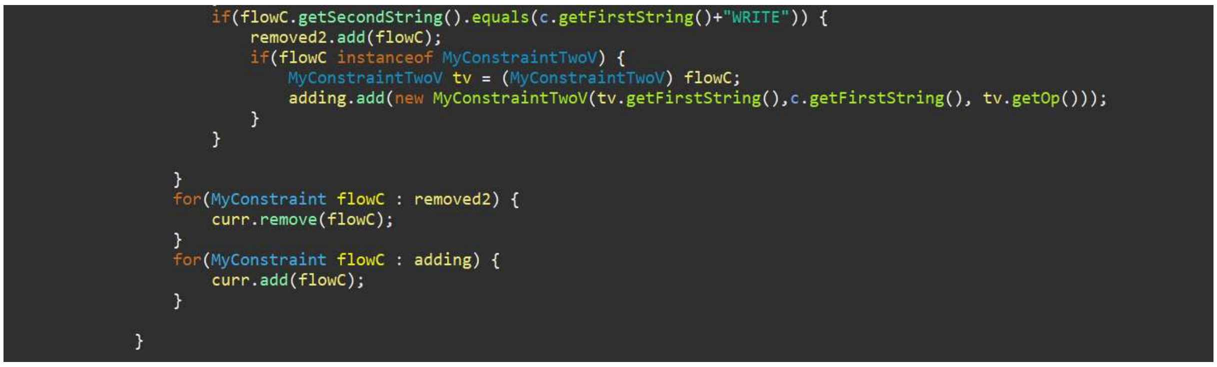
```
if(c.getRW() == MyConstraint.WRITEV) {

   if(c instanceof MyConstraintTwoV) {
       curr.add(new MyConstraintTwoV(c.getFirstString()+"WRITE", c.getSecondString(), c.getOp()));
   }
   else {
       curr.add(c);
   }

   curr = saturate(curr);
```

Figure 6.12: The third part of the New Constraint Set Algorithm



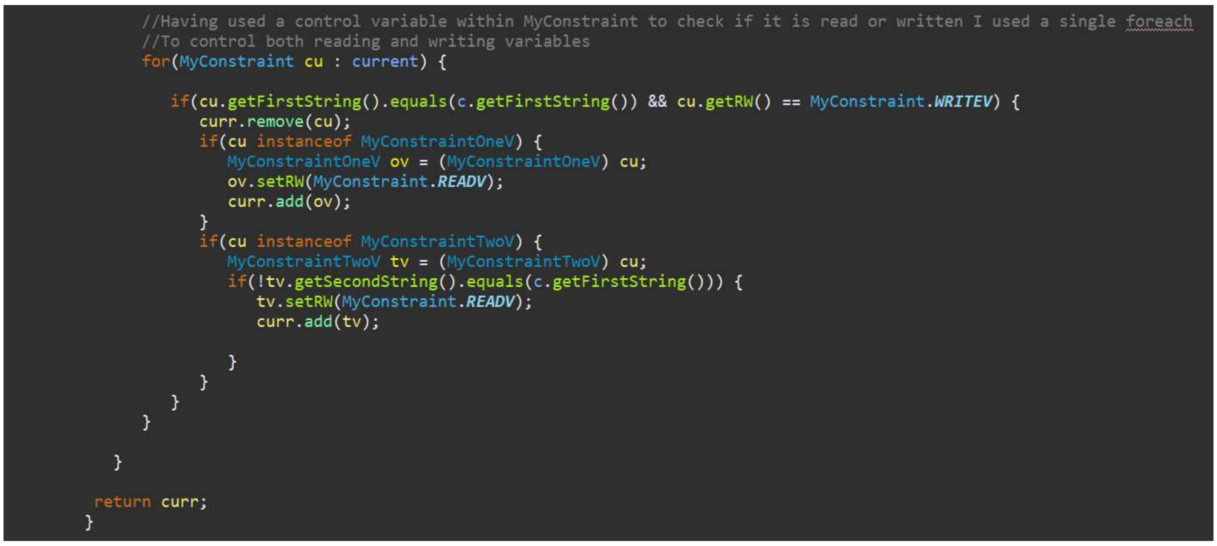
```
//1) Remove each occurrence of the previous variable
Set<MyConstraint> removed = new HashSet<>();
for(MyConstraint flowC : curr) {
    if(flowC.getFirstString().equals(c.getFirstString()) || flowC.getSecondString().equals(c.getFirstString())) {
        removed.add(flowC);
    }
}
for(MyConstraint flowC : removed) {
    curr.remove(flowC);
}
//2) Change the name of the variable+"WRITE" in variable
Set<MyConstraint> removed2 = new HashSet<>();
Set<MyConstraint> adding = new HashSet<>();
for(MyConstraint flowC : curr) {
    if(flowC.getFirstString().equals(c.getFirstString()+"WRITE")) {
        removed2.add(flowC);
        if(flowC instanceof MyConstraintOneV) {
            MyConstraintOneV ov = (MyConstraintOneV) flowC;
            adding.add(new MyConstraintOneV(c.getFirstString(),ov.getConstant(), ov.getOp()));
        }
        if(flowC instanceof MyConstraintTwoV) {
            MyConstraintTwoV tv = (MyConstraintTwoV) flowC;
            adding.add(new MyConstraintTwoV(c.getFirstString(),tv.getSecondString(), tv.getOp()));
        }
    }
```

Figure 6.13: The fourth part of the New Constraint Set Algorithm



```
                    if(flowC.getSecondString().equals(c.getFirstString()+"WRITE")) {
                        removed2.add(flowC);
                        if(flowC instanceof MyConstraintTwoV) {
                            MyConstraintTwoV tv = (MyConstraintTwoV) flowC;
                            adding.add(new MyConstraintTwoV(tv.getFirstString(),c.getFirstString(), tv.getOp()));
                        }
                    }

                }
                for(MyConstraint flowC : removed2) {
                    curr.remove(flowC);
                }
                for(MyConstraint flowC : adding) {
                    curr.add(flowC);
                }

            }
```

Figure 6.14: The fifth part of the New Constraint Set Algorithm



```
            //Having used a control variable within MyConstraint to check if it is read or written I used a single foreach
            //To control both reading and writing variables
            for(MyConstraint cu : current) {

                if(cu.getFirstString().equals(c.getFirstString()) && cu.getRW() == MyConstraint.WRITEV) {
                    curr.remove(cu);
                    if(cu instanceof MyConstraintOneV) {
                        MyConstraintOneV ov = (MyConstraintOneV) cu;
                        ov.setRW(MyConstraint.READV);
                        curr.add(ov);
                    }
                    if(cu instanceof MyConstraintTwoV) {
                        MyConstraintTwoV tv = (MyConstraintTwoV) cu;
                        if(!tv.getSecondString().equals(c.getFirstString())) {
                            tv.setRW(MyConstraint.READV);
                            curr.add(tv);

                        }
                    }
                }
            }

        }

     return curr;
    }
```

Figure 6.15: The sixth part of the New Constraint Set Algorithm

CHAPTER 7

Construction of the Constraint Graph

Once the first algorithm that deals with the construction of the new Constraint Set starting from a given guard has been defined, we proceed with the construction of a new node. A new node will not always be created: in fact if the new node will be equivalent to another existing node it will simply use the already created node or in other situations it may happen that the new node must not be created and there must be an interruption of the algorithm since you would be creating a Constraint Graph with an infinite number of nodes. All these aspects will be explored in this chapter.

7.1 The Satisfiability Function

This algorithm allows to verify if a given set of constraints given in input is satisfiable. The Satisfiable feature described here involves using Z3 as the SMT Solver that you interact with via pipe at the command prompt. Then it refers to the Z3Interation class whose implementation details will be described later.

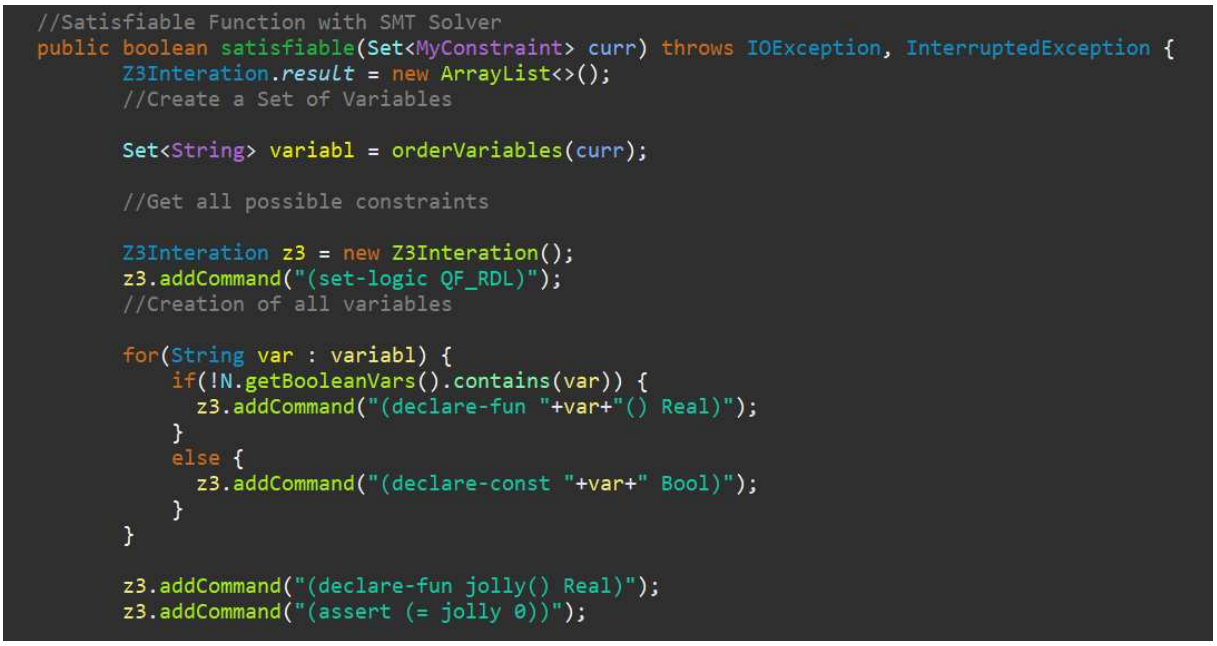
The first step involves defining an object of class Z3Interation to tell the program that it needs to interact with Z3. Subsequently a Set of Strings is created where inside all the variables of the set of constraints are taken through the orderVariables function. Then the actual interaction with Z3 begins by starting to send commands. The first command specifies the logic with which Z3 will have to work: i.e. the logic that allows real type constraints to be processed: QF_RDL. By scrolling through the previously defined Set of Strings, all the variables within Z3 are declared, both real and boolean. Finally, a "Wildcard" variable is created and initialized to zero, to allow to represent unary constraints as difference constraints.

The next step is to insert assertions inside Z3 to specify all the various variable constraints, then with a for loop iterates all the constraints of the current set. In the case of a variable-constant constraint, then essentially two main factors must be verified:

- If the constant value is greater or less than zero;
- If the operator is a "!=" operator.

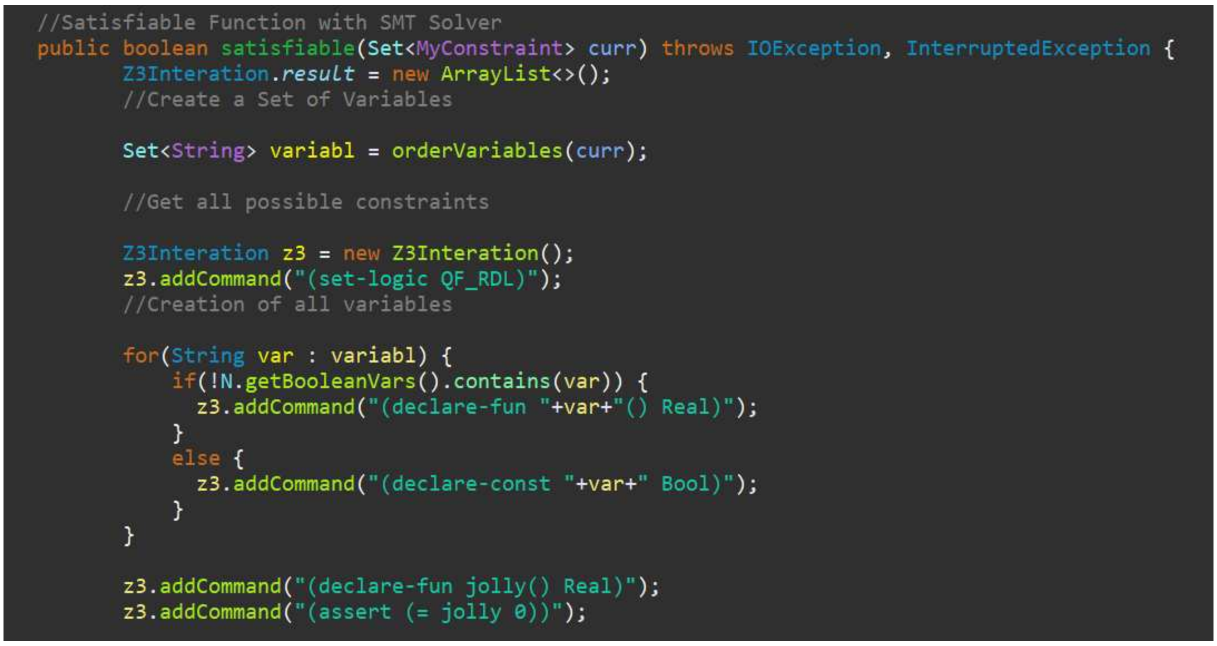
Both cases are related to the syntax of Z3, in fact a number less than 0 must be inserted in round brackets, since the '-' is read as an operator between two values, while in the second case the operator "!=" as for example represented in a PNML generated with ProM, it is represented as (not (= . . .)). These operations are performed if the variable does not belong to the variable returned by getBooleanVars of the Petrinet class, ie if it is not a boolean variable.

The other case is that of a variable-variable constraint, in this case the check on the positive or negative constant is eliminated and only the check of the operator '!=' remains. If, on the other hand, it were to be a constraint on a Boolean variable, a simple assert is added with the constraint, represented in the else branch. The reason why the operator is '=' is that constraints on a boolean variable can be of 5 types:



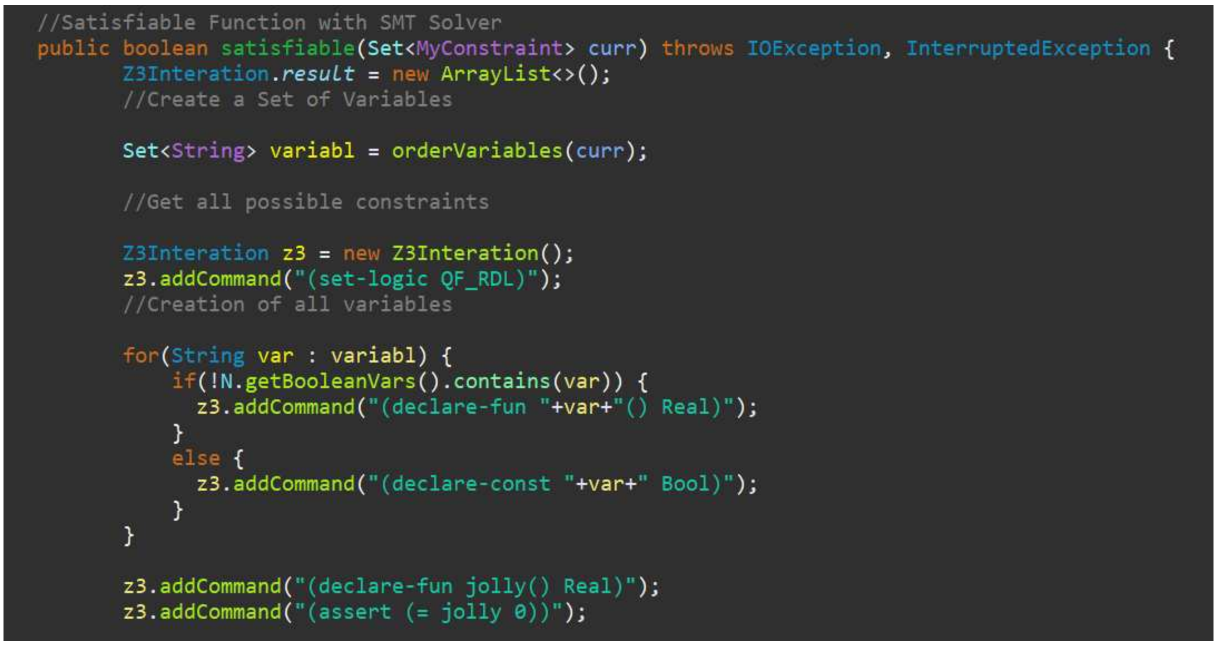
```
//Satisfiable Function with SMT Solver
public boolean satisfiable(Set<MyConstraint> curr) throws IOException, InterruptedException {
        Z3Interation.result = new ArrayList<>();
        //Create a Set of Variables

        Set<String> variabl = orderVariables(curr);

        //Get all possible constraints

        Z3Interation z3 = new Z3Interation();
        z3.addCommand("(set-logic QF_RDL)");
        //Creation of all variables

        for(String var : variabl) {
            if(!N.getBooleanVars().contains(var)) {
              z3.addCommand("(declare-fun "+var+"() Real)");
            }
            else {
              z3.addCommand("(declare-const "+var+" Bool)");
            }
        }

        z3.addCommand("(declare-fun jolly() Real)");
        z3.addCommand("(assert (= jolly 0))");
```

Figure 7.1: The first part of Satisfiable Algorithm



```
//Satisfiable Function with SMT Solver
public boolean satisfiable(Set<MyConstraint> curr) throws IOException, InterruptedException {
        Z3Interation.result = new ArrayList<>();
        //Create a Set of Variables

        Set<String> variabl = orderVariables(curr);

        //Get all possible constraints

        Z3Interation z3 = new Z3Interation();
        z3.addCommand("(set-logic QF_RDL)");
        //Creation of all variables

        for(String var : variabl) {
            if(!N.getBooleanVars().contains(var)) {
              z3.addCommand("(declare-fun "+var+"() Real)");
            }
            else {
              z3.addCommand("(declare-const "+var+" Bool)");
            }
        }

        z3.addCommand("(declare-fun jolly() Real)");
        z3.addCommand("(assert (= jolly 0))");
```

Figure 7.2: The second part of Satisfiable Algorithm



```
//Satisfiable Function with SMT Solver
public boolean satisfiable(Set<MyConstraint> curr) throws IOException, InterruptedException {
        Z3Interation.result = new ArrayList<>();
        //Create a Set of Variables

        Set<String> variabl = orderVariables(curr);

        //Get all possible constraints

        Z3Interation z3 = new Z3Interation();
        z3.addCommand("(set-logic QF_RDL)");
        //Creation of all variables

        for(String var : variabl) {
            if(!N.getBooleanVars().contains(var)) {
              z3.addCommand("(declare-fun "+var+"() Real)");
            }
            else {
              z3.addCommand("(declare-const "+var+" Bool)");
            }
        }

        z3.addCommand("(declare-fun jolly() Real)");
        z3.addCommand("(assert (= jolly 0))");
```

Figure 7.3: The third part of Satisfiable Algorithm

- variable = true;
- variable = false;
- variable != true;
- variable != false;
- variable = “” (undefined value)

In satisfiable if the variable has no defined value then it is simply not parsed. As far as the operator ’!=’ is concerned, it has simply been transformed into ’=’ when it is inserted in the list of strings of boolean variables present in the Petrinet class: in fact variable != false is equivalent to variable = true, while variable != true is equivalent to variable = false. Finally, once all the constraints have been entered in Z3, the command (check-sat) is added to verify whether the set of constraints is satisfiable (SAT) or not (UNSAT).

The last part of the method instead checks the returned result and in turn returns the value true or false. The value of the global variable "result" is taken which always takes into account the last interaction and the Pipe is closed. The reason the width of the list is checked is that it should return a single "SAT" or "UNSAT".

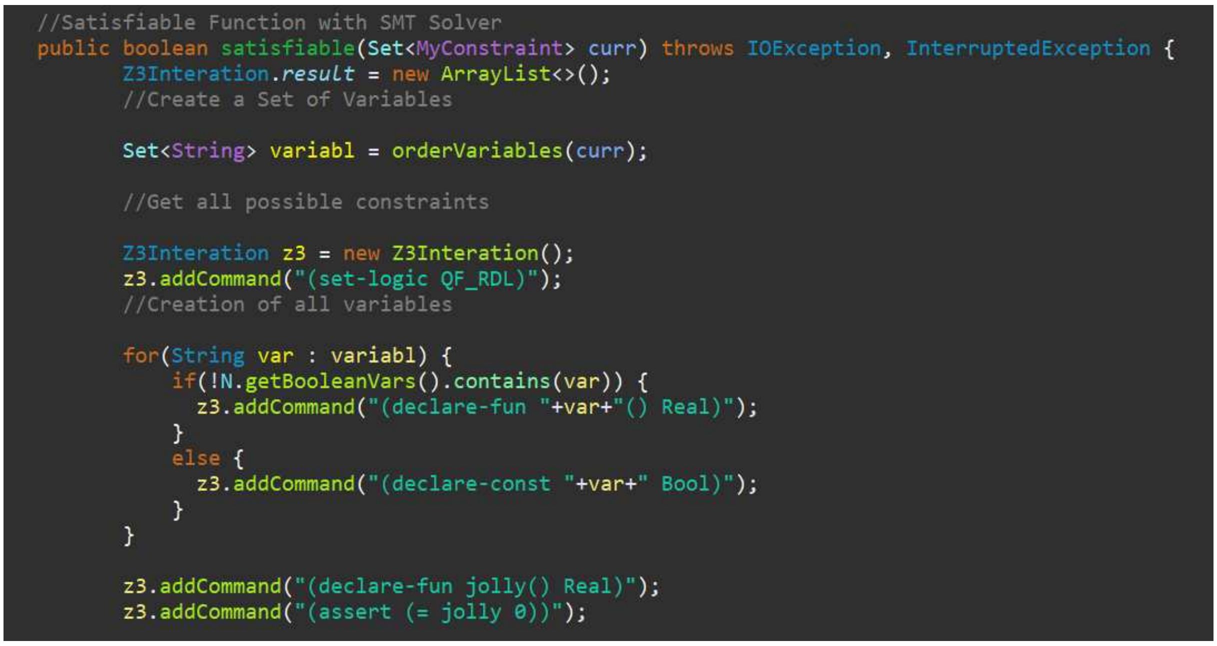
7.2 Check Equivalence Nodes

To verify that two nodes are equivalent, the *checkCGClone* method has been implemented.

This algorithm is used to check whether two sets of constraints subjected to saturation would obtain the same set of constraints, without using the saturation itself, thus making the program more efficient. First a new object of class Z3Interation is defined to create a pipe to which to send the various Z3 commands. Then a set of strings and another set of constraints are created where all the variables present and all the constraints present in newS are inserted respectively.

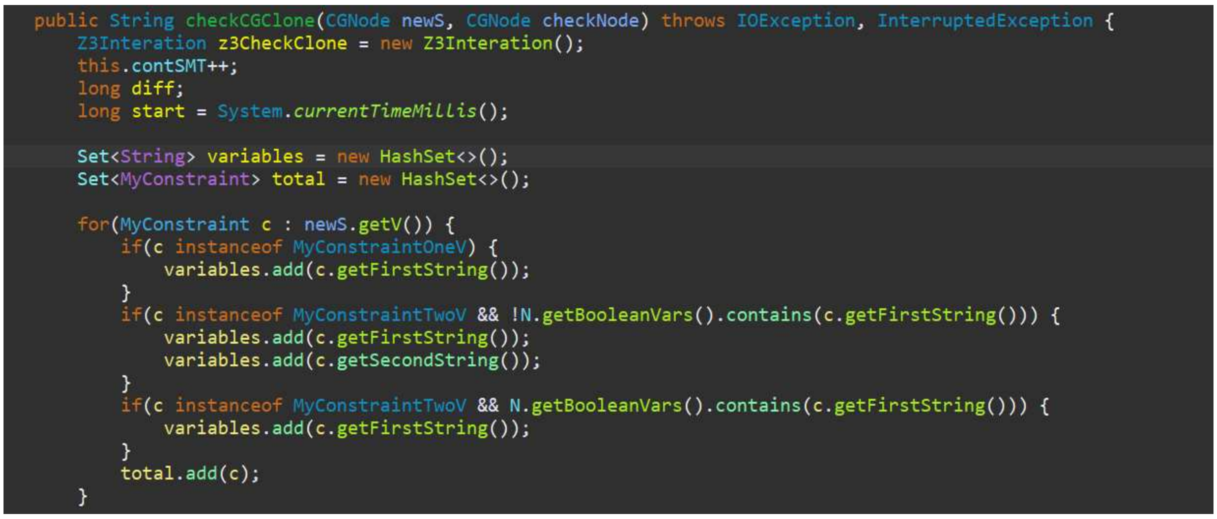
Then all the variables present in the entire set of nodes of the Constraint Graph are added to *S*

We start by declaring all variables both boolean and non-boolean and specifying the logic to use for the constraints, i.e. QF_RDL. As in the case of Satisfiable, the wildcard variable initialized to 0 is also defined which will be used to define the various constraints in Z3.



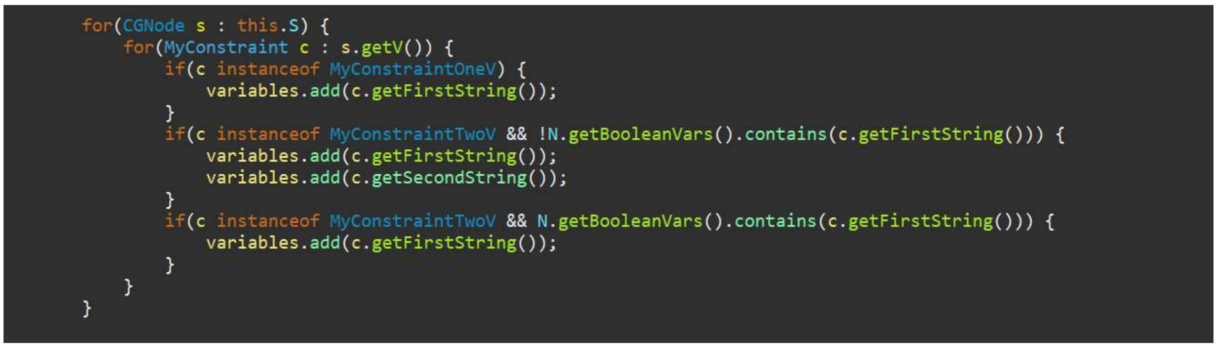
```java
//Satisfiable Function with SMT Solver
public boolean satisfiable(Set<MyConstraint> curr) throws IOException, InterruptedException {
       Z3Interation.result = new ArrayList<>();
       //Create a Set of Variables

       Set<String> variabl = orderVariables(curr);

       //Get all possible constraints

       Z3Interation z3 = new Z3Interation();
       z3.addCommand("(set-logic QF_RDL)");
       //Creation of all variables

       for(String var : variabl) {
           if(!N.getBooleanVars().contains(var)) {
             z3.addCommand("(declare-fun "+var+"() Real)");
           }
           else {
             z3.addCommand("(declare-const "+var+" Bool)");
           }
       }

       z3.addCommand("(declare-fun jolly() Real)");
       z3.addCommand("(assert (= jolly 0))");
```

Figure 7.4: The fourth part of Satisfiable Algorithm



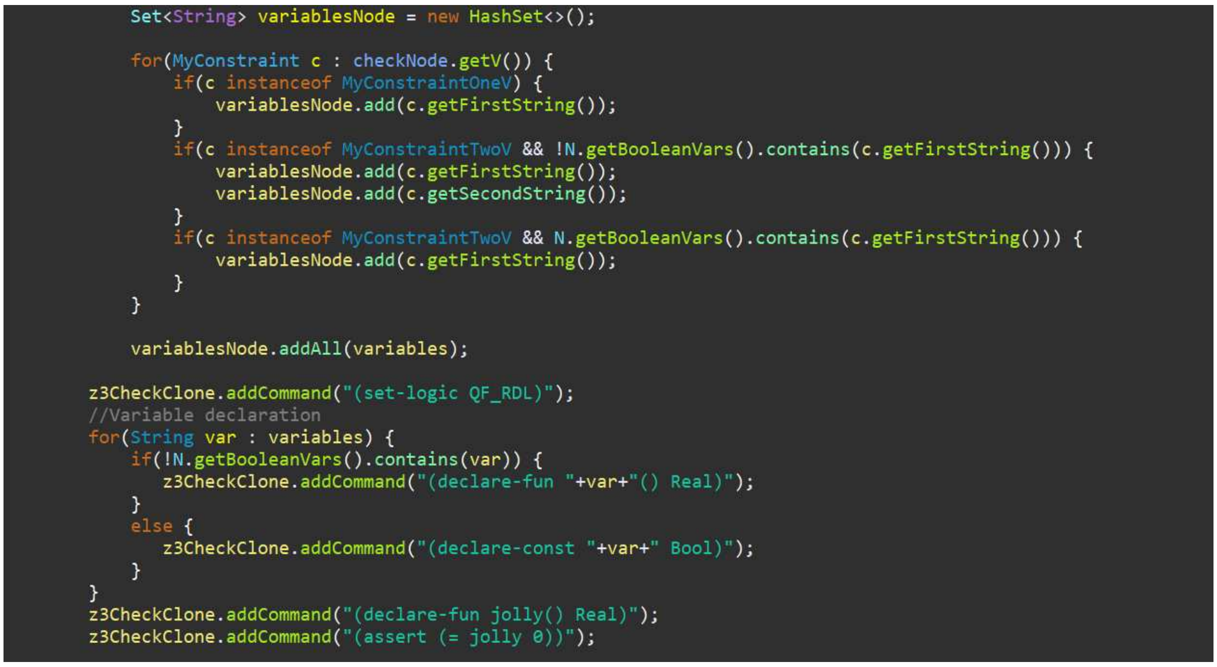
```java
public String checkCGClone(CGNode newS, CGNode checkNode) throws IOException, InterruptedException {
    Z3Interation z3CheckClone = new Z3Interation();
    this.contSMT++;
    long diff;
    long start = System.currentTimeMillis();

    Set<String> variables = new HashSet<>();
    Set<MyConstraint> total = new HashSet<>();

    for(MyConstraint c : newS.getV()) {
        if(c instanceof MyConstraintOneV) {
            variables.add(c.getFirstString());
        }
        if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
            variables.add(c.getFirstString());
            variables.add(c.getSecondString());
        }
        if(c instanceof MyConstraintTwoV && N.getBooleanVars().contains(c.getFirstString())) {
            variables.add(c.getFirstString());
        }
        total.add(c);
    }
```

Figure 7.5: The first part of the algorithm to check the equivalence of two nodes



```java
    for(CGNode s : this.S) {
        for(MyConstraint c : s.getV()) {
            if(c instanceof MyConstraintOneV) {
                variables.add(c.getFirstString());
            }
            if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
                variables.add(c.getFirstString());
                variables.add(c.getSecondString());
            }
            if(c instanceof MyConstraintTwoV && N.getBooleanVars().contains(c.getFirstString())) {
                variables.add(c.getFirstString());
            }
        }
    }
```

Figure 7.6: The second part of the algorithm to check the equivalence of two nodes



```
    Set<String> variablesNode = new HashSet<>();

    for(MyConstraint c : checkNode.getV()) {
        if(c instanceof MyConstraintOneV) {
            variablesNode.add(c.getFirstString());
        }
        if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
            variablesNode.add(c.getFirstString());
            variablesNode.add(c.getSecondString());
        }
        if(c instanceof MyConstraintTwoV && N.getBooleanVars().contains(c.getFirstString())) {
            variablesNode.add(c.getFirstString());
        }
    }

    variablesNode.addAll(variables);

z3CheckClone.addCommand("(set-logic QF_RDL)");
//Variable declaration
for(String var : variables) {
    if(!N.getBooleanVars().contains(var)) {
        z3CheckClone.addCommand("(declare-fun "+var+"() Real)");
    }
    else {
        z3CheckClone.addCommand("(declare-const "+var+" Bool)");
    }
}
z3CheckClone.addCommand("(declare-fun jolly() Real)");
z3CheckClone.addCommand("(assert (= jolly 0))");
```

Figure 7.7: The third part of the algorithm to check the equivalence of two nodes

The same thing is done for the variables inside checkNode and once this is finished the two sets of variables are merged to then move on to sending commands to Z3. We start by declaring all variables both boolean and non-boolean and specifying the logic to use for the constraints, i.e. QF_RDL. As in the case of Satisfiable, the Jolly variable initialized to 0 is also defined which will be used to define the various constraints in Z3.

Once you have finished inserting the constraints of the first node, you need to insert the constraints of the second node linked by the conjunction and.

The insertion of the constraints of the second node is then also completed.

Finally the operation is performed (check-sat) and the result is returned as output. The pipe is also closed and all parentheses are closed.

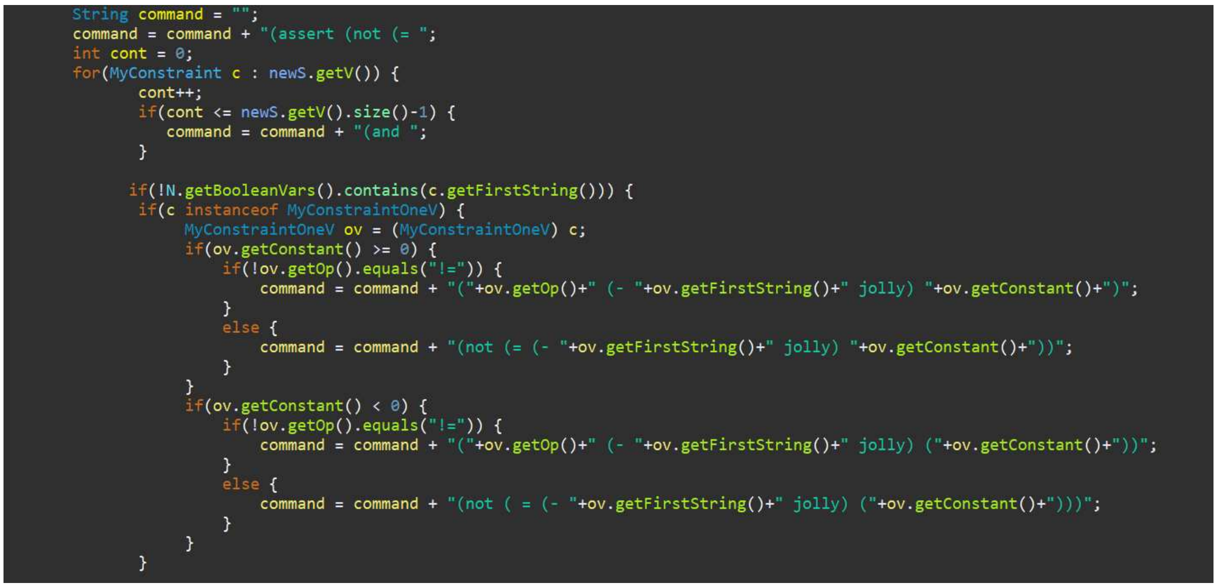
7.3 Construction of the Constraint Graph

Now there are all the elements to implement the final algorithm that builds the Constraint Graph and verifies its Soundness properties.

In the practical implementation, an initial markup, a list of initial constraints v and the Petri net N are taken as input. The counter will be used only in test phases to check the number of nodes of the Constraint Graph. Three constraint sets are initialized C0 which will contain the initial constraint set, C1 which will contain the transition constraint set and C2 which will contain the constraint set to be processed for the Silent Transition. A control variable unlimitedCG is set to false and is used to verify that the Constraint Graph is not unlimited, in this way by clicking on the graphic button of the "Check Data-Aware Soundness" program the writing "Unlimited Data Petri Net" appears. The subsequent operations initialize a HashMap of tokens associated with the various places and a set of Places associated with the Places with tokens greater than 0 in the initial marking.

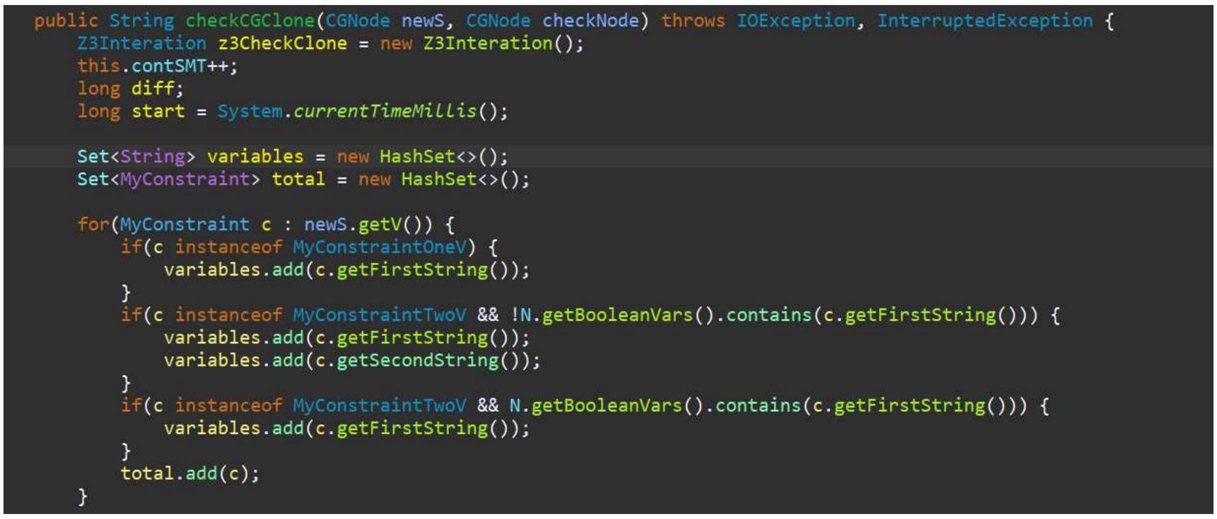
In this way it is possible to create the node C0, which is the initial node of the Constraint Graph and is also added to the Sets S and L.

The next step is a while loop that ends only when the node set of the Constraint Graph L is empty. Within the cycle, the current node is extracted and copied within a copy set to avoid passages by address of the objects present within it and to be able to carry out modification operations without affecting the original node. The visit of the DPN takes place by scrolling through the various arcs



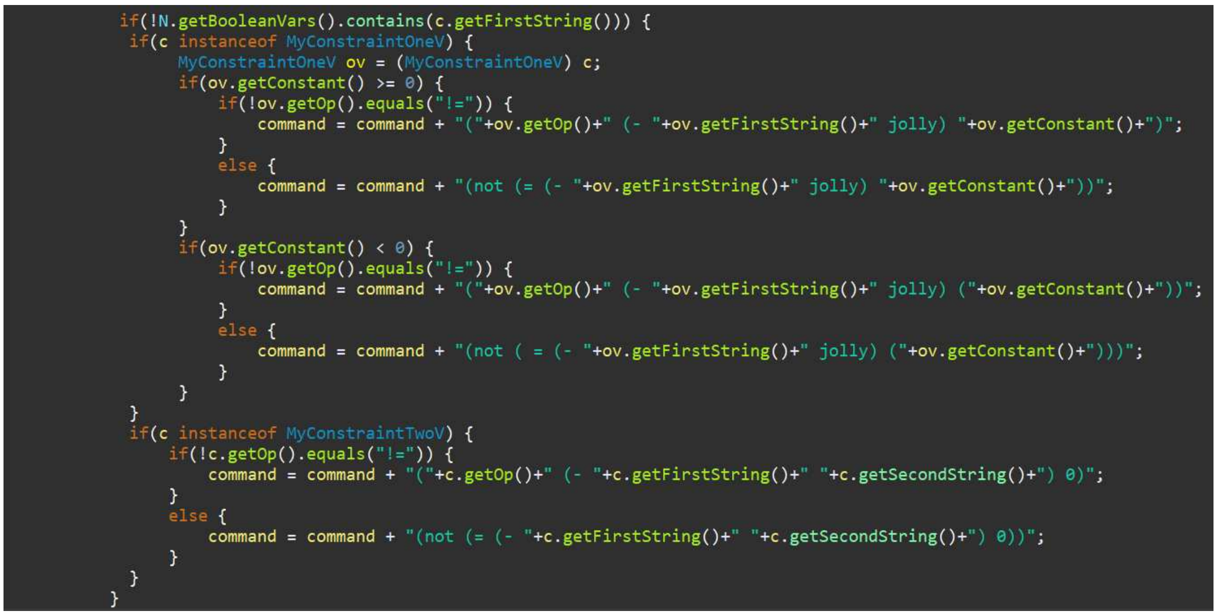
```
String command = "";
command = command + "(assert (not (= ";
int cont = 0;
for(MyConstraint c : newS.getV()) {
        cont++;
        if(cont <= newS.getV().size()-1) {
            command = command + "(and ";
        }

       if(!N.getBooleanVars().contains(c.getFirstString())) {
        if(c instanceof MyConstraintOneV) {
              MyConstraintOneV ov = (MyConstraintOneV) c;
              if(ov.getConstant() >= 0) {
                   if(!ov.getOp().equals("!=")) {
                        command = command + "("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+")";
                   }
                   else {
                        command = command + "(not (= (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+"))";
                   }
              }
              if(ov.getConstant() < 0) {
                   if(!ov.getOp().equals("!=")) {
                        command = command + "("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+"))";
                   }
                   else {
                        command = command + "(not ( = (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+")))";
                   }
              }
        }
```

Figure 7.8: The fourth part of the algorithm to check the equivalence of two nodes



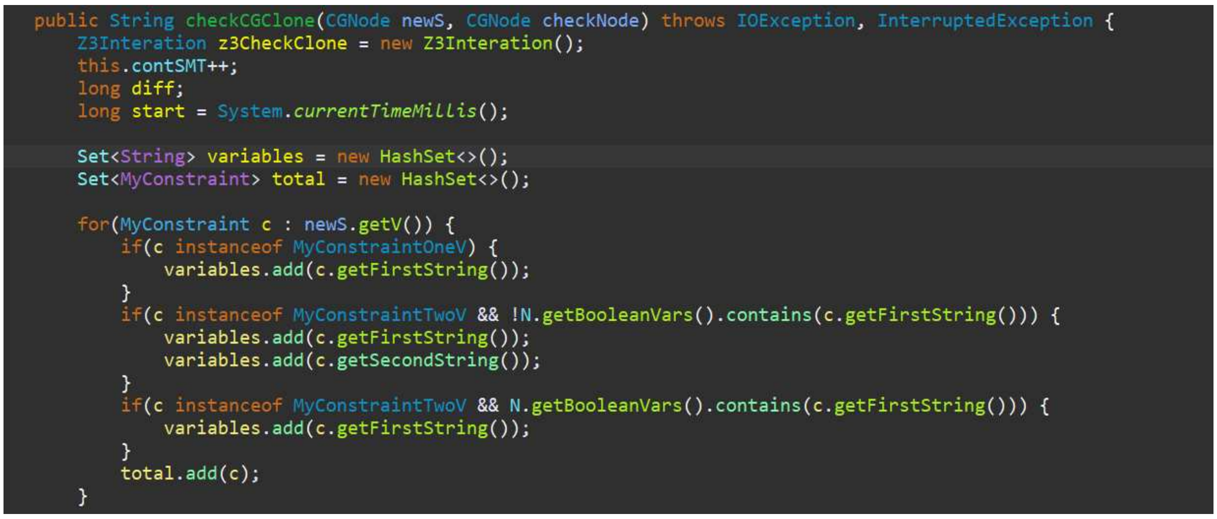
```
public String checkCGClone(CGNode newS, CGNode checkNode) throws IOException, InterruptedException {
    Z3Interation z3CheckClone = new Z3Interation();
    this.contSMT++;
    long diff;
    long start = System.currentTimeMillis();

    Set<String> variables = new HashSet<>();
    Set<MyConstraint> total = new HashSet<>();

    for(MyConstraint c : newS.getV()) {
        if(c instanceof MyConstraintOneV) {
            variables.add(c.getFirstString());
        }
        if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
            variables.add(c.getFirstString());
            variables.add(c.getSecondString());
        }
        if(c instanceof MyConstraintTwoV && N.getBooleanVars().contains(c.getFirstString())) {
            variables.add(c.getFirstString());
        }
        total.add(c);
    }
```

Figure 7.9: The fifth part of the algorithm to check the equivalence of two nodes



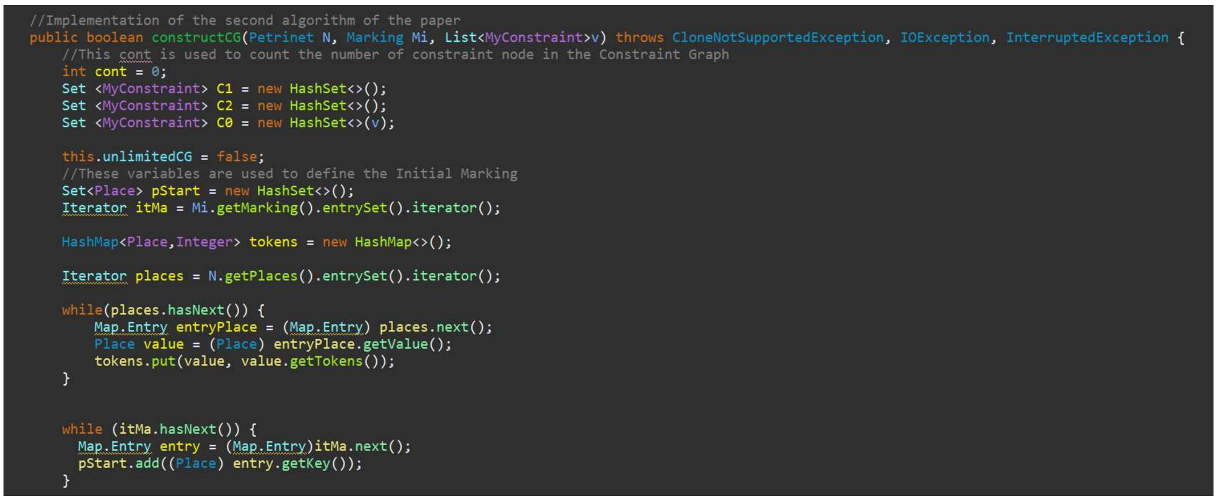
```
            if(!N.getBooleanVars().contains(c.getFirstString())) {
             if(c instanceof MyConstraintOneV) {
                  MyConstraintOneV ov = (MyConstraintOneV) c;
                  if(ov.getConstant() >= 0) {
                      if(!ov.getOp().equals("!=")) {
                          command = command + "("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+")";
                      }
                      else {
                          command = command + "(not (= (- "+ov.getFirstString()+" jolly) "+ov.getConstant()+"))";
                      }
                  }
                  if(ov.getConstant() < 0) {
                      if(!ov.getOp().equals("!=")) {
                          command = command + "("+ov.getOp()+" (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+"))";
                      }
                      else {
                          command = command + "(not ( = (- "+ov.getFirstString()+" jolly) ("+ov.getConstant()+")))";
                      }
                  }
             }
             if(c instanceof MyConstraintTwoV) {
                 if(!c.getOp().equals("!=")) {
                     command = command + "("+c.getOp()+" (- "+c.getFirstString()+" "+c.getSecondString()+") 0)";
                 }
                 else {
                     command = command + "(not (= (- "+c.getFirstString()+" "+c.getSecondString()+") 0))";
                 }
             }
           }
```

Figure 7.10: The sixth part of the algorithm to check the equivalence of two nodes



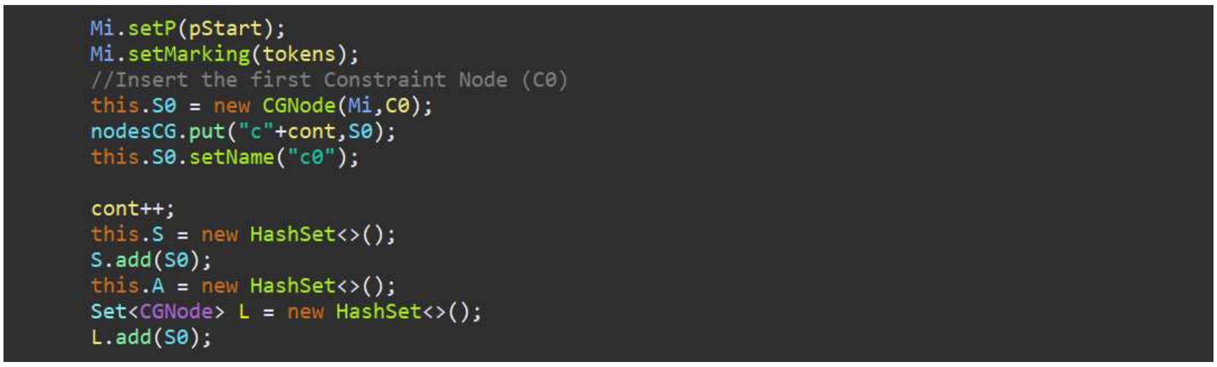
```
   public String checkCGClone(CGNode newS, CGNode checkNode) throws IOException, InterruptedException {
       Z3Interation z3CheckClone = new Z3Interation();
       this.contSMT++;
       long diff;
       long start = System.currentTimeMillis();

       Set<String> variables = new HashSet<>();
       Set<MyConstraint> total = new HashSet<>();

       for(MyConstraint c : newS.getV()) {
           if(c instanceof MyConstraintOneV) {
               variables.add(c.getFirstString());
           }
           if(c instanceof MyConstraintTwoV && !N.getBooleanVars().contains(c.getFirstString())) {
               variables.add(c.getFirstString());
               variables.add(c.getSecondString());
           }
           if(c instanceof MyConstraintTwoV && N.getBooleanVars().contains(c.getFirstString())) {
               variables.add(c.getFirstString());
           }
           total.add(c);
       }
```

Figure 7.11: The seventh part of the algorithm to check the equivalence of two nodes



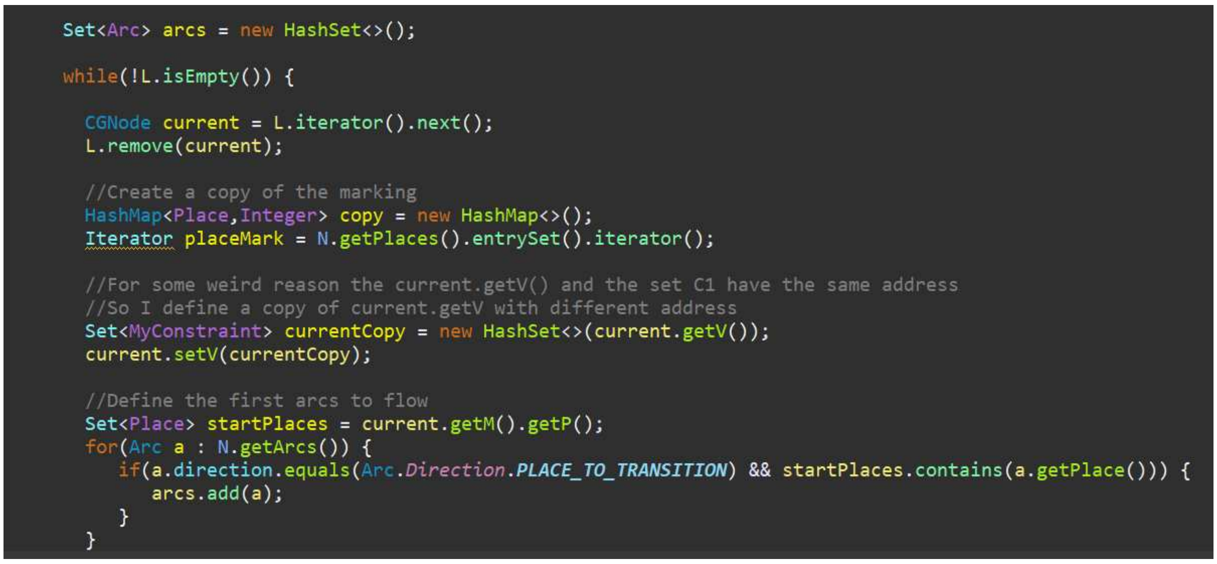
```java
//Implementation of the second algorithm of the paper
public boolean constructCG(Petrinet N, Marking Mi, List<MyConstraint>v) throws CloneNotSupportedException, IOException, InterruptedException {
    //This cont is used to count the number of constraint node in the Constraint Graph
    int cont = 0;
    Set <MyConstraint> C1 = new HashSet<>();
    Set <MyConstraint> C2 = new HashSet<>();
    Set <MyConstraint> C0 = new HashSet<>(v);

    this.unlimitedCG = false;
    //These variables are used to define the Initial Marking
    Set<Place> pStart = new HashSet<>();
    Iterator itMa = Mi.getMarking().entrySet().iterator();

    HashMap<Place,Integer> tokens = new HashMap<>();

    Iterator places = N.getPlaces().entrySet().iterator();

    while(places.hasNext()) {
        Map.Entry entryPlace = (Map.Entry) places.next();
        Place value = (Place) entryPlace.getValue();
        tokens.put(value, value.getTokens());
    }


    while (itMa.hasNext()) {
      Map.Entry entry = (Map.Entry)itMa.next();
      pStart.add((Place) entry.getKey());
    }
```

Figure 7.12: The first part of the algorithm to construct the Constraint Graph



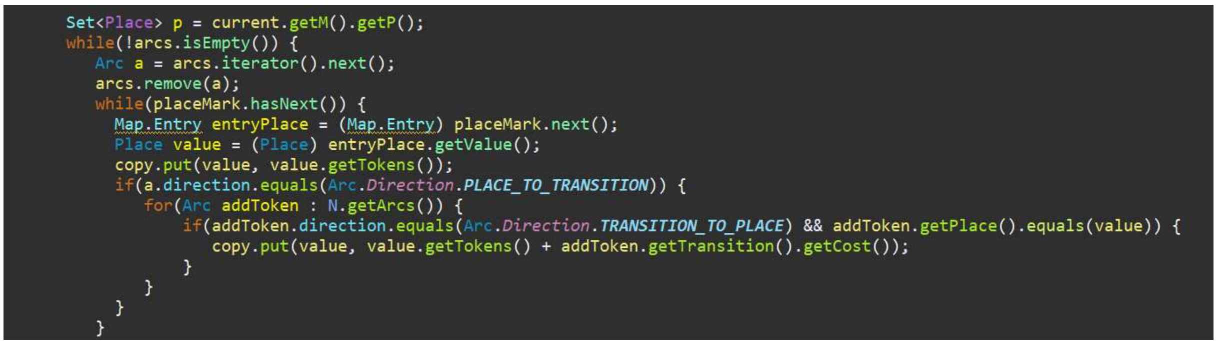
```java
    Mi.setP(pStart);
    Mi.setMarking(tokens);
    //Insert the first Constraint Node (C0)
    this.S0 = new CGNode(Mi,C0);
    nodesCG.put("c"+cont,S0);
    this.S0.setName("c0");

    cont++;
    this.S = new HashSet<>();
    S.add(S0);
    this.A = new HashSet<>();
    Set<CGNode> L = new HashSet<>();
    L.add(S0);
```

Figure 7.13: The second part of the algorithm to construct the Constraint Graph



```
Set<Arc> arcs = new HashSet<>();

while(!L.isEmpty()) {

  CGNode current = L.iterator().next();
  L.remove(current);

  //Create a copy of the marking
  HashMap<Place,Integer> copy = new HashMap<>();
  Iterator placeMark = N.getPlaces().entrySet().iterator();

  //For some weird reason the current.getV() and the set C1 have the same address
  //So I define a copy of current.getV with different address
  Set<MyConstraint> currentCopy = new HashSet<>(current.getV());
  current.setV(currentCopy);

  //Define the first arcs to flow
  Set<Place> startPlaces = current.getM().getP();
  for(Arc a : N.getArcs()) {
     if(a.direction.equals(Arc.Direction.PLACE_TO_TRANSITION) && startPlaces.contains(a.getPlace())) {
        arcs.add(a);
     }
  }
```

Figure 7.14: The third part of the algorithm to construct the Constraint Graph



```
Set<Place> p = current.getM().getP();
while(!arcs.isEmpty()) {
   Arc a = arcs.iterator().next();
   arcs.remove(a);
   while(placeMark.hasNext()) {
     Map.Entry entryPlace = (Map.Entry) placeMark.next();
     Place value = (Place) entryPlace.getValue();
     copy.put(value, value.getTokens());
     if(a.direction.equals(Arc.Direction.PLACE_TO_TRANSITION)) {
        for(Arc addToken : N.getArcs()) {
            if(addToken.direction.equals(Arc.Direction.TRANSITION_TO_PLACE) && addToken.getPlace().equals(value)) {
               copy.put(value, value.getTokens() + addToken.getTransition().getCost());
            }
        }
     }
   }
```

Figure 7.15: The fourth part of the algorithm to construct the Constraint Graph

present within N, for this reason the set of arcs takes as its initial elements the arcs that start from an initial Place.

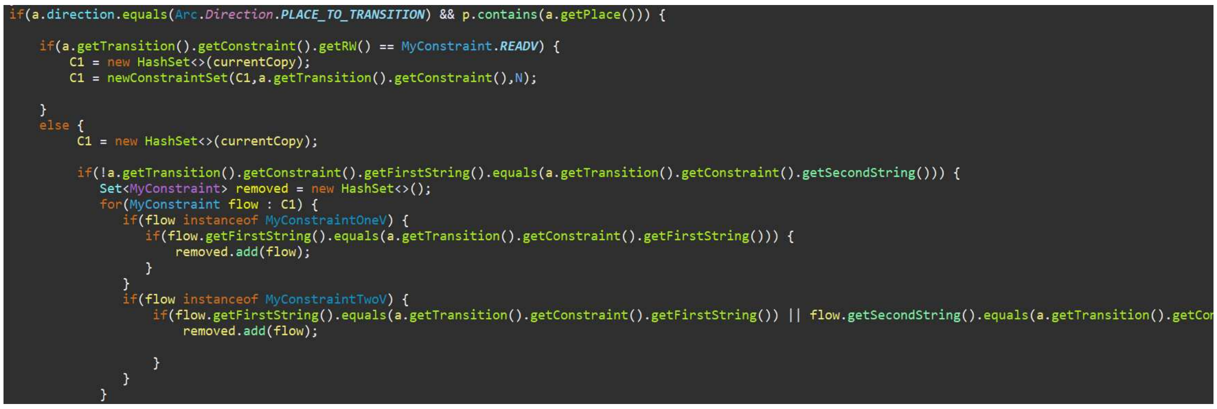
Then another while loop is opened which will repeat as long as there are arcs within the set arcs. Each iteration extracts the current arc and extracts the markup and copy inside copy.

Subsequently, it is verified whether the arc is of type Place to Transition and if Set p contains the place pointed to by the arc. If the condition is true, then it is verified that the constraint associated with the transition of the arc is in reading or writing. In the first case a new set C1 is simply created and the new transition constraint is added with *newConstraintSet()*. In the case of writing instead, set C1 is created with a copy of the constraints of the current Set. subsequently, if the constraint does not fall within the particular case of v op v, whereby the first and second variable are the same variable, then all the constraints containing the variable being written are removed.

Also at the end of the write operation the new constraint is added with *newConstraintSet()*. Next we define Set C2 initialized to the constraint set of the current node. Subsequently, the negation of the constraint of the transition to C2 is added if the operation is in read and the variable is not boolean. Neg's RW flag is also set to READV to prevent the neg flag from being set to write for any reason.

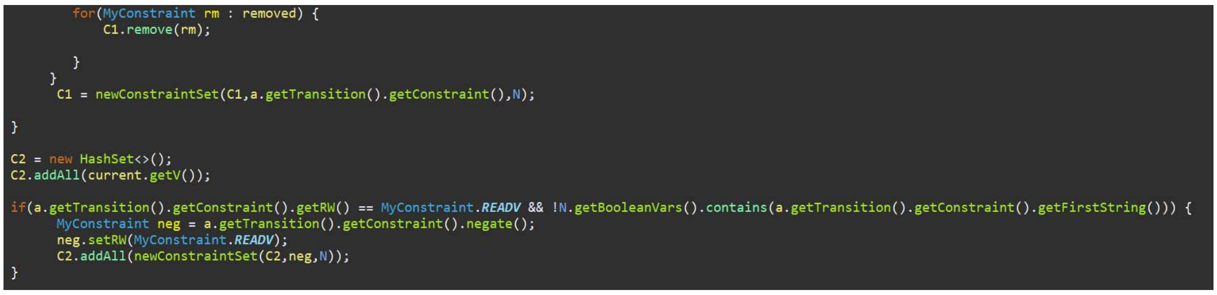
The subsequent operations instead deal with the boolean variables and therefore the variables are contained within N.getBooleanVars(). Constraints are added to C1, but if they are variable != true or variable != false they are converted to variable = false and variable = true respectively. Then the previous modified constraints are removed from C1.

Subsequently, it is verified whether the Set of constraints C1 is satisfiable and if it is true, a flag is defined initially initialized to true and then possibly modified later in case of a counterexample.



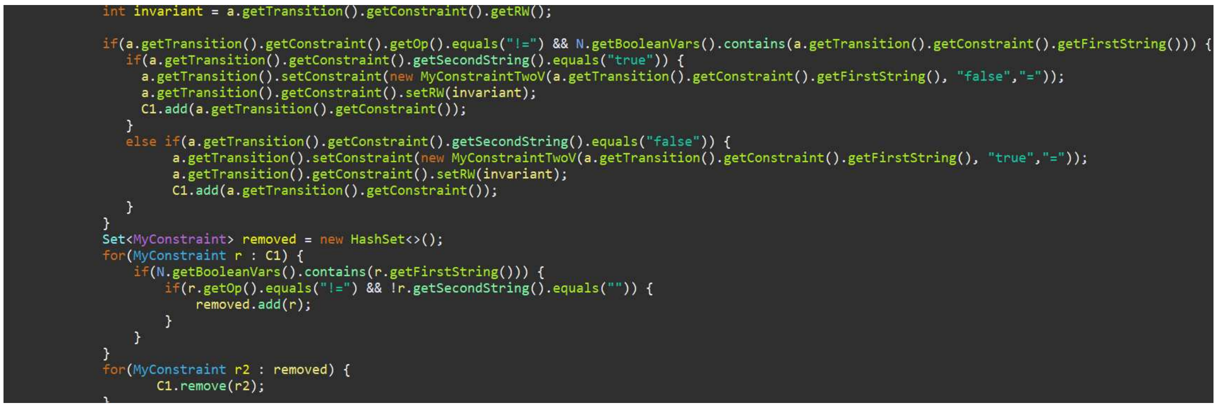
```
if(a.direction.equals(Arc.Direction.PLACE_TO_TRANSITION) && p.contains(a.getPlace())) {

    if(a.getTransition().getConstraint().getRW() == MyConstraint.READV) {
        C1 = new HashSet<>(currentCopy);
        C1 = newConstraintSet(C1,a.getTransition().getConstraint(),N);

    }
    else {
         C1 = new HashSet<>(currentCopy);

         if(!a.getTransition().getConstraint().getFirstString().equals(a.getTransition().getConstraint().getSecondString())) {
            Set<MyConstraint> removed = new HashSet<>();
            for(MyConstraint flow : C1) {
               if(flow instanceof MyConstraintOneV) {
                  if(flow.getFirstString().equals(a.getTransition().getConstraint().getFirstString())) {
                      removed.add(flow);
                  }
               }
               if(flow instanceof MyConstraintTwoV) {
                   if(flow.getFirstString().equals(a.getTransition().getConstraint().getFirstString()) || flow.getSecondString().equals(a.getTransition().getCon
                       removed.add(flow);

                   }
               }
            }
```

Figure 7.16: The fifth part of the algorithm to construct the Constraint Graph



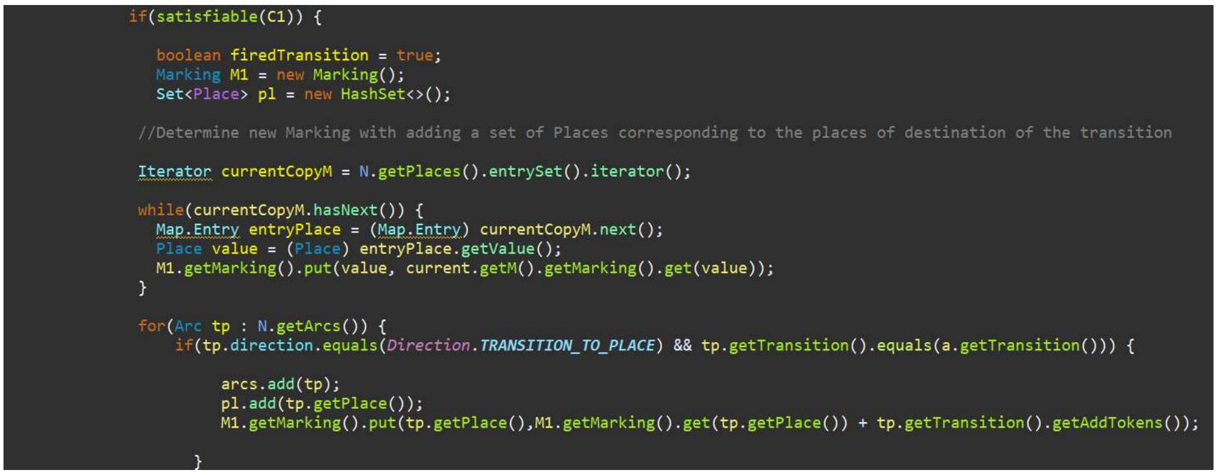
```
        for(MyConstraint rm : removed) {
            C1.remove(rm);

        }
      }
       C1 = newConstraintSet(C1,a.getTransition().getConstraint(),N);

}

C2 = new HashSet<>();
C2.addAll(current.getV());

if(a.getTransition().getConstraint().getRW() == MyConstraint.READV && !N.getBooleanVars().contains(a.getTransition().getConstraint().getFirstString())) {
       MyConstraint neg = a.getTransition().getConstraint().negate();
       neg.setRW(MyConstraint.READV);
       C2.addAll(newConstraintSet(C2,neg,N));
}
```

Figure 7.17: The sixth part of the algorithm to construct the Constraint Graph



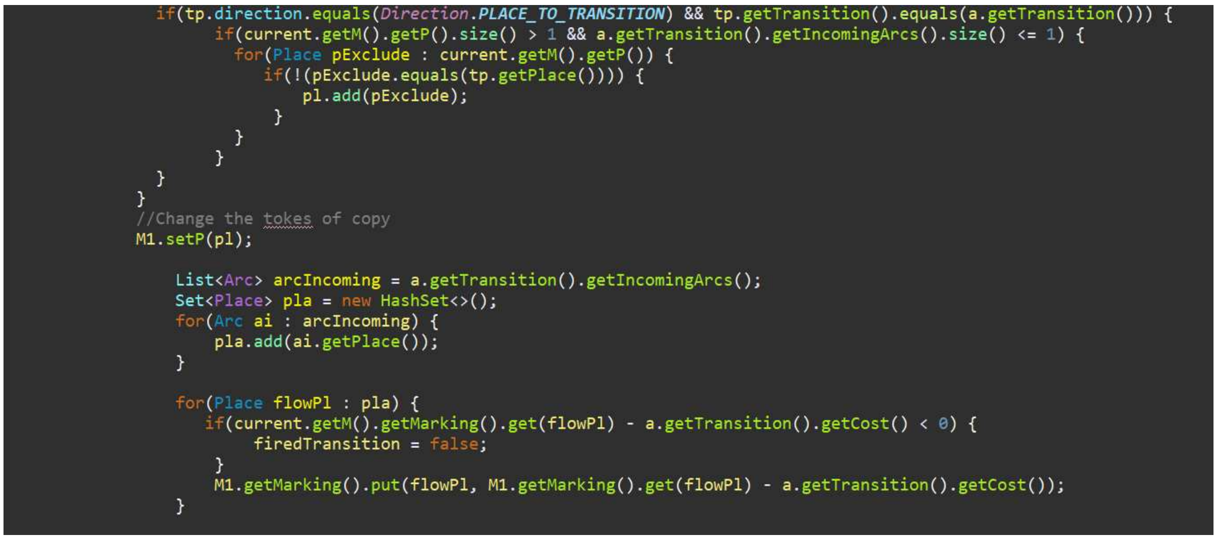
```
            int invariant = a.getTransition().getConstraint().getRW();

            if(a.getTransition().getConstraint().getOp().equals("!=") && N.getBooleanVars().contains(a.getTransition().getConstraint().getFirstString())) {
               if(a.getTransition().getConstraint().getSecondString().equals("true")) {
                 a.getTransition().setConstraint(new MyConstraintTwoV(a.getTransition().getConstraint().getFirstString(), "false","="));
                 a.getTransition().getConstraint().setRW(invariant);
                 C1.add(a.getTransition().getConstraint());
               }
               else if(a.getTransition().getConstraint().getSecondString().equals("false")) {
                     a.getTransition().setConstraint(new MyConstraintTwoV(a.getTransition().getConstraint().getFirstString(), "true","="));
                     a.getTransition().getConstraint().setRW(invariant);
                     C1.add(a.getTransition().getConstraint());
               }
            }
            Set<MyConstraint> removed = new HashSet<>();
            for(MyConstraint r : C1) {
                if(N.getBooleanVars().contains(r.getFirstString())) {
                    if(r.getOp().equals("!=") && !r.getSecondString().equals("")) {
                        removed.add(r);
                    }
                }
            }
            for(MyConstraint r2 : removed) {
                   C1.remove(r2);
```

Figure 7.18: The seventh part of the algorithm to construct the Constraint Graph



```
if(satisfiable(C1)) {

   boolean firedTransition = true;
   Marking M1 = new Marking();
   Set<Place> pl = new HashSet<>();

 //Determine new Marking with adding a set of Places corresponding to the places of destination of the transition

 Iterator currentCopyM = N.getPlaces().entrySet().iterator();

 while(currentCopyM.hasNext()) {
   Map.Entry entryPlace = (Map.Entry) currentCopyM.next();
   Place value = (Place) entryPlace.getValue();
   M1.getMarking().put(value, current.getM().getMarking().get(value));
 }

 for(Arc tp : N.getArcs()) {
     if(tp.direction.equals(Direction.TRANSITION_TO_PLACE) && tp.getTransition().equals(a.getTransition())) {

          arcs.add(tp);
          pl.add(tp.getPlace());
          M1.getMarking().put(tp.getPlace(),M1.getMarking().get(tp.getPlace()) + tp.getTransition().getAddTokens());

       }
```

Figure 7.19: The eigth part of the algorithm to construct the Constraint Graph



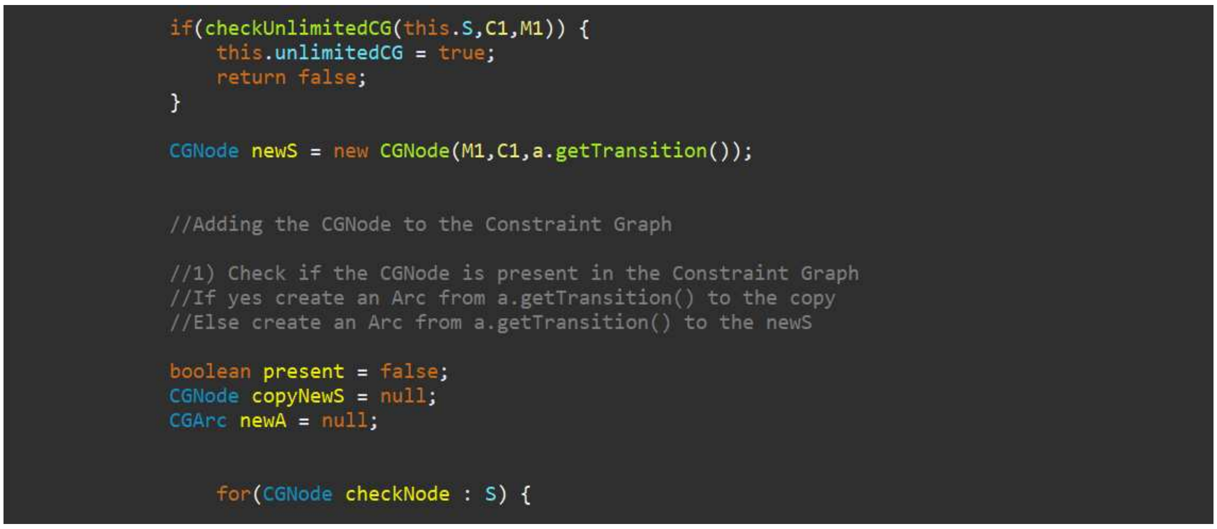
```
  if(tp.direction.equals(Direction.PLACE_TO_TRANSITION) && tp.getTransition().equals(a.getTransition())) {
        if(current.getM().getP().size() > 1 && a.getTransition().getIncomingArcs().size() <= 1) {
          for(Place pExclude : current.getM().getP()) {
             if(!(pExclude.equals(tp.getPlace()))) {
                 pl.add(pExclude);
              }
          }
        }
  }
}
//Change the tokes of copy
M1.setP(pl);

    List<Arc> arcIncoming = a.getTransition().getIncomingArcs();
    Set<Place> pla = new HashSet<>();
    for(Arc ai : arcIncoming) {
        pla.add(ai.getPlace());
    }

    for(Place flowPl : pla) {
       if(current.getM().getMarking().get(flowPl) - a.getTransition().getCost() < 0) {
            firedTransition = false;
        }
        M1.getMarking().put(flowPl, M1.getMarking().get(flowPl) - a.getTransition().getCost());
    }
```

Figure 7.20: The nineth part of the algorithm to construct the Constraint Graph

The marking M1 and the Set of Place pl are also defined in order to be able to define the marking of the new node that will eventually have to be created. The initial numbering is initially initialized to a copy of the current node's numbering. Then we start to iterate all the arcs and the first case occurs: if the arc is of type transition to place and the transition corresponds to that of the arc then the new arc is added to the set of arcs to be iterated in the previous while loop , the place of the arc is added to the set pl and finally the marking of M1 is updated by adding the number of tokens it takes from the transition to which it is connected.

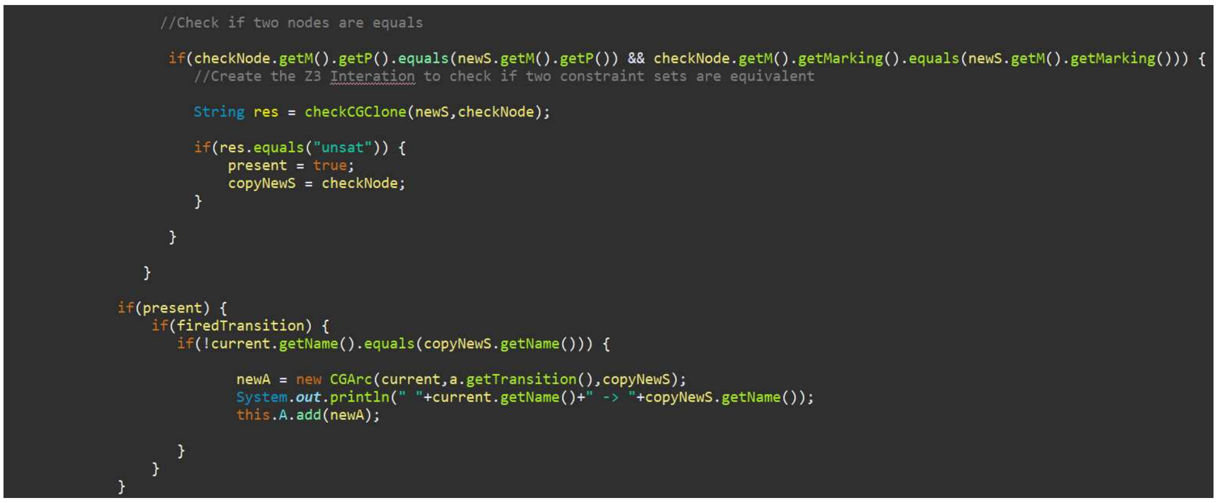
The second case instead is to check if the arc is of the place to transition type and if the transition associated with the arc is the same as the transition of the current node. In that case it is necessary to check if the transition is able to trigger and therefore it is necessary to see if all the places that have an arc with destination in that transition have a number of tokens greater than or equal to the cost required by the transition. If even just one of these places does not have a sufficient number of tokens, then the firedtransition flag is set to false. The marking of the new node will have the values decremented by the cost of its transition into the previously defined incoming places.

The next step is to check through the checkUnlimitedCG function if the Constraint Graph is unlimited and therefore it will have to return false to avoid an infinite loop. If this is not the case, then a new node is created with a current transition, the marking M1 and the set of nodes C1, however to be included in the Set of nodes S it is necessary to verify that the node is not already present (i.e. has



```
        if(checkUnlimitedCG(this.S,C1,M1)) {
            this.unlimitedCG = true;
            return false;
        }

        CGNode newS = new CGNode(M1,C1,a.getTransition());


        //Adding the CGNode to the Constraint Graph

        //1) Check if the CGNode is present in the Constraint Graph
        //If yes create an Arc from a.getTransition() to the copy
        //Else create an Arc from a.getTransition() to the newS

        boolean present = false;
        CGNode copyNewS = null;
        CGArc newA = null;


            for(CGNode checkNode : S) {
```

Figure 7.21: The tenth part of the algorithm to construct the Constraint Graph



```
            //Check if two nodes are equals

             if(checkNode.getM().getP().equals(newS.getM().getP()) && checkNode.getM().getMarking().equals(newS.getM().getMarking())) {
                //Create the Z3 Interation to check if two constraint sets are equivalent

                String res = checkCGClone(newS,checkNode);

                if(res.equals("unsat")) {
                    present = true;
                    copyNewS = checkNode;
                }

             }

         }

    if(present) {
        if(firedTransition) {
           if(!current.getName().equals(copyNewS.getName())) {

                newA = new CGArc(current,a.getTransition(),copyNewS);
                System.out.println(" "+current.getName()+" -> "+copyNewS.getName());
                this.A.add(newA);

           }
        }
    }
```

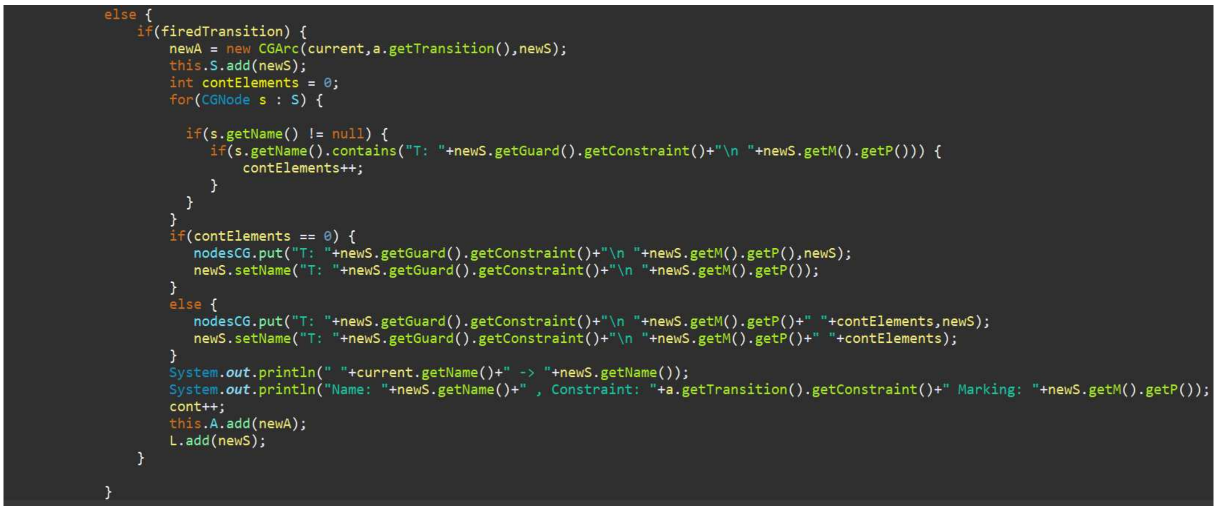
Figure 7.22: The eleventh part of the algorithm to construct the Constraint Graph

the same set of constraints taking into account the constraints involved, but also the marking itself). In this regard, the present flag is used, initially set to false and which will eventually become true if a counterexample is found. We then start scrolling through all the nodes of the Constraint Graph already built in S.

If the current node in S and the newly created one have the same marking and *checkCGClone()* verifies that they also have the same set of constraints (thus the result of checkCGClone must be "unsat"), then the flag present is set to true.

If the present flag is true, it checks if the firedTransition flag is also true and if so, it checks that the name of the current node and the new one do not have the same name and the arc is created between the two nodes already existing.

If it is not present, the new arc is created, the new node inserted in Set S and the contElements flag is used to avoid nodes with the same name: in fact, in the graphic representation for each node of the constraint graph only the guard constraint and place names having tokens greater than zero, so in case of nodes with the same guard, same places with tokens greater than zero, but with a different number of tokens between the places would give rise to a single node in the visualization of the Constraint Graph. Finally the new node is added to Set L which will be subsequently extracted in one of the next iterations of while(!L.isEmpty()).



```
else {
    if(firedTransition) {
        newA = new CGArc(current,a.getTransition(),newS);
        this.S.add(newS);
        int contElements = 0;
        for(CGNode s : S) {

          if(s.getName() != null) {
             if(s.getName().contains("T: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP())) {
                 contElements++;
             }
          }
        }
        if(contElements == 0) {
           nodesCG.put("T: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP(),newS);
           newS.setName("T: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP());
        }
        else {
           nodesCG.put("T: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP()+" "+contElements,newS);
           newS.setName("T: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP()+" "+contElements);
        }
        System.out.println(" "+current.getName()+" -> "+newS.getName());
        System.out.println("Name: "+newS.getName()+" , Constraint: "+a.getTransition().getConstraint()+" Marking: "+newS.getM().getP());
        cont++;
        this.A.add(newA);
        L.add(newS);
    }

}
```

Figure 7.23: The twelveth part of the algorithm to construct the Constraint Graph

If C1 is not satisfiable, then the negation of the constraint is added to the current set in order to make the comparison between set C2 and the current set true. In fact, if the Set of constraints should not be satisfiable then automatically the negation of the constraint is an implication of the current set of constraints and also the marking would remain the same.

The next part instead concerns the management of the boolean variables as occurred in a similar way to the case of C1.

It then checks that C2 is satisfiable, that the constraint set C2 is different from the current one, and that the transition *flagRW* is not set to write. In the event that these three conditions are met, the new node is created but before inserting it into S, it is checked, as done in the case of C1, if a similar node does not already exist.

Then, as in the case of C1, the same operations are carried out depending on whether the node is already present or not.

Finally, the last part of this method is to run the analyzeConstraintGraph function to tell the program if the constraint graph is sound:

7.4 Verification of Data-Aware Soundness properties

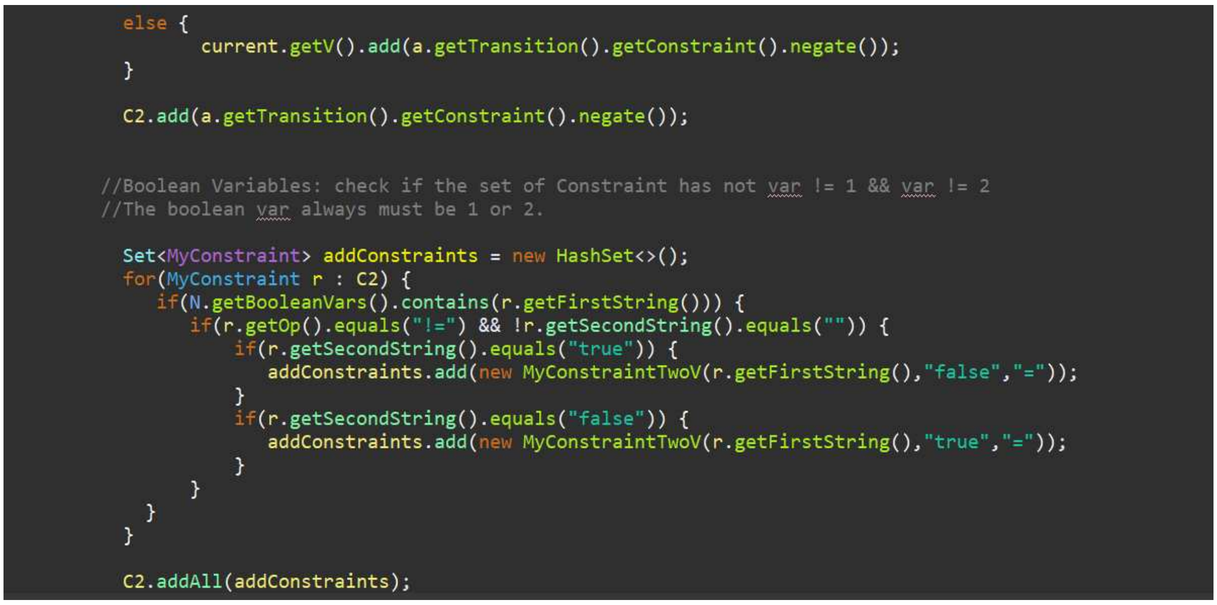
In the pseudocode presented in chapter 6.1, reference is made to the *AnalyzeConstraintGraph()* function, in which the three data-aware soundness properties are checked, which we summarize as:

- $\forall(\mathrm{M}, \alpha) \in \mathrm{Reach}_N \,.\, \exists\, \alpha' \,.\, (\mathrm{M}, \alpha) \rightarrow^* (\mathrm{M}_f, \alpha')$;
- $\forall(\mathrm{M}, \alpha) \in \mathrm{Reach}_N \,.\, \mathrm{M}\ ge\ \mathrm{M}_f \implies (\mathrm{M} = \mathrm{M}_f)$;
- $\forall \mathrm{t} \in \mathrm{T}.\ \exists \mathrm{M}_1,\mathrm{M}_2, \alpha 1, \alpha 2, \beta$, $(\mathrm{M1},\alpha 1) \in \mathrm{Reach}_N$ e $(\mathrm{M}_1, \alpha_1) \rightarrow (\mathrm{M}_2, \alpha_2)$ attraverso (t,β).

These properties are checked individually and if even one of these three properties is false then the whole system will not be data-aware sound. The other situation that could occur is that a Constraint Graph has an infinite number of nodes and for which the algorithm must automatically return the result *false*. In that case the condition occurs:

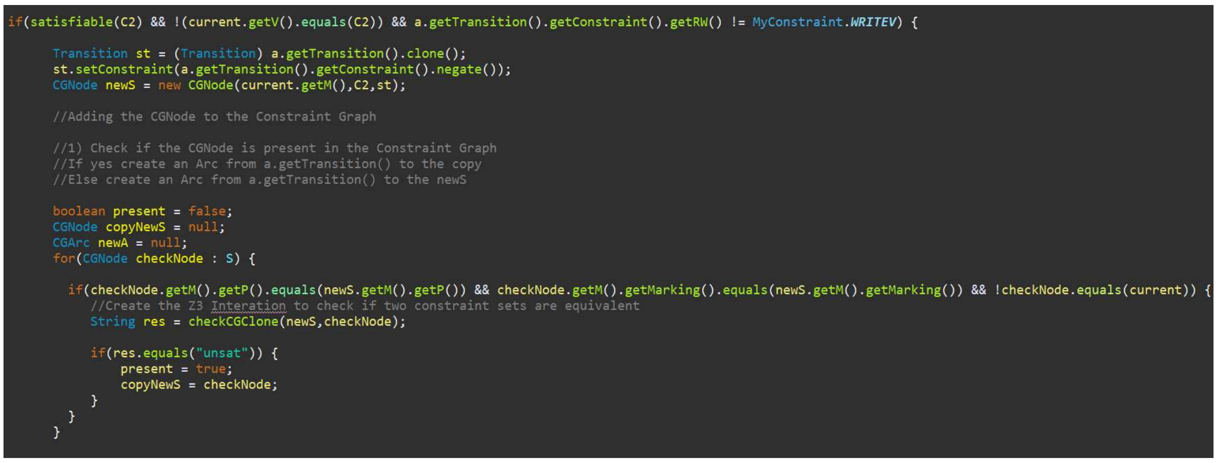
- $\exists(\overline{M}, \overline{\mathrm{C}} \in \mathrm{S}\ \mathrm{s.t}\ \mathrm{M}' > \overline{M} \wedge \mathrm{C}' = \overline{\mathrm{C}})$

As already seen in the pseudocode in chapter 6.1 .



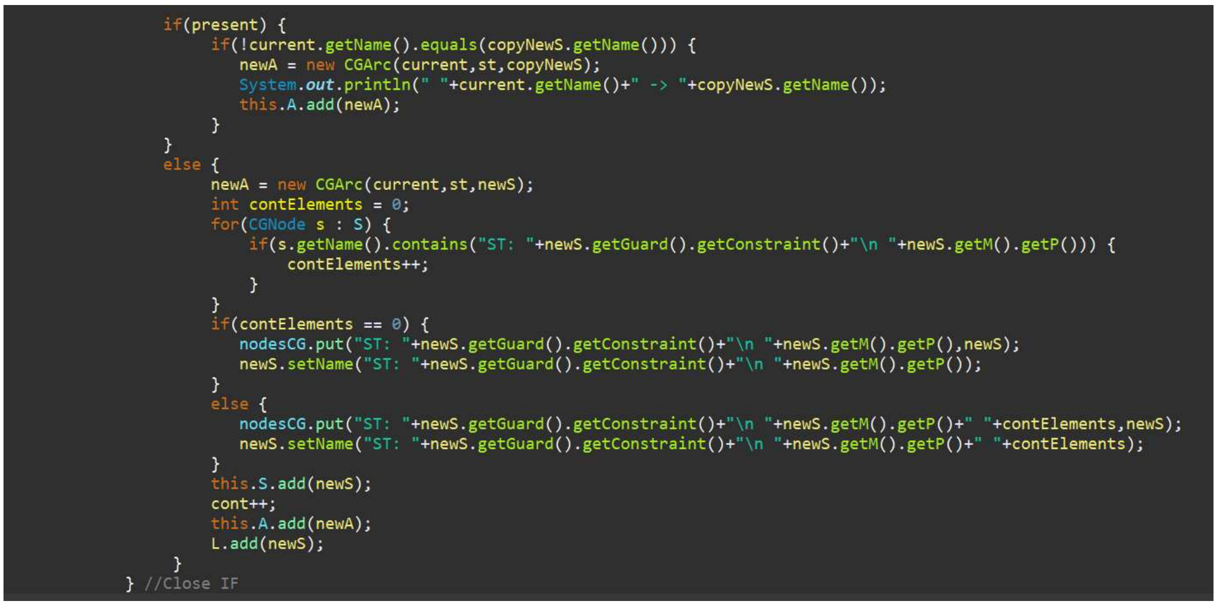
```
        else {
                current.getV().add(a.getTransition().getConstraint().negate());
        }

        C2.add(a.getTransition().getConstraint().negate());


      //Boolean Variables: check if the set of Constraint has not var != 1 && var != 2
      //The boolean var always must be 1 or 2.

        Set<MyConstraint> addConstraints = new HashSet<>();
        for(MyConstraint r : C2) {
           if(N.getBooleanVars().contains(r.getFirstString())) {
              if(r.getOp().equals("!=") && !r.getSecondString().equals("")) {
                  if(r.getSecondString().equals("true")) {
                     addConstraints.add(new MyConstraintTwoV(r.getFirstString(),"false","="));
                  }
                  if(r.getSecondString().equals("false")) {
                     addConstraints.add(new MyConstraintTwoV(r.getFirstString(),"true","="));
                  }
              }
           }
        }

        C2.addAll(addConstraints);
```

Figure 7.24: The thirteenth part of the algorithm to construct the Constraint Graph



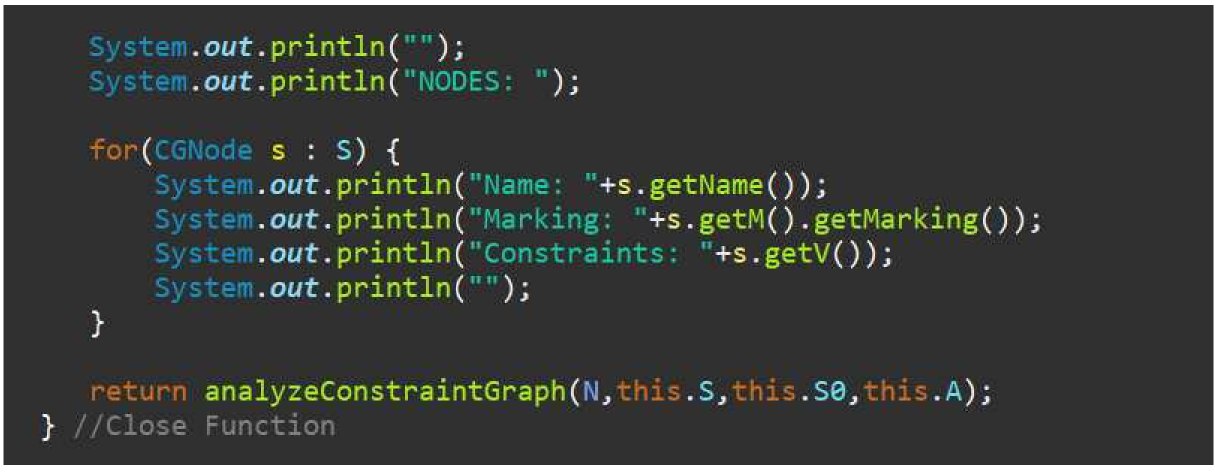
```
if(satisfiable(C2) && !(current.getV().equals(C2)) && a.getTransition().getConstraint().getRW() != MyConstraint.WRITEV) {

      Transition st = (Transition) a.getTransition().clone();
      st.setConstraint(a.getTransition().getConstraint().negate());
      CGNode newS = new CGNode(current.getM(),C2,st);

      //Adding the CGNode to the Constraint Graph

      //1) Check if the CGNode is present in the Constraint Graph
      //If yes create an Arc from a.getTransition() to the copy
      //Else create an Arc from a.getTransition() to the newS

      boolean present = false;
      CGNode copyNewS = null;
      CGArc newA = null;
      for(CGNode checkNode : S) {

        if(checkNode.getM().getP().equals(newS.getM().getP()) && checkNode.getM().getMarking().equals(newS.getM().getMarking()) && !checkNode.equals(current)) {
           //Create the Z3 Interation to check if two constraint sets are equivalent
           String res = checkCGClone(newS,checkNode);

           if(res.equals("unsat")) {
               present = true;
               copyNewS = checkNode;
           }
        }
      }
```

Figure 7.25: The fourteenth part of the algorithm to construct the Constraint Graph



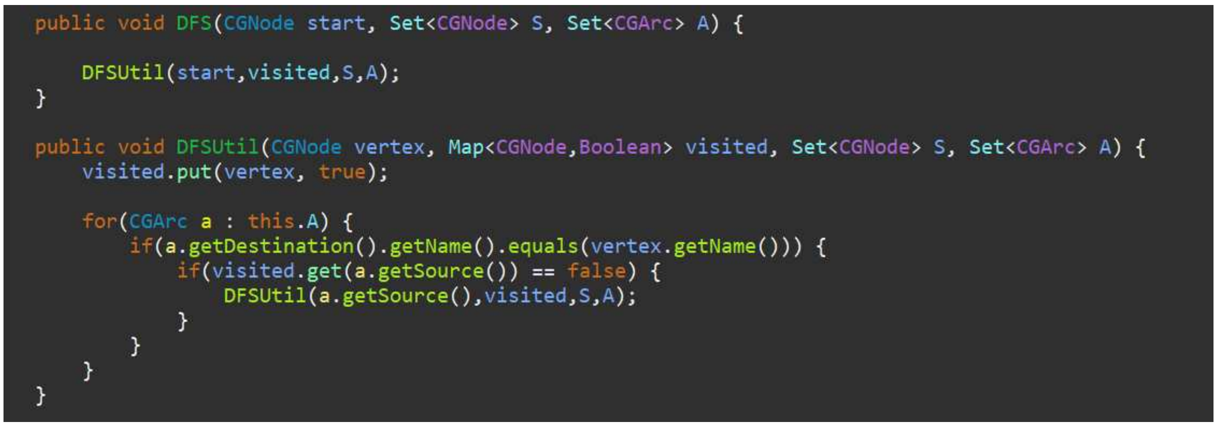
```
        if(present) {
            if(!current.getName().equals(copyNewS.getName())) {
               newA = new CGArc(current,st,copyNewS);
               System.out.println(" "+current.getName()+" -> "+copyNewS.getName());
               this.A.add(newA);
            }
        }
        else {
            newA = new CGArc(current,st,newS);
            int contElements = 0;
            for(CGNode s : S) {
                if(s.getName().contains("ST: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP())) {
                    contElements++;
                }
            }
            if(contElements == 0) {
               nodesCG.put("ST: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP(),newS);
               newS.setName("ST: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP());
            }
            else {
               nodesCG.put("ST: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP()+" "+contElements,newS);
               newS.setName("ST: "+newS.getGuard().getConstraint()+"\n "+newS.getM().getP()+" "+contElements);
            }
            this.S.add(newS);
            cont++;
            this.A.add(newA);
            L.add(newS);
         }
    } //Close IF
```

Figure 7.26: The fifteenth part of the algorithm to construct the Constraint Graph



```
    System.out.println("");
    System.out.println("NODES: ");

    for(CGNode s : S) {
        System.out.println("Name: "+s.getName());
        System.out.println("Marking: "+s.getM().getMarking());
        System.out.println("Constraints: "+s.getV());
        System.out.println("");
    }

    return analyzeConstraintGraph(N,this.S,this.S0,this.A);
} //Close Function
```

Figure 7.27: The sixteenth part of the algorithm to construct the Constraint Graph



```
public void DFS(CGNode start, Set<CGNode> S, Set<CGArc> A) {

    DFSUtil(start,visited,S,A);
}

public void DFSUtil(CGNode vertex, Map<CGNode,Boolean> visited, Set<CGNode> S, Set<CGArc> A) {
    visited.put(vertex, true);

    for(CGArc a : this.A) {
        if(a.getDestination().getName().equals(vertex.getName())) {
            if(visited.get(a.getSource()) == false) {
                DFSUtil(a.getSource(),visited,S,A);
            }
        }
    }
}
```

Figure 7.28: Implementation of DFS Algorithm

7.4.1 Property 1

The first property $(\forall(M, \alpha) \in Reach_N \,.\, M \; ge \; M_f \implies (M = M_f))$ it is verified using the *Depth First Search* algorithm, i.e. the depth search algorithm on a graph as Cormen et al. shows in [6]

The DFS algorithm is implemented using two methods. These two methods implement the depth-visitation algorithm of a graph starting from an initial node. In this case, however, a modification has been made: in fact, in the original algorithm the initial node is the root one and the exploration continues up to the last nodes, called leaves, instead in this case the final nodes (without outgoing arcs) are the nodes and continue the visit up to the initial nodes.

After the DFS algorithm it is possible to implement the *checkProperty1()* method which implements the verification of property 1 of Data-Aware Soundness, ie it verifies that it is possible to arrive at a node with final marking for any possible path of the graph. We use the set visited (global variable within the class) to insert all the nodes that have already been visited and we use the previously defined DFS algorithm to fill the elements of the set visited. In the final part it checks that all elements of visited are set to true, otherwise false is returned.

7.4.2 Property 2

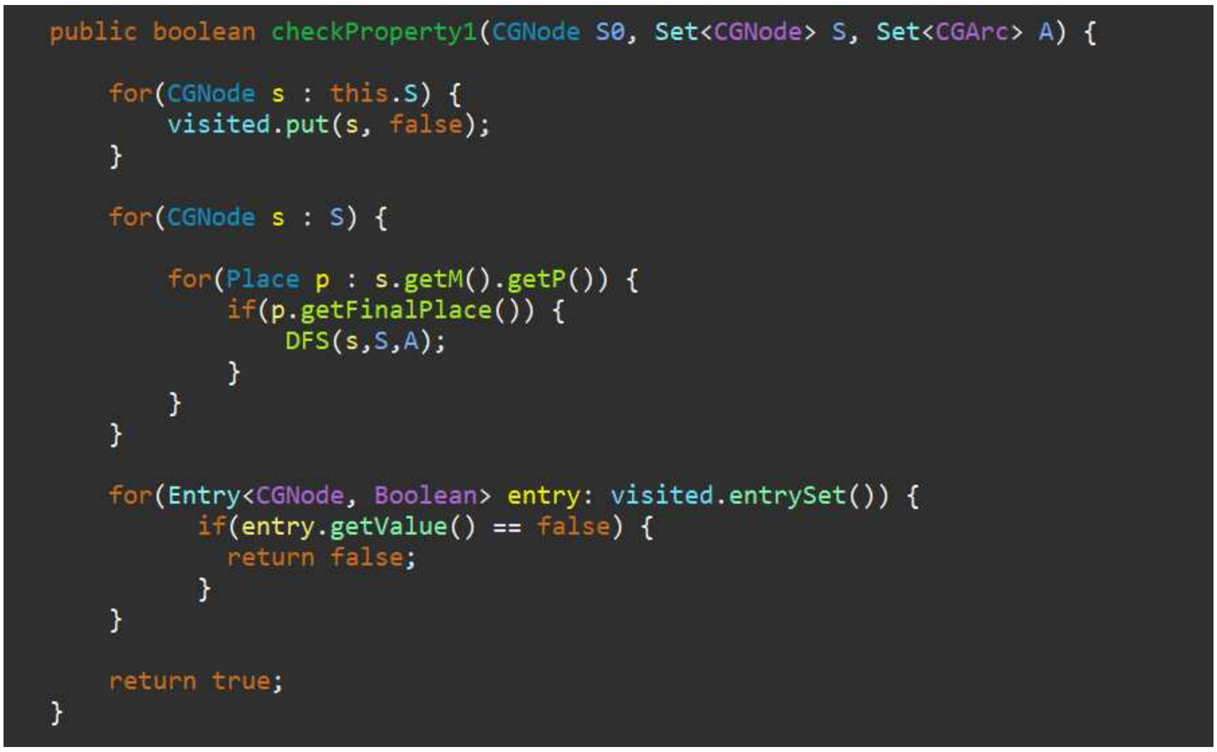
Property 2 $(\forall(M, \alpha) \in Reach_N \,.\, M \; ge \; M_f \implies (M = M_f))$ it is implemented using a function that allows to obtain the final marking. This function, given a set of nodes of the Constraint Graph and a set of arcs, creates a new set of nodes which contains the final nodes of the Constraint Graph, within which there are also the final markings.

Then the *checkProperty2()* function is implemented. This function takes as input the Petri net N and the Node set S which represents the created nodes of the Constraint Graph. For each node of S the comparison with the final marking is made. The first if inside the while loop checks if the number of tokens inside each CGS Place is less than that of the final markup. If even only one case should be true, it means that the condition for which M must be greater than or equal to Mf is not respected, therefore in this the while loop is interrupted and one moves on to the next CGS node. If it is only equal, a counter is increased which indicates how many Places of the marking are valid for respecting property 2. If instead it is greater, the counter is increased, but a flag is also set to 1 to indicate that the CGS marking can actually be greater than or equal to the final one.

Finally, the next part provides for the check that property 2 is violated by verifying three conditions jointly with the logical operation of AND:

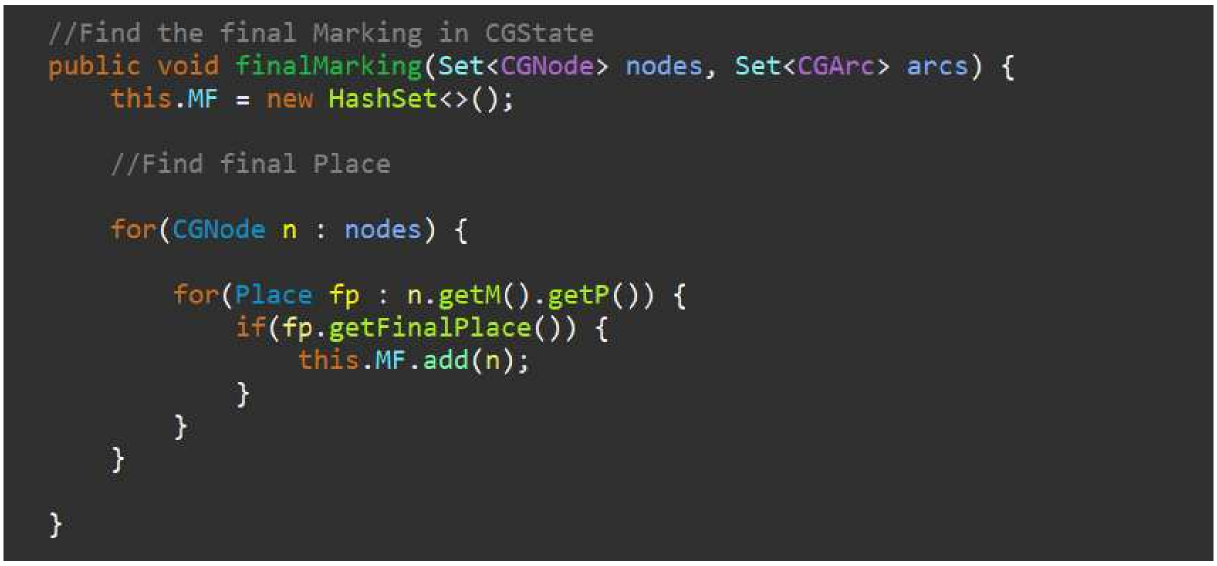
- That the counter is equal to the size of the mark, otherwise it means that the cycle would have terminated early and consequently it is not greater than or equal;
- Che flagMax = 1, which indicates that at least one token greater than the final markup is present.
- That the Places with values greater than 0 in CGS and in the final markup are equivalent

If all 3 of these conditions are true then false is returned, otherwise true is returned.



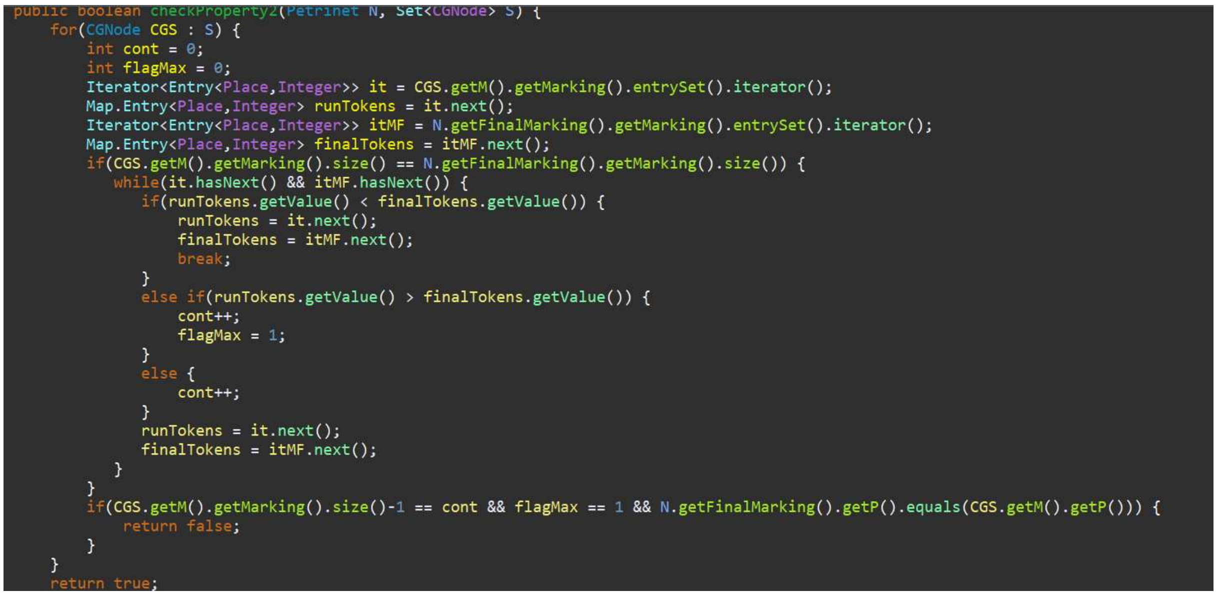
```
public boolean checkProperty1(CGNode S0, Set<CGNode> S, Set<CGArc> A) {

    for(CGNode s : this.S) {
        visited.put(s, false);
    }

    for(CGNode s : S) {

        for(Place p : s.getM().getP()) {
            if(p.getFinalPlace()) {
                DFS(s,S,A);
            }
        }
    }

    for(Entry<CGNode, Boolean> entry: visited.entrySet()) {
          if(entry.getValue() == false) {
            return false;
          }
    }

    return true;
}
```

Figure 7.29: The Implementation of the algorithm to check Property 1 of Data-Aware Soundness



```
//Find the final Marking in CGState
public void finalMarking(Set<CGNode> nodes, Set<CGArc> arcs) {
    this.MF = new HashSet<>();

    //Find final Place

    for(CGNode n : nodes) {

        for(Place fp : n.getM().getP()) {
            if(fp.getFinalPlace()) {
                this.MF.add(n);
            }
        }
    }

}
```

Figure 7.30: The algorithm to get the Final Marking of DPN



```
public boolean checkProperty2(Petrinet N, Set<CGNode> S) {
    for(CGNode CGS : S) {
        int cont = 0;
        int flagMax = 0;
        Iterator<Entry<Place,Integer>> it = CGS.getM().getMarking().entrySet().iterator();
        Map.Entry<Place,Integer> runTokens = it.next();
        Iterator<Entry<Place,Integer>> itMF = N.getFinalMarking().getMarking().entrySet().iterator();
        Map.Entry<Place,Integer> finalTokens = itMF.next();
        if(CGS.getM().getMarking().size() == N.getFinalMarking().getMarking().size()) {
           while(it.hasNext() && itMF.hasNext()) {
              if(runTokens.getValue() < finalTokens.getValue()) {
                  runTokens = it.next();
                  finalTokens = itMF.next();
                  break;
              }
              else if(runTokens.getValue() > finalTokens.getValue()) {
                  cont++;
                  flagMax = 1;
              }
              else {
                  cont++;
              }
              runTokens = it.next();
              finalTokens = itMF.next();
           }
        }
        if(CGS.getM().getMarking().size()-1 == cont && flagMax == 1 && N.getFinalMarking().getP().equals(CGS.getM().getP())) {
            return false;
        }
    }
    return true;
```

Figure 7.31: The Implementation of the algorithm to check Property 2 of Data-Aware Soundness

7.4.3 Property 3

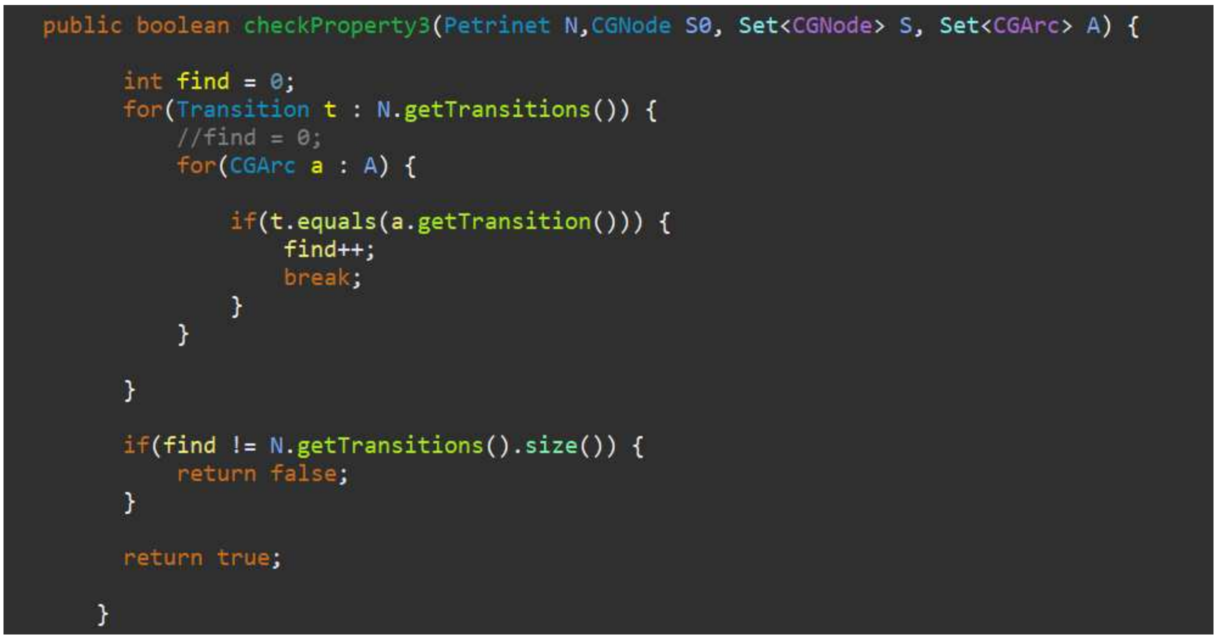
Property 3 ($\forall t \in T.\ \exists M_1, M_2, \alpha 1, \alpha 2, \beta$, $(M1, \alpha 1) \in Reach_N$ *and* $(M_1, \alpha_1) \rightarrow (M_2, \alpha_2)$ *through* (t,β)) is implemented through the *checkProperty3()* method. That is, it is necessary to verify the absence of dead transitions within the DPN. A dead transition can occur in two ways within a DPN:

- If a transition cannot be triggered due to an insufficient number of tokens inside the incoming Place (as in traditional petri nets);
- If a transition cannot fire due to a guard that can never be verified.

So to verify this property we scroll through all the transitions of the DPN present in Petrinet N, another loop is nested within the for loop which scrolls through all the arcs of the Constraint Graph and verifies that the transition associated with that node is the same as the transition current. In that case you stop iterating through the edges, increment a find flag that says the transition has been found, and move on to the next transition. Once the two for loops have ended, it is verified that the find flag has a value different from the number of transitions, if it is true the algorithm returns false, otherwise true.

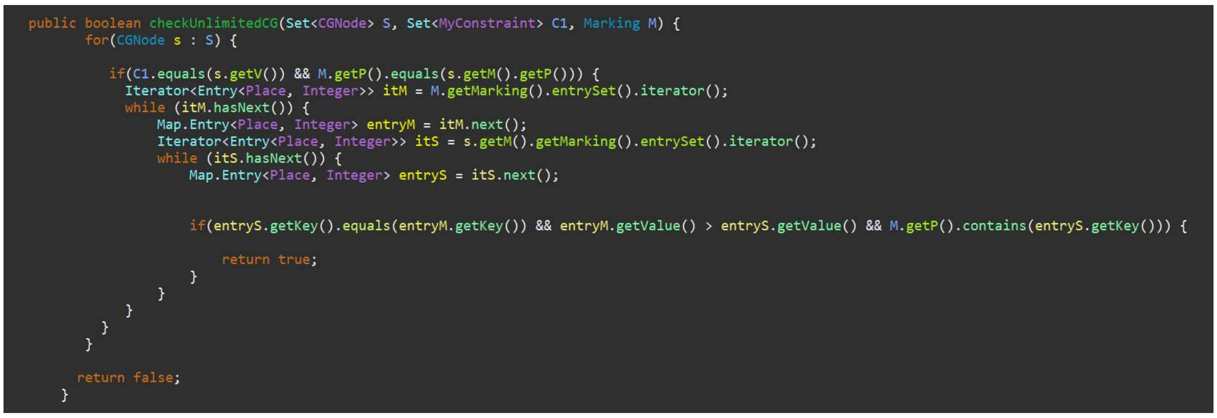
7.4.4 Unlimited Constraint Graph

To verify that a Constraint Graph is unbounded ($\exists(\overline{M}, \overline{C} \in S$ *s.t* $M' > \overline{M}$ *land* $C' = \overline{C}$)) the *checkUnlimitedCG()* method is implemented. This function checks if the Constraint Graph being generated is unbounded and consequently needs the ConstructCG method to finish, otherwise it would go into an infinite loop. The parameters passed as input are a Set of constraints C1, a Set of already created Nodes of the Constraint Graph S and a Marking M. The visit of the nodes present in S is carried out, it is checked if the set of constraints C1 is identical to the set of constraints present in node s and if they have the same places with tokens greater than 0. If the case is true, then the marking M is scrolled together with that of node s and it is verified whether the marking of s has at least one token higher than that of M. If this last condition is true then the algorithm returns the true value which will return the false value to the constructCG method to say that the Soundness cannot be verified since the Constraint Graph is infinite.



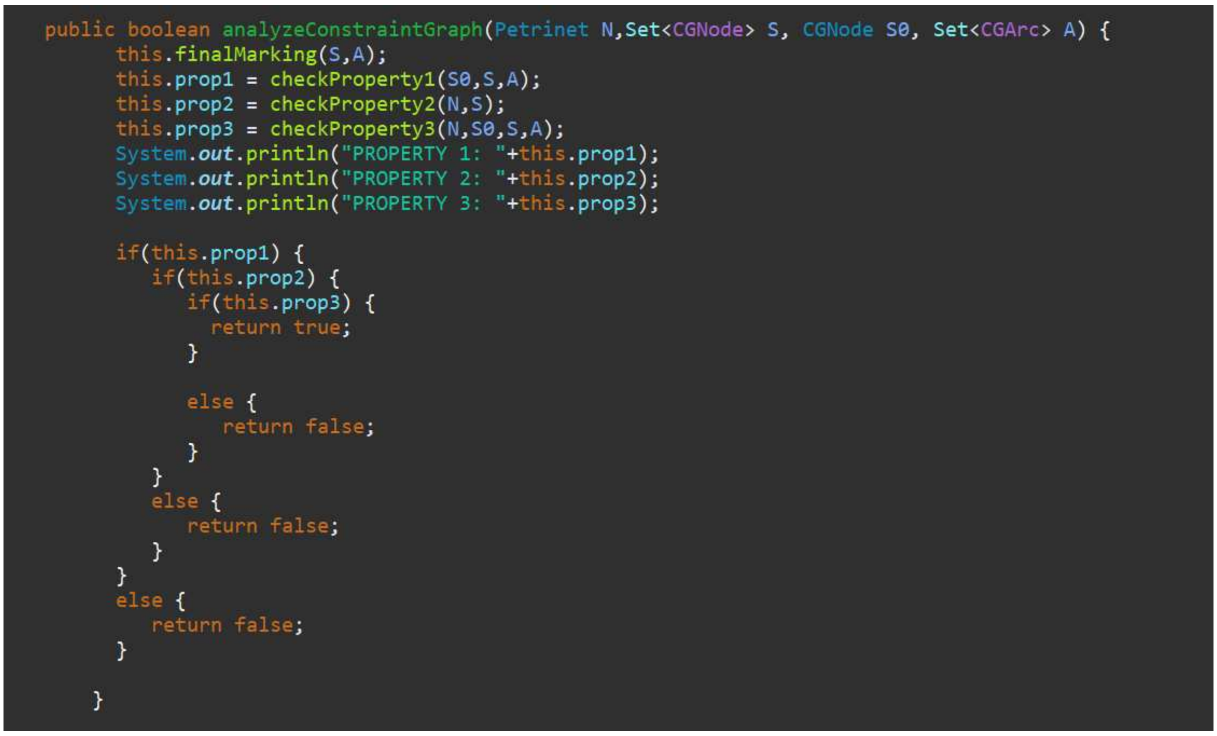
```
public boolean checkProperty3(Petrinet N,CGNode S0, Set<CGNode> S, Set<CGArc> A) {

      int find = 0;
      for(Transition t : N.getTransitions()) {
          //find = 0;
          for(CGArc a : A) {

              if(t.equals(a.getTransition())) {
                  find++;
                  break;
              }
          }

      }

      if(find != N.getTransitions().size()) {
          return false;
      }

      return true;

  }
```

Figure 7.32: The Implementation of the algorithm to check Property 3 of Data-Aware Soundness



```
public boolean checkUnlimitedCG(Set<CGNode> S, Set<MyConstraint> C1, Marking M) {
       for(CGNode s : S) {

          if(C1.equals(s.getV()) && M.getP().equals(s.getM().getP())) {
            Iterator<Entry<Place, Integer>> itM = M.getMarking().entrySet().iterator();
            while (itM.hasNext()) {
                Map.Entry<Place, Integer> entryM = itM.next();
                Iterator<Entry<Place, Integer>> itS = s.getM().getMarking().entrySet().iterator();
                while (itS.hasNext()) {
                    Map.Entry<Place, Integer> entryS = itS.next();


                    if(entryS.getKey().equals(entryM.getKey()) && entryM.getValue() > entryS.getValue() && M.getP().contains(entryS.getKey())) {

                        return true;
                    }
                }
            }
        }
      }

      return false;
   }
```

Figure 7.33: The algorithm to check if the Constraint Graph is unlimited



```
public boolean analyzeConstraintGraph(Petrinet N,Set<CGNode> S, CGNode S0, Set<CGArc> A) {
      this.finalMarking(S,A);
      this.prop1 = checkProperty1(S0,S,A);
      this.prop2 = checkProperty2(N,S);
      this.prop3 = checkProperty3(N,S0,S,A);
      System.out.println("PROPERTY 1: "+this.prop1);
      System.out.println("PROPERTY 2: "+this.prop2);
      System.out.println("PROPERTY 3: "+this.prop3);

      if(this.prop1) {
         if(this.prop2) {
            if(this.prop3) {
              return true;
            }

            else {
               return false;
            }
         }
         else {
            return false;
         }
      }
      else {
         return false;
      }

    }
```

Figure 7.34: The final algorithm to check Data-Aware Soundness properties

7.4.5 The complete algorithm

Finally we now have all the elements to implement *AnalyzeConstraintGraph()*. In this part the three CheckProperty functions and the final marking one are called and the Data-Aware Soundness of the DPN is verified. If all three properties are true then the algorithm returns true, otherwise it returns false.

Part III

Software Test

CHAPTER 8

The Petri Net Markup Language

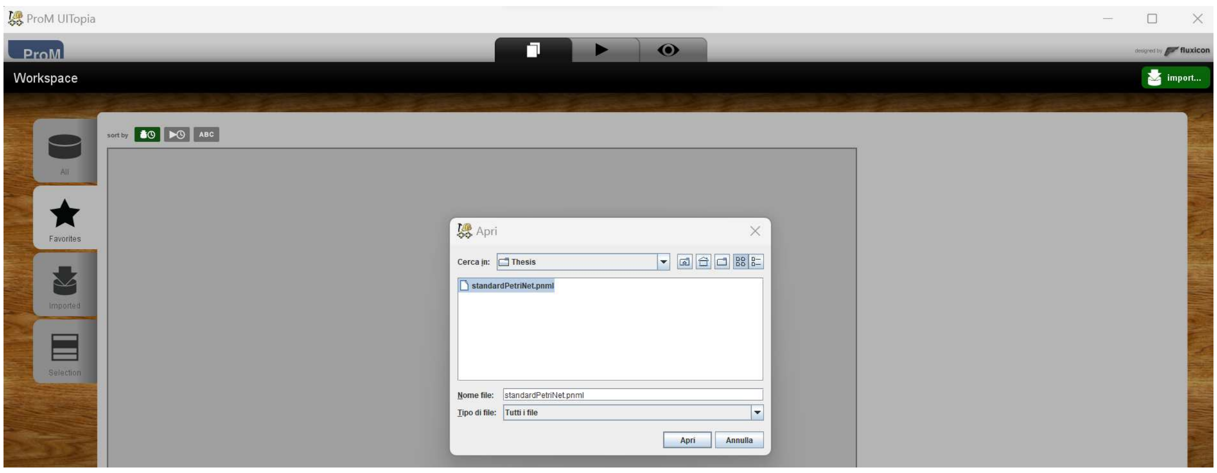
The common format for representing a Petri Net is the Petri Net Markup Language (PNML) format. Through the ProM software it is possible to take a .pnml file that represents a standard Petri Net and extend it with variables to then generate the new PNML adapted to the Data Petri Net. Once the file in .pnml format has been created, the Constraint Graph construction program must be able to read its content and transform it into a Data Petri Net on which to create its algorithms.

8.1 Introduction to ProM

ProM stands for Process Mining and is a Toolkit, i.e. a set of basic software tools, for Process Mining and Business Process Management, used to facilitate and standardize the development of more complex derivative applications. Process mining is concerned with extracting knowledge about a (business) process from its process execution logs. ProM is an extensible framework that supports a wide variety of process mining techniques in the form of plug-ins. It is platform independent as it is implemented in Java.

ProM is a software that has many applications and in this document the relevant part concerns the management of Petri Nets and Data Petri Nets. Within the program it is possible to import a file in .pnml format which represents a standard Petri net with no data (Figure 8.1)

Once the .pnml file has been imported, it is possible to transform it into a Data Petri Net. First you have to click on the "Use resource" button indicated in green (Figure 8.2) to select the desired petri net:



Cerca in: Thesis

standardPetriNet.pnml

Nome file: standardPetriNet.pnml

Tipo di file: Tutti i file

Figure 8.1: ProM software : select PNML File

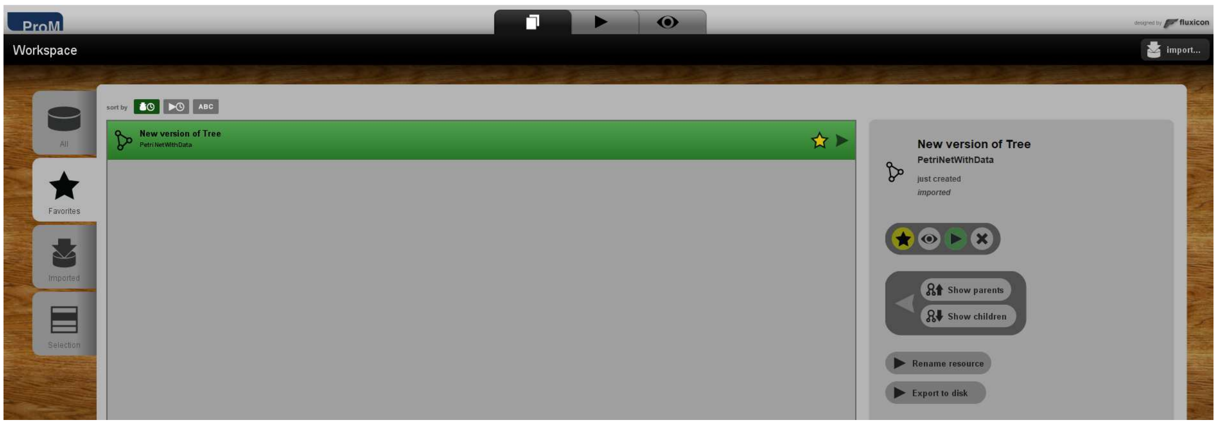


Figure 8.2: ProM software : Use Resource Button

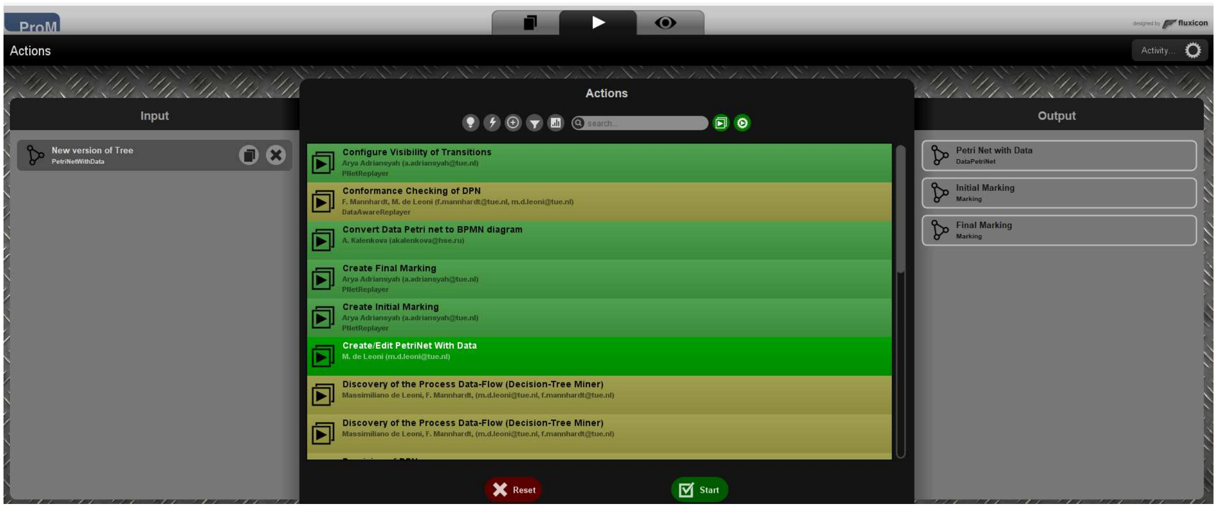

ProM
Actions
Activity...
Input
New version of Tree
PetriNetWithData
Actions
search...
Configure Visibility of Transitions
Arya Adriansyah (a.adriansyah@tue.nl)
PNetReplayer
Conformance Checking of DPN
F. Mannhardt, M. de Leoni (f.mannhardt@tue.nl, m.d.leoni@tue.nl)
DataAwareReplayer
Convert Data Petri net to BPMN diagram
A. Kalenkova (akalenkova@hse.ru)
Create Final Marking
Arya Adriansyah (a.adriansyah@tue.nl)
PNetReplayer
Create Initial Marking
Arya Adriansyah (a.adriansyah@tue.nl)
PNetReplayer
Create/Edit PetriNet With Data
M. de Leoni (m.d.leoni@tue.nl)
Discovery of the Process Data-Flow (Decision-Tree Miner)
Massimiliano de Leoni, F. Mannhardt, (m.d.leoni@tue.nl, f.mannhardt@tue.nl)
Discovery of the Process Data-Flow (Decision-Tree Miner)
Massimiliano de Leoni, F. Mannhardt, (m.d.leoni@tue.nl, f.mannhardt@tue.nl)
Reset
Start
Output
Petri Net with Data
DataPetriNet
Initial Marking
Marking
Final Marking
Marking

Figure 8.3: ProM software : Create/Edit Petri Net

In the screen that appears, select the item in green "Create/Edit Petri Net with Data" and then press the "Start" button.

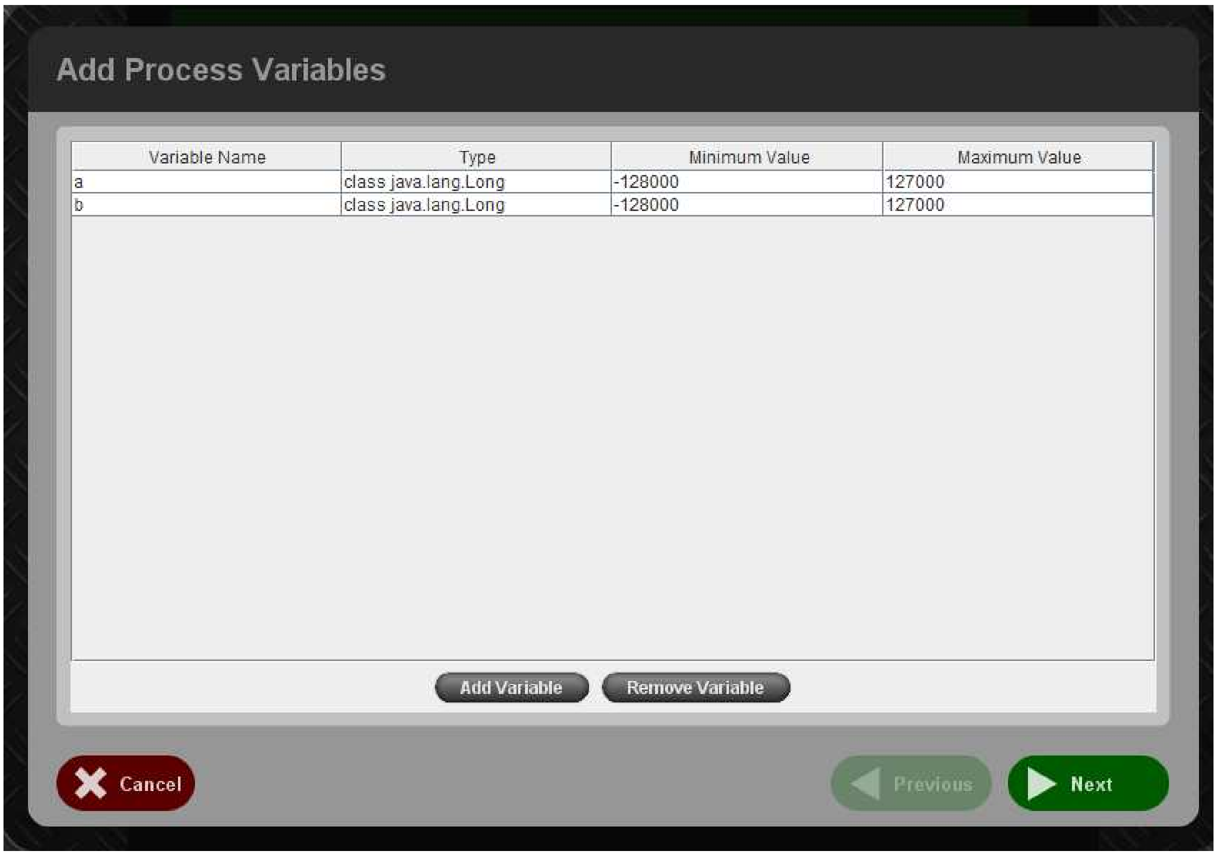
In the screen that appears next, click on "create variable" to create a new variable. You must specify the type of the variable which can be of five types:

- java.lang.String : A String;
- java.lang.Long : An Integer;
- java.lang.Double : A Real;
- java.lang.Date : A date;
- java.lang.Boolean: A Boolean.

In addition to this, the name of the variable and its minimum and maximum value must also be specified when the type is real or integer. Once the variables have been entered, click on "Next".

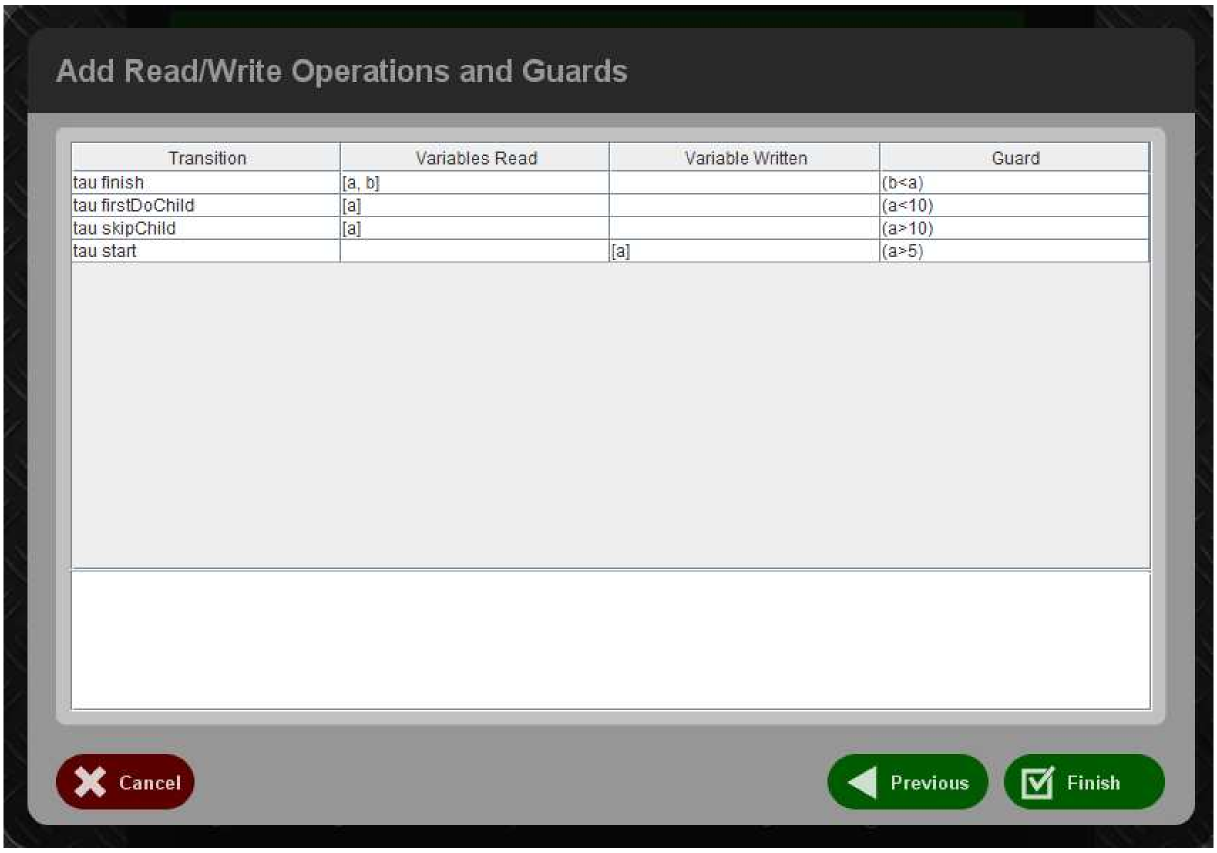
In the following screen, all the guards of the various transitions must be entered, specifying which variables of the constraint are in read and which in write. Transition names are written inside the <*text*> tag of the .pnml format.

Finally, the Data Petri Net is created and displayed. The variables are represented inside yellow hexagons, while the Places and the Transitions are represented as seen previously. Guards are inserted into the arc entering the transition.



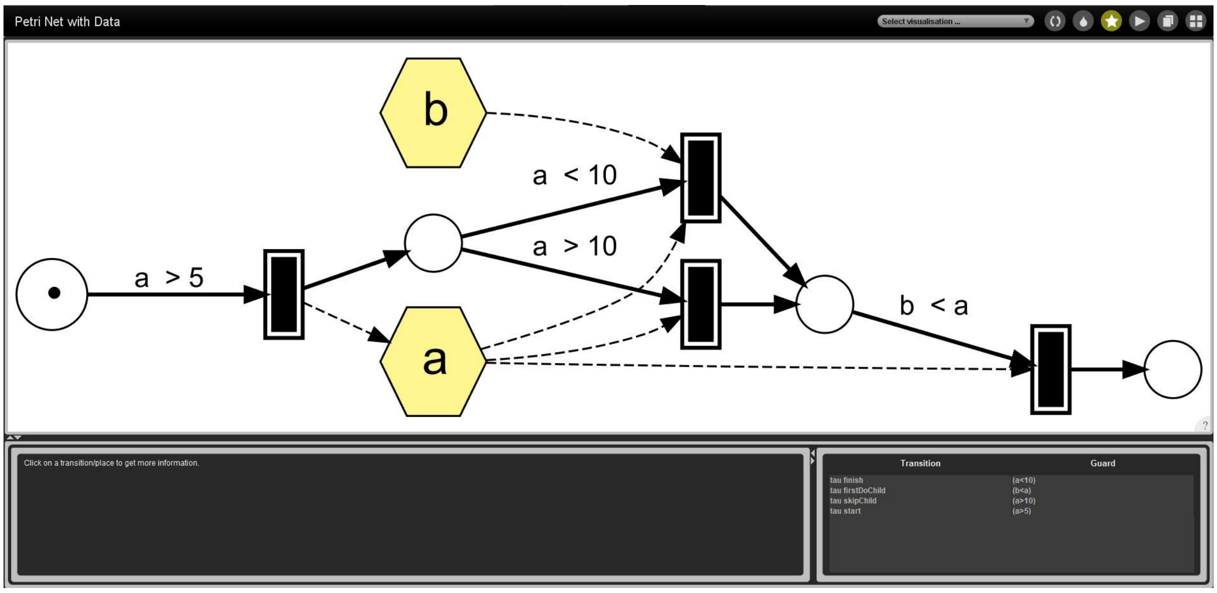
Variable Name	Type	Minimum Value	Maximum Value
a	class java.lang.Long	-128000	127000
b	class java.lang.Long	-128000	127000

Figure 8.4: ProM software : Insert the Variables



Transition	Variables Read	Variable Written	Guard
tau finish	[a, b]		(b<a)
tau firstDoChild	[a]		(a<10)
tau skipChild	[a]		(a>10)
tau start		[a]	(a>5)

Figure 8.5: ProM software : Insert Variables



Transition	Guard
tau finish	(a<10)
tau firstDoChild	(b<a)
tau skipChild	(a>10)
tau start	(a>5)

Figure 8.6: ProM software : Visualization of DPN

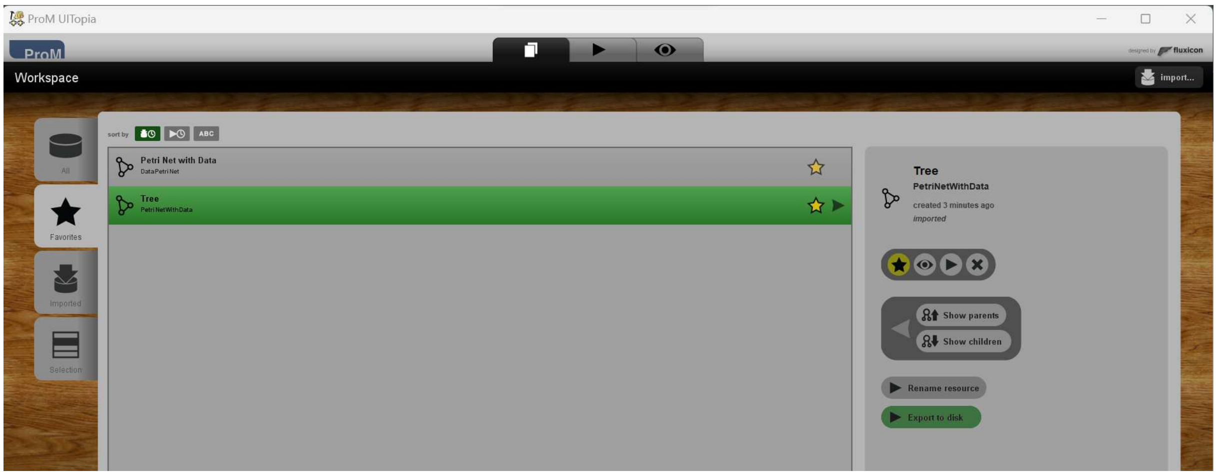


Figure 8.7: ProM software : Export to Disk

Finally, the last part involves clicking on "Export to disk", to save the new .pnml format which also takes into account the variables and guards.

Now the new .pnml file has been generated which extends a standard Petri Net into a Data Petri Net and it is essential to understand the main markers found within the generated file in order to allow the Constraint Graph construction program to generate the corresponding DPN.

8.2 Petri Net Markup Language main markers

This section analyzes the .pnml file generated by analyzing the various markers that compose it.

8.2.1 The generated File

The generated file includes the following content:

8.2.2 Definition of Markers

Below is the definition of all the markers present in the generated file:

- <*pnml*> : indicates the opening of a Petri Net Markup Language;
- <*net*> : indicates the opening of a network, among its attributes it specifies the network identifier and its type;
- <*name*>: indicates the name of the Petri net;
- <*text*> : indicates a text to insert;
- <*place*> : A fundamental element of the program that indicates the opening of a Place in which its identifier is specified;
- <*graphics*> : Element used only for PrOM which defines how it should be displayed, inside it contains the tags *position* and *dimension* which respectively specify the position and size of the element to be displayed graphically;
- <*initialMarking*> : This tag allows you to specify an initial marking;
- <*transition*> : Another fundamental element of the constraint graph construction software which requires an identifier exactly like the Places. The difference with the Petri Net standards is the presence of the guard attribute which specifies the transition guard.
- <*writeVariable*> : Indicates a variable being written;

- <*readVariable*> : Indicates a variable being read;
- <*arc*> : Another fundamental element of the Data Petri Net and indicates an arc that connects a Place to a Transition or vice versa. Equipped with an identifier and two attributes *source* and *target* which respectively indicate the id of the source element and that of the target element;
- <*arctype*> : indicates the type of arc;
- <*finalMarkings*> : final marking container;
- <*marking*> : Contains a markup, i.e. it contains <*place*> tags which instead of the *id* attribute have the *idref* attribute since they refer to a place already defined and the number of tokens is written inside the <*text*> tag;
- <*variables*> : container of all the variables present within the Data Petri Net, this attribute is also present only in a PNML of a Data Petri Net and not of a Standard Petri Net
- <*variable* defines a variable with the attributes *maxValue* and *minValue* to respectively indicate the maximum and minimum value and the attribute *type* which indicates the type of the variable . The name of the variable is specified inside the <*name*> tag.

8.3 Creation of the PNML Reader

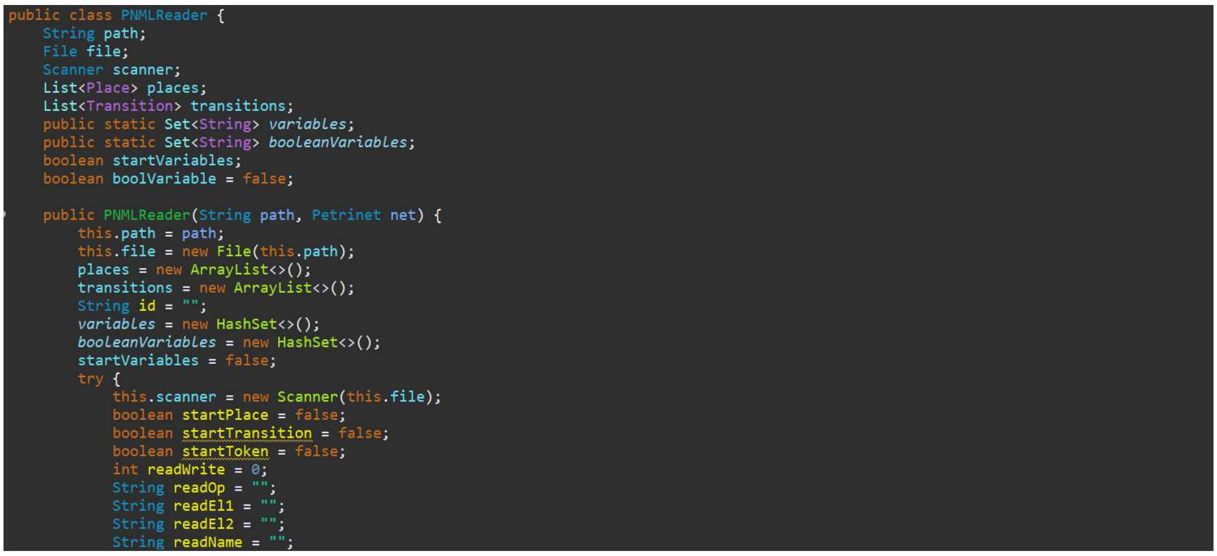
The program for constructing the Constraint Graph implements a class called *PNMLReader* which allows to extract the information needed to generate the DPN. This class is used to take an input PNML file and create the DPN based on what is specified in the input. So this class simply implements a file read and string parsing operation. The main elements of a PNML file are:

- <*place id="name"*> : where <place indicates a place and id indicates its name;
- <*transition guard="V" id="name"*> : where <transition indicates a transition, the guard its associated constraint and id the name of the transition;
- <*arc id="name" source="name1" target="name2"*> : where <arc indicates an arc, id indicates the name of the arc, source the name of the source element and target the recipient item name;
- <*variables*> : indicates the variables present in the document.

As for the V constraint, it consists of two operands and one operator. The strings of the DPN variables are contained within the <variables> tag, if the second operator does not fall within those variables, then it is a constant and the constraint is therefore variable-constant.
While for the operators the following 6 operators are defined:

Operator	Symbol
Not Equal	!=
Equal	=
Greater	&gt;
Lower	&lt;
Greater or Equal	&ge;
Lower or Equal	&le;

The implementation of the PNML Reader is shown in Figure 8.8 - Figure 8.19. This class simply consists of a file reading operation to store the data necessary for the construction of the DPN within variables.
This first part shows the initialization of the variables (Figure 8.8). A series of variables are initialized as a series of empty strings which will concatenate read characters as soon as the element they refer to begins. To verify that a certain tag and an attribute associated with it have been identified when reading the .pnml file, a series of boolean variables are also initialized to false which will become true as soon as the element is identified.



```java
public class PNMLReader {
    String path;
    File file;
    Scanner scanner;
    List<Place> places;
    List<Transition> transitions;
    public static Set<String> variables;
    public static Set<String> booleanVariables;
    boolean startVariables;
    boolean boolVariable = false;

    public PNMLReader(String path, Petrinet net) {
        this.path = path;
        this.file = new File(this.path);
        places = new ArrayList<>();
        transitions = new ArrayList<>();
        String id = "";
        variables = new HashSet<>();
        booleanVariables = new HashSet<>();
        startVariables = false;
        try {
            this.scanner = new Scanner(this.file);
            boolean startPlace = false;
            boolean startTransition = false;
            boolean startToken = false;
            int readWrite = 0;
            String readOp = "";
            String readEl1 = "";
            String readEl2 = "";
            String readName = "";
```

Figure 8.8: PNML Reader : Initialization of Variables

The next step is reading information about places (Figure 8.9). Subsequently, the file is read line by line and the places are identified as the first element. Once <*place id*= has been identified, it begins to scroll the line character by character and as soon as it encounters the quotation marks (with ascii code 34) the place start flag is set to true. It then concatenates subsequent characters until the quotes are read again.

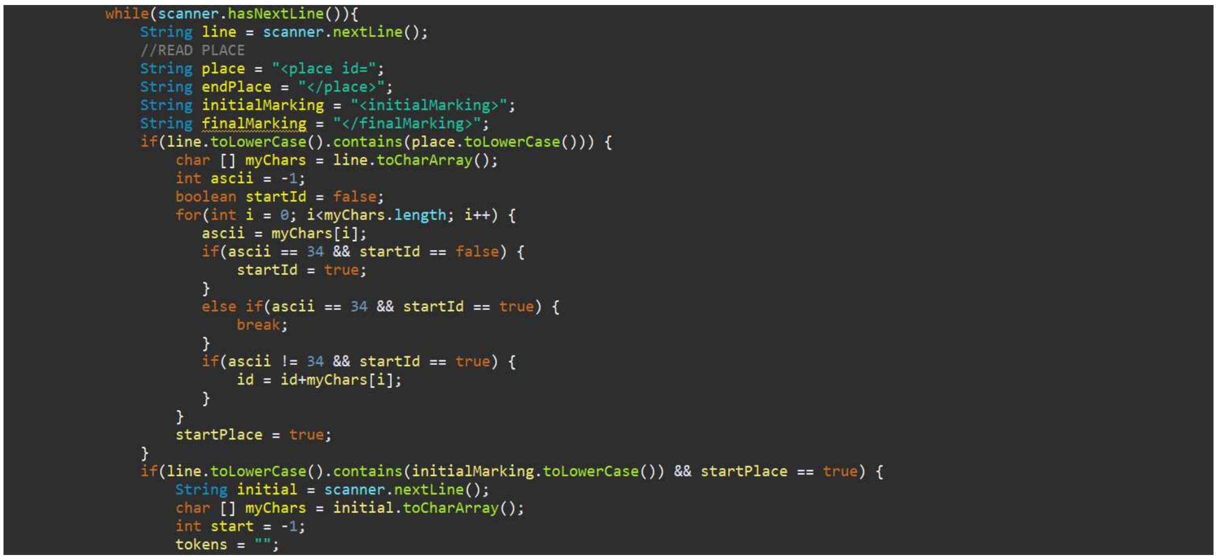
Once the id of a place has been read, the *startPlace* flag is set to true and will return to false only when the closing </*place*> tag is found. In the meantime it is checked whether there is an initial marking. This operation involves checking if the <*initialMarking*> tag is present inside the line and if present it starts reading character by character and starts concatenating the number of tokens after finding the closing angle bracket of the tag. The characters that are concatenated must necessarily be digits. Finally, once all the data has been obtained, the place is created with its specific id and its possible initial marking by converting the string of tokens into an integer value (Figure 8.10).

The next Part define the reading of transitions information (Figure 8.11 - 8.14). The operation of reading transitions is very similar to that of places. The transition starts when the <*transition*> tag is encountered. The following lines may contain the tag <*readVariable*> or <*writeVariable*> and indicate whether the variable indicated in the first operand is read or written and this is specified by a special flag which is set to 0 if it is reading and to 1 if it is writing. The initiation of the constraint occurs when the *guard* attribute is encountered in the transition line. The first element to be read will be the first variable, then the relational operator and finally the second value. If the second value has a value composed exclusively of digits then a variable-constant constraint will be created, otherwise variable-variable.

The next part concerns the reading of the arcs that connect the elements of the graph. arc reading starts when a line with tag <*arc*> is read, then arc id in attribute *id*, source element in attribute *will* be read source and the destination element in the *destination* attribute. Finally it runs through all the places and transitions to determine if it is a place-transition arc or the other way around (Figure 8.15 - 8.17)

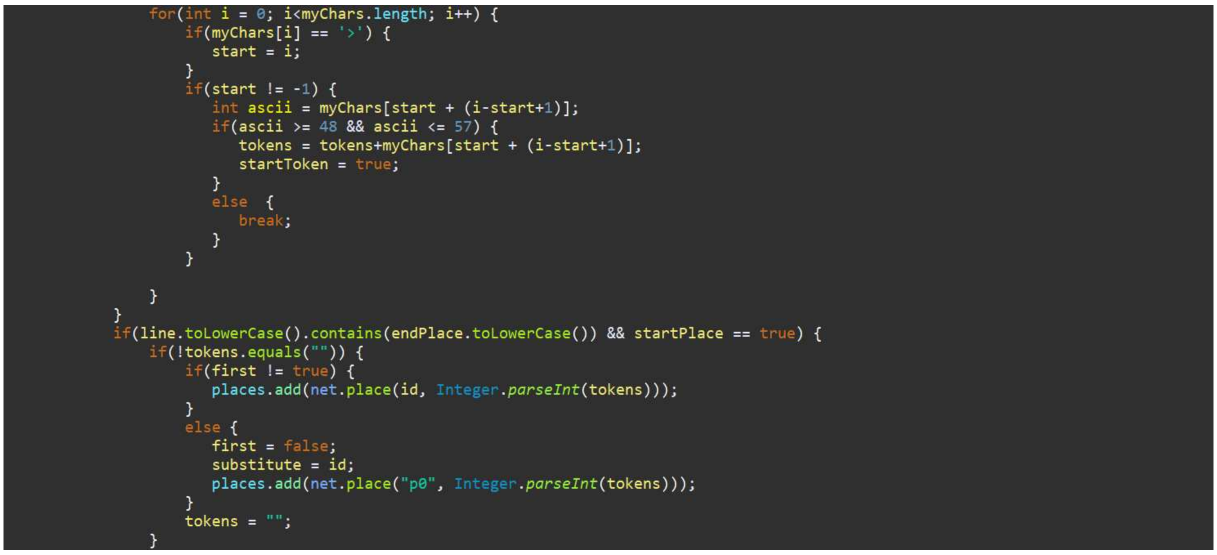
The last part concerns the reading of the variable names which starts when the <*variable*> tag is read and more specifically it starts concatenating the characters when it has read the <*name*> tag (Figure 8.18 - 8.19)

Now it is possible to generate DPN starting from ProM through the .pnml format and read them inside the Constraint Graph Construction software.



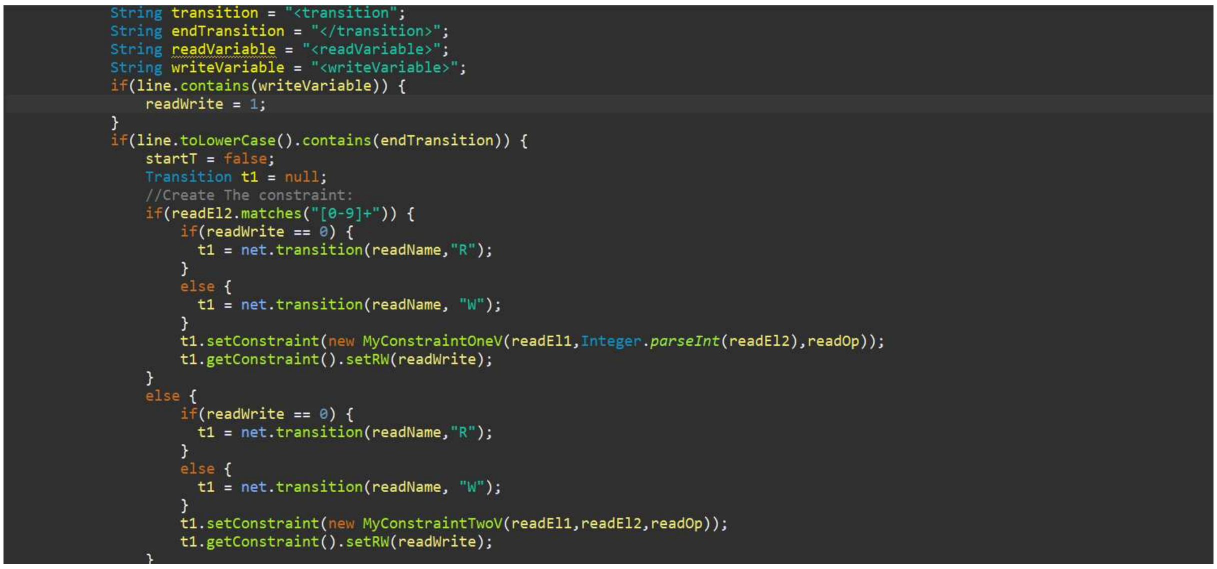
```
while(scanner.hasNextLine()){
    String line = scanner.nextLine();
    //READ PLACE
    String place = "<place id=";
    String endPlace = "</place>";
    String initialMarking = "<initialMarking>";
    String finalMarking = "</finalMarking>";
    if(line.toLowerCase().contains(place.toLowerCase())) {
        char [] myChars = line.toCharArray();
        int ascii = -1;
        boolean startId = false;
        for(int i = 0; i<myChars.length; i++) {
           ascii = myChars[i];
           if(ascii == 34 && startId == false) {
               startId = true;
           }
           else if(ascii == 34 && startId == true) {
               break;
           }
           if(ascii != 34 && startId == true) {
               id = id+myChars[i];
           }
        }
        startPlace = true;
    }
    if(line.toLowerCase().contains(initialMarking.toLowerCase()) && startPlace == true) {
        String initial = scanner.nextLine();
        char [] myChars = initial.toCharArray();
        int start = -1;
        tokens = "";
```

Figure 8.9: PNML Reader : Information about places part 1



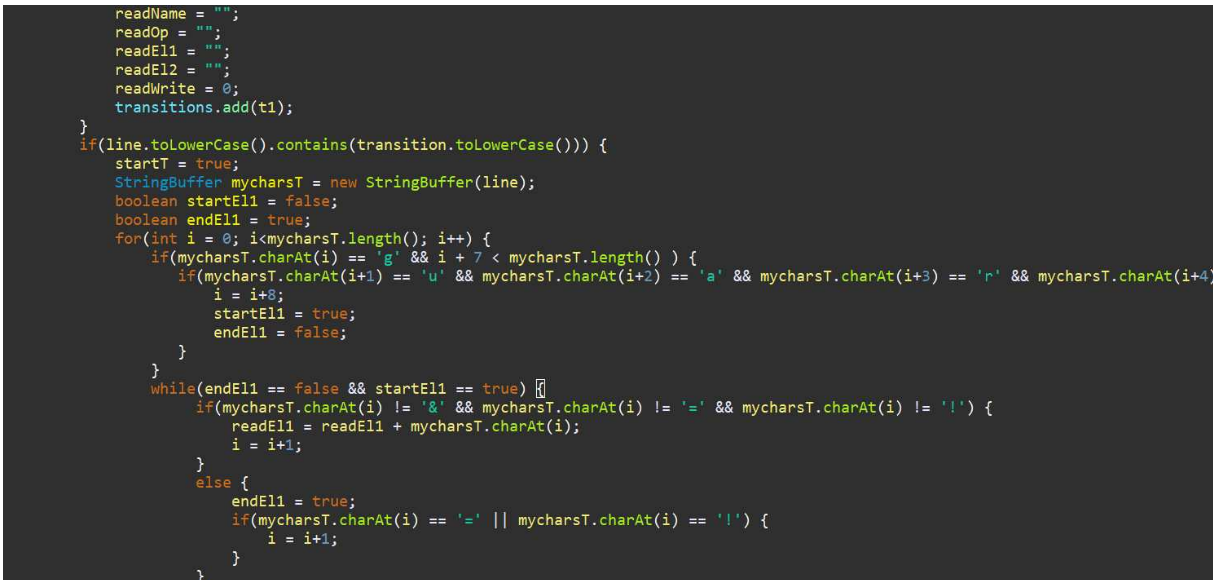
```
            for(int i = 0; i<myChars.length; i++) {
                if(myChars[i] == '>') {
                   start = i;
                }
                if(start != -1) {
                   int ascii = myChars[start + (i-start+1)];
                   if(ascii >= 48 && ascii <= 57) {
                      tokens = tokens+myChars[start + (i-start+1)];
                      startToken = true;
                   }
                   else  {
                      break;
                   }
                }

            }
        }
        if(line.toLowerCase().contains(endPlace.toLowerCase()) && startPlace == true) {
            if(!tokens.equals("")) {
                if(first != true) {
                   places.add(net.place(id, Integer.parseInt(tokens)));
                }
                else {
                   first = false;
                   substitute = id;
                   places.add(net.place("p0", Integer.parseInt(tokens)));
                }
                tokens = "";
            }
```

Figure 8.10: PNML Reader : Information about places part 2



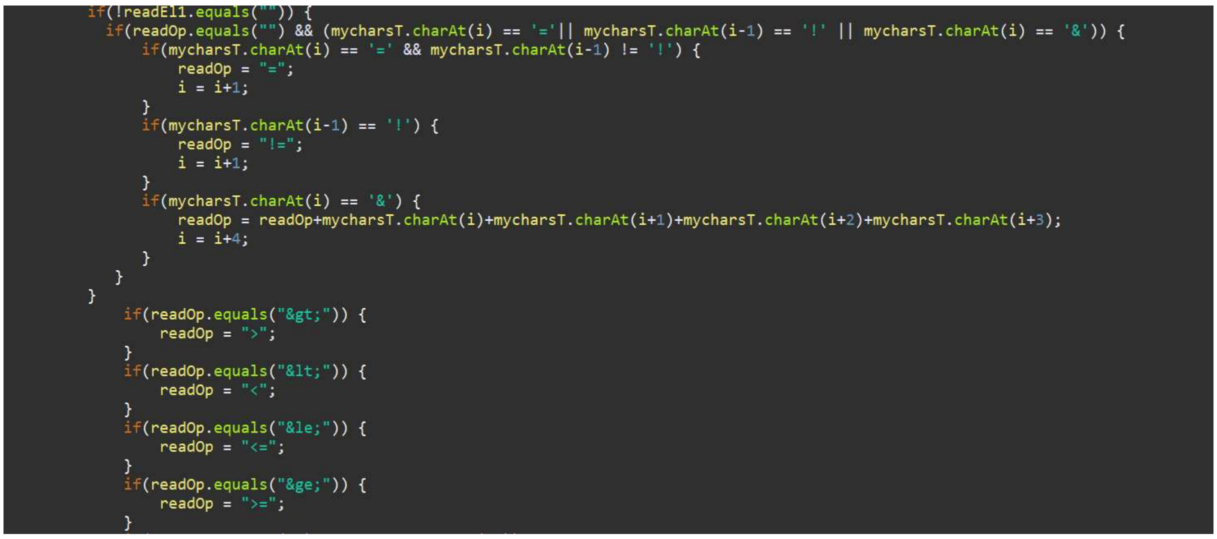
```
            String transition = "<transition";
            String endTransition = "</transition>";
            String readVariable = "<readVariable>";
            String writeVariable = "<writeVariable>";
            if(line.contains(writeVariable)) {
                readWrite = 1;
            }
            if(line.toLowerCase().contains(endTransition)) {
                startT = false;
                Transition t1 = null;
                //Create The constraint:
                if(readEl2.matches("[0-9]+")) {
                    if(readWrite == 0) {
                      t1 = net.transition(readName,"R");
                    }
                    else {
                      t1 = net.transition(readName, "W");
                    }
                    t1.setConstraint(new MyConstraintOneV(readEl1,Integer.parseInt(readEl2),readOp));
                    t1.getConstraint().setRW(readWrite);
                }
                else {
                    if(readWrite == 0) {
                      t1 = net.transition(readName,"R");
                    }
                    else {
                      t1 = net.transition(readName, "W");
                    }
                    t1.setConstraint(new MyConstraintTwoV(readEl1,readEl2,readOp));
                    t1.getConstraint().setRW(readWrite);
                }
```

Figure 8.11: PNML Reader : Information about Transition part 1



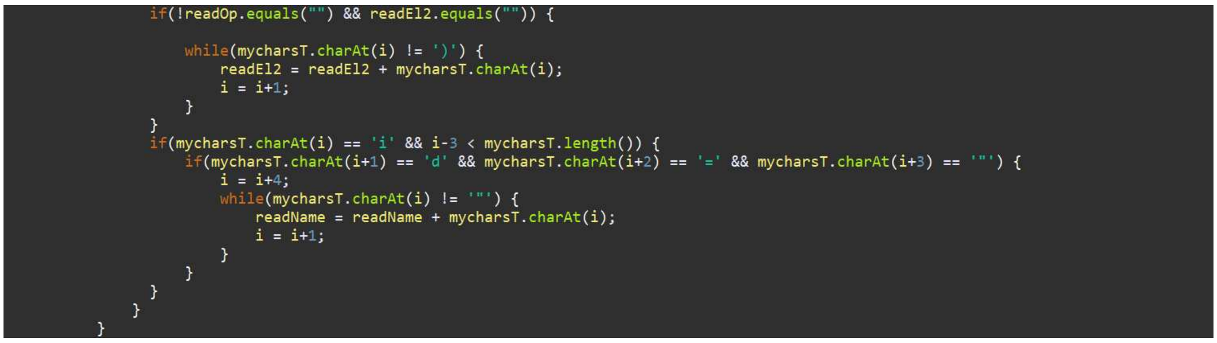
```
            readName = "";
            readOp = "";
            readEl1 = "";
            readEl2 = "";
            readWrite = 0;
            transitions.add(t1);
        }
        if(line.toLowerCase().contains(transition.toLowerCase())) {
            startT = true;
            StringBuffer mycharsT = new StringBuffer(line);
            boolean startEl1 = false;
            boolean endEl1 = true;
            for(int i = 0; i<mycharsT.length(); i++) {
                if(mycharsT.charAt(i) == 'g' && i + 7 < mycharsT.length() ) {
                   if(mycharsT.charAt(i+1) == 'u' && mycharsT.charAt(i+2) == 'a' && mycharsT.charAt(i+3) == 'r' && mycharsT.charAt(i+4)
                       i = i+8;
                       startEl1 = true;
                       endEl1 = false;
                   }
                }
                while(endEl1 == false && startEl1 == true) {
                     if(mycharsT.charAt(i) != '&' && mycharsT.charAt(i) != '=' && mycharsT.charAt(i) != '!') {
                         readEl1 = readEl1 + mycharsT.charAt(i);
                         i = i+1;
                     }
                     else {
                         endEl1 = true;
                         if(mycharsT.charAt(i) == '=' || mycharsT.charAt(i) == '!') {
                             i = i+1;
                         }
                     }
```

Figure 8.12: PNML Reader : Information about Transition part 2



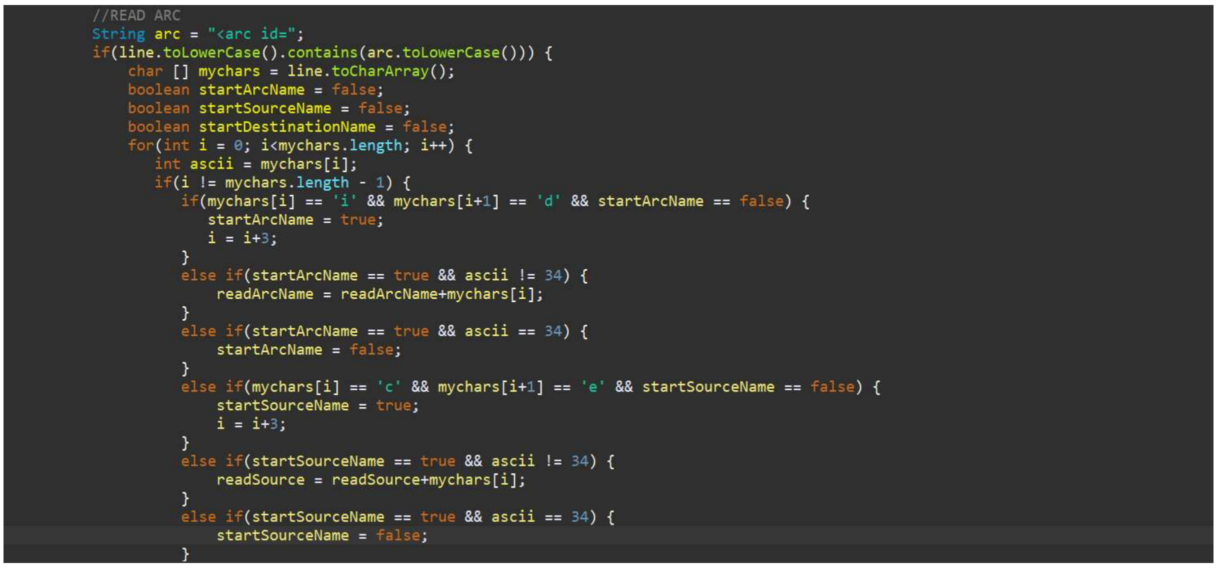
```
        if(!readEl1.equals("")) {
          if(readOp.equals("") && (mycharsT.charAt(i) == '='|| mycharsT.charAt(i-1) == '!' || mycharsT.charAt(i) == '&')) {
              if(mycharsT.charAt(i) == '=' && mycharsT.charAt(i-1) != '!') {
                  readOp = "=";
                  i = i+1;
              }
              if(mycharsT.charAt(i-1) == '!') {
                  readOp = "!=";
                  i = i+1;
              }
              if(mycharsT.charAt(i) == '&') {
                  readOp = readOp+mycharsT.charAt(i)+mycharsT.charAt(i+1)+mycharsT.charAt(i+2)+mycharsT.charAt(i+3);
                  i = i+4;
              }
          }
        }
            if(readOp.equals("&gt;")) {
                readOp = ">";
            }
            if(readOp.equals("&lt;")) {
                readOp = "<";
            }
            if(readOp.equals("&le;")) {
                readOp = "<=";
            }
            if(readOp.equals("&ge;")) {
                readOp = ">=";
            }
```

Figure 8.13: PNML Reader : Information about Transition part 3



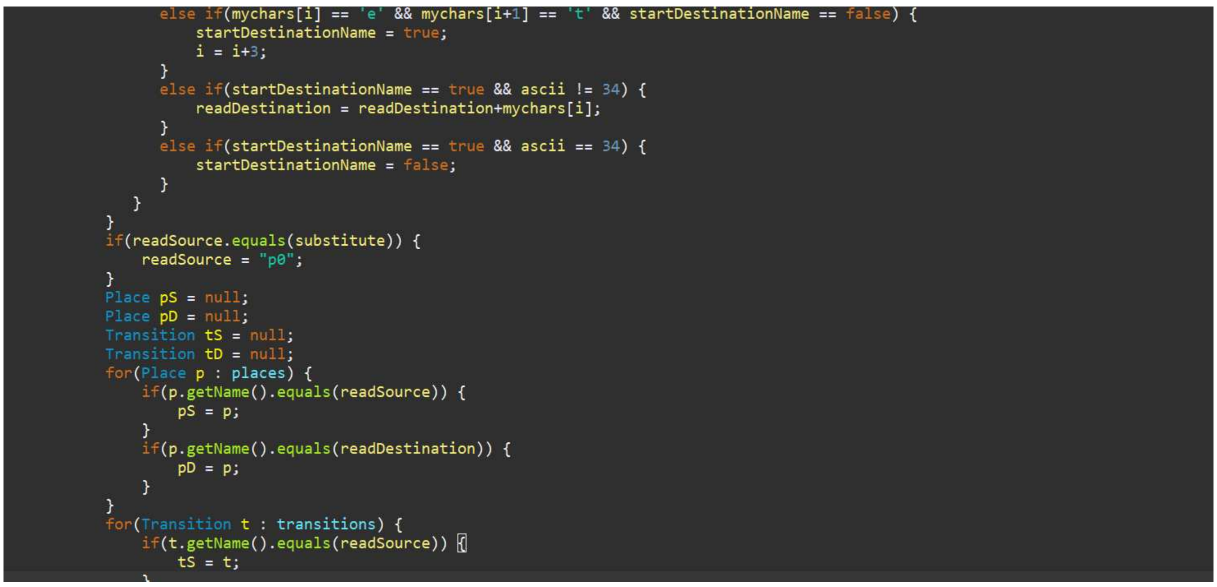
```
            if(!readOp.equals("") && readEl2.equals("")) {

                while(mycharsT.charAt(i) != ')') {
                    readEl2 = readEl2 + mycharsT.charAt(i);
                    i = i+1;
                }
            }
            if(mycharsT.charAt(i) == 'i' && i-3 < mycharsT.length()) {
                if(mycharsT.charAt(i+1) == 'd' && mycharsT.charAt(i+2) == '=' && mycharsT.charAt(i+3) == '"') {
                    i = i+4;
                    while(mycharsT.charAt(i) != '"') {
                        readName = readName + mycharsT.charAt(i);
                        i = i+1;
                    }
                }
            }
          }
        }
```

Figure 8.14: PNML Reader : Information about Transition part 4



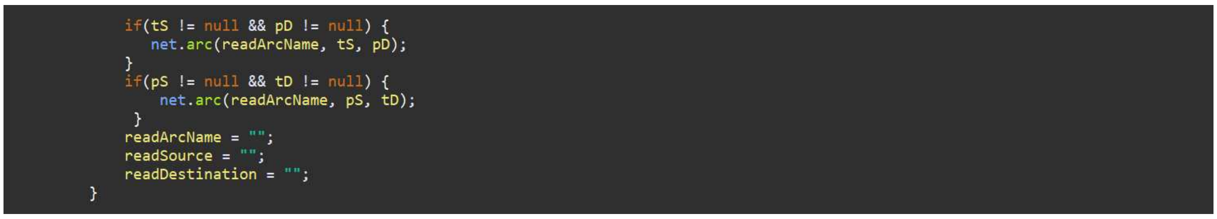
```
        //READ ARC
        String arc = "<arc id=";
        if(line.toLowerCase().contains(arc.toLowerCase())) {
            char [] mychars = line.toCharArray();
            boolean startArcName = false;
            boolean startSourceName = false;
            boolean startDestinationName = false;
            for(int i = 0; i<mychars.length; i++) {
                int ascii = mychars[i];
                if(i != mychars.length - 1) {
                    if(mychars[i] == 'i' && mychars[i+1] == 'd' && startArcName == false) {
                        startArcName = true;
                        i = i+3;
                    }
                    else if(startArcName == true && ascii != 34) {
                        readArcName = readArcName+mychars[i];
                    }
                    else if(startArcName == true && ascii == 34) {
                        startArcName = false;
                    }
                    else if(mychars[i] == 'c' && mychars[i+1] == 'e' && startSourceName == false) {
                        startSourceName = true;
                        i = i+3;
                    }
                    else if(startSourceName == true && ascii != 34) {
                        readSource = readSource+mychars[i];
                    }
                    else if(startSourceName == true && ascii == 34) {
                        startSourceName = false;
                    }
```

Figure 8.15: PNML Reader : Information about Arcs part 1



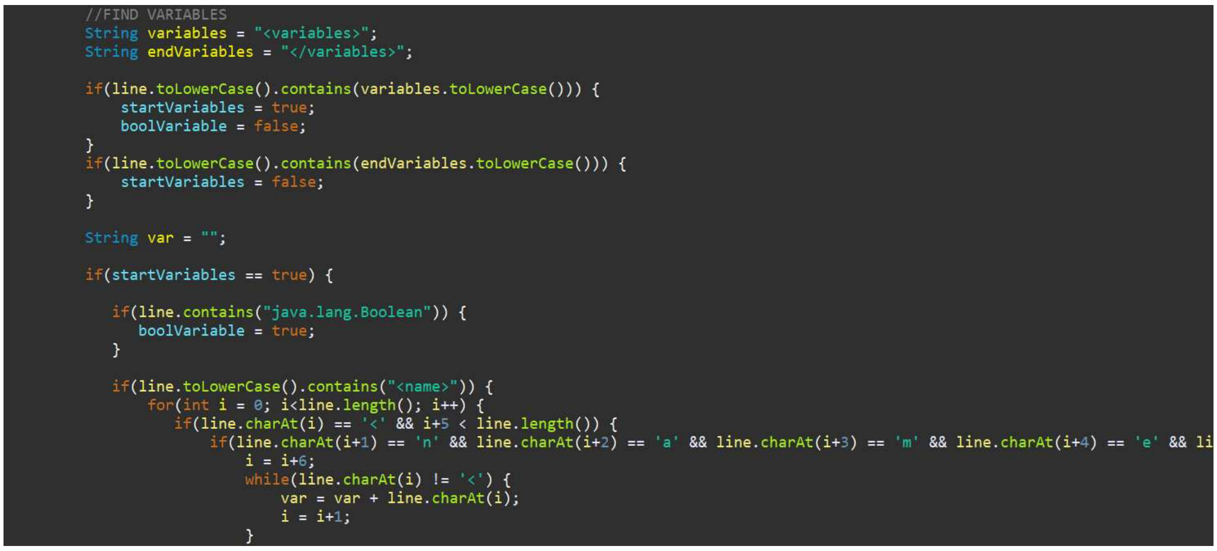
```
      else if(mychars[i] == 'e' && mychars[i+1] == 't' && startDestinationName == false) {
          startDestinationName = true;
          i = i+3;
      }
      else if(startDestinationName == true && ascii != 34) {
          readDestination = readDestination+mychars[i];
      }
      else if(startDestinationName == true && ascii == 34) {
          startDestinationName = false;
      }
   }
}
if(readSource.equals(substitute)) {
    readSource = "p0";
}
Place pS = null;
Place pD = null;
Transition tS = null;
Transition tD = null;
for(Place p : places) {
    if(p.getName().equals(readSource)) {
        pS = p;
    }
    if(p.getName().equals(readDestination)) {
        pD = p;
    }
}
for(Transition t : transitions) {
    if(t.getName().equals(readSource)) {
        tS = t;
```

Figure 8.16: PNML Reader : Information about Arcs part 2



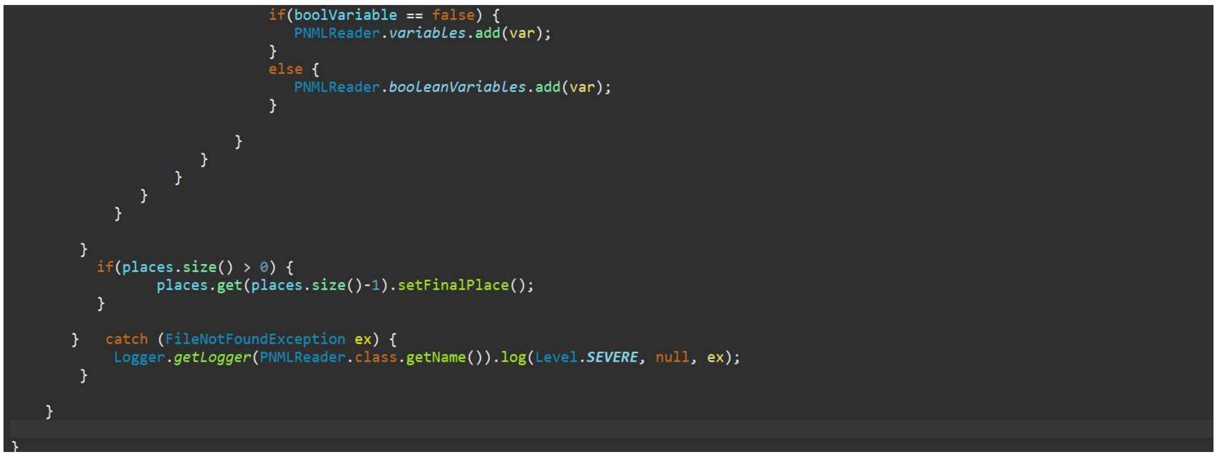
```
    if(tS != null && pD != null) {
       net.arc(readArcName, tS, pD);
    }
    if(pS != null && tD != null) {
        net.arc(readArcName, pS, tD);
     }
    readArcName = "";
    readSource = "";
    readDestination = "";
}
```

Figure 8.17: PNML Reader : Information about Arcs part 3



```
//FIND VARIABLES
String variables = "<variables>";
String endVariables = "</variables>";

if(line.toLowerCase().contains(variables.toLowerCase())) {
    startVariables = true;
    boolVariable = false;
}
if(line.toLowerCase().contains(endVariables.toLowerCase())) {
    startVariables = false;
}

String var = "";

if(startVariables == true) {

   if(line.contains("java.lang.Boolean")) {
      boolVariable = true;
   }

   if(line.toLowerCase().contains("<name>")) {
       for(int i = 0; i<line.length(); i++) {
          if(line.charAt(i) == '<' && i+5 < line.length()) {
              if(line.charAt(i+1) == 'n' && line.charAt(i+2) == 'a' && line.charAt(i+3) == 'm' && line.charAt(i+4) == 'e' && li
                  i = i+6;
                  while(line.charAt(i) != '<') {
                      var = var + line.charAt(i);
                      i = i+1;
                  }
```

Figure 8.18: PNML Reader : Information about Variables part 1



```
                                if(boolVariable == false) {
                                    PNMLReader.variables.add(var);
                                }
                                else {
                                    PNMLReader.booleanVariables.add(var);
                                }

                            }
                        }
                    }
                }
            }

        }
          if(places.size() > 0) {
                places.get(places.size()-1).setFinalPlace();
          }

        }   catch (FileNotFoundException ex) {
            Logger.getLogger(PNMLReader.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
}
```

Figure 8.19: PNML Reader : Information about Variables part 2

8.4 Generate Random Petri Nets with Processes and Logs Generator (PLG)

One of the major criticalities that occurs when trying to create random Data Petri Nets is that it is generally very complex to create algorithms that expect to create scalable Data Petri Nets, due to the fact that the Petri Nets and to an even greater extent the Data Petri Net follow strict rules and a small violation is enough to prevent the creation of complex Constraint Graphs. A situation of Unlimited Petri Data Net or the presence of dead transitions will be sufficient to have a heavy truncation of the number of elements of the generated graph.

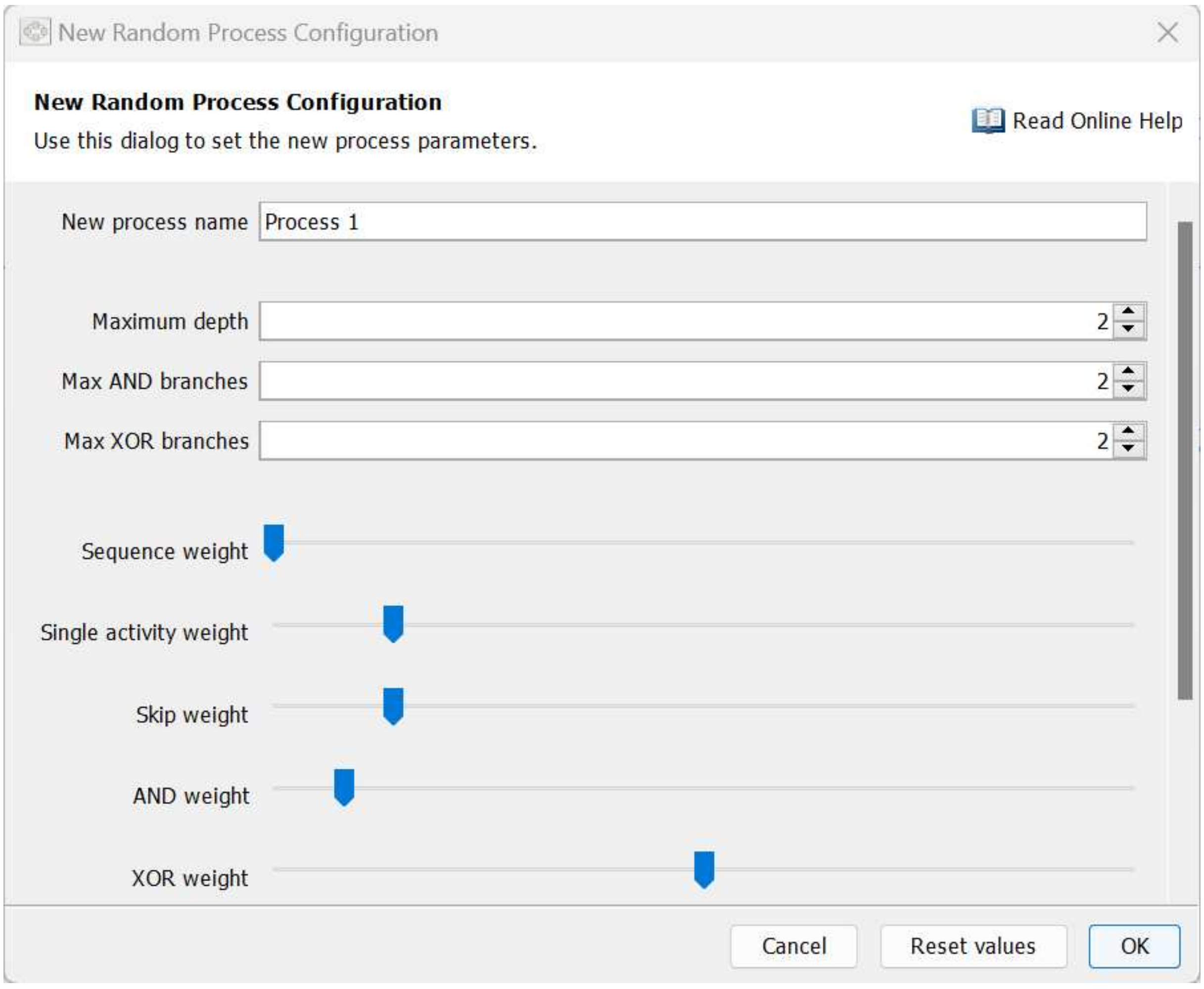
To generate test cases, the *Processes and Logs Generator* software was also used, which allows you to generate processes with a structure very similar to that of a standard Petri Net, then the part relating to the data, then variable initialization and transitions guards is carried out manually.

After installing the program, first click on *New Process* present at the top left of the program. The following screen will then open, on which you have to work on the parameters that are listed below

- **New process name** : The name of the process;
- **Max And branches** : The maximum number of transitions in parallel;
- **Max XOR branches** : The maximum number of branches outgoing from a place.
- **Sequence weight** : This should be set to a minimum value, since transitions connected to other transitions should not be generated.

For example, with the parameters set in the previous image, the following graph is generated where the Activities represent the transitions, while both x and + depicted with a yellow rhombus represent the places.

Finally we proceed manually to enter all the parts concerning the data.


New Random Process Configuration

New Random Process Configuration

Use this dialog to set the new process parameters.

New process name	Process 1
Maximum depth	2
Max AND branches	2
Max XOR branches	2

Figure 8.20: PLG : Selection of Elements

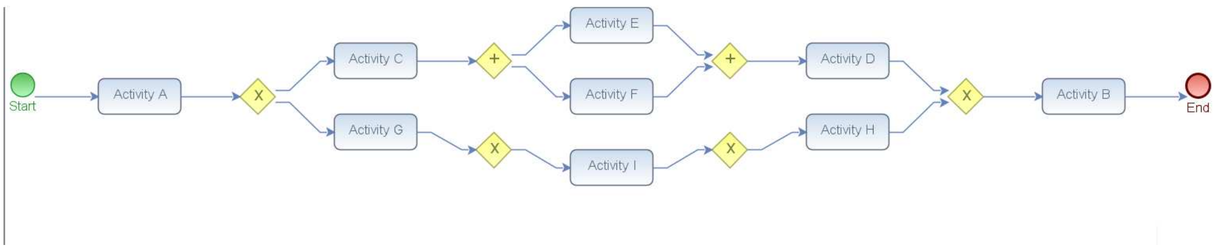


Figure 8.21: PLG : Visualization of the Petri Net

CHAPTER 9

Test on Examples of DPN

In the final part of this document the behavior of the program is verified by generating some examples and showing their output. For each example we will explain the behavior of the program and why it returns a correct result.

9.1 Generation of DPNs

DPN generation within the software is possible through three options:

- Reading a .pnml file with PNML Reader (See Chapter 8);
- Generation of a random DPN with a random number of Places, Transitions, variables and depth of the tree structure;
- Generation of a DPN with a fixed number of tree depth and variables;

Once the program has been executed, the menu will appear in Figure 9.1:
By selecting "Import a PNML File" a screen will appear in which it is possible to specify the path of the file in the appropriate text box in .pnml format and then press "Submit" to generate it.

9.1.1 The Automatic DPN Generator

To automate the process described above, a DPN Generator was created directly within the software, capable of generating a series of transitions, places and relative arcs to then be given as

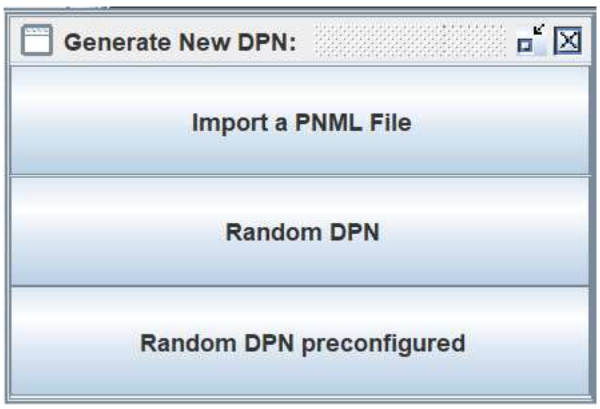


Figure 9.1: The Buttons to decide how to generate a DPN

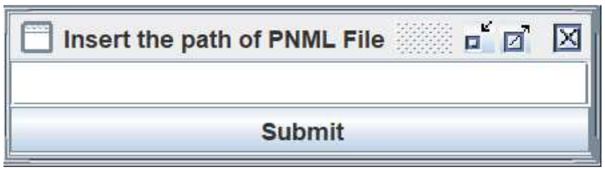


Figure 9.2: The Textfield in which insert the PNML File

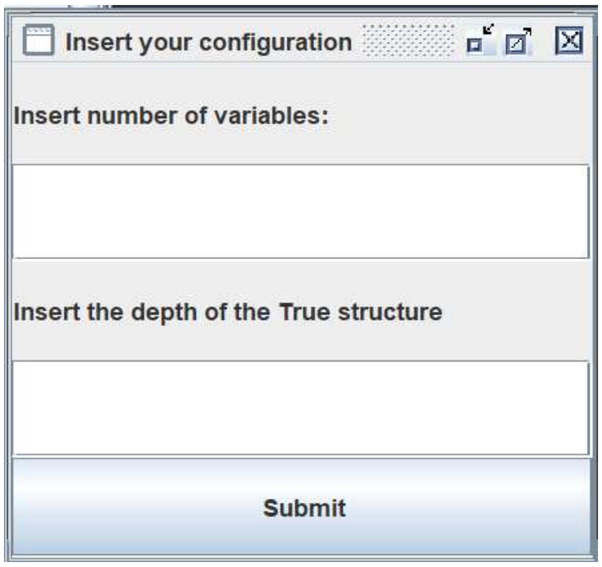

Insert your configuration
Insert number of variables:
Insert the depth of the True structure
Submit

Figure 9.3: The random DPN preconfigured menu

input for the construction of the Constraint Graph. The DPN Generator is used in Chapter 10 to test the performance of the program.

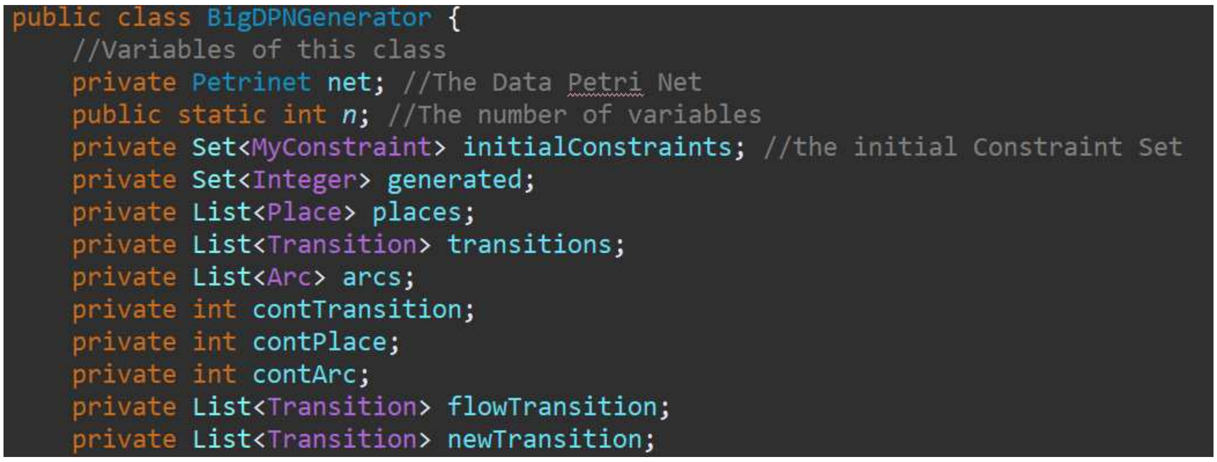
The function that generates a DPN with all tokens set to 1 both in the places and in the costs/additions of the transitions takes the name of *generateDPNwithoutToken*.

The logic of this algorithm is to generate a DPN which has a tree structure. Furthermore, to avoid dead transitions truncating the Constraint Graph, thus making the generation of Constraint Graphs with large number of nodes very difficult, only constraints using the operator *!=* (*not* =) are used. To avoid repeated constants, a Set *generated* is used in which all the constants that are inserted in the constraints are stored, in this way each node originates both the node for the transition and for the Silent Transition. The algorithm verifies that each Transition has only one incoming Place, while more branches can start from the Places for the transitions (branching factor) You can specify a number given by the n variable which indicates the maximum depth of the tree. So to determine the maximum number of Places in a graph is given by (branching factor)^n. The initial constraint is initialized to a random value which also uses the *!=* operator and is represented in the *initialize()* function. In the last operation, once the tree structure has been created, all the transitions are reunited in a Place which will be the final Place of the DPN and therefore the final Place will be the only Place to have multiple Transitions as parents thus transforming the data structure from Directed Acyclic Graph (DAG) tree.

The Automatic DPN Generator also allows you to specify parameters in order to partially customize the generated DPN. (Figure 9.5)

Among its input parameters that can be modified by the user are the number of variables and the depth of the tree, which however excludes the last level of depth which will be represented by the final place which rejoins all the parallel transitions. This data structure provides a high parallelization and thus allows to better control the creation of new nodes in terms of number of nodes generated. The code shown in the various figures is a simplification of the algorithm, in fact in its final version it also allows you to insert constraints of the type *variable != variable*.

Figure 9.4 shows the initialization of the variables, including a list of Places, Transitions and Arcs that will be the constituent elements of the Constraint Graph. In addition to this there is an integer list to check for integers that have already been generated as a constant for the constraints.



```
public class BigDPNGenerator {
    //Variables of this class
    private Petrinet net; //The Data Petri Net
    public static int n; //The number of variables
    private Set<MyConstraint> initialConstraints; //the initial Constraint Set
    private Set<Integer> generated;
    private List<Place> places;
    private List<Transition> transitions;
    private List<Arc> arcs;
    private int contTransition;
    private int contPlace;
    private int contArc;
    private List<Transition> flowTransition;
    private List<Transition> newTransition;
```

Figure 9.4: DPN Generator : Variable Initialization

Figure 9.5 shows the method of initializing variables when there is only one variable. In the case of multiple variables, a for loop is added and a list of strings is used instead of the *"x"* variable.

In the constructor the variables defined in 9.4 are initialized and the initial Place is initialized together with a transition and connected with a Place-Transition arc (Figures 9.6 - 9.7).

Then the tree structure is created through a loop that extracts each transition and connects it to one or more places. Places, on the other hand, can be connected exclusively to a single transition, in this way it will be sufficient for a single place to have a sufficient number of tokens to trigger the transition (Figure 9.8 - 9.9)

Each time a place is created it will have a transition at the output, therefore in order to create a place that has a final markup it is necessary to join all the leaf transitions to the final place, thus transforming the data structure from tree to acyclic directed graph (Figure 9.10)

An example of a possible final result is given by the example in Figure 9.11. The generated graph has depth (depth+1) as it also adds the final place which joins all the transitions.

9.2 Examples

The following paragraphs will show the results obtained for some example DPNs created with ProM, trying to show all the various cases that could occur.

9.2.1 Example Constraint Graph Data-Aware Sound

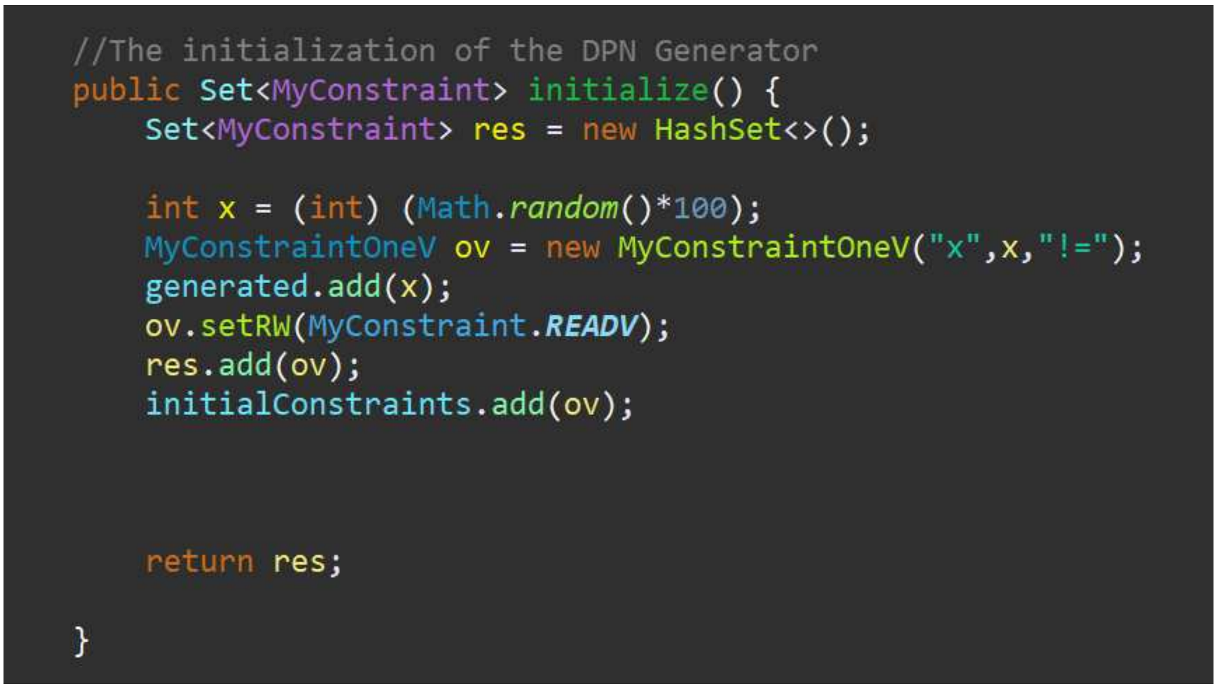
Figure 9.12 shows an example of a DPN that is completely data-aware sound. This is demonstrated in Figure 9.13 with the construction of a Constraint Graph capable of respecting all three data-aware soundness properties. The graphical result of the three properties of data-aware soundness is shown in figure 9.14.

9.2.2 Property 1 Violation Example

Figure 9.16 shows an example of a generated Constraint Graph in which the first property of Data-Aware Soundness is violated, in fact in the case where a = 10, or in the case where a < 10 it is not possible to arrive at a final node. Indeed, the Constraint Graph contains two "dead nodes" with no outgoing transitions (c7 and c2) that violates property 1, as correctly detected by the program (Figure 9.17). Figure 9.15 shows the DPN associated with Figure 9.16.

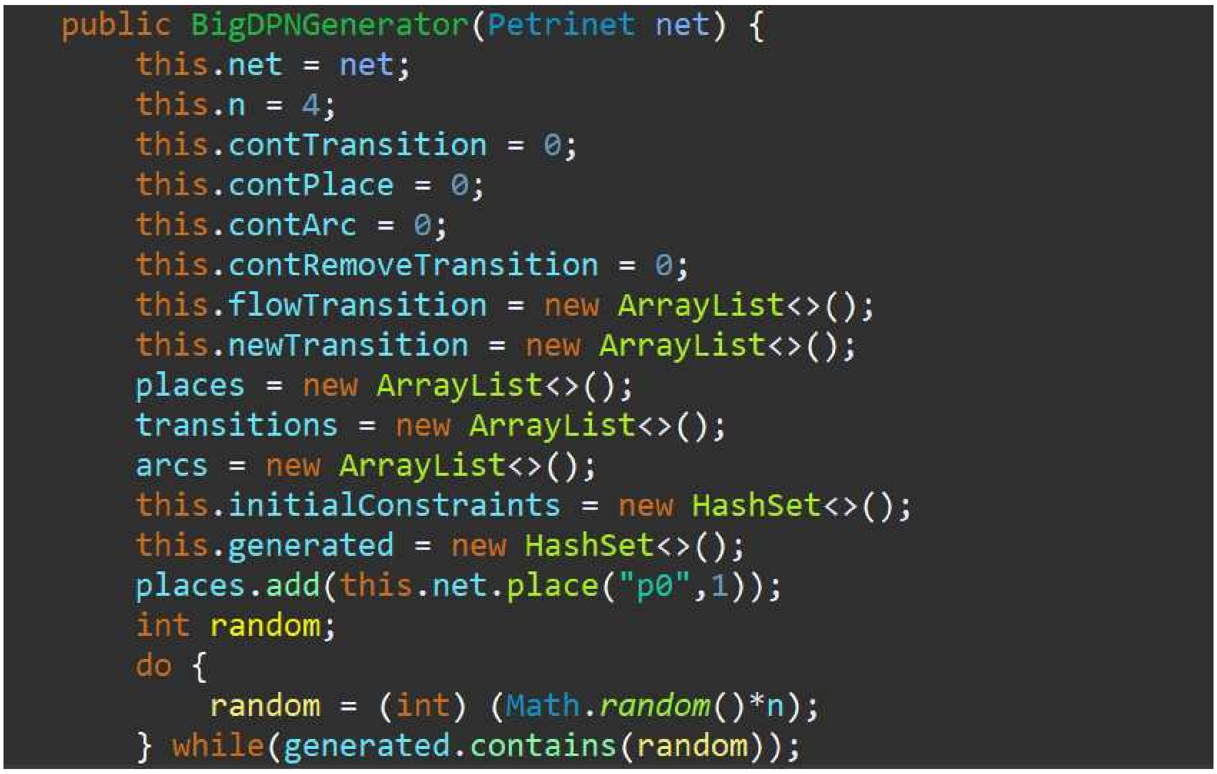
9.2.3 Violation Property 2 Example

Figure 9.19 shows an example of a generated Constraint Graph in which the first property of Data-Aware Soundness is violated, In fact analyzing all the possible paths there are two nodes of the



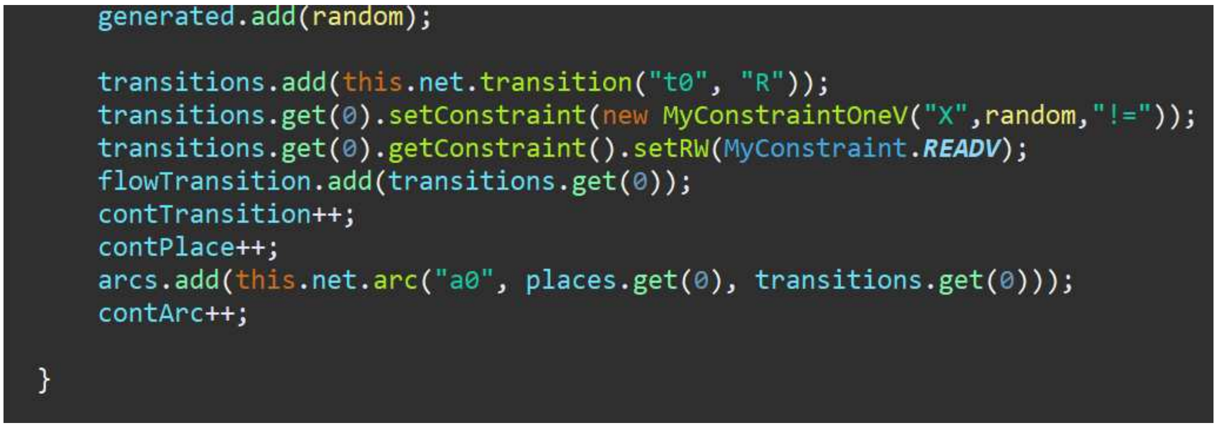
```
//The initialization of the DPN Generator
public Set<MyConstraint> initialize() {
    Set<MyConstraint> res = new HashSet<>();

    int x = (int) (Math.random()*100);
    MyConstraintOneV ov = new MyConstraintOneV("x",x,"!=");
    generated.add(x);
    ov.setRW(MyConstraint.READV);
    res.add(ov);
    initialConstraints.add(ov);



    return res;

}
```

Figure 9.5: DPN Generator : Constraint Initialization



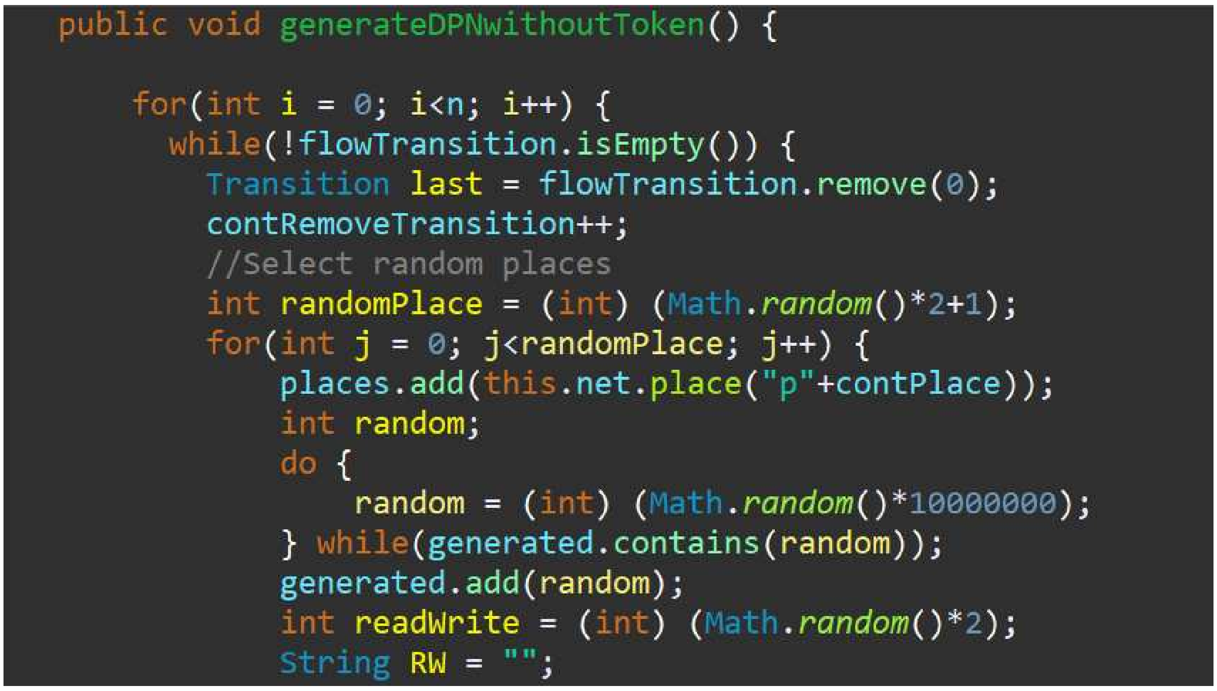
```
public BigDPNGenerator(Petrinet net) {
    this.net = net;
    this.n = 4;
    this.contTransition = 0;
    this.contPlace = 0;
    this.contArc = 0;
    this.contRemoveTransition = 0;
    this.flowTransition = new ArrayList<>();
    this.newTransition = new ArrayList<>();
    places = new ArrayList<>();
    transitions = new ArrayList<>();
    arcs = new ArrayList<>();
    this.initialConstraints = new HashSet<>();
    this.generated = new HashSet<>();
    places.add(this.net.place("p0",1));
    int random;
    do {
        random = (int) (Math.random()*n);
    } while(generated.contains(random));
```

Figure 9.6: DPN Generator : Constructor part 1



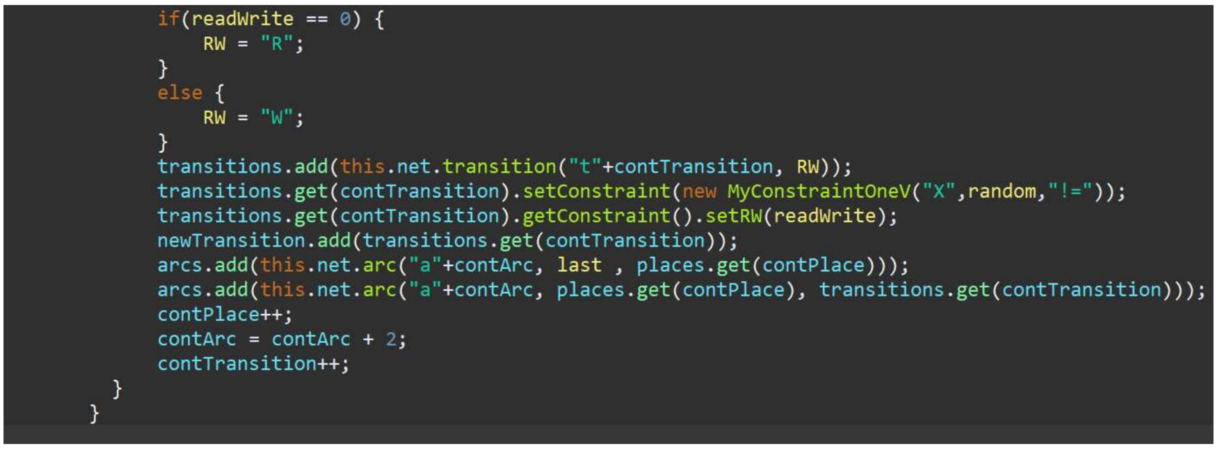
```
    generated.add(random);

    transitions.add(this.net.transition("t0", "R"));
    transitions.get(0).setConstraint(new MyConstraintOneV("X",random,"!="));
    transitions.get(0).getConstraint().setRW(MyConstraint.READV);
    flowTransition.add(transitions.get(0));
    contTransition++;
    contPlace++;
    arcs.add(this.net.arc("a0", places.get(0), transitions.get(0)));
    contArc++;

}
```

Figure 9.7: DPN Generator : Constructor part 2



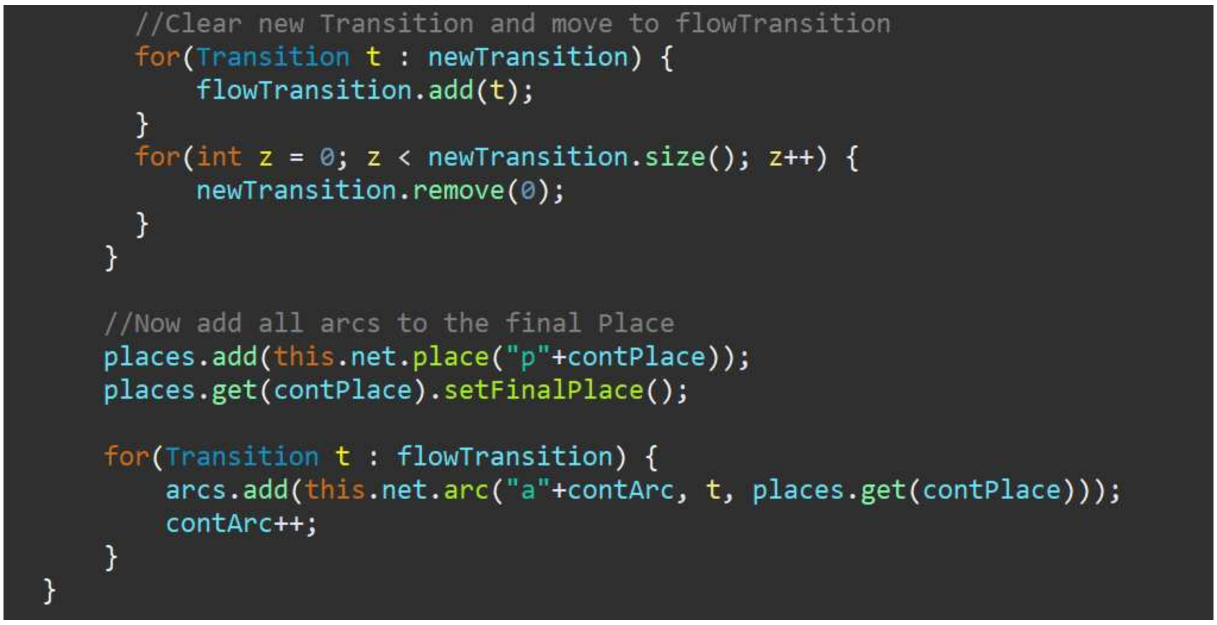
```
public void generateDPNwithoutToken() {

    for(int i = 0; i<n; i++) {
      while(!flowTransition.isEmpty()) {
        Transition last = flowTransition.remove(0);
        contRemoveTransition++;
        //Select random places
        int randomPlace = (int) (Math.random()*2+1);
        for(int j = 0; j<randomPlace; j++) {
            places.add(this.net.place("p"+contPlace));
            int random;
            do {
                random = (int) (Math.random()*10000000);
            } while(generated.contains(random));
            generated.add(random);
            int readWrite = (int) (Math.random()*2);
            String RW = "";
```

Figure 9.8: DPN Generator : Generate DPN without token part 1



```
            if(readWrite == 0) {
                RW = "R";
            }
            else {
                RW = "W";
            }
            transitions.add(this.net.transition("t"+contTransition, RW));
            transitions.get(contTransition).setConstraint(new MyConstraintOneV("X",random,"!="));
            transitions.get(contTransition).getConstraint().setRW(readWrite);
            newTransition.add(transitions.get(contTransition));
            arcs.add(this.net.arc("a"+contArc, last , places.get(contPlace)));
            arcs.add(this.net.arc("a"+contArc, places.get(contPlace), transitions.get(contTransition)));
            contPlace++;
            contArc = contArc + 2;
            contTransition++;
        }
    }
```

Figure 9.9: DPN Generator : Generate DPN without token part 2



```
            //Clear new Transition and move to flowTransition
            for(Transition t : newTransition) {
                flowTransition.add(t);
            }
            for(int z = 0; z < newTransition.size(); z++) {
                newTransition.remove(0);
            }
        }

        //Now add all arcs to the final Place
        places.add(this.net.place("p"+contPlace));
        places.get(contPlace).setFinalPlace();

        for(Transition t : flowTransition) {
            arcs.add(this.net.arc("a"+contArc, t, places.get(contPlace)));
            contArc++;
        }
    }
```

Figure 9.10: DPN Generator : generate last Place of DPN

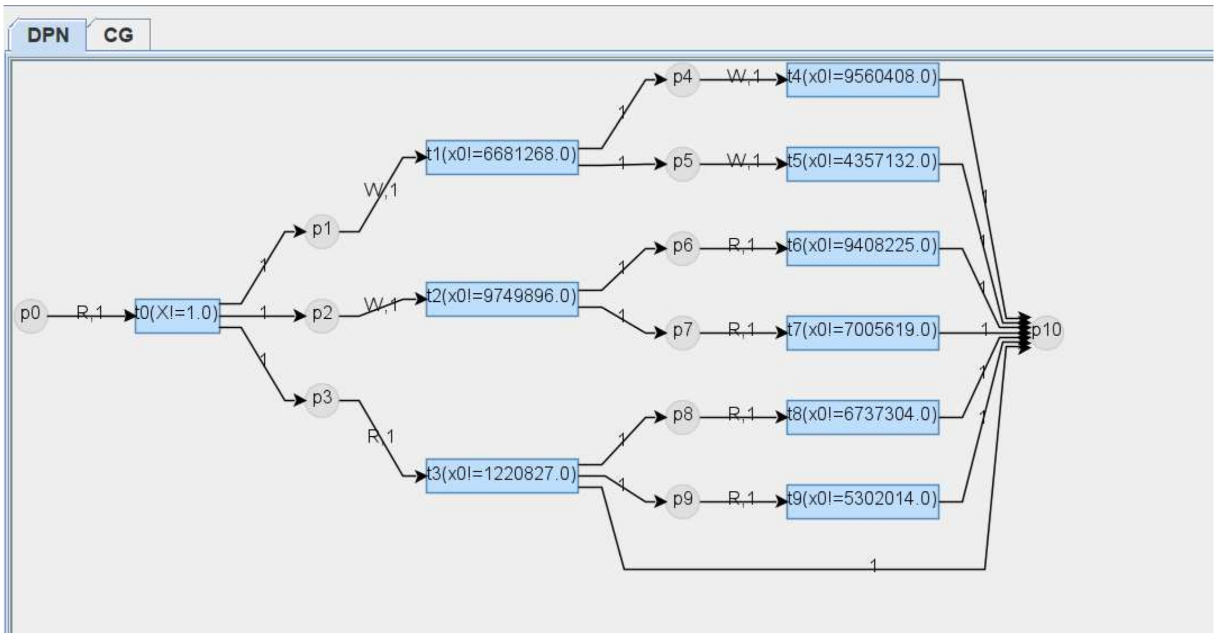
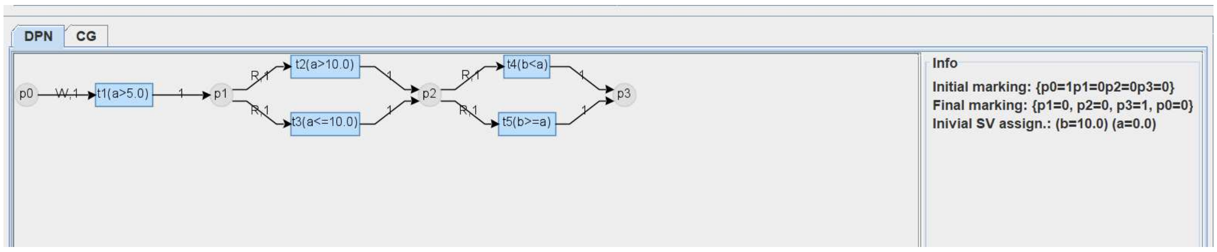


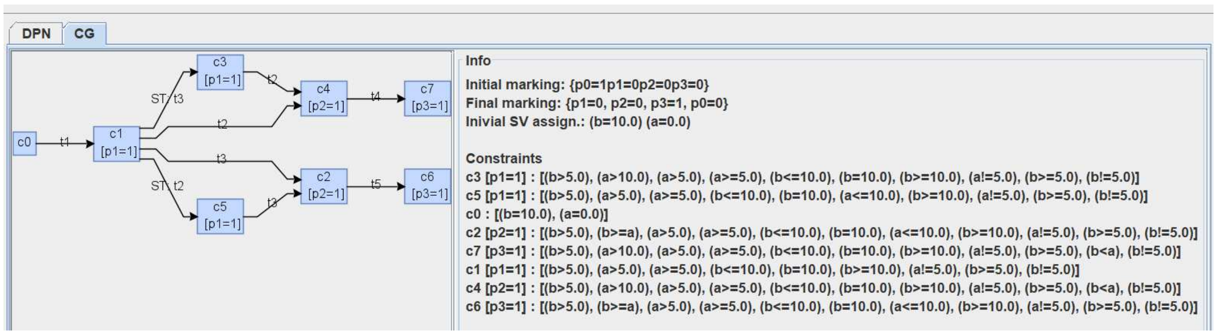
Figure 9.11: The Example of DPN generated by the algorithm



Info

Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Figure 9.12: DPN of Example 1

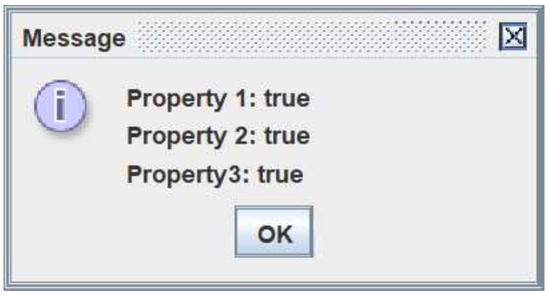


Info

Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Constraints
c3 [p1=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c5 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c0 : [(b=10.0), (a=0.0)]
c2 [p2=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c7 [p3=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c1 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c4 [p2=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c6 [p3=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]

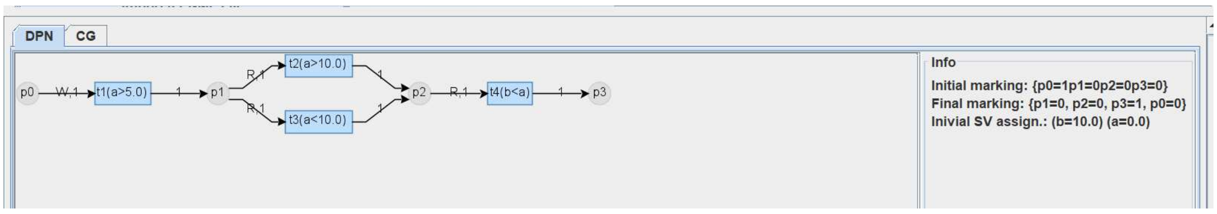
Figure 9.13: Constraint Graph of Example 1



Message

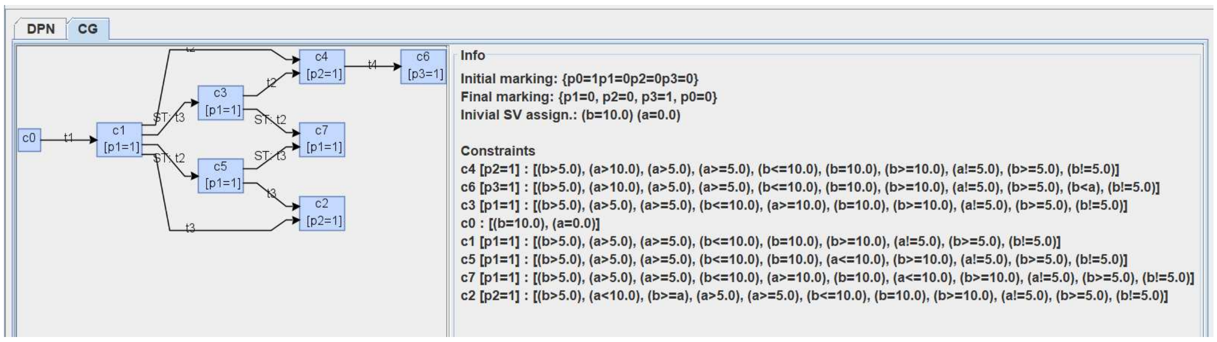
Property 1: true
Property 2: true
Property3: true

Figure 9.14: Soundness of Example 1



Info
Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

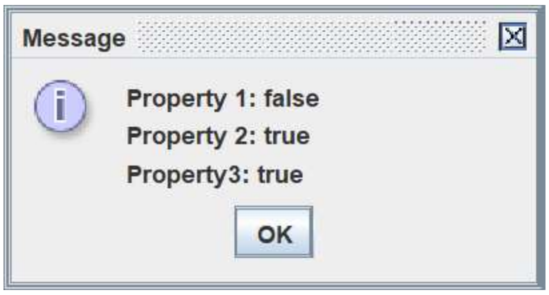
Figure 9.15: DPN of Example 2



Info
Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Constraints
c4 [p2=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c6 [p3=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c3 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (a>=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c0 : [(b=10.0), (a=0.0)]
c1 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c5 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c7 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (a>=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c2 [p2=1] : [(b>5.0), (a<10.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]

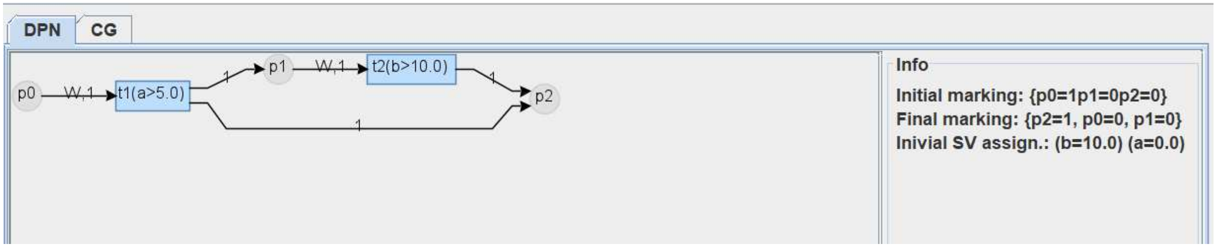
Figure 9.16: Constraint Graph of Example 2



Message

Property 1: false
Property 2: true
Property3: true

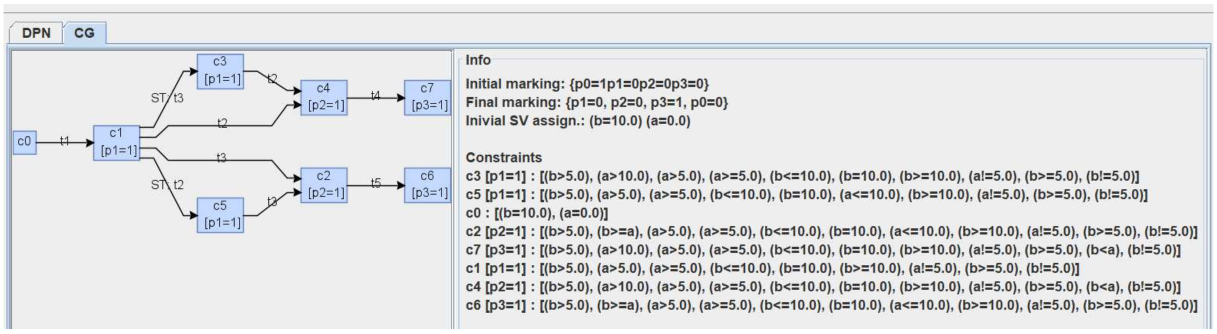
Figure 9.17: Soundness of Example 2



Info

Initial marking: {p0=1p1=0p2=0}
Final marking: {p2=1, p0=0, p1=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Figure 9.18: DPN of Example 3



Info

Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

Constraints
c3 [p1=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c5 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c0 : [(b=10.0), (a=0.0)]
c2 [p2=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c7 [p3=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c1 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c4 [p2=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b<a), (b!=5.0)]
c6 [p3=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]

Figure 9.19: Constraint Graph of Example 3

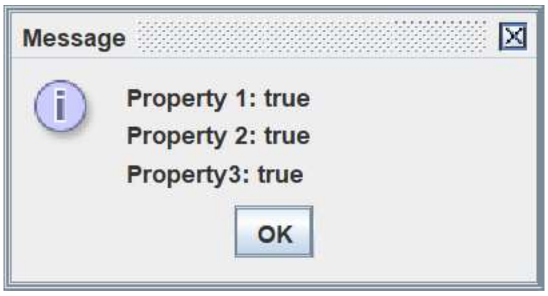
Constraint Graph in which one will have marking [P1 = 0, P2 = 1] and another will have marking [P1 = 0, P2 = 2] which are compatible and one is strictly greater than other as correctly detected by the program (Figure 9.20). Figure 9.18 shows the DPN associated with Figure 9.19.

9.2.4 Violation Property 3 Example

Figure 9.22 an example of a generated Constraint Graph where the third property of Data-Aware Soundness is violated. This is the example seen in Figure 9.12 but with a modification that leads to the violation of property 3 as correctly detected by the program (Figure 9.23). This happens because in one of the transitions a cost of 2 has been inserted instead of 1, in this way the incoming Place having only 1 token has a number < of the tokens required by the transition and therefore this transition cannot occur. Figure 9.21 shows the DPN associated with Figure 9.22.

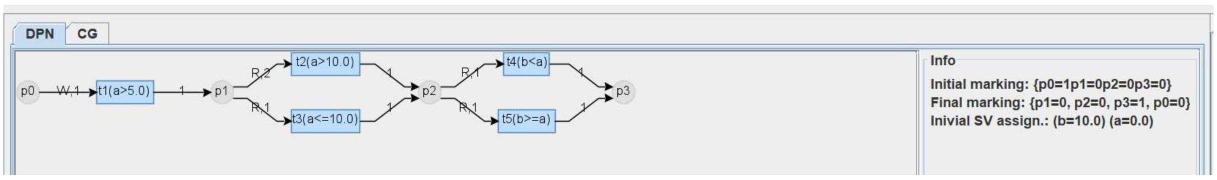
9.2.5 Unlimited Petri Data Net Example

An example of Unlimited Petri Data Net is shown in Figure 9.25, in fact it is not possible to determine a maximum number of tokens for the second node, it could potentially increase to infinity and since each node with a number of different tokens must originate a new node, yes would give rise



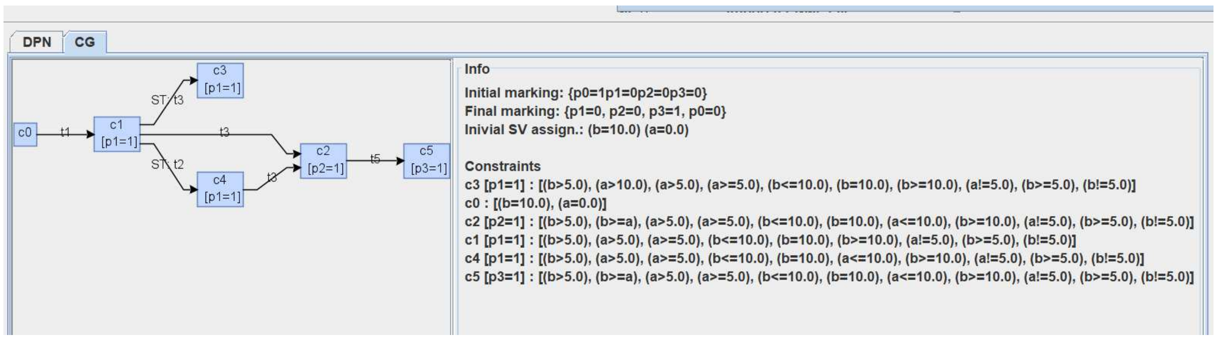
Property 1: true
Property 2: true
Property3: true

Figure 9.20: Soundness of Example 3



Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

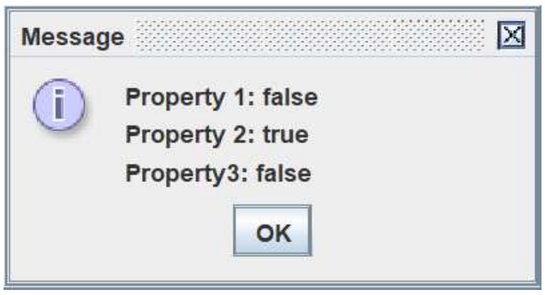
Figure 9.21: DPN of Example 4



Initial marking: {p0=1p1=0p2=0p3=0}
Final marking: {p1=0, p2=0, p3=1, p0=0}
Inivial SV assign.: (b=10.0) (a=0.0)

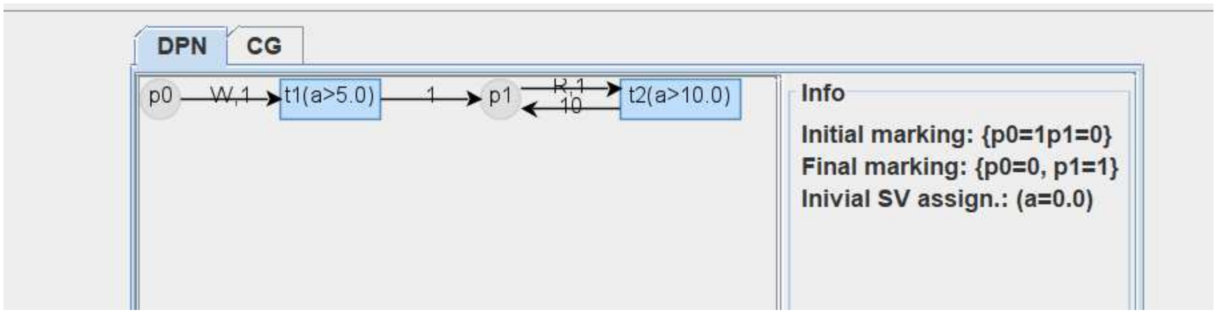
Constraints
c3 [p1=1] : [(b>5.0), (a>10.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c0 : [(b=10.0), (a=0.0)]
c2 [p2=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c1 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c4 [p1=1] : [(b>5.0), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]
c5 [p3=1] : [(b>5.0), (b>=a), (a>5.0), (a>=5.0), (b<=10.0), (b=10.0), (a<=10.0), (b>=10.0), (a!=5.0), (b>=5.0), (b!=5.0)]

Figure 9.22: Constraint Graph of Example 4



Property 1: false
Property 2: true
Property3: false

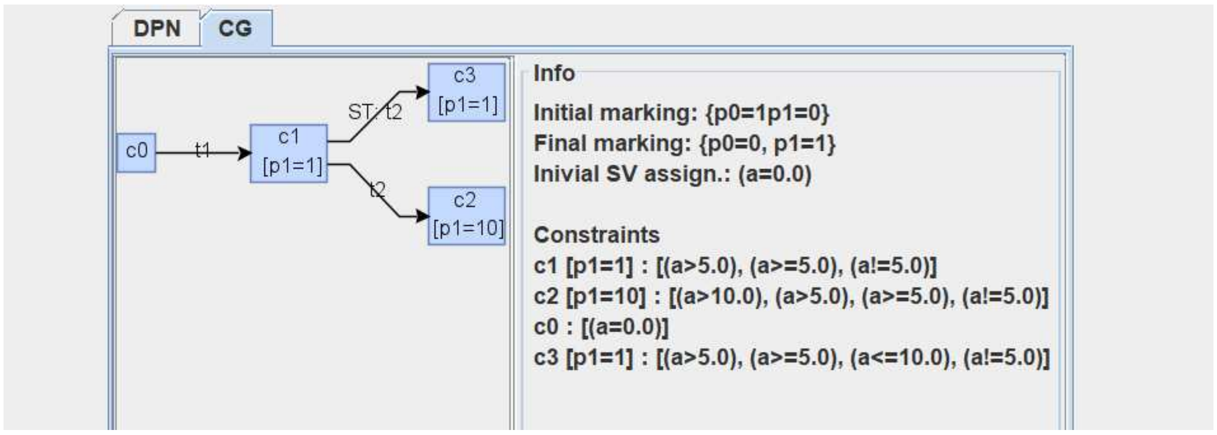
Figure 9.23: Soundness of Example 4



Info

Initial marking: {p0=1p1=0}
Final marking: {p0=0, p1=1}
Inivial SV assign.: (a=0.0)

Figure 9.24: DPN of Example 5



Info

Initial marking: {p0=1p1=0}
Final marking: {p0=0, p1=1}
Inivial SV assign.: (a=0.0)

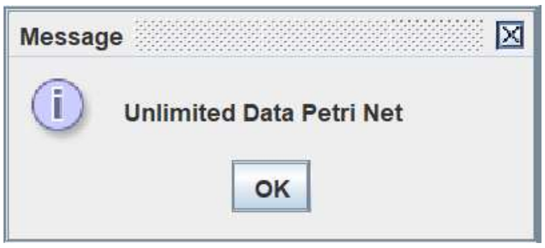
Constraints
c1 [p1=1] : [(a>5.0), (a>=5.0), (a!=5.0)]
c2 [p1=10] : [(a>10.0), (a>5.0), (a>=5.0), (a!=5.0)]
c0 : [(a=0.0)]
c3 [p1=1] : [(a>5.0), (a>=5.0), (a<=10.0), (a!=5.0)]

Figure 9.25: Constraint Graph of Example 5

to an infinite loop and consequently the Constraint Graph cannot be calculated since it is of infinite size as correctly detected by Figure 9.26. Figure 9.24 shows the DPN associated with Figure 9.25.

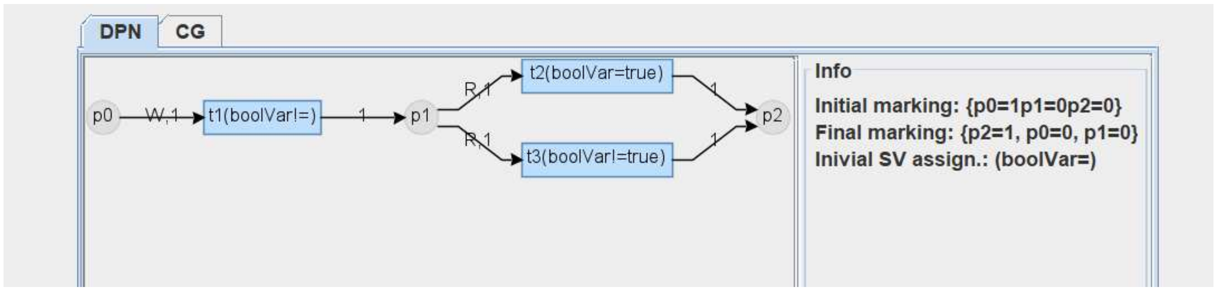
9.2.6 Boolean Variables Example

Here is an example using boolean variables. Boolean type variables have some advantages, but also disadvantages compared to real variables. Its main advantage is that it has a binary domain, i.e. it can only have *true* or *false* values, and it also has only two possible operators = or $\neq$. Consequently, all Boolean variable constraints can be represented with a single operator. In fact, if *variable* $\neq$ *true*, then it can be transformed into *variable* = *false* since it is equivalent. Using only the = operator do not allow to represent transitions where an arbitrary value is written to a boolean variable. Hence the PNML syntax has been extended by allowing expressions with empty right hand side boolVar="" in transition.



Unlimited Data Petri Net

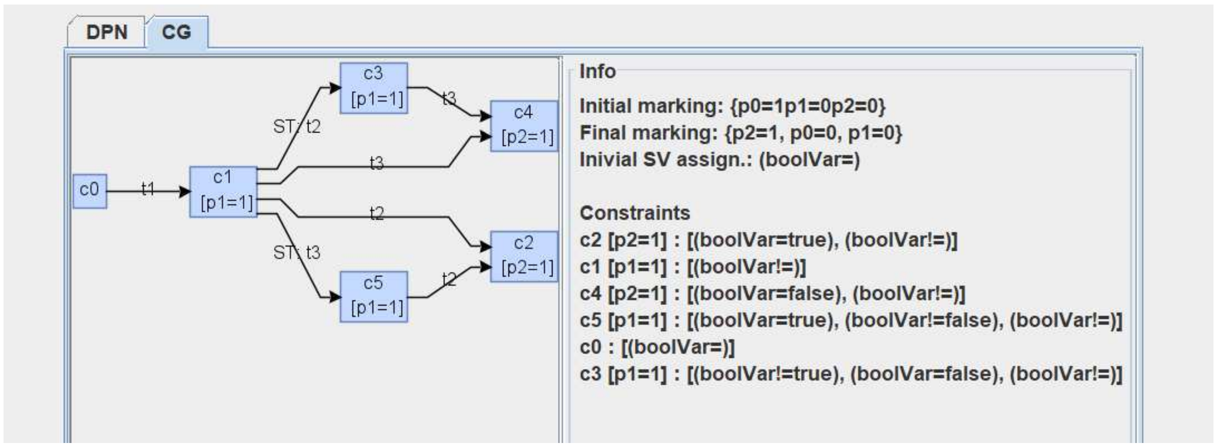
Figure 9.26: Soundness of Example 5



Info

Initial marking: {p0=1p1=0p2=0}
Final marking: {p2=1, p0=0, p1=0}
Inivial SV assign.: (boolVar=)

Figure 9.27: DPN of Example Boolean Variables

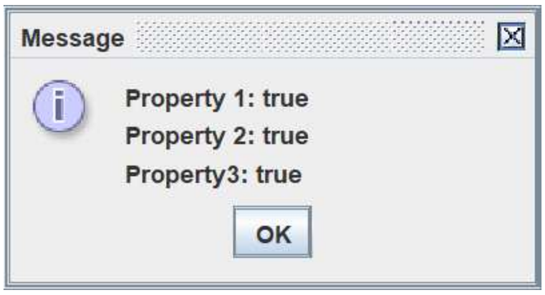


Info

Initial marking: {p0=1p1=0p2=0}
Final marking: {p2=1, p0=0, p1=0}
Inivial SV assign.: (boolVar=)

Constraints
c2 [p2=1] : [(boolVar=true), (boolVar!=)]
c1 [p1=1] : [(boolVar!=)]
c4 [p2=1] : [(boolVar=false), (boolVar!=)]
c5 [p1=1] : [(boolVar=true), (boolVar!=false), (boolVar!=)]
c0 : [(boolVar=)]
c3 [p1=1] : [(boolVar!=true), (boolVar=false), (boolVar!=)]

Figure 9.28: Constraint Graph of Example Boolean Variables



Property 1: true
Property 2: true
Property3: true

Figure 9.29: Soundness of Example Boolean Variables

CHAPTER 10

Constraint Graph Scalar Times Test

As Final part of the document we test the execution times of the algorithm at the scalar size of the Constraint Graph. In this chapter we will therefore display a series of tables and graphs showing the behavior of the software as the size of the Constraint Graph and the number of variables increases. Finally, a brief conclusion will be given on the work and the possible future developments it could involve.

10.1 Hardware

The computer used to run the program has the following hardware features:

- **CPU** : Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz;
- **RAM** : 8,00 GB (7,89 GB usable);
- **Operating System** : Windows 10, 64-bit operating system, x64-based processor

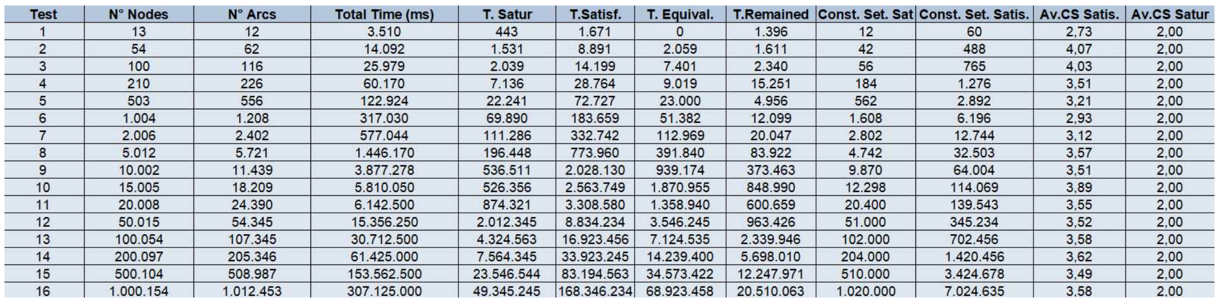
10.2 Experiments with a single variable and small Constraint Sets.

In this first part we test small instances, ie an average Constraint Set of the nodes of the Constraint Graph equal to 2 and with a single variable for the entire Constraint Graph. In subsequent extensions these two parameters will be extended and the various results obtained will be compared.

10.2.1 Time versus number of nodes

In this section, an analysis of time versus the number of nodes will be made, taking into consideration the saturation time, satisfiability time, the equivalence verification algorithm and finally the remaining time. The time measurement table is shown in figure 10.1 . As it is possible to see from the various figures, all the times follow an almost linear trend, i.e. as the number of nodes increases there is a directly proportional increase in the various types of time, in particular in the saturation and Satisfiable time.
The table of measurements contains the following elements:

- Identifier Test;
- N° Nodes : the number of nodes in Constraint Graph;
- N° Arcs : the number of arcs in Constraint Graph;
- Total Time (ms) : The total time execution of the program;
- Saturation Time (ms) : The time of Saturation taken by the program;

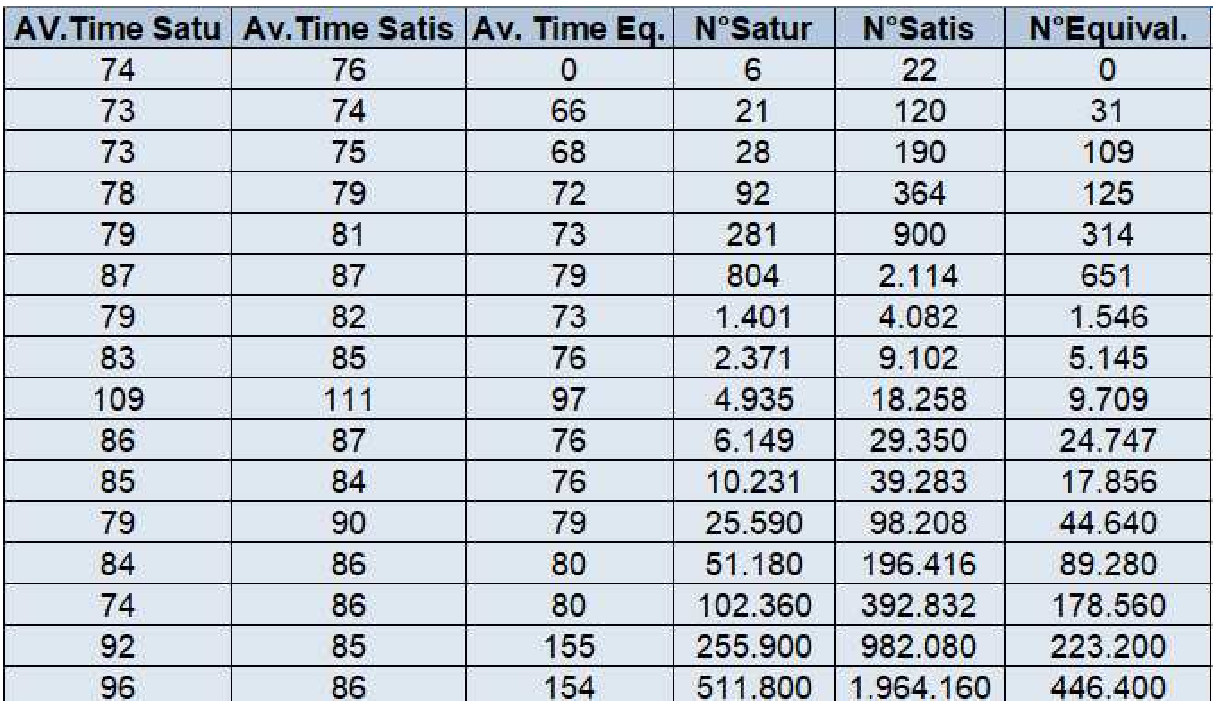


Test	N° Nodes	N° Arcs	Total Time (ms)	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Sat	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	3.510	443	1.671	0	1.396	12	60	2,73	2,00
2	54	62	14.092	1.531	8.891	2.059	1.611	42	488	4,07	2,00
3	100	116	25.979	2.039	14.199	7.401	2.340	56	765	4,03	2,00
4	210	226	60.170	7.136	28.764	9.019	15.251	184	1.276	3,51	2,00
5	503	556	122.924	22.241	72.727	23.000	4.956	562	2.892	3,21	2,00
6	1.004	1.208	317.030	69.890	183.659	51.382	12.099	1.608	6.196	2,93	2,00
7	2.006	2.402	577.044	111.286	332.742	112.969	20.047	2.802	12.744	3,12	2,00
8	5.012	5.721	1.446.170	196.448	773.960	391.840	83.922	4.742	32.503	3,57	2,00
9	10.002	11.439	3.877.278	536.511	2.028.130	939.174	373.463	9.870	64.004	3,51	2,00
10	15.005	18.209	5.810.050	526.356	2.563.749	1.870.955	848.990	12.298	114.069	3,89	2,00
11	20.008	24.390	6.142.500	874.321	3.308.580	1.358.940	600.659	20.400	139.543	3,55	2,00
12	50.015	54.345	15.356.250	2.012.345	8.834.234	3.546.245	963.426	51.000	345.234	3,52	2,00
13	100.054	107.345	30.712.500	4.324.563	16.923.456	7.124.535	2.339.946	102.000	702.456	3,58	2,00
14	200.097	205.346	61.425.000	7.564.345	33.923.245	14.239.400	5.698.010	204.000	1.420.456	3,62	2,00
15	500.104	508.987	153.562.500	23.546.544	83.194.563	34.573.422	12.247.971	510.000	3.424.678	3,49	2,00
16	1.000.154	1.012.453	307.125.000	49.345.245	168.346.234	68.923.458	20.510.063	1.020.000	7.024.635	3.58	2.00

Figure 10.1: Time Measurement Table (Part 1)

- Satisfiable Time (ms): The time of Satisfiable taken by the program;
- Check Equivalence Time (ms) : The time of Check Equivalence of two nodes taken by the program;
- Remained Time (ms) : The time taken by the program to run other operations;
- Constraint Set Saturation : the sum of Constraint Set Size of all nodes passed as argument to Saturation ;
- Constraint Set Satisfiable : the sum of Constraint Set Size of all nodes passed as argument to Satisfiable;
- Average Constraint Set Saturation : Ratio between the Constraint Set Size Saturation and the number of saturation functions performed;
- Average Constraint Set Saturation : Ratio between the Constraint Set Size Satisfiable and the number of satisfiable functions performed;
- Average Time Saturation (ms): Ratio between the Saturation time and the number of saturation functions executed;
- Average Time Satisfiable (ms): Ratio between the Satisfiable time and the number of satisfiable functions executed;
- Average Time Check Equivalence (ms): Ratio between the Check Equivalence of two nodes time and the number of check equivalence functions executed;
- number of saturation functions;
- number of satisfiable functions;
- number of check equivalence of two nodes functions.

Figure 10.1 and Figure 10.2 contains the Tabular data of the test. Figures 10.3 to 10.8 plots time measurements with respect to the number of nodes of the constraint graph for Saturation Time, Satisfiable Time, Check Equivalence of two Nodes Time, Remained Time and Total Time. Remained Time is obtained by the Total Time minus Satisfiable, Check Equivalence and Saturation Time. Finally Figure 10.8 summarize all results.



AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
74	76	0	6	22	0
73	74	66	21	120	31
73	75	68	28	190	109
78	79	72	92	364	125
79	81	73	281	900	314
87	87	79	804	2.114	651
79	82	73	1.401	4.082	1.546
83	85	76	2.371	9.102	5.145
109	111	97	4.935	18.258	9.709
86	87	76	6.149	29.350	24.747
85	84	76	10.231	39.283	17.856
79	90	79	25.590	98.208	44.640
84	86	80	51.180	196.416	89.280
74	86	80	102.360	392.832	178.560
92	85	155	255.900	982.080	223.200
96	86	154	511.800	1.964.160	446.400

Figure 10.2: Time Measurement Table (Part 2)

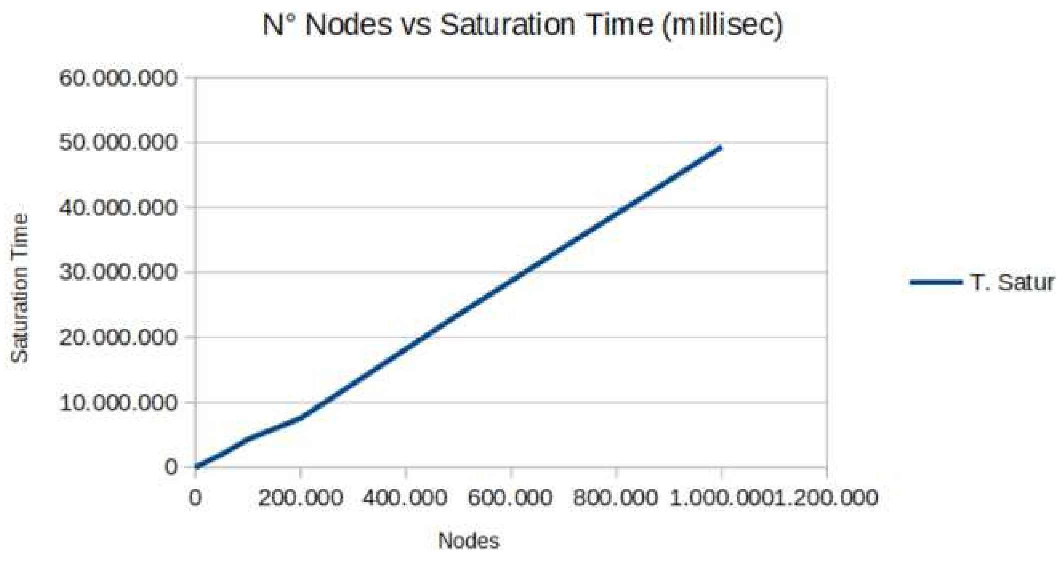


Figure 10.3: N° Nodes vs Saturation Time

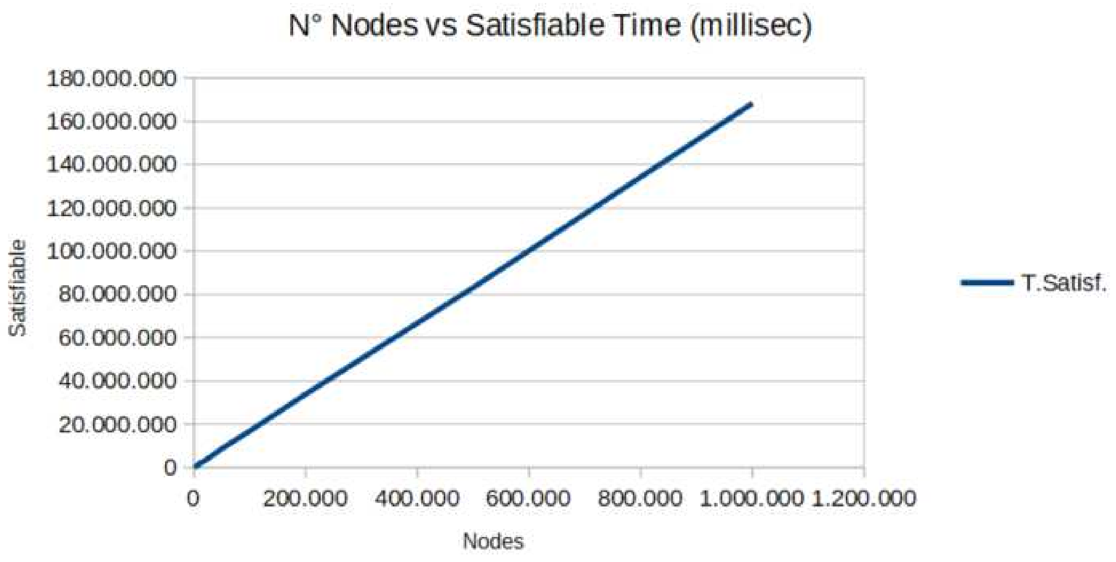


Figure 10.4: N° Nodes vs Satisfiable Time

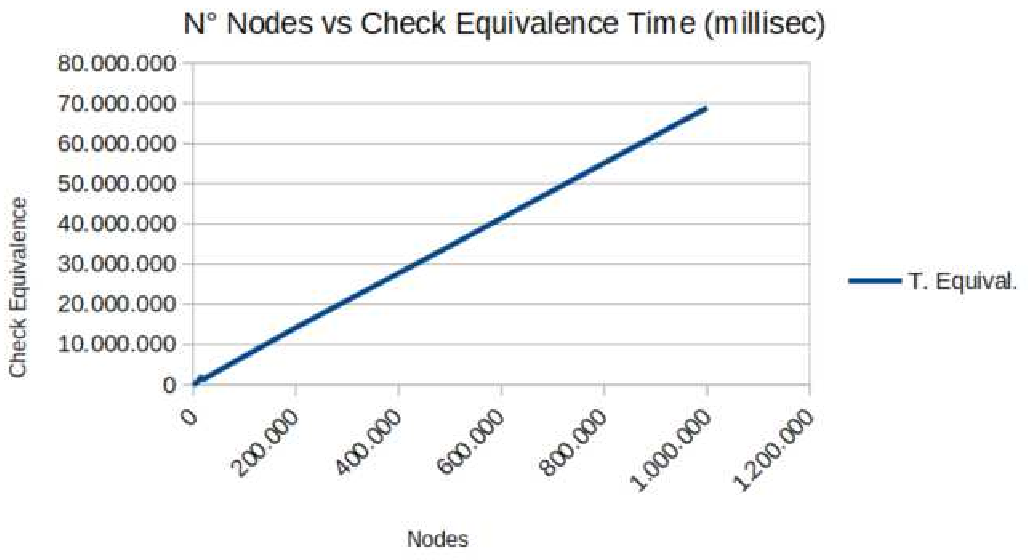


Figure 10.5: N° Nodes vs Check Equivalence Time

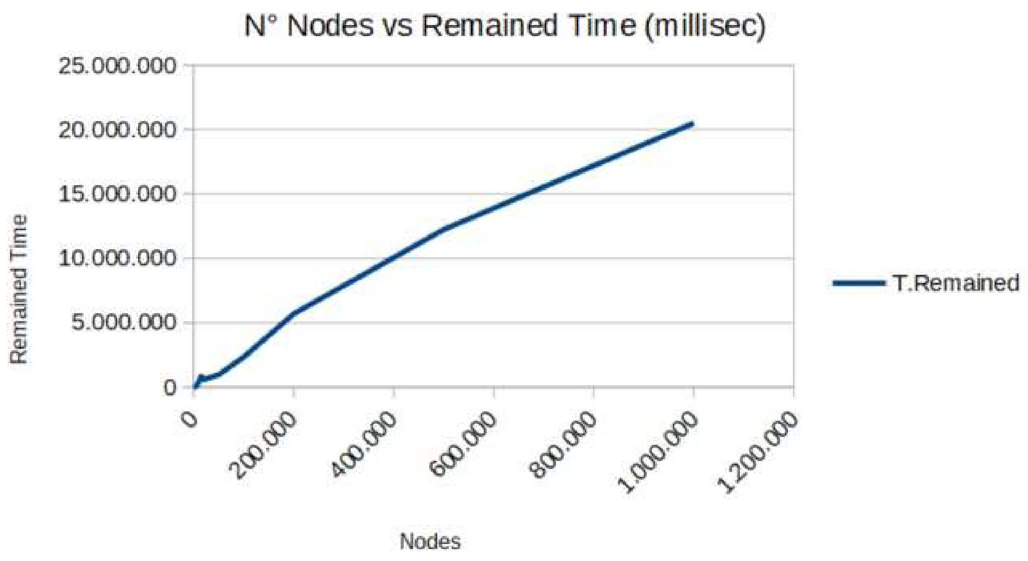


Figure 10.6: N° Nodes vs Remained Time

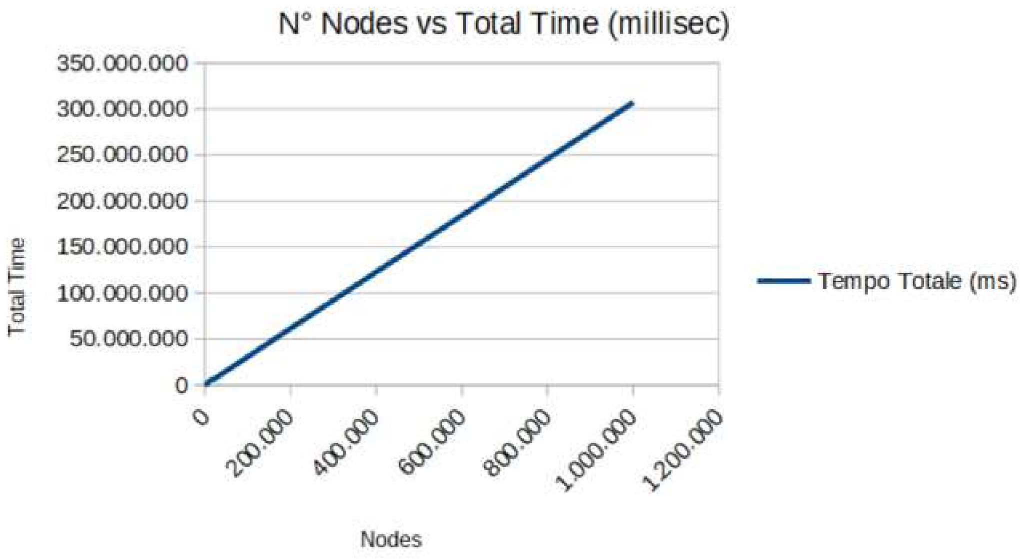


Figure 10.7: N° Nodes vs Total Time

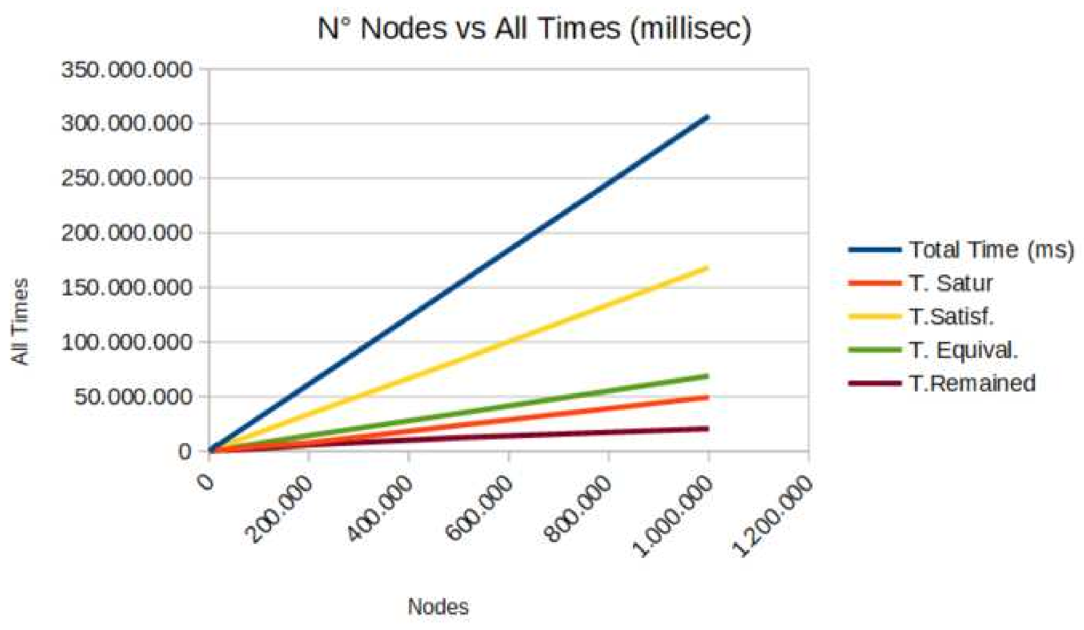


Figure 10.8: N° Nodes vs All Times

10.2.2 Number of Operations Measures

Figures 10.9 - 10.12 analyzes the number of Saturation, satisfiability and equivalence check operations that have been performed. With only one variable the number of satisfiability functions is greater than with the other two functions.

10.3 Experiment with varying the size of the Constraint Set

The experiments conducted so far have limited themselves to analyzing cases in which the Constraint Set has about 2 constraints for each satisfiability and saturation operation. It should be borne in mind that the Saturation and Satisfiability algorithms take a Constraint Set as input, while the equivalence verification algorithm takes two nodes of the Constraint Graph as input and analyzes its Constraint Sets with Z3, so it is essential to understand how the program behaves when the size of Constraint Sets change. Now we will examine the behavior of the program as the size of the Constraint Set increases, in particular we will try three other cases: 10, 100 and 1000 constraints for each Constraint Set.

Figure 10.14 to 10.19 shown the complete table of all cases.

10.3.1 Time analysis as the size of the Constraint Set varies

Analyzing the timing of the various functions, it can be seen that the most inefficient function is the saturation function and we analyzed his behavior by analyzing his average time and reporting it on a relative graph (Figure 10.20). The most efficient function, on the other hand, is that of satisfiability, which grows by a few units as the Constraint Set increases (Figure 10.21). The control function of the equivalence of two nodes grows to a greater extent than the satisfiability function, but still remains much lower than the saturation function and therefore its advantage in terms of efficiency is confirmed (Figure 10.22). Finally, in its total time it is possible to observe the entire timing occupied by the program as the Constraint Set varies (Figure 10.23).

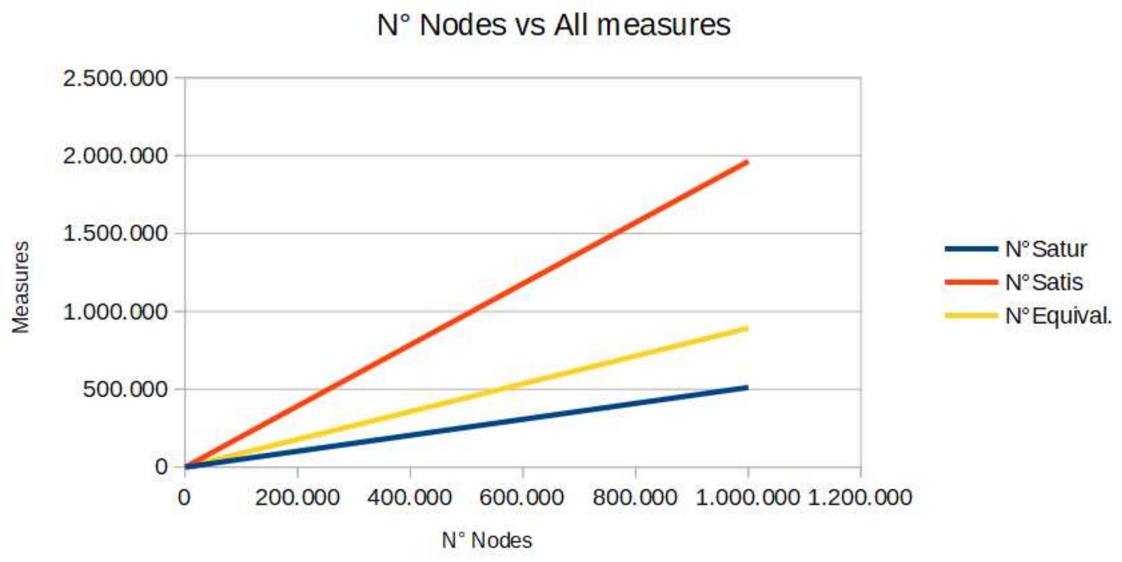


Figure 10.9: N° Nodes vs All Measures

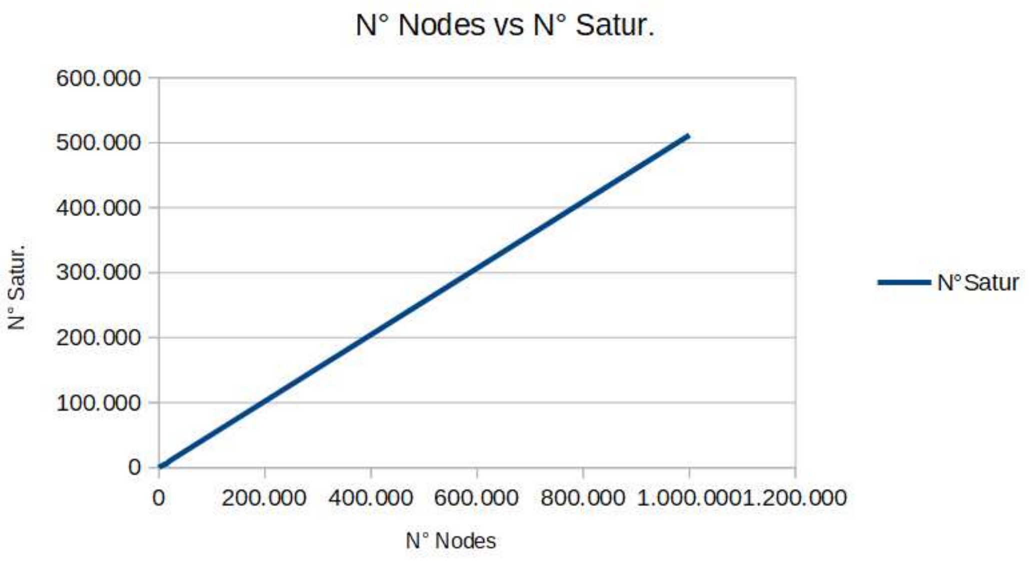


Figure 10.10: N° Nodes vs N° Saturation

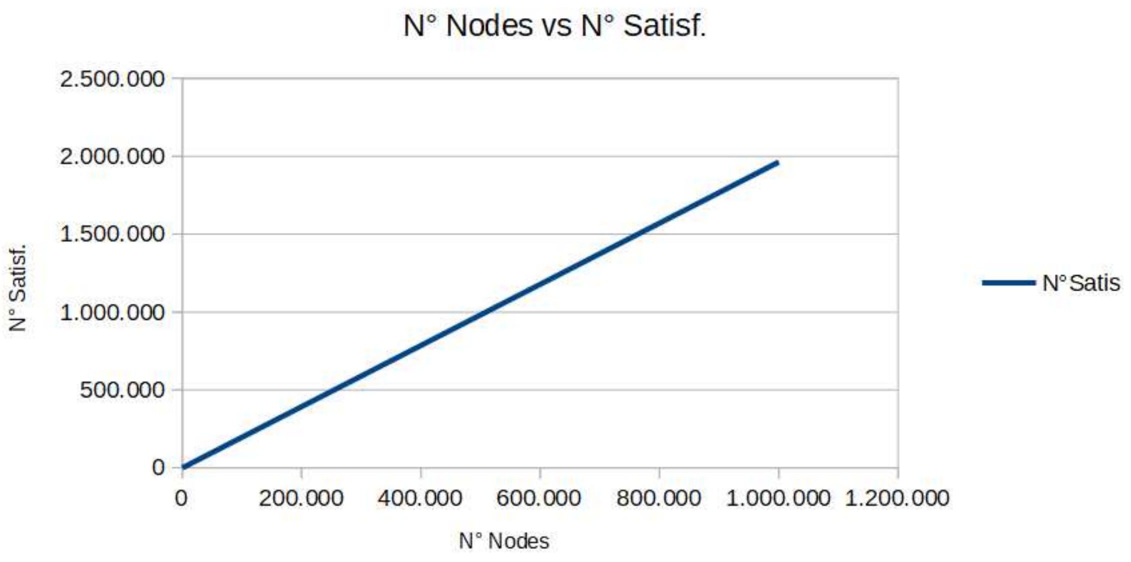


Figure 10.11: N° Nodes vs N° Satisfiable

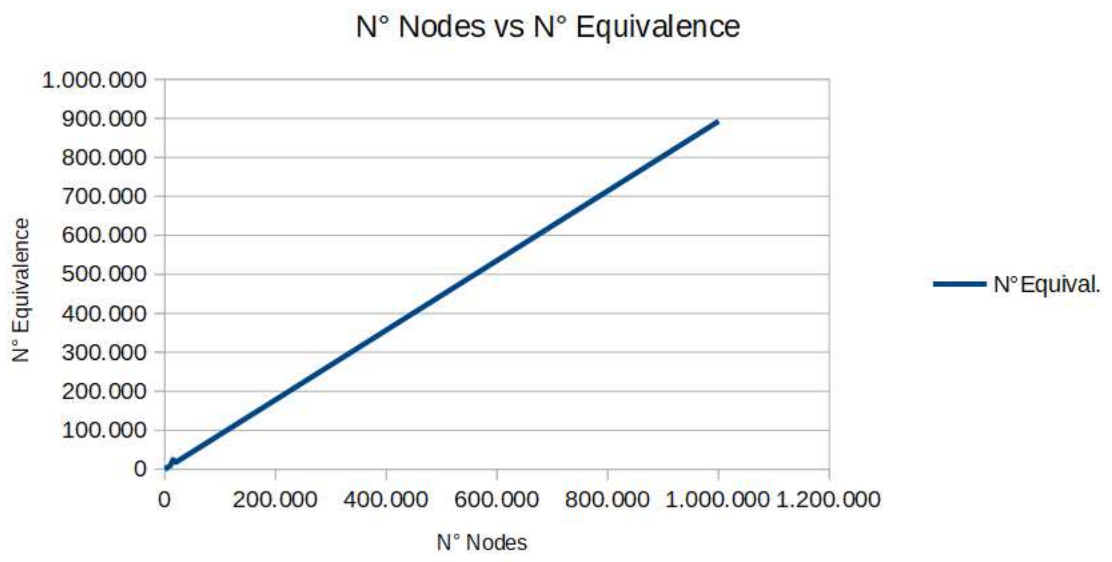
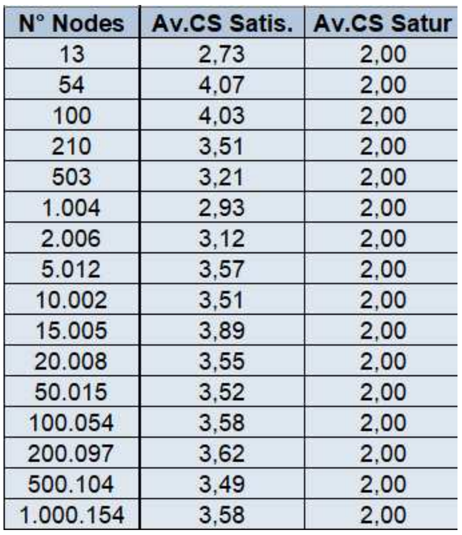
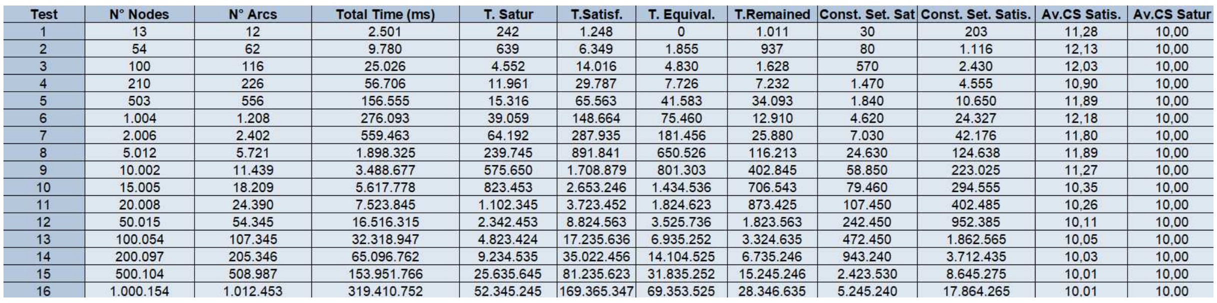


Figure 10.12: N° Nodes vs N° Check Equivalence



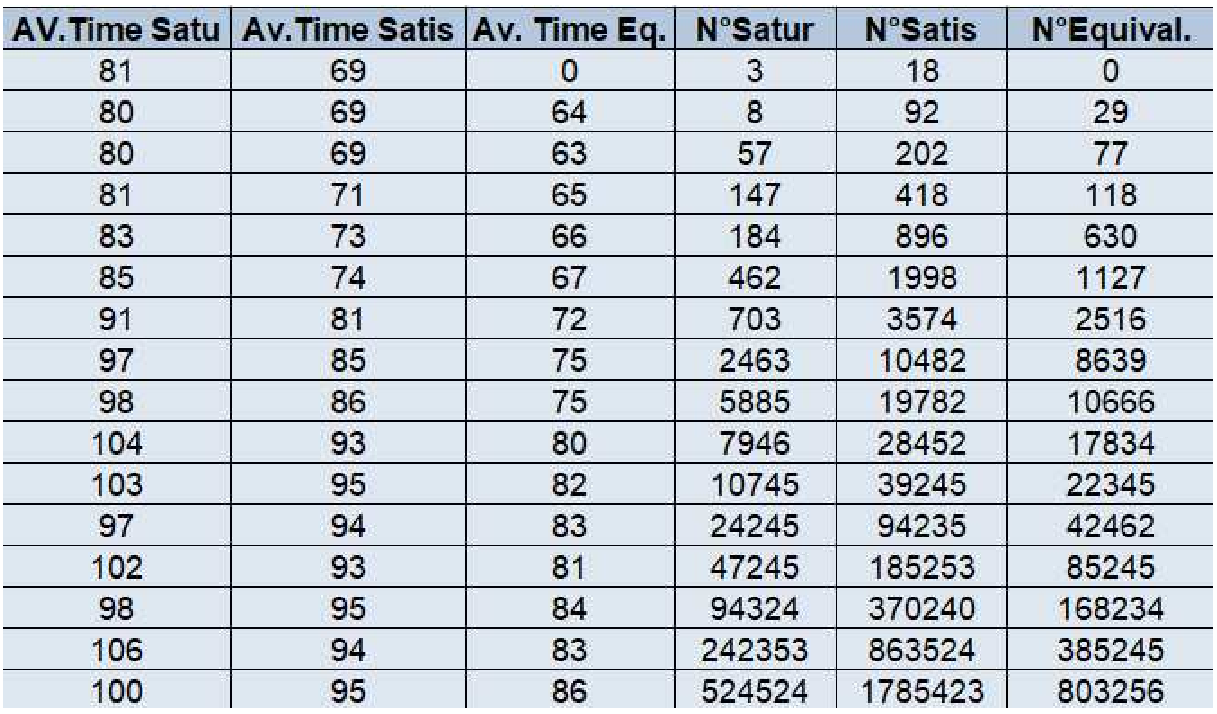
N° Nodes	Av.CS Satis.	Av.CS Satur
13	2,73	2,00
54	4,07	2,00
100	4,03	2,00
210	3,51	2,00
503	3,21	2,00
1.004	2,93	2,00
2.006	3,12	2,00
5.012	3,57	2,00
10.002	3,51	2,00
15.005	3,89	2,00
20.008	3,55	2,00
50.015	3,52	2,00
100.054	3,58	2,00
200.097	3,62	2,00
500.104	3,49	2,00
1.000.154	3,58	2,00

Figure 10.13: Tables that shows the average Constraint Set Dimension (the Dimension is defined by the quantities of constraints) for each test executed.



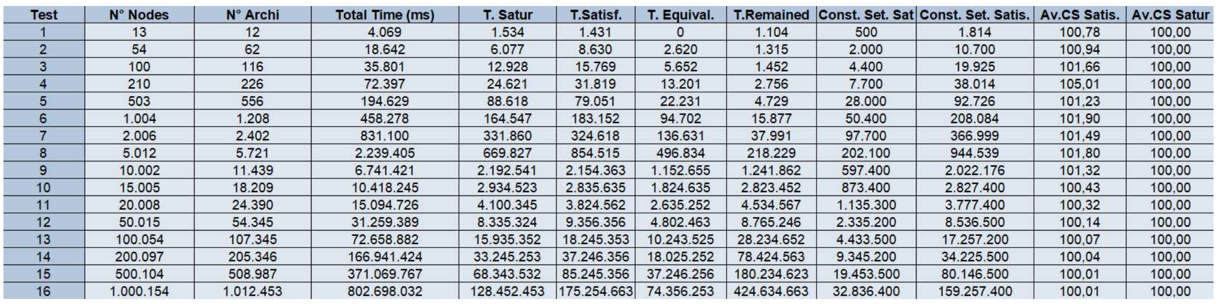
Test	N° Nodes	N° Arcs	Total Time (ms)	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Sat	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	2.501	242	1.248	0	1.011	30	203	11,28	10,00
2	54	62	9.780	639	6.349	1.855	937	80	1.116	12,13	10,00
3	100	116	25.026	4.552	14.016	4.830	1.628	570	2.430	12,03	10,00
4	210	226	56.706	11.961	29.787	7.726	7.232	1.470	4.555	10,90	10,00
5	503	556	156.555	15.316	65.563	41.583	34.093	1.840	10.650	11,89	10,00
6	1.004	1.208	276.093	39.059	148.664	75.460	12.910	4.620	24.327	12,18	10,00
7	2.006	2.402	559.463	64.192	287.935	181.456	25.880	7.030	42.176	11,80	10,00
8	5.012	5.721	1.898.325	239.745	891.841	650.526	116.213	24.630	124.638	11,89	10,00
9	10.002	11.439	3.488.677	575.650	1.708.879	801.303	402.845	58.850	223.025	11,27	10,00
10	15.005	18.209	5.617.778	823.453	2.653.246	1.434.536	706.543	79.460	294.555	10,35	10,00
11	20.008	24.390	7.523.845	1.102.345	3.723.452	1.824.623	873.425	107.450	402.485	10,26	10,00
12	50.015	54.345	16.516.315	2.342.453	8.824.563	3.525.736	1.823.563	242.450	952.385	10,11	10,00
13	100.054	107.345	32.318.947	4.823.424	17.235.636	6.935.252	3.324.635	472.450	1.862.565	10,05	10,00
14	200.097	205.346	65.096.762	9.234.535	35.022.456	14.104.525	6.735.246	943.240	3.712.435	10,03	10,00
15	500.104	508.987	153.951.766	25.635.645	81.235.623	31.835.252	15.245.246	2.423.530	8.645.275	10,01	10,00
16	1.000.154	1.012.453	319.410.752	52.345.245	169.365.347	69.353.525	28.346.635	5.245.240	17.864.265	10.01	10.00

Figure 10.14: Table with all measures with Dimension 10 of Constraint Set. (part 1)



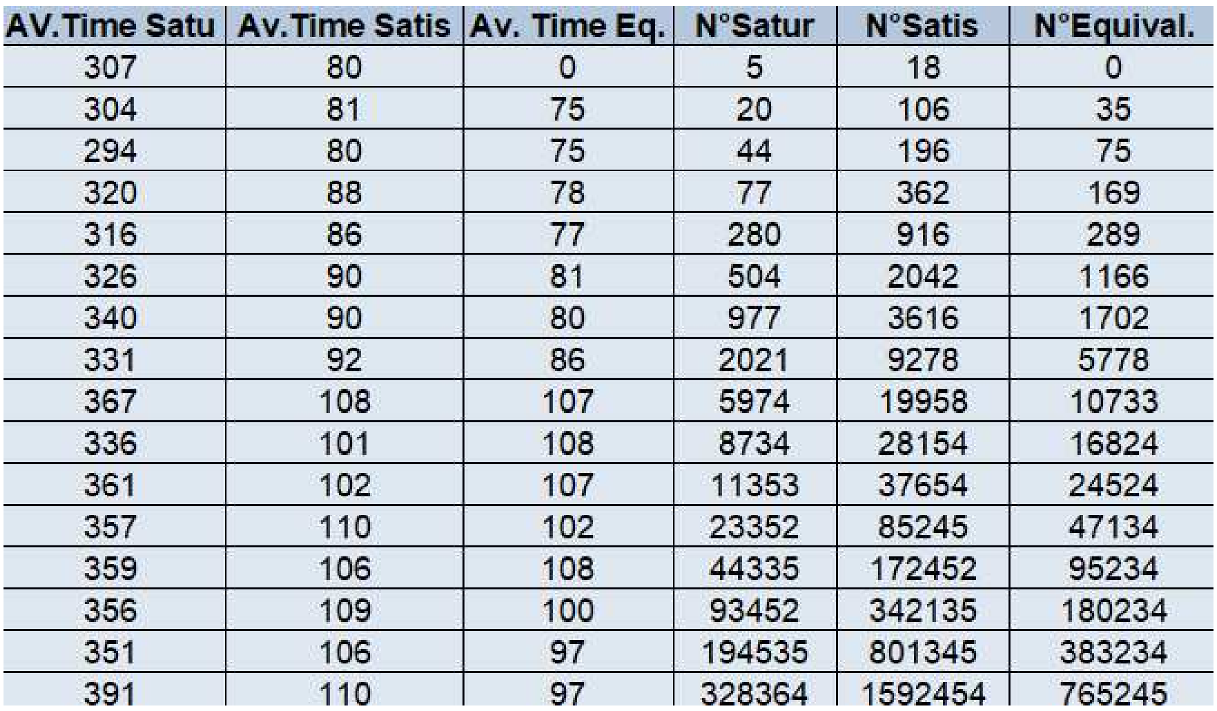
AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
81	69	0	3	18	0
80	69	64	8	92	29
80	69	63	57	202	77
81	71	65	147	418	118
83	73	66	184	896	630
85	74	67	462	1998	1127
91	81	72	703	3574	2516
97	85	75	2463	10482	8639
98	86	75	5885	19782	10666
104	93	80	7946	28452	17834
103	95	82	10745	39245	22345
97	94	83	24245	94235	42462
102	93	81	47245	185253	85245
98	95	84	94324	370240	168234
106	94	83	242353	863524	385245
100	95	86	524524	1785423	803256

Figure 10.15: Table with all measures with Dimension 10 of Constraint Set. (part 2)



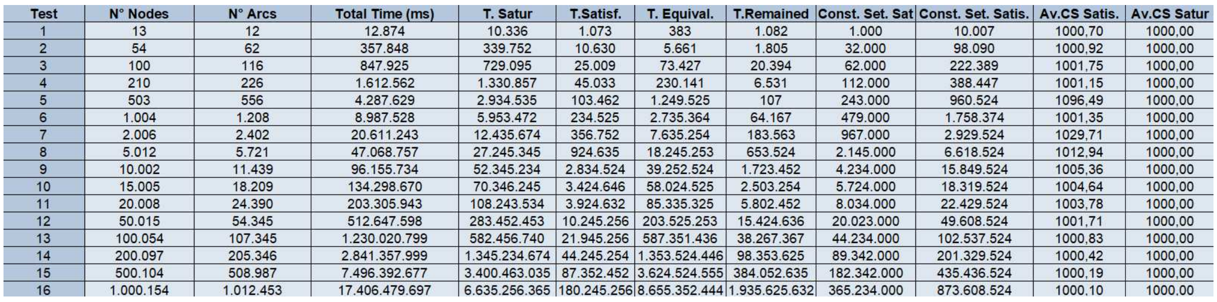
Test	N° Nodes	N° Archi	Total Time (ms)	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Sat	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	4.069	1.534	1.431	0	1.104	500	1.814	100,78	100,00
2	54	62	18.642	6.077	8.630	2.620	1.315	2.000	10.700	100,94	100,00
3	100	116	35.801	12.928	15.769	5.652	1.452	4.400	19.925	101,66	100,00
4	210	226	72.397	24.621	31.819	13.201	2.756	7.700	38.014	105,01	100,00
5	503	556	194.629	88.618	79.051	22.231	4.729	28.000	92.726	101,23	100,00
6	1.004	1.208	458.278	164.547	183.152	94.702	15.877	50.400	208.084	101,90	100,00
7	2.006	2.402	831.100	331.860	324.618	136.631	37.991	97.700	366.999	101,49	100,00
8	5.012	5.721	2.239.405	669.827	854.515	496.834	218.229	202.100	944.539	101,80	100,00
9	10.002	11.439	6.741.421	2.192.541	2.154.363	1.152.655	1.241.862	597.400	2.022.176	101,32	100,00
10	15.005	18.209	10.418.245	2.934.523	2.835.635	1.824.635	2.823.452	873.400	2.827.400	100,43	100,00
11	20.008	24.390	15.094.726	4.100.345	3.824.562	2.635.252	4.534.567	1.135.300	3.777.400	100,32	100,00
12	50.015	54.345	31.259.389	8.335.324	9.356.356	4.802.463	8.765.246	2.335.200	8.536.500	100,14	100,00
13	100.054	107.345	72.658.882	15.935.352	18.245.353	10.243.525	28.234.652	4.433.500	17.257.200	100,07	100,00
14	200.097	205.346	166.941.424	33.245.253	37.246.356	18.025.252	78.424.563	9.345.200	34.225.500	100,04	100,00
15	500.104	508.987	371.069.767	68.343.532	85.245.356	37.246.256	180.234.623	19.453.500	80.146.500	100,01	100,00
16	1.000.154	1.012.453	802.698.032	128.452.453	175.254.663	74.356.253	424.634.663	32.836.400	159.257.400	100,01	100,00

Figure 10.16: Table with all measures with Dimension 100 of Constraint Set. (part 1)



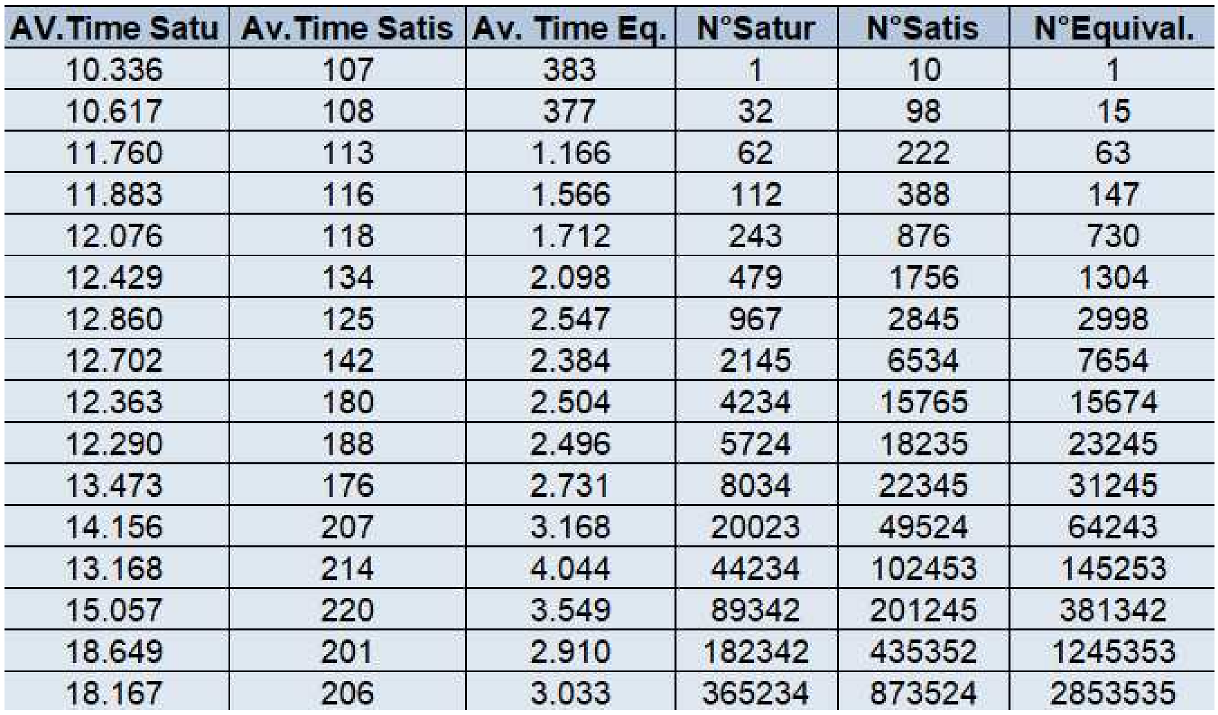
AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
307	80	0	5	18	0
304	81	75	20	106	35
294	80	75	44	196	75
320	88	78	77	362	169
316	86	77	280	916	289
326	90	81	504	2042	1166
340	90	80	977	3616	1702
331	92	86	2021	9278	5778
367	108	107	5974	19958	10733
336	101	108	8734	28154	16824
361	102	107	11353	37654	24524
357	110	102	23352	85245	47134
359	106	108	44335	172452	95234
356	109	100	93452	342135	180234
351	106	97	194535	801345	383234
391	110	97	328364	1592454	765245

Figure 10.17: Table with all measures with Dimension 10 of Constraint Set. (part 2)



Test	N° Nodes	N° Arcs	Total Time (ms)	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Sat	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	12.874	10.336	1.073	383	1.082	1.000	10.007	1000,70	1000,00
2	54	62	357.848	339.752	10.630	5.661	1.805	32.000	98.090	1000,92	1000,00
3	100	116	847.925	729.095	25.009	73.427	20.394	62.000	222.389	1001,75	1000,00
4	210	226	1.612.562	1.330.857	45.033	230.141	6.531	112.000	388.447	1001,15	1000,00
5	503	556	4.287.629	2.934.535	103.462	1.249.525	107	243.000	960.524	1096,49	1000,00
6	1.004	1.208	8.987.528	5.953.472	234.525	2.735.364	64.167	479.000	1.758.374	1001,35	1000,00
7	2.006	2.402	20.611.243	12.435.674	356.752	7.635.254	183.563	967.000	2.929.524	1029,71	1000,00
8	5.012	5.721	47.068.757	27.245.345	924.635	18.245.253	653.524	2.145.000	6.618.524	1012,94	1000,00
9	10.002	11.439	96.155.734	52.345.234	2.834.524	39.252.524	1.723.452	4.234.000	15.849.524	1005,36	1000,00
10	15.005	18.209	134.298.670	70.346.245	3.424.646	58.024.525	2.503.254	5.724.000	18.319.524	1004,64	1000,00
11	20.008	24.390	203.305.943	108.243.534	3.924.632	85.335.325	5.802.452	8.034.000	22.429.524	1003,78	1000,00
12	50.015	54.345	512.647.598	283.452.453	10.245.256	203.525.253	15.424.636	20.023.000	49.608.524	1001,71	1000,00
13	100.054	107.345	1.230.020.799	582.456.740	21.945.256	587.351.436	38.267.367	44.234.000	102.537.524	1000,83	1000,00
14	200.097	205.346	2.841.357.999	1.345.234.674	44.245.254	1.353.524.446	98.353.625	89.342.000	201.329.524	1000,42	1000,00
15	500.104	508.987	7.496.392.677	3.400.463.035	87.352.452	3.624.524.555	384.052.635	182.342.000	435.436.524	1000,19	1000,00
16	1.000.154	1.012.453	17.406.479.697	6.635.256.365	180.245.256	8.655.352.444	1.935.625.632	365.234.000	873.608.524	1000,10	1000,00

Figure 10.18: Table with all measures with Dimension 1000 of Constraint Set. (part 1)



AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
10.336	107	383	1	10	1
10.617	108	377	32	98	15
11.760	113	1.166	62	222	63
11.883	116	1.566	112	388	147
12.076	118	1.712	243	876	730
12.429	134	2.098	479	1756	1304
12.860	125	2.547	967	2845	2998
12.702	142	2.384	2145	6534	7654
12.363	180	2.504	4234	15765	15674
12.290	188	2.496	5724	18235	23245
13.473	176	2.731	8034	22345	31245
14.156	207	3.168	20023	49524	64243
13.168	214	4.044	44234	102453	145253
15.057	220	3.549	89342	201245	381342
18.649	201	2.910	182342	435352	1245353
18.167	206	3.033	365234	873524	2853535

Figure 10.19: Table with all measures with Dimension 10 of Constraint Set. (part 2)

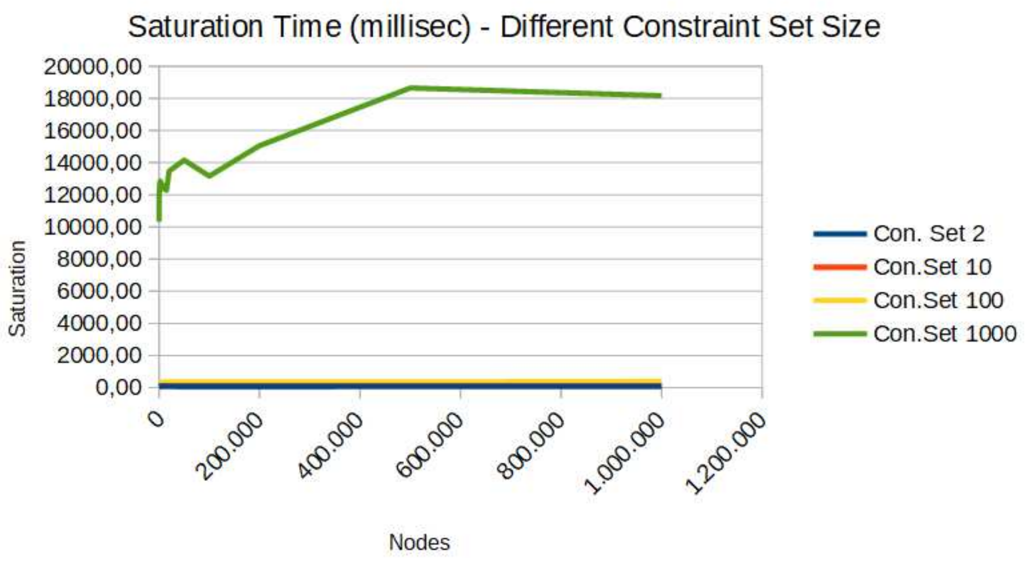


Figure 10.20: The analysis of Saturation Time when Constraint Set Dimension increase.

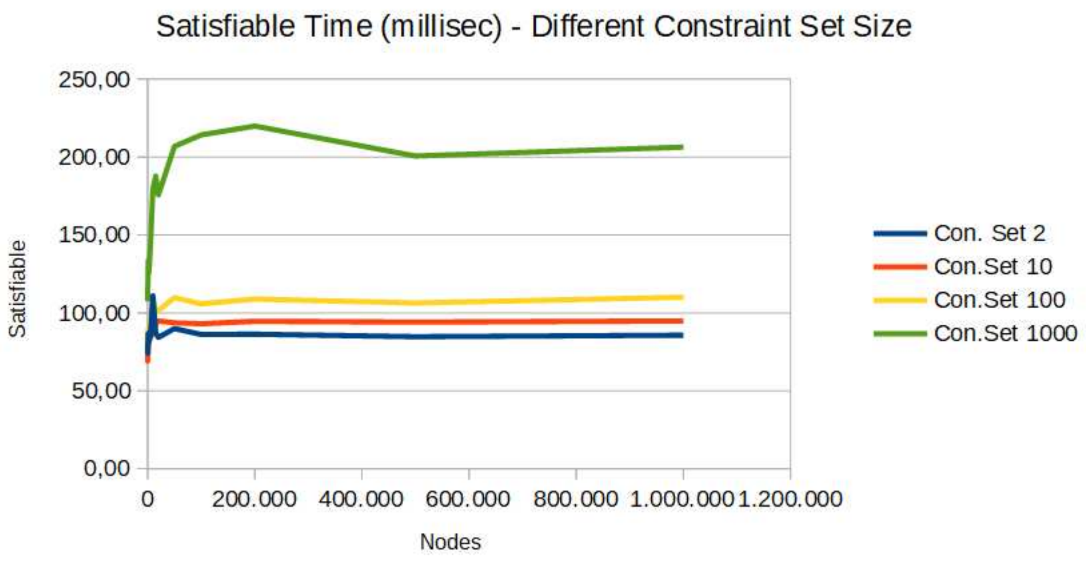


Figure 10.21: The analysis of Satisfiable Time when Constraint Set Dimension increase.

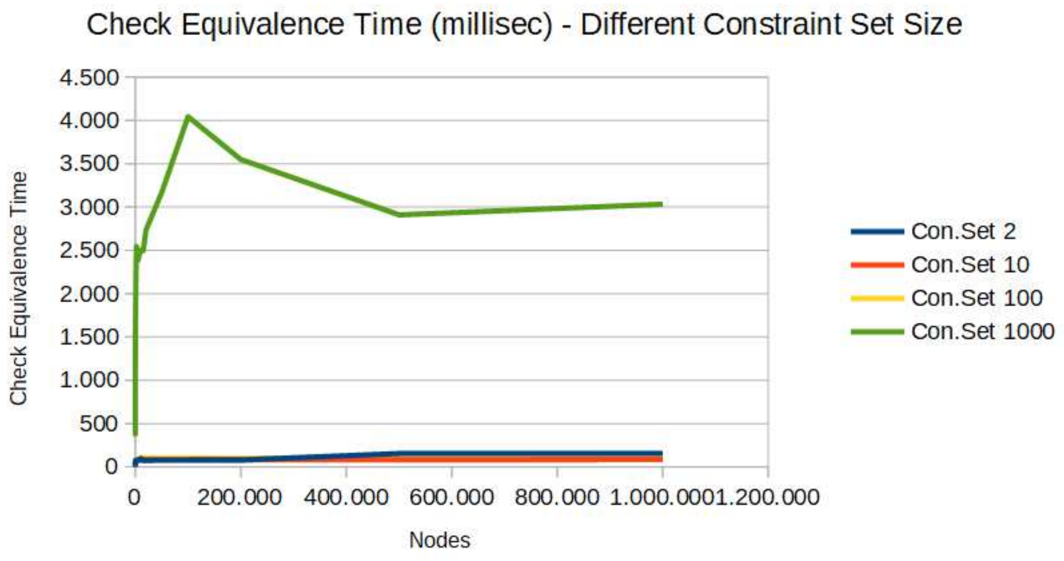


Figure 10.22: The analysis of Check Equivalence of two Nodes Time when Constraint Set Dimension increase.

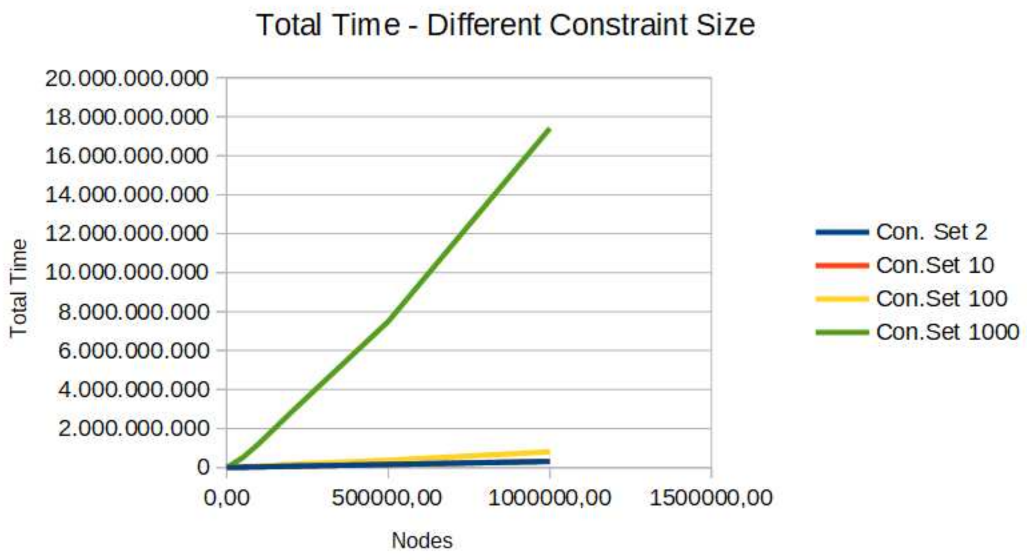


Figure 10.23: The analysis of the Total Time of program when Constraint Set Dimension increase.

10.3.2 Total Time Composition Analysis

Once the various timings and the total time have been analyzed, it is important to understand which of these algorithms takes up more or less time. For this reason, its percentage of occupation has been calculated for each function. The Tables are shown in Figures 10.24, 10.25, 10.26, 10.27.

Now you can average each dimension of the Constraint Set and see how the time composition changes as the Constraint Set increases. As can be seen in Figures the satisfiability function remains much more constant as input increases than the equivalence checking algorithm and saturation, and as input increases it occupies an increasingly smaller percentage of the total time . It is a different matter for saturation that as the Constraint Set and the inputs increase, both its time and the percentage of total time occupied increase significantly. Also in the equivalence control algorithm there is an increase as the size of the Constraint Set increases, but it is still much more efficient than saturation.

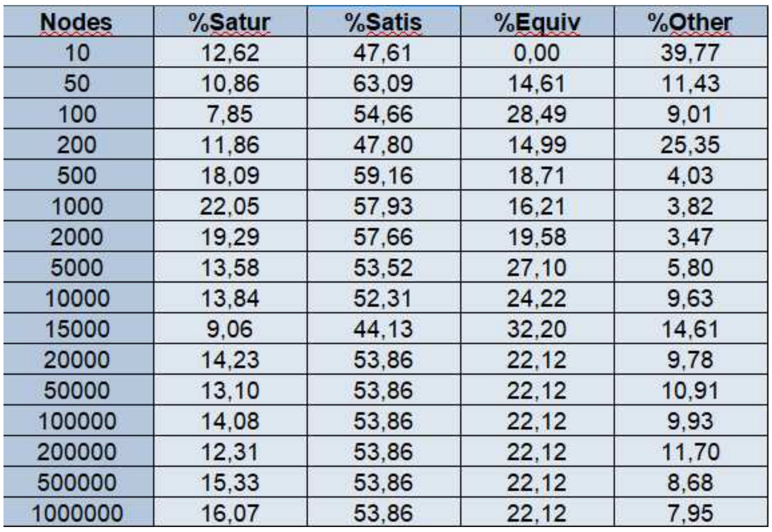
10.4 Time saved from using two-node equivalence checking

It is now possible to estimate the time saved by using the two-node equivalency check algorithm instead of using double saturation. In the double saturation algorithm, saturation occurs every time the algorithm for building the new Constraint Set is called up, in addition to the cases already foreseen by the saturation which is not removed. Considering the average saturation value for each program execution, multiply it by the number of calls to the new Constraint Set construction algorithm. To obtain the new total time, the total time of the algorithm with equivalence control is considered and the time taken for the equivalence function is subtracted and the previously calculated product is added.

In Figures 10.30, 10.31, 10.32, 10.33 tables are shown showing the estimate of the total time with double saturation and the time calculated for the equivalence algorithm and it is shown how much percentage of time is saved. Finally, Figure 10.31 shows an average of savings for each Constraint Set and shows how this average changes as the size of the Constraint Set increases.

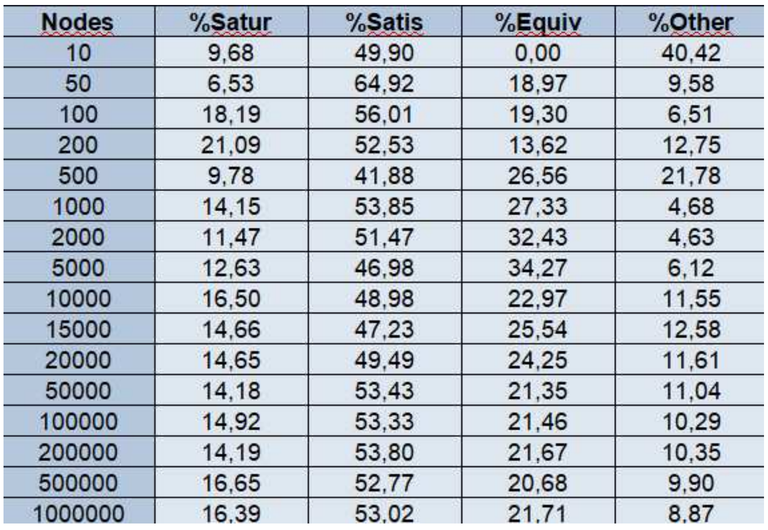
10.5 Experiments with different number of variables.

The complexity of the algorithms that use SMT Solver is not due only to the increase in the number of constraints within the Constraint Set, but also by the number of variables that are present within it. Up to now, Constraint Sets that only included one variable have been tested and in this section the



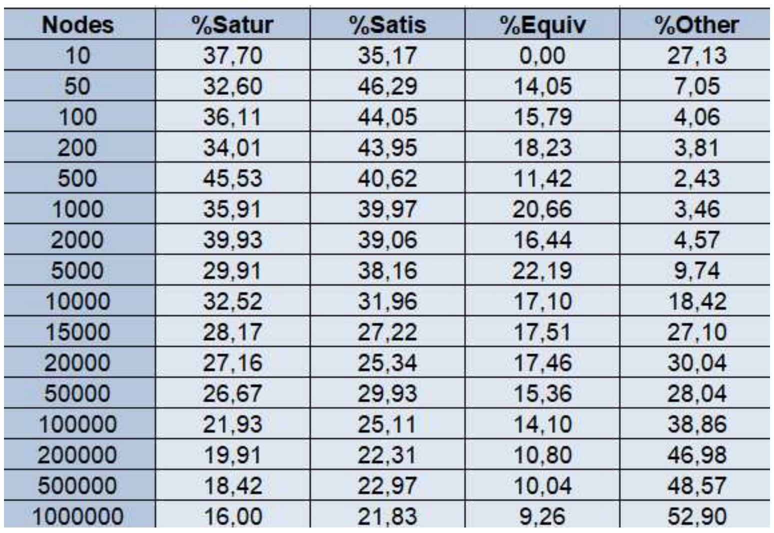
Nodes	%Satur	%Satis	%Equiv	%Other
10	12,62	47,61	0,00	39,77
50	10,86	63,09	14,61	11,43
100	7,85	54,66	28,49	9,01
200	11,86	47,80	14,99	25,35
500	18,09	59,16	18,71	4,03
1000	22,05	57,93	16,21	3,82
2000	19,29	57,66	19,58	3,47
5000	13,58	53,52	27,10	5,80
10000	13,84	52,31	24,22	9,63
15000	9,06	44,13	32,20	14,61
20000	14,23	53,86	22,12	9,78
50000	13,10	53,86	22,12	10,91
100000	14,08	53,86	22,12	9,93
200000	12,31	53,86	22,12	11,70
500000	15,33	53,86	22,12	8,68
1000000	16,07	53,86	22,12	7,95

Figure 10.24: Composition of Total Time with Constraint Set Dimension 2



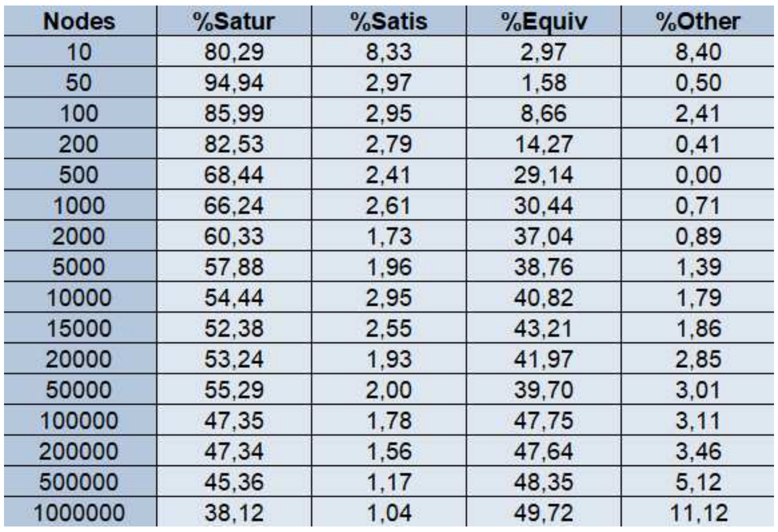
Nodes	%Satur	%Satis	%Equiv	%Other
10	9,68	49,90	0,00	40,42
50	6,53	64,92	18,97	9,58
100	18,19	56,01	19,30	6,51
200	21,09	52,53	13,62	12,75
500	9,78	41,88	26,56	21,78
1000	14,15	53,85	27,33	4,68
2000	11,47	51,47	32,43	4,63
5000	12,63	46,98	34,27	6,12
10000	16,50	48,98	22,97	11,55
15000	14,66	47,23	25,54	12,58
20000	14,65	49,49	24,25	11,61
50000	14,18	53,43	21,35	11,04
100000	14,92	53,33	21,46	10,29
200000	14,19	53,80	21,67	10,35
500000	16,65	52,77	20,68	9,90
1000000	16,39	53,02	21,71	8,87

Figure 10.25: Composition of Total Time with Constraint Set Dimension 10



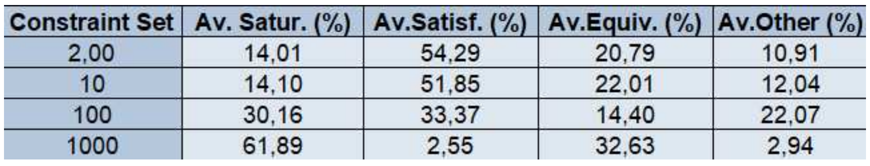
Nodes	%Satur	%Satis	%Equiv	%Other
10	37,70	35,17	0,00	27,13
50	32,60	46,29	14,05	7,05
100	36,11	44,05	15,79	4,06
200	34,01	43,95	18,23	3,81
500	45,53	40,62	11,42	2,43
1000	35,91	39,97	20,66	3,46
2000	39,93	39,06	16,44	4,57
5000	29,91	38,16	22,19	9,74
10000	32,52	31,96	17,10	18,42
15000	28,17	27,22	17,51	27,10
20000	27,16	25,34	17,46	30,04
50000	26,67	29,93	15,36	28,04
100000	21,93	25,11	14,10	38,86
200000	19,91	22,31	10,80	46,98
500000	18,42	22,97	10,04	48,57
1000000	16,00	21,83	9,26	52,90

Figure 10.26: Composition of Total Time with Constraint Set Dimension 100



Nodes	%Satur	%Satis	%Equiv	%Other
10	80,29	8,33	2,97	8,40
50	94,94	2,97	1,58	0,50
100	85,99	2,95	8,66	2,41
200	82,53	2,79	14,27	0,41
500	68,44	2,41	29,14	0,00
1000	66,24	2,61	30,44	0,71
2000	60,33	1,73	37,04	0,89
5000	57,88	1,96	38,76	1,39
10000	54,44	2,95	40,82	1,79
15000	52,38	2,55	43,21	1,86
20000	53,24	1,93	41,97	2,85
50000	55,29	2,00	39,70	3,01
100000	47,35	1,78	47,75	3,11
200000	47,34	1,56	47,64	3,46
500000	45,36	1,17	48,35	5,12
1000000	38,12	1,04	49,72	11,12

Figure 10.27: Composition of Total Time with Constraint Set Dimension 1000



Constraint Set	Av. Satur. (%)	Av.Satisf. (%)	Av.Equiv. (%)	Av.Other (%)
2,00	14,01	54,29	20,79	10,91
10	14,10	51,85	22,01	12,04
100	30,16	33,37	14,40	22,07
1000	61,89	2,55	32,63	2,94

Figure 10.28: Table of Average Percentage

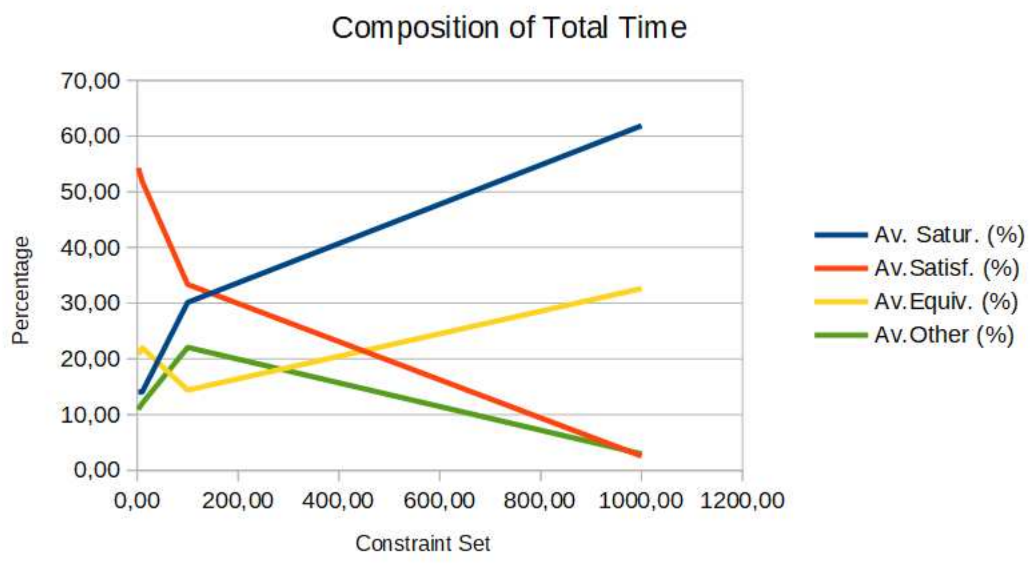


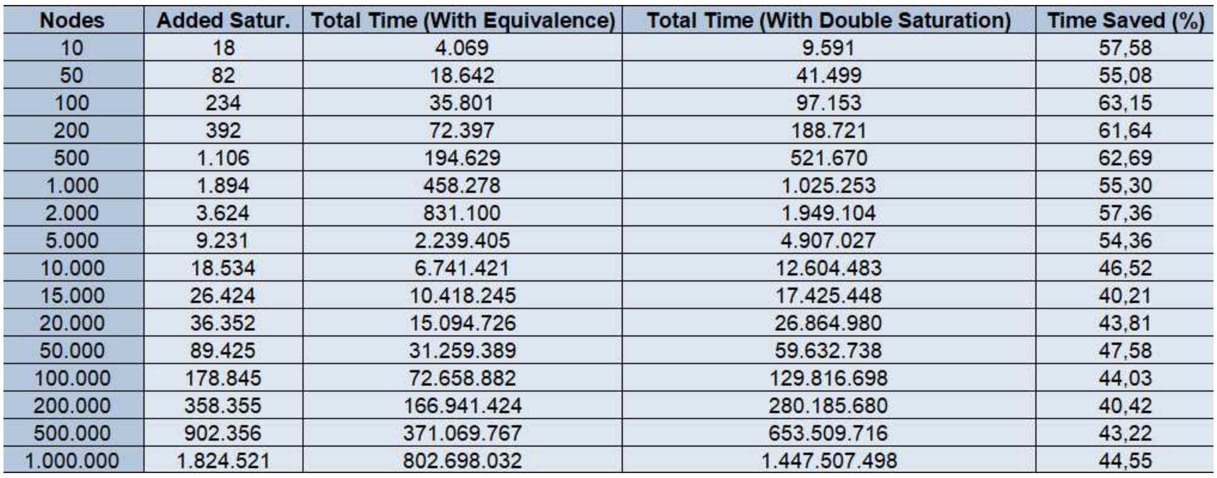
Figure 10.29: The variation of Composition of Total Time

Nodes	Added Satur.	Total Time (With Equivalence)	Total Time (With Double Saturation)	Time Saved (%)
10	18	3.510	4.839	27,46
50	82	14.092	18.011	21,76
100	234	25.979	35.618	27,06
200	392	60.170	81.557	26,22
500	1.106	122.924	187.463	34,43
1.000	1.894	317.030	430.289	26,32
2.000	3.624	577.044	751.941	23,26
5.000	9.231	1.446.170	1.819.160	20,50
10.000	18.534	3.877.278	4.953.037	21,72
15.000	26.424	5.810.050	6.200.996	6,30
20.000	36.352	6.142.500	7.890.130	22,15
50.000	89.425	15.356.250	18.842.203	18,50
100.000	178.845	30.712.500	38.699.854	20,64
200.000	358.355	61.425.000	73.667.828	16,62
500.000	902.356	153.562.500	202.019.032	23,99
1.000.000	1.824.521	307.125.000	414.112.905	25,84

Figure 10.30: The variation of Saved Time Percentage for Constraint Set Size 2

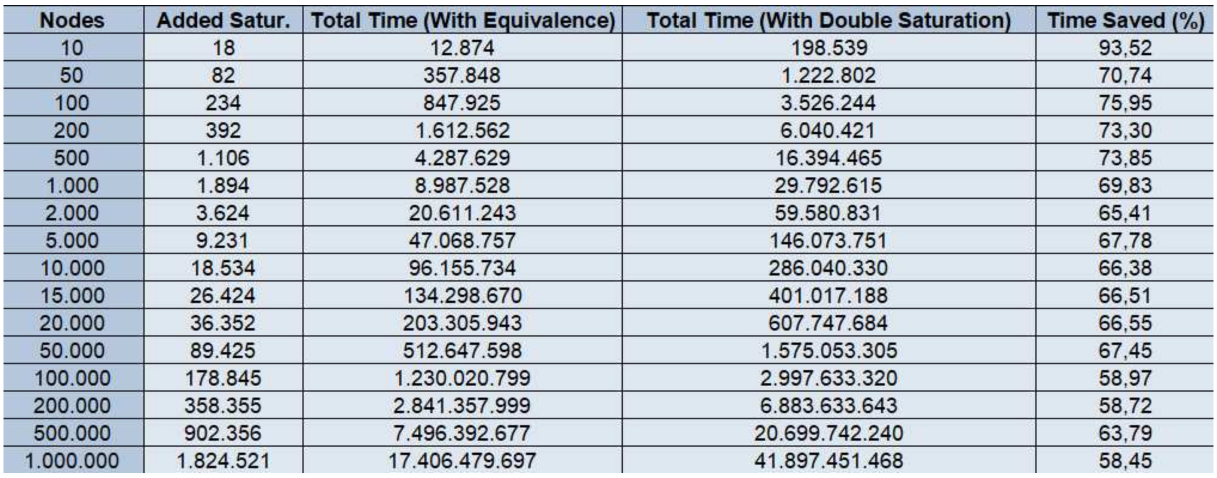
Nodes	Added Satur.	Total Time (With Equivalence)	Total Time (With Double Saturation)	Time Saved (%)
10	18	2.501	3.953	36,73
50	82	9.780	14.475	32,43
100	234	25.026	38.883	35,64
200	392	56.706	80.876	29,89
500	1.106	156.555	207.034	24,38
1.000	1.894	276.093	360.758	23,47
2.000	3.624	559.463	708.920	21,08
5.000	9.231	1.898.325	2.146.332	11,55
10.000	18.534	3.488.677	4.500.305	22,48
15.000	26.424	5.617.778	6.921.591	18,84
20.000	36.352	7.523.845	9.428.626	20,20
50.000	89.425	16.516.315	21.630.458	23,64
100.000	178.845	32.318.947	43.642.670	25,95
200.000	358.355	65.096.762	86.076.010	24,37
500.000	902.356	153.951.766	217.566.036	29,24
1.000.000	1.824.521	319.410.752	432.136.596	26,09

Figure 10.31: The variation of Saved Time Percentage for Constraint Set Size 10.



Nodes	Added Satur.	Total Time (With Equivalence)	Total Time (With Double Saturation)	Time Saved (%)
10	18	4.069	9.591	57,58
50	82	18.642	41.499	55,08
100	234	35.801	97.153	63,15
200	392	72.397	188.721	61,64
500	1.106	194.629	521.670	62,69
1.000	1.894	458.278	1.025.253	55,30
2.000	3.624	831.100	1.949.104	57,36
5.000	9.231	2.239.405	4.907.027	54,36
10.000	18.534	6.741.421	12.604.483	46,52
15.000	26.424	10.418.245	17.425.448	40,21
20.000	36.352	15.094.726	26.864.980	43,81
50.000	89.425	31.259.389	59.632.738	47,58
100.000	178.845	72.658.882	129.816.698	44,03
200.000	358.355	166.941.424	280.185.680	40,42
500.000	902.356	371.069.767	653.509.716	43,22
1.000.000	1.824.521	802.698.032	1.447.507.498	44,55

Figure 10.32: The variation of Saved Time Percentage for Constraint Set Size 100.



Nodes	Added Satur.	Total Time (With Equivalence)	Total Time (With Double Saturation)	Time Saved (%)
10	18	12.874	198.539	93,52
50	82	357.848	1.222.802	70,74
100	234	847.925	3.526.244	75,95
200	392	1.612.562	6.040.421	73,30
500	1.106	4.287.629	16.394.465	73,85
1.000	1.894	8.987.528	29.792.615	69,83
2.000	3.624	20.611.243	59.580.831	65,41
5.000	9.231	47.068.757	146.073.751	67,78
10.000	18.534	96.155.734	286.040.330	66,38
15.000	26.424	134.298.670	401.017.188	66,51
20.000	36.352	203.305.943	607.747.684	66,55
50.000	89.425	512.647.598	1.575.053.305	67,45
100.000	178.845	1.230.020.799	2.997.633.320	58,97
200.000	358.355	2.841.357.999	6.883.633.643	58,72
500.000	902.356	7.496.392.677	20.699.742.240	63,79
1.000.000	1.824.521	17.406.479.697	41.897.451.468	58,45

Figure 10.33: The variation of Saved Time Percentage for Constraint Set Size 1000.

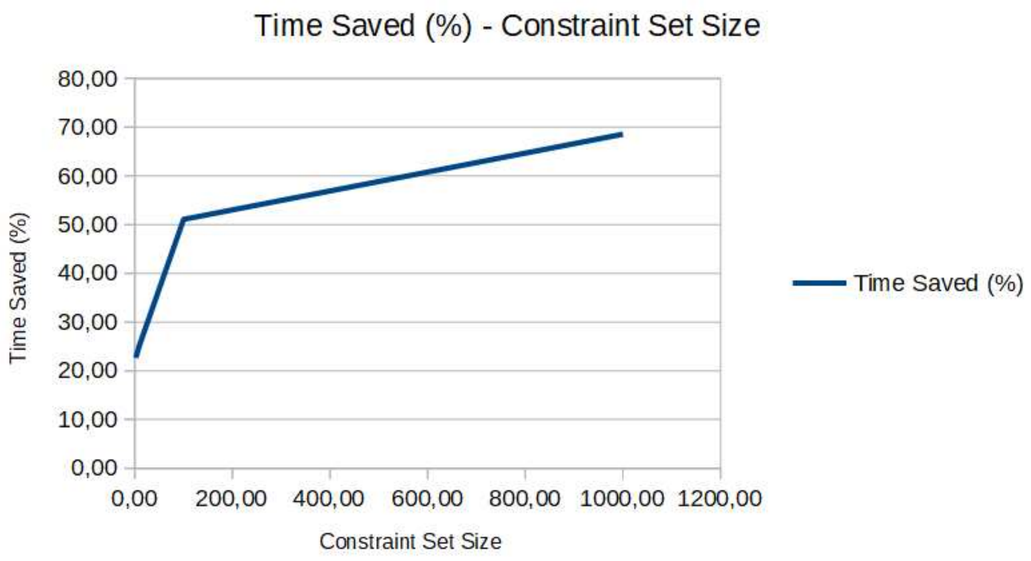
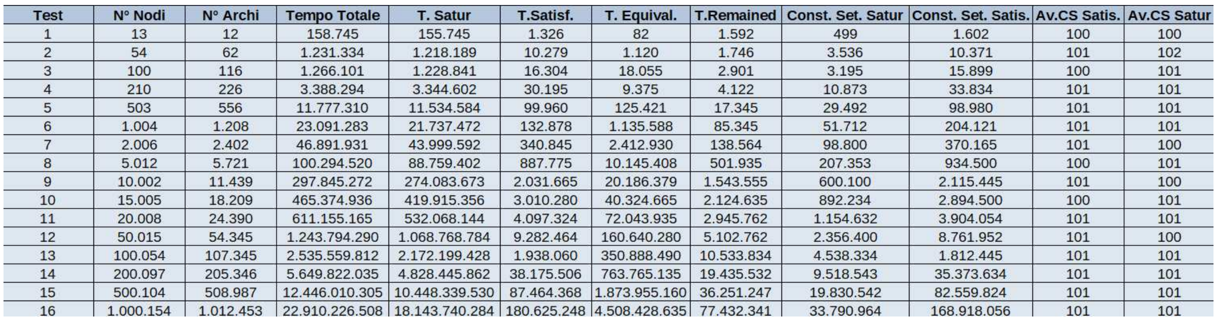


Figure 10.34: The variation of Average Saved Time Percentage for each Constraint Set.



Test	N° Nodi	N° Archi	Tempo Totale	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Satur	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	158.745	155.745	1.326	82	1.592	499	1.602	100	100
2	54	62	1.231.334	1.218.189	10.279	1.120	1.746	3.536	10.371	101	102
3	100	116	1.266.101	1.228.841	16.304	18.055	2.901	3.195	15.899	100	101
4	210	226	3.388.294	3.344.602	30.195	9.375	4.122	10.873	33.834	101	101
5	503	556	11.777.310	11.534.584	99.960	125.421	17.345	29.492	98.980	101	101
6	1.004	1.208	23.091.283	21.737.472	132.878	1.135.588	85.345	51.712	204.121	101	101
7	2.006	2.402	46.891.931	43.999.592	340.845	2.412.930	138.564	98.800	370.165	101	100
8	5.012	5.721	100.294.520	88.759.402	887.775	10.145.408	501.935	207.353	934.500	100	101
9	10.002	11.439	297.845.272	274.083.673	2.031.665	20.186.379	1.543.555	600.100	2.115.445	101	100
10	15.005	18.209	465.374.936	419.915.356	3.010.280	40.324.665	2.124.635	892.234	2.894.500	100	101
11	20.008	24.390	611.155.165	532.068.144	4.097.324	72.043.935	2.945.762	1.154.632	3.904.054	101	101
12	50.015	54.345	1.243.794.290	1.068.768.784	9.282.464	160.640.280	5.102.762	2.356.400	8.761.952	101	100
13	100.054	107.345	2.535.559.812	2.172.199.428	1.938.060	350.888.490	10.533.834	4.538.334	1.812.445	101	101
14	200.097	205.346	5.649.822.035	4.828.445.862	38.175.506	763.765.135	19.435.532	9.518.543	35.373.634	101	101
15	500.104	508.987	12.446.010.305	10.448.339.530	87.464.368	1.873.955.160	36.251.247	19.830.542	82.559.824	101	101
16	1.000.154	1.012.453	22.910.226.508	18.143.740.284	180.625.248	4.508.428.635	77.432.341	33.790.964	168.918.056	101	101

Figure 10.35: Measurement Table with Constraint Set Size 100 and number of variables 100 (part 1)

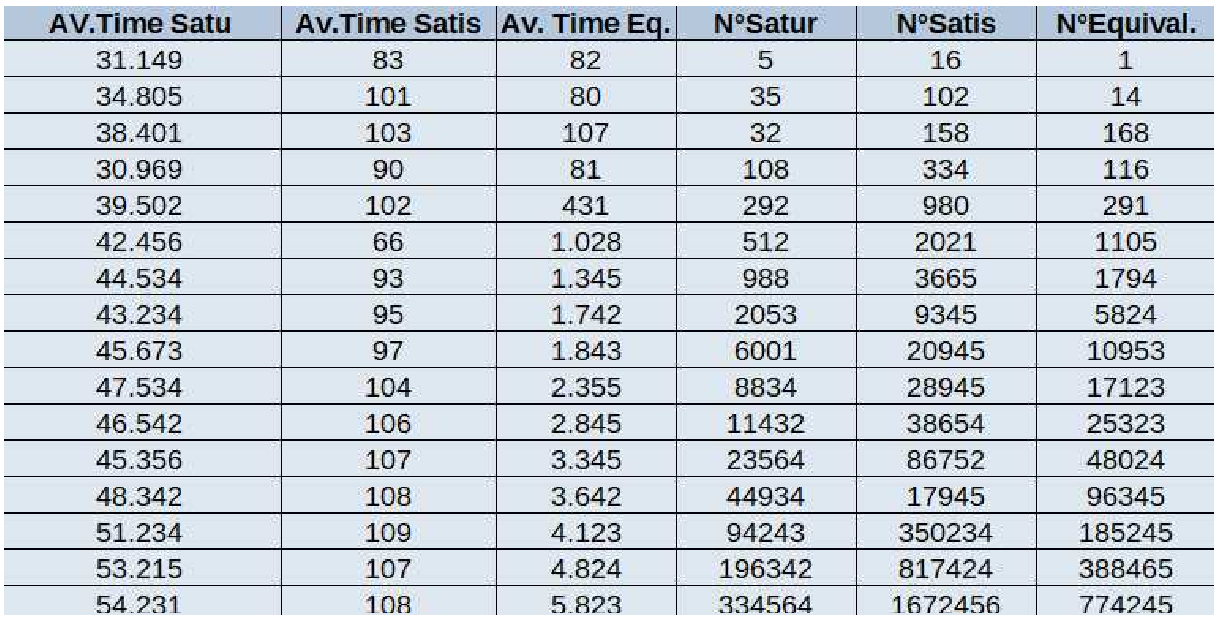
tests will be enriched with many variables. In particular, to guarantee compatibility with the previous examples, Constraint Sets have been generated with constraints on 100 and 1000 variables and their behavior is compared.

The possibility of adding variables and also having variable - variable constraints that always use the != operator is added in the DPN Automatic Generator in order to complicate the operations that the solver has to do.

In the case of a Constraint Set with 100 variables there are at least 100 constraint initialization, one for each variable, consequently the tested Constraint Sets cannot have less than 100 constraints. For this reason the tests were performed on Constraint Sets of average size around 100 and 1000 constraints. With 100 constraints we obtain the data present in the table, with the percentage composition shown in the table. While with 1000 constraints the measures are present in the table with the percentage composition shown in the table. Observing the data, it is possible to notice how the execution time increases considerably and the saturation algorithm comes to occupy over 90% of the total time of the algorithm, thus confirming that it is the bottleneck of the program.

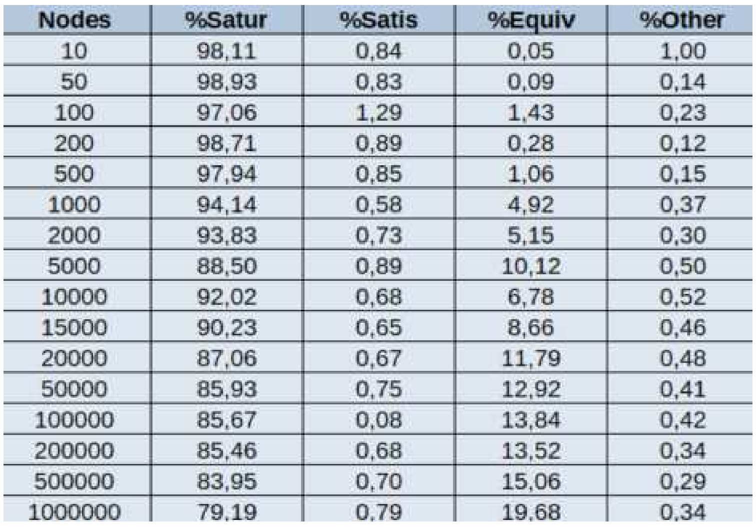
The program has also been tested with 500 variables and a Constraint Set of size 1000. The measurement table is shown in Figure 10.41 and 10.42, while the percentage composition table is shown in Figure 10.43.

Finally, an average Constraint Size of 1000 is applied with a number of variables in the constraints of 1000. In Figures 10.37 there is the table of the measurements performed, while in Figure 10.44 and 10.45 there is the percentage composition of the various functions. Subsequently, the variation of the



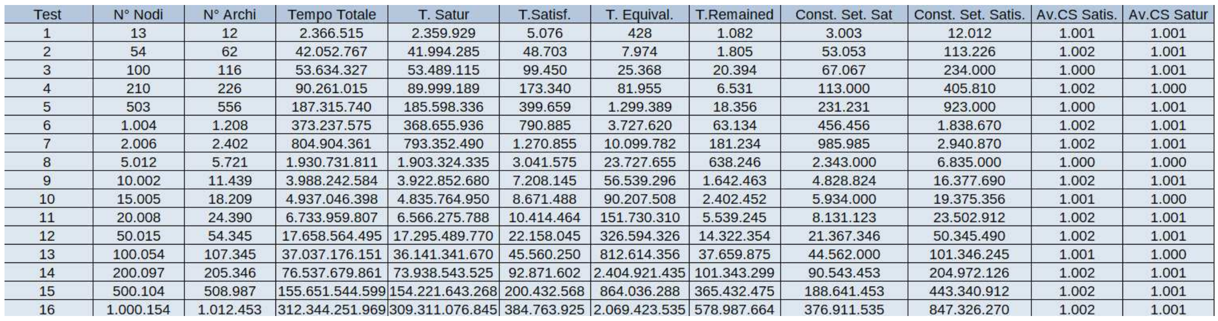
AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
31.149	83	82	5	16	1
34.805	101	80	35	102	14
38.401	103	107	32	158	168
30.969	90	81	108	334	116
39.502	102	431	292	980	291
42.456	66	1.028	512	2021	1105
44.534	93	1.345	988	3665	1794
43.234	95	1.742	2053	9345	5824
45.673	97	1.843	6001	20945	10953
47.534	104	2.355	8834	28945	17123
46.542	106	2.845	11432	38654	25323
45.356	107	3.345	23564	86752	48024
48.342	108	3.642	44934	17945	96345
51.234	109	4.123	94243	350234	185245
53.215	107	4.824	196342	817424	388465
54.231	108	5.823	334564	1672456	774245

Figure 10.36: Measurement Table with Constraint Set Size 100 and number of variables 100 (part 2)



Nodes	%Satur	%Satis	%Equiv	%Other
10	98,11	0,84	0,05	1,00
50	98,93	0,83	0,09	0,14
100	97,06	1,29	1,43	0,23
200	98,71	0,89	0,28	0,12
500	97,94	0,85	1,06	0,15
1000	94,14	0,58	4,92	0,37
2000	93,83	0,73	5,15	0,30
5000	88,50	0,89	10,12	0,50
10000	92,02	0,68	6,78	0,52
15000	90,23	0,65	8,66	0,46
20000	87,06	0,67	11,79	0,48
50000	85,93	0,75	12,92	0,41
100000	85,67	0,08	13,84	0,42
200000	85,46	0,68	13,52	0,34
500000	83,95	0,70	15,06	0,29
1000000	79,19	0,79	19,68	0,34

Figure 10.37: Percentage Composition of Constraint Size 100 and number of variables 100



Test	N° Nodi	N° Archi	Tempo Totale	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Sat	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
1	13	12	2.366.515	2.359.929	5.076	428	1.082	3.003	12.012	1.001	1.001
2	54	62	42.052.767	41.994.285	48.703	7.974	1.805	53.053	113.226	1.002	1.001
3	100	116	53.634.327	53.489.115	99.450	25.368	20.394	67.067	234.000	1.000	1.001
4	210	226	90.261.015	89.999.189	173.340	81.955	6.531	113.000	405.810	1.002	1.000
5	503	556	187.315.740	185.598.336	399.659	1.299.389	18.356	231.231	923.000	1.000	1.001
6	1.004	1.208	373.237.575	368.655.936	790.885	3.727.620	63.134	456.456	1.838.670	1.002	1.001
7	2.006	2.402	804.904.361	793.352.490	1.270.855	10.099.782	181.234	985.985	2.940.870	1.002	1.001
8	5.012	5.721	1.930.731.811	1.903.324.335	3.041.575	23.727.655	638.246	2.343.000	6.835.000	1.000	1.000
9	10.002	11.439	3.988.242.584	3.922.852.680	7.208.145	56.539.296	1.642.463	4.828.824	16.377.690	1.002	1.001
10	15.005	18.209	4.937.046.398	4.835.764.950	8.671.488	90.207.508	2.402.452	5.934.000	19.375.356	1.001	1.000
11	20.008	24.390	6.733.959.807	6.566.275.788	10.414.464	151.730.310	5.539.245	8.131.123	23.502.912	1.002	1.001
12	50.015	54.345	17.658.564.495	17.295.489.770	22.158.045	326.594.326	14.322.354	21.367.346	50.345.490	1.002	1.001
13	100.054	107.345	37.037.176.151	36.141.341.670	45.560.250	812.614.356	37.659.875	44.562.000	101.346.245	1.001	1.000
14	200.097	205.346	76.537.679.861	73.938.543.525	92.871.602	2.404.921.435	101.343.299	90.543.453	204.972.126	1.002	1.001
15	500.104	508.987	155.651.544.599	154.221.643.268	200.432.568	864.036.288	365.432.475	188.641.453	443.340.912	1.002	1.001
16	1.000.154	1.012.453	312.344.251.969	309.311.076.845	384.763.925	2.069.423.535	578.987.664	376.911.535	847.326.270	1.002	1.001

Figure 10.38: Measurement Table with Constraint Set Size 1000 and number of variables 100 (part 1)

AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
786.643	423	428	3	12	1
792.345	431	443	53	113	18
798.345	425	453	67	234	56
796.453	428	443	113	405	185
803.456	433	1.703	231	923	763
808.456	431	2.430	456	1835	1534
805.434	433	3.123	985	2935	3234
812.345	445	2.813	2343	6835	8435
813.195	441	3.564	4824	16345	15864
814.925	448	3.823	5934	19356	23596
808.356	444	4.534	8123	23456	33465
810.245	441	5.102	21346	50245	64013
811.035	450	5.634	44562	101245	144234
817.425	454	6.323	90453	204563	380345
818.356	453	6.834	188453	442456	126432
821.467	455	7.103	376535	845635	291345

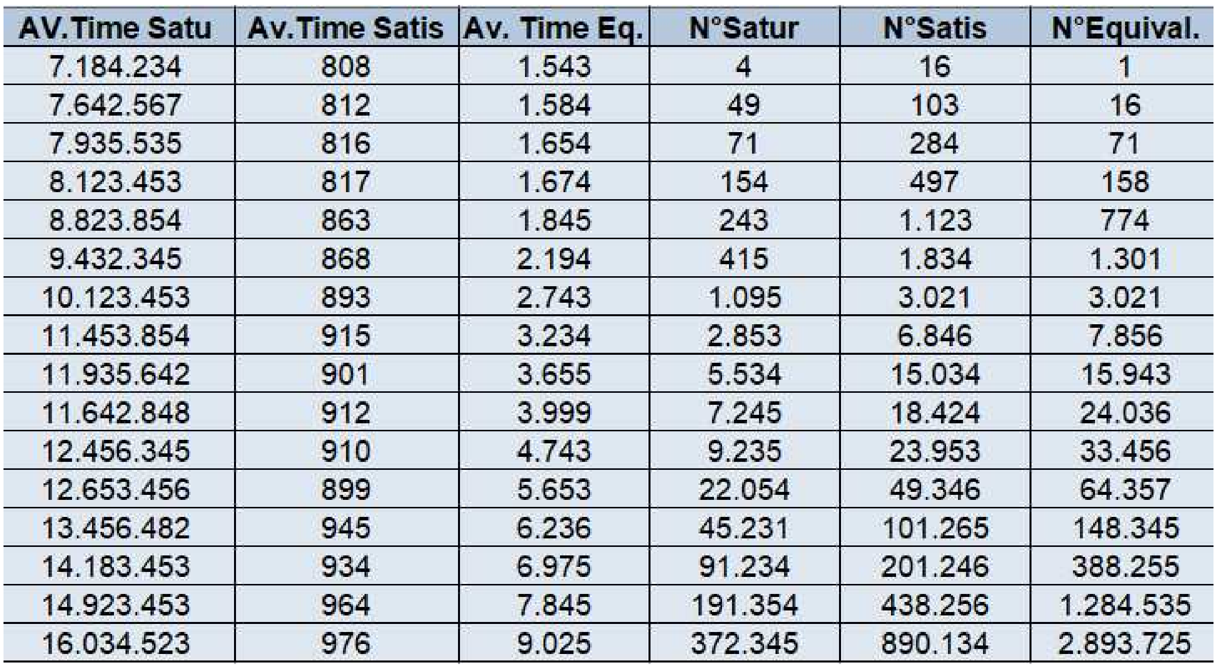
Figure 10.39: Measurement Table with Constraint Set Size 1000 and number of variables 100 (part 2)

Nodes	%Satur	%Satis	%Equiv	%Other
10	99,72	0,21	0,02	0,05
50	99,86	0,12	0,02	0,00
100	99,73	0,19	0,05	0,04
200	99,71	0,19	0,09	0,01
500	99,08	0,21	0,69	0,01
1000	98,77	0,21	1,00	0,02
2000	98,56	0,16	1,25	0,02
5000	98,58	0,16	1,23	0,03
10000	98,36	0,18	1,42	0,04
15000	97,95	0,18	1,83	0,05
20000	97,51	0,15	2,25	0,08
50000	97,94	0,13	1,85	0,08
100000	97,58	0,12	2,19	0,10
200000	96,60	0,12	3,14	0,13
500000	99,08	0,13	0,56	0,23
1000000	99,03	0,12	0,66	0,19

Figure 10.40: Percentage Composition of Constraint Size 1000 and number of variables 100

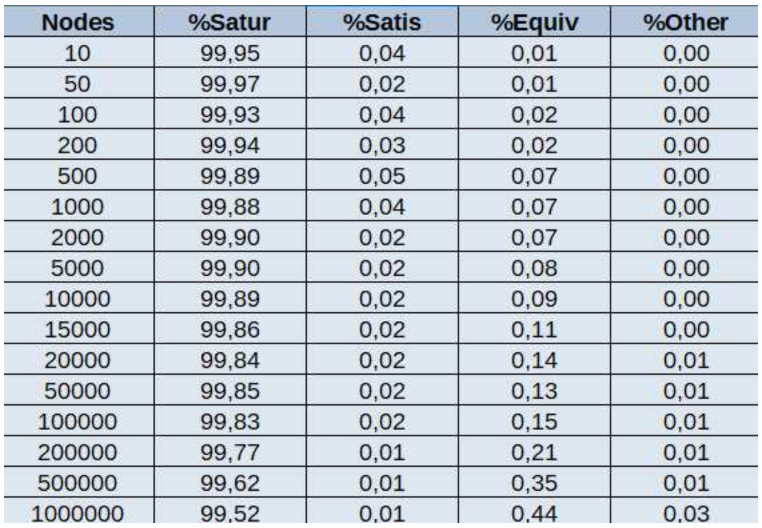
N° Nodes	N° Arcs	Total Time (ms)	T. Satur	T.Satisf.	T. Equival.	T.Remained	Const. Set. Satur	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
13	12	28.752.530	28.736.936	12.928	1.543	1.123	2.004	8.032	501	502
54	62	374.596.506	374.485.783	83.636	25.344	1.743	24.598	51.603	502	501
100	116	563.794.625	563.422.985	231.744	117.434	22.462	35.571	142.568	501	502
210	226	1.251.720.755	1.251.011.762	406.049	264.492	38.452	77.000	249.494	500	502
503	556	2.146.640.046	2.144.196.522	969.149	1.428.030	46.345	121.743	562.623	501	501
1.004	1.208	3.918.939.726	3.914.423.175	1.591.912	2.854.394	70.245	207.915	920.668	501	502
2.006	2.402	11.096.353.625	11.085.181.035	2.697.753	8.286.603	188.234	549.690	1.516.542	502	502
5.012	5.721	32.710.199.590	32.677.845.462	6.264.090	25.406.304	683.734	1.426.500	3.436.692	500	502
10.002	11.439	66.125.539.474	66.051.842.828	13.545.634	58.271.665	1.879.347	2.772.534	7.532.034	501	501
15.005	18.209	84.468.070.799	84.352.433.760	16.802.688	96.119.964	2.714.387	3.629.745	9.248.848	501	502
20.008	24.390	115.220.768.523	115.034.346.075	21.797.230	158.681.808	5.943.410	4.626.735	12.024.406	501	502
50.015	54.345	279.485.936.116	279.059.318.624	44.362.054	363.810.121	18.445.317	11.027.000	24.771.692	500	502
100.054	107.345	609.707.344.359	608.650.137.342	95.695.425	925.079.420	36.432.172	22.660.731	50.835.030	501	502
200.097	205.346	1.297.017.535.180	1.294.013.151.002	187.963.764	2.708.078.625	108.341.789	45.799.468	100.824.246	502	501
500.104	508.987	2.866.551.113.097	2.855.662.425.362	422.478.784	10.077.177.075	389.031.876	95.868.354	220.004.512	501	502
1.000.154	1.012.453	5.999.412.373.233	5.970.374.466.435	868.770.784	26.115.868.125	2.053.267.889	186.544.845	446.847.268	501	502

Figure 10.41: Measurement Table with Constraint Set Size 1000 and number of variables 500 (part 1)



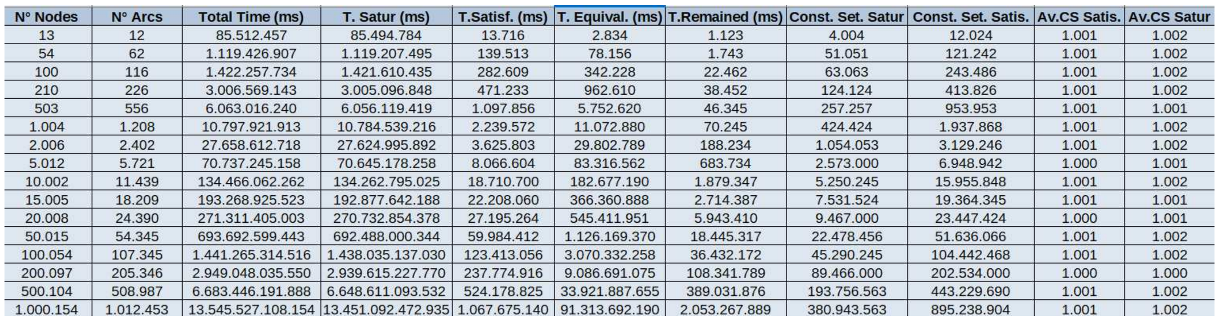
AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
7.184.234	808	1.543	4	16	1
7.642.567	812	1.584	49	103	16
7.935.535	816	1.654	71	284	71
8.123.453	817	1.674	154	497	158
8.823.854	863	1.845	243	1.123	774
9.432.345	868	2.194	415	1.834	1.301
10.123.453	893	2.743	1.095	3.021	3.021
11.453.854	915	3.234	2.853	6.846	7.856
11.935.642	901	3.655	5.534	15.034	15.943
11.642.848	912	3.999	7.245	18.424	24.036
12.456.345	910	4.743	9.235	23.953	33.456
12.653.456	899	5.653	22.054	49.346	64.357
13.456.482	945	6.236	45.231	101.265	148.345
14.183.453	934	6.975	91.234	201.246	388.255
14.923.453	964	7.845	191.354	438.256	1.284.535
16.034.523	976	9.025	372.345	890.134	2.893.725

Figure 10.42: Measurement Table with Constraint Set Size 1000 and number of variables 500 (part 2)



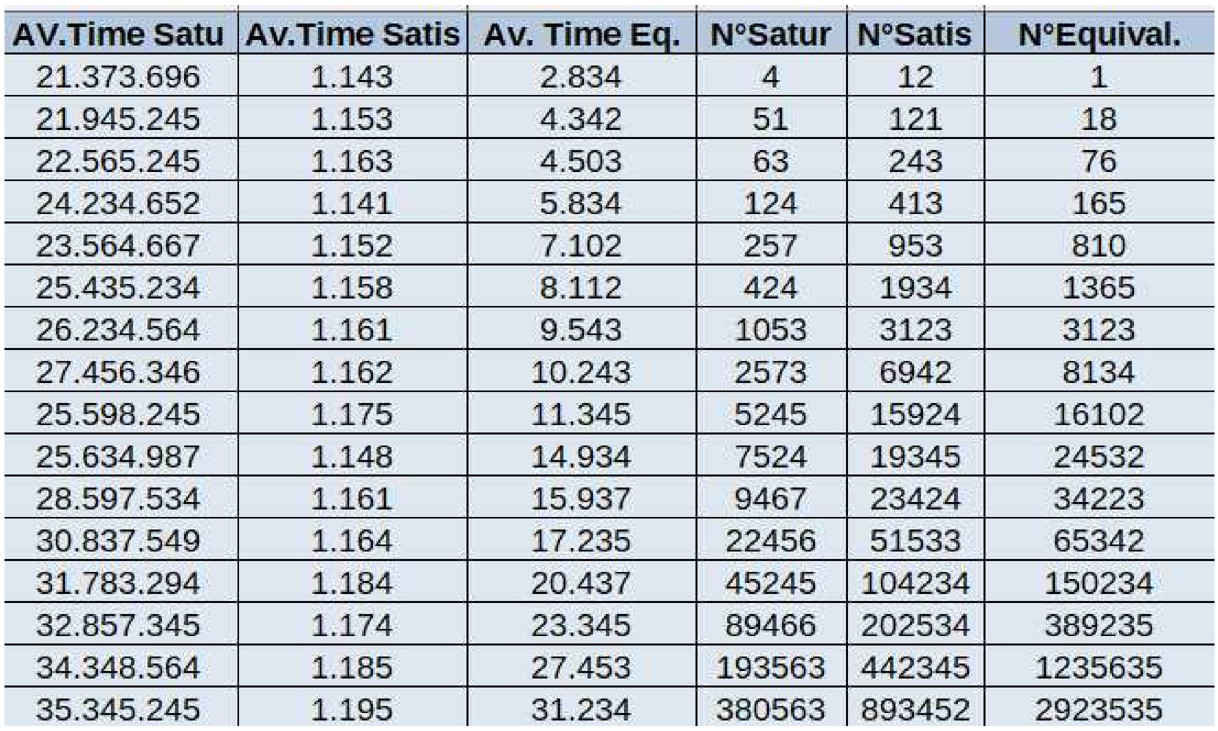
Nodes	%Satur	%Satis	%Equiv	%Other
10	99,95	0,04	0,01	0,00
50	99,97	0,02	0,01	0,00
100	99,93	0,04	0,02	0,00
200	99,94	0,03	0,02	0,00
500	99,89	0,05	0,07	0,00
1000	99,88	0,04	0,07	0,00
2000	99,90	0,02	0,07	0,00
5000	99,90	0,02	0,08	0,00
10000	99,89	0,02	0,09	0,00
15000	99,86	0,02	0,11	0,00
20000	99,84	0,02	0,14	0,01
50000	99,85	0,02	0,13	0,01
100000	99,83	0,02	0,15	0,01
200000	99,77	0,01	0,21	0,01
500000	99,62	0,01	0,35	0,01
1000000	99,52	0,01	0,44	0,03

Figure 10.43: Percentage Composition of Constraint Size 1000 and number of variables 500



N° Nodes	N° Arcs	Total Time (ms)	T. Satur (ms)	T.Satisf. (ms)	T. Equival. (ms)	T.Remained (ms)	Const. Set. Satur	Const. Set. Satis.	Av.CS Satis.	Av.CS Satur
13	12	85.512.457	85.494.784	13.716	2.834	1.123	4.004	12.024	1.001	1.002
54	62	1.119.426.907	1.119.207.495	139.513	78.156	1.743	51.051	121.242	1.001	1.002
100	116	1.422.257.734	1.421.610.435	282.609	342.228	22.462	63.063	243.486	1.001	1.002
210	226	3.006.569.143	3.005.096.848	471.233	962.610	38.452	124.124	413.826	1.001	1.002
503	556	6.063.016.240	6.056.119.419	1.097.856	5.752.620	46.345	257.257	953.953	1.001	1.001
1.004	1.208	10.797.921.913	10.784.539.216	2.239.572	11.072.880	70.245	424.424	1.937.868	1.001	1.002
2.006	2.402	27.658.612.718	27.624.995.892	3.625.803	29.802.789	188.234	1.054.053	3.129.246	1.001	1.002
5.012	5.721	70.737.245.158	70.645.178.258	8.066.604	83.316.562	683.734	2.573.000	6.948.942	1.000	1.001
10.002	11.439	134.466.062.262	134.262.795.025	18.710.700	182.677.190	1.879.347	5.250.245	15.955.848	1.001	1.002
15.005	18.209	193.268.925.523	192.877.642.188	22.208.060	366.360.888	2.714.387	7.531.524	19.364.345	1.001	1.001
20.008	24.390	271.311.405.003	270.732.854.378	27.195.264	545.411.951	5.943.410	9.467.000	23.447.424	1.000	1.001
50.015	54.345	693.692.599.443	692.488.000.344	59.984.412	1.126.169.370	18.445.317	22.478.456	51.636.066	1.001	1.002
100.054	107.345	1.441.265.314.516	1.438.035.137.030	123.413.056	3.070.332.258	36.432.172	45.290.245	104.442.468	1.001	1.002
200.097	205.346	2.949.048.035.550	2.939.615.227.770	237.774.916	9.086.691.075	108.341.789	89.466.000	202.534.000	1.000	1.000
500.104	508.987	6.683.446.191.888	6.648.611.093.532	524.178.825	33.921.887.655	389.031.876	193.756.563	443.229.690	1.001	1.002
1.000.154	1.012.453	13.545.527.108.154	13.451.092.472.935	1.067.675.140	91.313.692.190	2.053.267.889	380.943.563	895.238.904	1.001	1.002

Figure 10.44: Measurement Table with Constraint Set Size 1000 and number of variables 1000 (part 1)



AV.Time Satu	Av.Time Satis	Av. Time Eq.	N°Satur	N°Satis	N°Equival.
21.373.696	1.143	2.834	4	12	1
21.945.245	1.153	4.342	51	121	18
22.565.245	1.163	4.503	63	243	76
24.234.652	1.141	5.834	124	413	165
23.564.667	1.152	7.102	257	953	810
25.435.234	1.158	8.112	424	1934	1365
26.234.564	1.161	9.543	1053	3123	3123
27.456.346	1.162	10.243	2573	6942	8134
25.598.245	1.175	11.345	5245	15924	16102
25.634.987	1.148	14.934	7524	19345	24532
28.597.534	1.161	15.937	9467	23424	34223
30.837.549	1.164	17.235	22456	51533	65342
31.783.294	1.184	20.437	45245	104234	150234
32.857.345	1.174	23.345	89466	202534	389235
34.348.564	1.185	27.453	193563	442345	1235635
35.345.245	1.195	31.234	380563	893452	2923535

Figure 10.45: Measurement Table with Constraint Set Size 1000 and number of variables 1000 (part 2)

various average times in absolute values of the three functions that interact with the SMT Solver (Saturation, Satisfiable and Check Equivalence) is analyzed using as a reference the test with the highest magnitude number of the Constraint Set (1000) as the number varies of variables (Figures 10.47 , 10.48, 10.49). Finally, the percentage change is shown as the number of variables increases, assuming we have a Constraint Set Size with 1000 constraints (Figure 10.50). As can be seen from the table and the graph, the average of the saturation percentages already becomes very close to 100% already with 100 variables and with 1000 variables it gets even closer. Opposite trend instead for the other functions that go towards 0%.

Nodes	%Satur	%Satis	%Equiv	%Other
10	99,98	0,02	0,00	0,00
50	99,98	0,01	0,01	0,00
100	99,95	0,02	0,02	0,00
200	99,95	0,02	0,03	0,00
500	99,89	0,02	0,09	0,00
1000	99,88	0,02	0,10	0,00
2000	99,88	0,01	0,11	0,00
5000	99,87	0,01	0,12	0,00
10000	99,85	0,01	0,14	0,00
15000	99,80	0,01	0,19	0,00
20000	99,79	0,01	0,20	0,00
50000	99,83	0,01	0,16	0,00
100000	99,78	0,01	0,21	0,00
200000	99,68	0,01	0,31	0,00
500000	99,48	0,01	0,51	0,01
1000000	99,30	0,01	0,67	0,02

Figure 10.46: Percentage Composition of Constraint Size 1000 and number of variables 1000

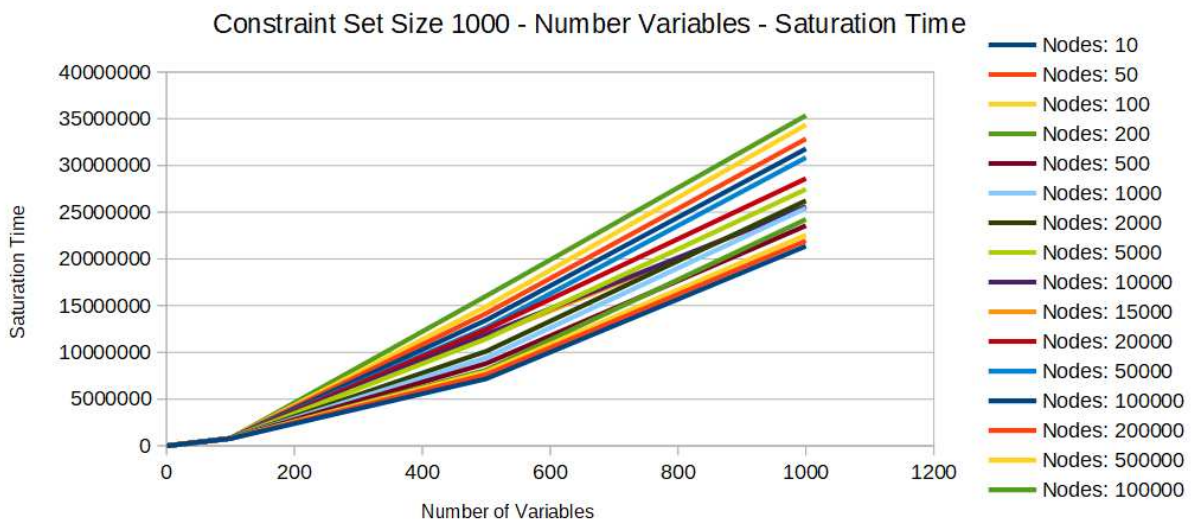


Figure 10.47: Variation of the Saturation time as the number of variables increases in an average Constraint Set per node with 1000 constraints.

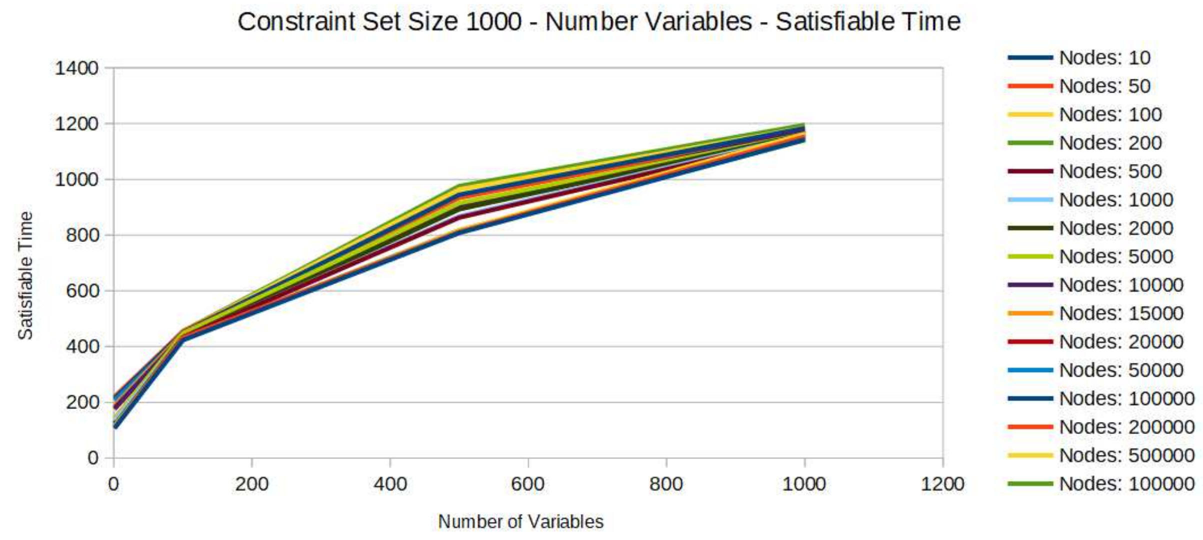


Figure 10.48: Variation of the Satisfiable time as the number of variables increases in an average Constraint Set per node with 1000 constraints.

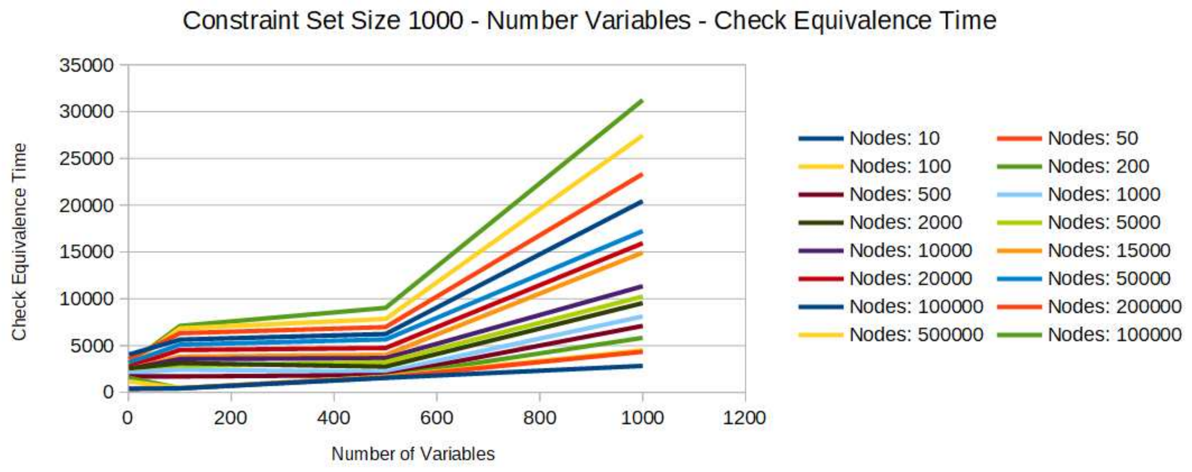


Figure 10.49: Variation of the Check Equivalence time as the number of variables increases in an average Constraint Set per node with 1000 constraints.

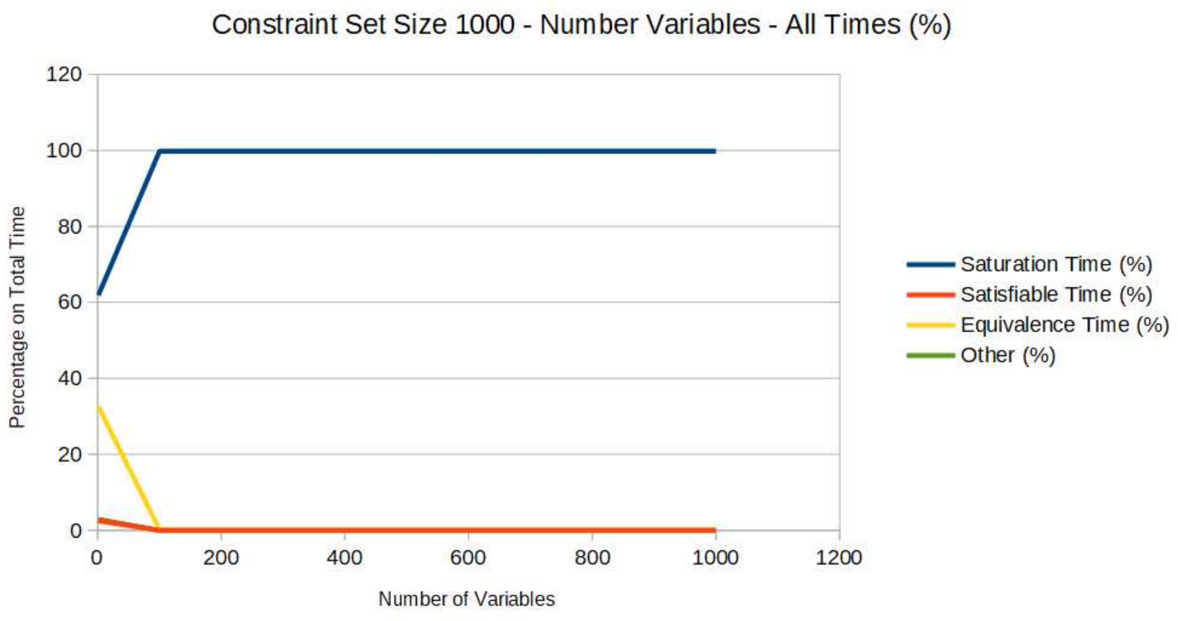


Figure 10.50: Variation of the percentage occupied for a certain time on the total time

Part IV

Conclusion

CHAPTER 11

Conclusion

11.1 Conclusion and Future Developments

In this thesis we wanted to perform a gradual approach by describing the implementation of each single step in order to then arrive at the final result. The project has a very practical purpose, in fact its main objective is to apply what has been prepared only at a theoretical level. With this work it is possible to verify if a certain Data Petri Net is always Sound in all the possible states it can be in or if it needs to be reformulated because it does not respect the three properties of data-aware soundness. In a simple DPN it is not possible to verify that these three properties are solidly verified, for this reason it is necessary to implement the Constraint Graph and then verify the three properties on it. To do this, a top-down methodology was used in which the global problem was taken and broken down into many sub-problems which gave rise to different algorithms integrated with each other and which together made it possible to achieve the final result. In the measurements performed, it was observed that the real bottleneck of this algorithm is saturation, so it will be important in the future to find more efficient algorithms that manage to obtain the same result and make the program more scalable. A possible future extension is to add debugging functions for which it is possible to pass back to the user what causes the three data-aware soundness properties to fail. Data Petri Nets are still a very specific topic unlike Petri Net Standards and the aim of this thesis is to add further material to this topic. During the last decades, different ways have been introduced to integrate business processes with different types of data according to the various specific needs of the business realities. The software can have important applications in Process Mining and in particular in all areas in which Petri nets are used. The software is implemented with the Java programming language and therefore requires a Java Virtual Machine to run and works for Linux or Windows based operating systems.

bibliography

[1] Marco Montali Paolo Felli, Massimiliano De Leoni. Soundness verification of data-aware process models with variable-to-variable conditions. fundamenta informaticae 182(1): 1-29 (2021). *Padua University*.

[2] Marco Montali Massimiliano de Leoni, Paolo Felli. A holistic approach for soundness verification of decision-aware process models. er 2018: 219-235. *Padua University*, 2018a.

[3] Marco Montali Massimiliano de Leoni, Paolo Felli. A holistic approach for soundness verification of decision-aware process models (extended version). corr abs/1804.02316 (2018). *Padua University*, 2018b.

[4] Joost Engelfriet Grzegorz Rozenberg. Elementary net systems. lectures on petri nets i: Basic models. acpn 1996. lecture notes in computer science, vol 1491. 1998.

[5] Nikolaj Bjørner Leonardo de Moura. Z3: an efficient smt solver. tools and algorithms for the construction and analysis of systems. 4963: 337–340. 2018.

[6] Charles E.; Rivest Ronald L.; Stein Clifford. Cormen, Thomas H.; Leiserson. Introduction to algorithms (3rd ed.). mit press and mcgraw-hill. 2009.

[7] Wolfgang Reisig Joerg Desel. Place/transition petri nets. lectures on petri nets i: Basic models. acpn 1996. lecture notes in computer science, vol 1491. 1998.

Websites consulted

- Fondamenti di Intelligenza Artificiale UniBo – `http://lia.deis.unibo.it/Courses/AI/applicationsAI2009-2010/Lucidi/07-CLP.pdf`
- Wikipedia – `www.wikipedia.org/`
- Microsoft – `www.microsoft.com`
- Choco Solver Tutorial - `https://choco-solver.org/tutos/`
- Choco Team – `https://github.com/chocoteam/choco-solver`
- ProM Tools – `promtools.org`
- mlWiki – `http://mlwiki.org/index.php/Workflow_Soundness`