FernUniversität in Hagen
Faculty of Mathematics and Computer Science

Artificial Intelligence Group

On SAT encodings for Quasi-Inconsistency

Bachelor's Thesis

in partial fulfillment of the requirements for
the degree of Bachelor of Science (B.Sc.)
in Informatik

submitted by
Michael Annen

First examiner:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Advisor:	Prof. Dr. Matthias Thimm Artificial Intelligence Group

Statement

	Yes	No
I agree to have this thesis published in the library.	☐	☐
I agree to have this thesis published on the webpage of the artificial intelligence group.	☐	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐
The source code is available under a GNU General Public License (GPLv3).	☐	☐
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☐	☐

. . . .

(Place, Date) (Signature)

Zusammenfassung

In dieser Bachelorarbeit werden die theoretischen Ergebnisse aus [CT20] genutzt, um einen Algorithmus zu entwerfen und zu implementieren, der Quasi-Inkonsistenz in Regelbasen für Geschäftsregeln erkennen kann. Da das Problem der Erkennung von Quasi-Inkonsistenz NP-vollständig ist, ist eine direkte effiziente Lösung nicht garantiert. Wir nutzen daher die großen Fortschritte, die in letzter Zeit im Bereich des "SAT-Solving" gemacht wurden. Dazu wird das Problem in ein Erfüllbarkeitsproblem (SAT) umgewandelt, welches dann von einem SAT-Solver gelöst werden kann. Anschließend wird ein Leistungsvergleich zwischen dem von uns entwickelten SAT basierten Algorithmus und einer traditionelleren "Brute Force" Lösung durchgeführt. In diesem Vergleich wird festgestellt, dass der SAT basierte Algorithmus die Probleme für die verwendeten Daten deutlich schneller lösen konnte.

Abstract

In this bachelor's thesis the theoretical results from [CT20] are used to develop an algorithm that can detect Quasi-Inconsistencies in rule bases for business rules. As the problem is NP-complete and we are not guaranteed to find an efficient solution for this directly. Therefore we are taking advantage of recent massive performance increases in SAT solving. This is done by encoding the underlying problem to a SAT instance, which is then run through a SAT solver. The performance of the SAT-based approach to Quasi-Inconsistency detection is evaluated against a more traditional brute force based implementation of inconsistency detection, where it shows much promise and significant performance benefits for the underlying data sets that were used.

Contents

1 Introduction

In this section, we will give an informal introduction into the topic without worrying too much about notation and strict formal correctness.

1.1 Motivation

A Business rule is "a directive that is intended to influence or guide business behavior [...] in response to risks, threats or opportunities" [Bus03]. The idea of business rules is therefore to derive (automated) decisions that are concurrent with a companies policies and goals. They are *declarative* statements, meaning that the outcome of the rule application is independent of the order in which the rules are applied [Gra06]. To get a better impression of how this could look, let us look at an example rule base:

$$\mathcal{B}_1 = \{invoicePayment \leftarrow regularCustomer; prePayment \leftarrow newCustomer\}$$

This rule base contains two rules that define how customers have to pay for their orders depending on their history, so this is pretty straightforward and there are no problems with this rule base.
In traditional knowledge representation knowledge bases are expressed similarly, but can also contain facts in addition to rules, for example:

$$\mathcal{K}_1 = \{a \leftarrow b; \neg a \leftarrow b; b\}$$

With this knowledge base we can see, that we have a fact b which leads to a as well as to ¬ a, which is obviously not correct and therefore call this knowledge base inconsistent (in the classical sense). This type of inconsistency can be measured and analyzed by existing traditional methods [CD19]. With business rules however we face a different problem. The actual facts are not known at the design time of the business rule base. Therefore we need a more extended definition of inconsistency. If we look at another example business rule base

$$\mathcal{B}_2 = \{\neg prePayment \leftarrow newCustomer; prePayment \leftarrow newCustomer\}$$

it is obvious that this does not make much sense, although it is consistent in a classical sense. Whenever a new customer places an order we would run into a conflict. We can clearly see that from a *Business Rule Management (BRM)* perspective this rule base does not help us out at all and is even harmful. We need a new concept to deal with these types of problems.

To address this, the idea of *Quasi-Inconsistency* has been introduced by [CD19]. The notion of this type of inconsistency deals with inconsistencies that cannot be detected by classical methods of inconsistency analysis at design time but will be activated at run time by the introduction of facts. This type of inconsistency is therefore especially relevant in the domain of business rules management since rule bases are always present at design time before the facts are introduced during the execution. Specifically, the

definition (which will be explained more formally later on) demands that rules that are always being activated together cannot be contradictory if a rule base is to be consistent in this way. Analyzing and detecting if a business rule base is Quasi-Inconsistent at design time before actually running into unresolvable conflicts in production could therefore turn out to be a valuable practical world application to ensure compliance and correct business decisions. In [CT20] a complexity analysis of the decision problem, whether a rule base is Quasi-Inconsistent was conducted and it was proven that it is NP-complete. While at first, this seems like a setback, it actually makes the problem more tractable since there are quite a few inconsistency problems that are even more difficult in computational complexity than "only" NP [CT20]. It also opens up the possibility for us to take advantage of modern SAT-solving techniques that have recently grown better and better at solving these types of "hard" problems.
As SAT is NP-complete, all other NP-complete problems can be reduced to SAT. So-called SAT solvers can nowadays effectively find solutions to this problem even with millions of variables and clauses. Since the inception of SAT solvers many new concepts like *Conflict Driven Clause Learning (CDCL)*, *Variable State Independent Decaying Sum (VSIDS)*, *Literal Block Distance (LBD)* and many more have led to massive performance increases that are still going on ([BHM21] [Bie21]). The recent performance increases can be seen in the figures Figure 11 and Figure 12, which show the improvements from 2004 to 2020 in the annual *SAT competition* and the massive size of problems that can be solved in reasonable timeframes. Due to this, many real-world science and engineering problems are now translated to SAT, for example software verification, Electronic Design Automation, hardware verification and many more. With this being the case we would like to try to add the decision problem for Quasi-Inconsistency to said list. Therefore in this Bachelor's thesis, we propose to develop a SAT encoding of the problem as well as develop and implement an algorithm to generate this SAT encoding from a business rule base so that it can be checked by SAT solvers. Finally, we will run a performance comparison between the SAT-based approach and a standard "brute force" implementation.

1.2 Chapter Overview

In our first chapter *Introduction*, we will informally introduce the notion of business rules, Quasi-Inconsistency and practical examples as our motivation for this thesis. The second chapter *Preliminaries* will contain the necessary groundwork, where we will define the notation and general formal concepts that will be used throughout the thesis. In the third chapter *Encoding*, we will look at ways how we can encode the decision problem for Quasi-Inconsistency into a propositional logic formula so that it can be solved by a SAT solver. The fourth chapter *Implementation* will deal with the implementation of the algorithm that was developed in chapter three in Java. For this, we will first define an input file format for business rules that can be read by our program. The program will then transform this problem into a CNF SAT encoding. As options for the output, we will be able to choose between either generating a file in DIMACS format that can be used to run the problem through any SAT solver or to call an integrated SAT solver directly, which will be able to return the issue that was found. After having implemented the program, in our fifth chapter *Evaluation*, we will benchmark our SAT-based algorithm against a more traditional brute force approach for finding Quasi-Inconsistencies and discuss the results. Following this, in chapter six *Outlook* we will give a brief recap on what we discovered in this thesis and how this work could be expanded on.

2 Preliminaries

In this chapter, we will lay out the formal groundwork for the ideas that were already informally mentioned in the introduction and explain some further concepts that we will need in the further sections. This section is mostly based on the groundwork laid out in [CT20] and a more comprehensive overview, an introduction to more detailed concepts and further explanation can be found there.

2.1 Propositional Logic

Variables are statements of facts or just abstract statements. For example *creditworthy* from the Introduction is a variable, but it could also be something just like x. A *literal* is either a variable or the negation of a variable. We will use the term *literal* and *fact* interchangeably. A disjunction (*OR*) of literals is called a *clause*. Boolean *formulas* are a combination (with logical *AND* or *OR*) of any number of literals. An assignment to a formula is if every literal in the formula is assigned a truth value (*TRUE* (1) or *FALSE* (0)). We call an assignment *satisfying* if the boolean formula evaluates to true with this assignment. A boolean formula is in *conjunctive normal form (CNF)* if it is a conjunction (*AND*) of clauses. Any boolean formula can be transformed to CNF, however by just using boolean algebraic transformation the formula's size can increase exponentially [BHM21].
The *boolean satisfiability problem* (SAT) is one of the most important problems in computer science [Knu15]. It was the first problem to be proven NP-complete [Coo71] and can be stated as follows: Given a Boolean formula $(F(x_1, \ldots, x_n))$ expressed in conjunctive normal form, can we satisfy F by assigning values to its variables in such a way that $(F(x_1, \ldots, x_n)) = 1$? Basically, we are asking if there exists a satisfying assignment for a given formula. As any formula can be expressed in CNF this problem can universally be answered for any boolean formula.

2.2 Business Rules

As was mentioned in the introduction business rules are declarative statements that guide business decisions [Gra06]. There are different formalisms to express business rules. For our purpose we will use the traditional form of "logic programs with classical

negation but without default negation" [CT20].
A business rule r has the form: $l_0 \leftarrow l_1, \ldots, l_n$. We call l_0 the head of the rule and $\{l_1, \ldots, l_n\}$ its body. Let us look at an example of this:

$$r_1 = \{\neg creditworthy \leftarrow lowIncome, newCustomer\}$$
$$head(r_1) = \neg creditworthy$$
$$body(r_1) = \{lowIncome, newCustomer\}$$

A business *rule base* $\mathcal{B}$ is a set of business rules. A set of literals X is called a *fact base*. "A set of literals X activates a set of rules R, iff there is a sequence $< r_1; \ldots; r_n >$ with $R = \{r_1, \ldots, r_n\}$ so that:

1. $body(r_1) \subseteq X$
2. for all i=2,...,n : $body(r_i) \subseteq \{head(r_1), \ldots, head(r_{i-1})\} \cup X$" [CT20]

We call the set $\{head(r_1), \ldots, head(r_{i-1})\}$ the *ActivatedSet* of (X_1, R_1). "A set of facts X minimally-activates [..] a set of rules R, iff X activates R and there is no proper subset of X that also activates R." [CT20]. Let us clarify this with an example:

$$\mathcal{B} = \{c \leftarrow b; b \leftarrow a\}$$

$\{a\}$ and $\{a, b\}$ are activation sets for $\mathcal{B}$. $\{a\}$ minimally-activates $\mathcal{B}$. A rule base R is $X - inconsistent$ wrt. to a fact base X if $X \cup ActivatedSet(X, R)$ is inconsistent (in the classic-logical sense).

2.3 Quasi-Inconsistency

Let $\mathcal{B}$ be a rule base, with R_1 and R_2 being rule bases which are subsets of $\mathcal{B}$. X_1 and X_2 are consistent fact bases of possible literals in $\mathcal{B}$. A tuple (X_1, X_2, R_1, R_2) is called an issue iff

1. $X_1 \subseteq X_2$
2. X_1 minimally-activates R_1
3. X_2 minimally-activates R_2
4. R_1 is X_1 consistent and R_2 is X_2 consistent
5. $R_1 \cup R_2$ is X_2 inconsistent

The rule base $\mathcal{B}$ is Quasi-Inconsistent, iff at least one issue can be found for this rule base. An issue (X_1, X_2, R_1, R_2) "describes a case where the activation of one set of rules R_1 implies the activation of a second set of rules R_2 and both sets together derive an inconsistency (while being consistent on their own)."[CT20] As has been expressed in [CD19] from these requirements we are able to conclude that Quasi-Inconsistency can only occur, iff there is a literal a, which is the conclusion of one rule and its negation $\neg a$ is the conclusion of another rule. This matches our intuitive understanding of Quasi-Inconsistency that the inconsistency has to be derived from rule conclusions and not by contradictions of input facts and rule conclusions. We want to check if the minimal activation requirement is actually necessary to find Quasi-Inconsistency as this is not clearly obvious from the start. We are going to look at a counterexample, that brings the point across really well and that was found by running the implementation without minimal activation enabled. Consider the (non)issue:

$$\begin{aligned}
X_1 &= \{a2\} \\
X_2 &= \{a2, \neg b2\} \\
R_1 &= \{b2 \leftarrow a2\} \\
R_2 &= \{\}
\end{aligned}$$

This “issue” fulfills all requirements for an issue but minimal activation, but we can clearly see that it does not fit our understanding of it. The inconsistency is not actually derived from different rule activations, but by just introducing an additional negated literal into X_2. We could even generalize this approach and almost always find an issue. We just need to set R_2 to the empty set and put one rule into R_1. Then we put the activation literals for this rule into X_1 and X_2 and add the negated head of the rule to X_2. We can clearly see that this is not the point of Quasi-Inconsistency. If we add minimal activation this does not work anymore, since R_2 cannot be the empty set, without X_2 being empty as well, which would lead to all sets being empty and violating the inconsistency rule.

From this point on we will call the decision problem if a rule base is Quasi-Inconsistent *DEC-QI*, which will return true if the rule base is Quasi-Inconsistent and false if it is not.

3 Encoding

To find out if a rule base is Quasi-Inconsistent we are going to look for issues as defined in subsection 2.3. By looking at the definition of these issues we can partition the problem into multiple easier problems. We will repeat this definition here since it is a vital concept for all the following sections. With X_1, X_2 being sets of facts and R_1, R_2 being sets of rules, we basically have to check each of the requirements for a set (X_1, X_2, R_1, R_2) being an issue separately, with those namely being:

1. $X_1 \subseteq X_2$
2. X_1 minimally-activates R_1
3. X_2 minimally-activates R_2
4. R_1 is X_1 consistent and R_2 is X_2 consistent
5. $R_1 \cup R_2$ is X_2 inconsistent

We will look at how to find constraints for each of those problems and encode them into a SAT formula. However we actually split these up into even more problems so that they are easier to deal with. Specifically since minimal activation is by far the hardest of these problems, we will divide points two and three into first checking the activation and then adding another constraint encoding to guarantee its minimality. All the different requirements can in the end be joined together by logical *AND*, since this means that all requirements have to be met by themselves and the only way the overall formula evaluates to true is if every requirement formula evaluates to true as well. Since every boolean formula can be transformed into a CNF, we will not focus on actually formulating our individual requirements in CNF, as this can be more easily and efficiently achieved by relying on external libraries that focus on handling this type of problem.

3.1 Notation

In this chapter, we are defining the notation that will be used for the rest of this paper. We will only define matters that will be used multiple times throughout the paper. If some notation is only used once it will be explained at the place where it is actually used. All rules of the input rule base $\mathcal{B}$ are indexed with the index k starting from 1 to m. $\mathcal{A}$ will be the set of all possible literals (which are mentioned in the rule base either in the head or body). $\mathcal{H}$ is the set of all possible literals that are rule heads. $a_q \in \mathcal{A}$ will be one possible literal and $h_l \in \mathcal{H}$ one possible head. X_j denotes a set of facts. R_i denotes a set of rules.

Variable $x_{j,a}$ is true if atom a (in non-negated form) is included in the set of facts X_j and false if it is not included. Variable $x_{j,-a}$ is true if the negated atom a ($\neg a$) is included in the set of facts X_j and false if it is not included.

It is very important to note here that $x_{j,-a} \neq \neg x_{j,a}$ and vice versa. One means the literal $\neg a$ is present in the fact base X_j, while the other means that the literal a is not present in the fact base. This becomes especially important when we talk about activation variables later on, for example $-a_{Act_1} \neq \neg a_{Act_1}$. a_{Act_1} meant that a is in the activated set of X_1, R_1 (the rule with head a was activated). $a_{Act_{1,2}}$ also describes, in which rule the activation happened. In this case a was activated in rule with index number 2. It is also important to realize that we do not and cannot look at the semantics of variable names. This means that *notCreditworthy* is not the same as $\neg$*creditworthy*. One is just a variable with the name *notCreditworthy*, the other one is the negated variable *creditworthy*.

$r_{i,k}$ is true if the rule with the index number k r_k is included in the rule set R_i. As we have seen we can split rules up into head and body with the head only being one literal. $head(r_k)$ will be denoted as h_k. The body of a rule r_k will be compromised of $b_{k,1}$ to b_{k,p_k} with o being the index for the different body variables and p_k the number of body literals for one rule k.

$a_{Act(1)}$ is true if a is the head of a rule that was activated in combination of X_1 and R_1. $a_{Act(1,2)}$ is true if a is the head of rule 2, which was activated in combination of X_1

and R_1.

From now on we will explain the concepts in this chapter by looking at the following rule base to demonstrate the examples:

$$\mathcal{B}_1 = \{c \leftarrow a, b\,;\ \neg c \leftarrow a, d\,;\ d \leftarrow b\}$$

First, we enumerate the rules so that we can refer to them easier and it is clear which rules we mean:

$$r_1 = c \leftarrow a, b\,;\ r_2 = \neg c \leftarrow a, d\,;\ r_3 = d \leftarrow b.$$

If we would like to look at different parts of the rules we can refer to them by the index notation described above. For example, we can write $h_1 = c, b_{1,1} = a, b_{1,2} = b$. For this rule base we can find the following issue:

$$\begin{aligned} X_1 &= \{a, b\} \\ X_2 &= \{a, b\} \\ R_1 &= \{c \leftarrow a, b\} \\ R_2 &= \{d \leftarrow b\,;\ \neg c \leftarrow a, d\} \end{aligned}$$

If we use the notation introduced above for our SAT encoding, this issue is:

$$ISSUE(\mathcal{B}_1) = \{x_{1,a}, x_{1,b}, x_{2,a}, x_{2,b}, r_{1,1}, r_{2,2}, r_{2,3}\}$$

An overview of all different notation concepts with their context is displayed in Table 1.

3.2 Consistency Fact Base

One constraint that is not directly mentioned in the rules above, is that the fact bases themselves have to be consistent. This basically means if a literal is part of a fact base, its negated counterpart is not allowed to be included in this fact base. We can constrain this by adding the following statements for all possible literals a_q (with n being the number of possible literals) and fact bases:

$$\bigwedge_{j=1}^{2} \bigwedge_{q=1}^{n} \neg(x_{j,a_q} \wedge x_{j,-a_q})$$

For our sample rule base $\mathcal{B}_1$ and a fact base X_j it looks like this:

$$Consistency(X_j) = \neg(x_{j,a} \wedge x_{j,-a}) \wedge \neg(x_{j,b} \wedge x_{j,-b}) \wedge \neg(x_{j,c} \wedge x_{j,-c}) \wedge \neg(x_{j,d} \wedge x_{j,-d})$$

In this equation we can already see that we skipped the clause $\neg(x_{j,-c} \wedge x_{j,-(-c)})$ since it is the same as $\neg(x_{j,c} \wedge x_{j,-c})$. So actually we do not need this for all possible literals, but only for all possible variables.

Letter	Description	Example	Explanation
$\mathcal{B}$	input rule base		
$\mathcal{A}$	set of all possible literals		
$\mathcal{H}$	set of all rule heads		
R	rule base in potential issue ($\subseteq \mathcal{B}$)	R_1	rule base 1
X	fact base in potential issue ($\subseteq \mathcal{A}$)	X_2	fact base 2
a	one literal $\in \mathcal{A}$	a_q	any possible fact
b	one body literal	$b_{1,2}$	second fact of the rule body of r_1
h	one head literal $\in \mathcal{H}$	h_1	head of the first rule
n	#possible literals		$\|\mathcal{A}\|$
m	#all rules		$\|\mathcal{B}\|$
p	#literals in body	p_k	#literals in body k
i	index of rule base	R_i	rule base number i
j	index of fact base	X_j	fact base number j
o	index of body literals	$b_{2,o}$	o-th fact of rule body r_2
l	index of head literals	h_k	head of the k-th rule
k	index for rules	r_k	k-th rule
q	index for a	a_q	
r	boolean value	$r_{1,3}$	rule 3 $\in R_1$
x	boolean value	$x_{2,a}$	fact $a \in X_2$
$a_{Act_{i,k}}$	boolean value	$a_{Act_{2,3}}$	rule 3 activated $a \in ActivatedSet(X_1, R_1)$
a_{Act_i}	boolean value	a_{Act_2}	fact a $\in ActivatedSet(X_1, R_1)$

Table 1: Notation

$x_{1,a}$	$x_{2,a}$	result
0	0	1
0	1	1
1	0	0
1	1	1

Table 2: truth table for set inclusion

3.3 Set Inclusion

As X_1 has to be a subset of X_2 this basically means, that each literal that is present in X_1 has also to be present in X_2. The other way around is however not required, a literal that is included in X_2 does not necessarily have to be included in X_1. Writing this into a truth table we get the result shown in Table 2. From this table we can see that this constraint corresponds to the logical implication from X_2 to X_1.

$$x_{2,a} \rightarrow x_{1,a} = \neg x_{2,a} \lor x_{1,a}$$

Therefore we can guarantee set inclusion by adding the following to our formula (for all possible literals a_q):

$$\bigwedge_{q=1}^{n} (x_{2,a_q} \rightarrow x_{1,a_q}) = \bigwedge_{q=1}^{n} (\neg x_{2,a_q} \lor x_{1,a_q})$$

Looking at our example rule base $\mathcal{B}_1$ it looks like this:

$$\begin{aligned} SetInclusion(X_1, X_2) = &(\neg x_{1,a} \lor x_{2,a}) \land (\neg x_{1,b} \lor x_{2,b}) \land (\neg x_{1,c} \lor x_{2,c}) \land \\ &(\neg x_{1,d} \lor x_{2,d}) \land (\neg x_{1,-c} \lor x_{2,-c}) \end{aligned}$$

3.4 Activation

As mentioned above for this we introduce helper variables $h_{Act(i)}$ for all literals l that are heads of rules in the overall rule base for our rule base R_i. The requirements for this basically states, that if a fact base is to activate a rule base, all body atoms of all rules have to be either in the fact set of the rule base or in the activation set and all rules have to be activated. Thinking of how to solve this we can partition our problem further and think of different ideas that need to be true to achieve that. First of all we actually need to be able to activate rules and thus add new literals to our activated set, which can in turn activate other rules. We do that by using the following formula:

$$\bigwedge_{j=1}^{2} \bigwedge_{k=1}^{m} ((\bigwedge_{o=1}^{p_k} (x_{j,b_{k,o}} \lor b_{k,o_{Act_j}}) \land r_{j,k}) \leftrightarrow h_{k_{Act_{j,k}}})$$

Take note that we use the same index variable j for rule and fact bases as we only compare X_1 to R_1 and X_2 to R_2, so the index can be the same and we do not need to unnecesarily complicate our formula. $r_{j,k}$ guarantees that a rule can only be activated if the rule is actually part of the rule base we are looking at. The conjunction of all $(x_{j,b_o} \vee b_{o_{Act_j}})$ checks if all body literals are either in the fact base X_j or in the activated set for this rule and fact base combination. If so, the rule will be activated and its head literal will be added to the activation set. We do this for all rules of the rule base for each of the two fact bases. Now that we made the actual rule activation happen, we need to add some constraints:

1. If a rule r_k is present in a rule base, its head has to be in the activation set. This also holds true for the other way around: If a literal is part of an activation set for a rule, it has to be activated by that rule. We can write this with our notation in propositional logic as: $h_{k_{Act_{j,k}}} \leftrightarrow r_{j,k}$
2. If a literal is activated in one rule r_k of the rule base R_i it is "active" everywhere else in this rule base, but not the other way around: $h_{k_{Act_{j,k}}} \rightarrow h_{k_{Act_j}}$
3. If a literal is part of an overall activation set in the rule, it has to have been activated somewhere (and cannot just be guessed as true by the solver): $h_{k_{Act_j}} \rightarrow \bigvee_{l=1}^{r} h_{l_{Act_{j,l}}}$ where r is the number of rules that contain h_l as their head. This constraint guarantees that at least one of the $h_{l_{Act_{j,l}}}$ is present.

We add these formulas for each rule of the rule base and with those we can check whether a set of facts X_j activates a rule base R_i. To summarize our overall equation for Activation now looks like this:

$$\bigwedge_{j=1}^{2} \bigwedge_{k=1}^{m} (((\bigwedge_{o=1}^{p_k} (x_{j,b_{k,o}} \vee b_{k,o_{Act_j}}) \wedge r_{j,k}) \leftrightarrow h_{k_{Act_{j,k}}}) \wedge (h_{k_{Act_{j,k}}} \leftrightarrow r_{j,k})$$
$$\wedge (h_{k_{Act_{j,k}}} \rightarrow h_{k_{Act_j}}) \wedge (h_{k_{Act_j}} \rightarrow \vee_{l=1}^{r} h_{l_{Act_{j,l}}}))$$

Let us look how this would look for our example rule base $\mathcal{B}_1$:

$$\begin{aligned} Activation(X_1, R_1) = &((r_{1,1} \wedge (x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})) \leftrightarrow c_{Act_{1,1}}) \wedge \\ &((r_{1,2} \wedge (x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1})) \leftrightarrow \neg c_{Act_{1,2}}) \wedge \\ &((r_{1,3} \wedge (x_{1,b} \vee b_{Act_1})) \leftrightarrow d_{Act_{1,3}}) \wedge \\ &(c_{Act_{1,1}} \leftrightarrow r_{1,1}) \wedge (-c_{Act_{1,2}} \leftrightarrow r_{1,2}) \wedge (d_{Act_{1,3}} \leftrightarrow r_{1,3}) \wedge \\ &(c_{Act_{1,1}} \rightarrow c_{Act_1}) \wedge (\neg c_{Act_{1,2}} \rightarrow \neg c_{Act_1}) \wedge (d_{Act_{1,3}} \rightarrow d_{Act_1}) \end{aligned}$$

As we can see the third constraint is not added, since all rules have different rule heads and therefore this is not needed. We do not actually have to add activation literals at every point. If a literal is not in the head of any rule it can obviously never be activated and thus does not have to be considered. Actually if we did write it like above, where we did not include this optimization to make the concept clearer, we would have to add constraints so that literals like b_{Act_1} can never be true. Otherwise the SAT solver could just guess these as true, although they are not in any rule head. We will avoid this problem by just never adding variables like this in the first place. Thus we remove the literals a_{Act_1}, b_{Act_1} as they cannot be activated (only c, $\neg c$ and d are heads and can be activated):

$$\begin{aligned}
Activation(X_1, R_1) = &((r_{1,1} \wedge x_{1,a} \wedge x_{1,b}) \leftrightarrow c_{Act_{1,1}}) \wedge \\
&((r_{1,2} \wedge x_{1,a} \wedge (x_{1,d} \vee d_{Act_1})) \leftrightarrow \neg c_{Act_{1,2}}) \wedge \\
&((r_{1,1} \wedge x_{1,b}) \leftrightarrow d_{Act_{1,3}}) \wedge \\
&(c_{Act_{1,1}} \leftrightarrow r_{1,1}) \wedge (-c_{Act_{1,2}} \leftrightarrow r_{1,2}) \wedge (d_{Act_{1,3}} \leftrightarrow r_{1,3}) \wedge \\
&(c_{Act_{1,1}} \rightarrow c_{Act_1}) \wedge (\neg c_{Act_{1,2}} \rightarrow \neg c_{Act_1}) \wedge (d_{Act_{1,3}} \rightarrow d_{Act_1})
\end{aligned}$$

3.5 Minimal Activation

Minimal activation means, that if we remove any one literal from a fact base it will not activate the belonging rule base anymore. To check this we try to think of an encoding that will remove exactly one literal each from the fact base and check if this leads to the activation not happening anymore. This needs to be true for every removed literal. We will have to develop an encoding that takes every literal and every rule into account, but at run time only checks the condition if the literal was included in the fact base in the first place. To take care of this the overall formula (for one literal a_q) has to be a disjunction with $\neg x_{i,a_q}$ and "not activation". This disjunction will let the solver ignore the other constraints, due to $\neg x_{i,a_q}$ being true if it is not in the fact base and thus satisfying the whole disjunction.
In the second part "not activation", we will have to check for all rules present, if removing the literal a_q from the fact base will lead to it not being activated anymore. A rule is activated if it is present and all its body literals are in the set, so we do not need to consider the head of rules as well as the activation, since this already happens in subsection 3.4. So in our second part we will set every occurrence of a_q to false. Let us look at how the "not activation" part will look like for one fact:

$$\neg(\bigvee_{k=1}^{m} \neg r_{j,k} \vee (\bigwedge_{o=1}^{p_k} (x_{i,b_{k,o}} \vee b_{k,o_{Act_j}})))$$

If we combine this to an overall expression for minimal activation we get:

$$\bigwedge_{j=1}^{2} \bigwedge_{q=1}^{n} (\neg x_{j,a_q} \vee \neg(\bigvee_{k=1}^{m} \neg r_{j,k} \vee (\bigwedge_{o=1}^{p_k} (x_{j,b_{k,o}} \vee b_{k,o_{Act_j}}))))$$

In this formula we will replace $(x_{j,b_{k,o}}$ for false, if $b_{k,o} = a_q$ in every different iteration of q. Note that in this case the index variable for facts x_j and rules r_j is the same since we always have one fact base and one rule base that we analyze. So let us see how our idea looks like for our example rule base

$$
\begin{aligned}
&MinimalActivation(X_1, R_1) = \\
&(\neg x_{1,a} \vee \neg((\neg r_{1,1} \vee ((false \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge \\
&(\neg r_{1,2} \vee ((false \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge \\
&(\neg x_{1,b} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (false \vee b_{Act_1})))\wedge \\
&(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (false \vee b_{Act_1}))))\wedge \\
&(\neg x_{1,c} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge \\
&(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge \\
&(\neg x_{1,d} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge \\
&(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (false \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge \\
&(\neg x_{1,-c} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge \\
&(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))
\end{aligned}
$$

3.6 Consistency Rule Base To Fact Base

We have already checked the fact bases for consistency within themselves in subsection 3.2. To achieve consistency between a fact base X_j and a rule base R_i we only have to guarantee that no rule head is activated, that is inconsistent with the fact base or its own activation. This can be easily done by adding the following constraints for all n possible rule heads $h_k \in \mathcal{H}$:

$$Consistency(X_1, R_1) = \bigwedge_{k=1}^{m} (h_{k_{Act_1}} \rightarrow \neg(x_{1,-h_k} \vee -h_{k_{Act_1}}))$$

Putting this into a combined formula it is:

$$Consistency = \bigwedge_{j=1}^{2} \bigwedge_{k=1}^{m} (h_{k_{Act_j}} \rightarrow \neg(x_{j,-h_k} \vee -h_{k_{Act_j}}))$$

Looking at our example it looks like this:

$$
\begin{aligned}
Consistency(X_1, R_1) =& (c_{Act_1} \rightarrow \neg(x_{1,-c} \vee -c_{Act_1})) \wedge (-c_{Act_1} \rightarrow \neg(x_{1,c} \vee c_{Act_1}))\wedge \\
& (d_{Act_1} \rightarrow \neg(x_{1,-d} \vee -d_{Act_1}))
\end{aligned}
$$

We can further optimize here. In the above formula we have included the variable $-d_{Act_1}$. As $\neg d$ is not in any rule head, it cannot possibly be activated. This means we do not need to consider it for our rule base $\mathcal{B}_1$ nor do we need to consider literals that are not rule heads as activation variables.

3.7 Inconsistency

Now we want to consider the next prerequisite that we have to meet for an issue: $R_1 \cup R_2$ is X_2-inconsistent. We already know that X_2 and R_2 together are consistent. This basically means, that if we add the rule base R_1 to our previously consistent combination, some rule from R_1 has to get activated which will cause an inconsistency. We use the same idea as in subsection 3.4 to activate the rule bases. However this time we model the constraints that we get at least one inconsistency. For this, we will need new activation variables though, which we will denote with index 3, e.g. $h_{k_{Act_3}}$. Let us look at the activation (for one rule k of the rule base) first:

$$((\bigwedge_{o=1}^{p_k}(x_{2,b_o} \vee b_{k,o_{Act_2}} \vee b_{k,o_{Act_3}})) \wedge (r_{1,k} \vee r_{2,k})) \leftrightarrow h_{k_{Act_{3,k}}}$$

As for the constraints, we can skip the first one mentioned in subsection 3.4, as we do not necessarily need to activate all the rules. The second and third constraints we do however need:

1. If a literal is activated in one rule r_k of the rule base R_3 it is "active" everywhere else in this rule base, but not the other way around: $h_{k_{Act(3,k)}} \rightarrow h_{k_{Act_3}}$
2. If a literal is part of an overall activation set in the rule, it has to have been activated somewhere (and cannot just be guessed as true by the solver): $h_{k_{Act_3}} \rightarrow \bigvee_{l=1}^{r} h_{l_{Act(3,l)}}$ (for notation see subsection 3.4)

Now we need to check, whether this activation will lead to an inconsistency. We do this by specifying that at least one $h_{k_{Act_3}}$ has to lead to an inconsistency, which is the case if $-h_k$ is present in X_2 or in one of the activation sets of Act_2 or Act_3.

$$\mathit{AtLeastOne}(h_{k_{Act_3}} \wedge (-h_{k_{Act_3}} \vee -h_{k_{Act_2}} \vee x_{2,-h_k})) =$$
$$\bigvee_{k=1}^{n}(h_{k_{Act_3}} \wedge (-h_{k_{Act_3}} \vee -h_{k_{Act_2}} \vee x_{2,-h_k}))$$

Combining this we get the formula for $Inconsistency(X_2, R_1 \cup R_2)$:

$$\bigwedge_{k=1}^{m}(((\bigwedge_{o=1}^{p_k}(x_{2,b_{k,o}} \vee b_{k,o_{Act_2}} \vee b_{k,o_{Act_3}})) \wedge (r_{1,k} \vee r_{2,k})) \leftrightarrow h_{k_{Act_{3,k}}} \wedge (h_{k_{Act(3,k)}} \rightarrow h_{k_{Act_3}}) \wedge$$
$$(h_{k_{Act_3}} \rightarrow \bigvee_{l=1}^{r} h_{l_{Act(3,l)}})) \wedge \bigvee_{k=1}^{n}(h_{k_{Act_3}} \wedge (-h_{k_{Act_3}} \vee -h_{k_{Act_2}} \vee x_{2,-h_k}))$$

If we transfer the whole concept to our example rule base, this is:

$$\begin{aligned}
&Inconsistency(X_2, R_1 \cup R_2) = \\
&(((x_{2,a} \lor a_{Act_2} \lor a_{Act_3}) \land (x_{2,b} \lor b_{Act_2} \lor b_{Act_3}) \land (r_{2,1} \lor r_{1,1})) \leftrightarrow c_{Act_{3,1}})\land \\
&(((x_{2,a} \lor a_{Act_2} \lor a_{Act_3}) \land (x_{2,d} \lor d_{Act_2} \lor d_{Act_3}) \land (r_{2,2} \lor r_{1,2})) \leftrightarrow -c_{Act_{3,2}})\land \\
&(((x_{2,b} \lor b_{Act_2} \lor b_{Act_3}) \land (r_{2,3} \lor r_{1,3})) \leftrightarrow d_{Act_{3,3}})\land \\
&(c_{Act_{3,1}} \rightarrow c_{Act_3}) \land (-d_{Act_{3,2}} \rightarrow -c_{Act_3}) \land (d_{Act_{3,3}} \rightarrow d_{Act_3})\land \\
&(c_{Act_3} \rightarrow c_{Act_{3,1}}) \land (-c_{Act_3} \rightarrow -d_{Act_{3,2}}) \land (d_{Act_3} \rightarrow d_{Act_{3,3}})\land \\
&((c_{Act_3} \land (x_{2,-c} \lor -c_{Act_3} \lor -c_{Act_2})) \lor ((-c_{Act_3} \land (x_{2,c} \lor c_{Act_3} \lor c_{Act_2})))\lor \\
&(d_{Act_3} \land (x_{2,-d} \lor -d_{Act_3} \lor -d_{Act_2})))
\end{aligned}$$

The first three lines are the rules that are directly responsible for the rule activation. In the next two lines, we guarantee that rule activation in one rule means activation in the whole set. The last two lines check that at least one inconsistency is found.

3.8 Summary

As we stated before, we need to connect all these separate requirements into a common formula by joining them together with a logical *AND*.

$$\begin{aligned}
DEC - QI(\mathcal{B}_1) =\, &Consistency(X_1) \land Consistency(X_2) \land SetInclusion(X_1, X_2)\land \\
&Activation(X_1, R_1) \land Activation(X_2, R_2)\land \\
&MinimalActivation(X_1, R_1) \land MinimalActivation(X_2, R_2)\land \\
&Consistency(X_1, R_1) \land Consistency(X_2, R_2)\land \\
&Inconsistency(X_2, R_1 \cup R_2)
\end{aligned}$$

If we substitute these with our developed encodings in the previous subsections we get the overall general formula for *DEC-QI* (for the notation see subsection 3.1 and Table 1):

$$DEC - QI(\mathcal{B}_1) =$$

$$\bigwedge_{j=1}^{2}\bigwedge_{q=1}^{n} \neg(x_{j,a_q} \wedge x_{j,-a_q}) \wedge \bigwedge_{q=1}^{n}(\neg x_{2,a_q} \vee x_{1,a_q})\wedge$$

$$\bigwedge_{j=1}^{2}\bigwedge_{k=1}^{m}(((\bigwedge_{o=1}^{p_k}(x_{j,b_{k,o}} \vee b_{k,o_{Act_j}}) \wedge r_{j,k}) \leftrightarrow h_{k_{Act_{j,k}}}) \wedge (h_{k_{Act_{j,k}}} \leftrightarrow r_{j,k})\wedge$$

$$(h_{k_{Act_{j,k}}} \to h_{k_{Act_j}}) \wedge (h_{k_{Act_j}} \to \vee_{l=1}^{r} h_{l_{Act_{j,l}}}))\wedge$$

$$\bigwedge_{j=1}^{2}\bigwedge_{k=1}^{m}(h_{k_{Act_j}} \to \neg(x_{j,-h_k} \vee -h_{k_{Act_j}}))\wedge$$

$$\bigwedge_{k=1}^{m}(((\bigwedge_{o=1}^{p_k}(x_{2,b_{k,o}} \vee b_{k,o_{Act_2}} \vee b_{k,o_{Act_3}})) \wedge (r_{1,k} \vee r_{2,k})) \leftrightarrow h_{k_{Act_{3,k}}} \wedge (h_{k_{Act(3,k)}} \to h_{k_{Act_3}})\wedge$$

$$(h_{k_{Act_3}} \to \bigvee_{l=1}^{r} h_{l_{Act(3,l)}})) \wedge \bigvee_{k=1}^{n}(h_{k_{Act_3}} \wedge (-h_{k_{Act_3}} \vee -h_{k_{Act_2}} \vee x_{2,-h_k}))\wedge$$

$$\bigwedge_{j=1}^{2}\bigwedge_{q=1}^{n}(\neg x_{j,a_q} \vee \neg(\bigvee_{k=1}^{m} \neg r_{j,k} \vee (\bigwedge_{o=1}^{p_k}(x_{j,b_{k,o}} \vee b_{k,o_{Act_j}}))))$$

Now let us take a peek at how all of this combined will look for our rule base $\mathcal{B}_1$:

$$SetInclusion(X_1, X_2) =$$
$$(\neg x_{1,a} \vee x_{2,a}) \wedge (\neg x_{1,b} \vee x_{2,b}) \wedge (\neg x_{1,c} \vee x_{2,c}) \wedge (\neg x_{1,d} \vee x_{2,d}) \wedge (\neg x_{1,-c} \vee x_{2,-c})$$

$$Consistency(X_1) =$$
$$\neg(x_{1,a} \wedge x_{1,-a}) \wedge \neg(x_{1,b} \wedge x_{1,-b}) \wedge \neg(x_{1,c} \wedge x_{1,-c}) \wedge \neg(x_{1,d} \wedge x_{1,-d})$$

$$Consistency(X_2) =$$
$$\neg(x_{2,a} \wedge x_{2,-a}) \wedge \neg(x_{2,b} \wedge x_{2,-b}) \wedge \neg(x_{2,c} \wedge x_{2,-c}) \wedge \neg(x_{2,d} \wedge x_{2,-d})$$

$$Activation(X_1, R_1) =$$
$$((r_{1,1} \wedge x_{1,a} \wedge x_{1,b}) \leftrightarrow c_{Act_{1,1}}) \wedge ((r_{1,2} \wedge x_{1,a} \wedge (x_{1,d} \vee d_{Act_1})) \leftrightarrow \neg c_{Act_{1,2}})\wedge$$
$$((r_{1,1} \wedge x_{1,b}) \leftrightarrow d_{Act_{1,3}}) \wedge (c_{Act_{1,1}} \leftrightarrow r_{1,1}) \wedge (-c_{Act_{1,2}} \leftrightarrow r_{1,2}) \wedge (d_{Act_{1,3}} \leftrightarrow r_{1,3})\wedge$$
$$(c_{Act_{1,1}} \to c_{Act_1}) \wedge (\neg c_{Act_{1,2}} \to \neg c_{Act_1}) \wedge (d_{Act_{1,3}} \to d_{Act_1})$$

$$Activation(X_2, R_2) =$$
$$((r_{2,1} \wedge x_{2,a} \wedge x_{2,b}) \leftrightarrow c_{Act_{2,1}}) \wedge ((r_{2,2} \wedge x_{2,a} \wedge (x_{2,d} \vee d_{Act_2})) \leftrightarrow \neg c_{Act_{2,2}})\wedge$$
$$((r_{2,1} \wedge x_{2,b}) \leftrightarrow d_{Act_{2,3}}) \wedge (c_{Act_{2,1}} \leftrightarrow r_{2,1}) \wedge (-c_{Act_{2,2}} \leftrightarrow r_{2,2}) \wedge (d_{Act_{2,3}} \leftrightarrow r_{2,3})\wedge$$
$$(c_{Act_{2,1}} \to c_{Act_2}) \wedge (\neg c_{Act_{2,2}} \to \neg c_{Act_2}) \wedge (d_{Act_{2,3}} \to d_{Act_2})$$

$Activation(X_2, R_2) =$
$(\neg r_{2,1} \vee (c_{Act(2,1)} \leftrightarrow ((x_{2,a} \vee a_{Act(2)}) \vee (x_{2,b} \vee b_{Act(2)})))))\wedge$
$(\neg r_{2,2} \vee (-c_{Act(2,2)} \leftrightarrow ((x_{2,a} \vee a_{Act(2)}) \vee (x_{2,d} \vee d_{Act(2)})))))\wedge$
$(\neg r_{2,3} \vee (d_{Act(2,3)} \leftrightarrow (x_{2,b} \vee b_{Act(2)})))\wedge$
$(c_{Act(2,1)} \leftrightarrow r_{2,1}) \wedge (-c_{Act(2,2)} \leftrightarrow r_{2,2}) \wedge (d_{Act(2,3)} \leftrightarrow r_{2,3})\wedge$
$(c_{Act(2,1)} \rightarrow c_{Act(2)}) \wedge (-c_{Act(2,2)} \rightarrow -c_{Act(2)}) \wedge (d_{Act(2,3)} \rightarrow d_{Act(2)})$
$(c_{Act(2)} \rightarrow c_{Act(2,1)}) \wedge (-c_{Act(2)} \rightarrow -c_{Act(2,2)}) \wedge (d_{Act(2)} \rightarrow d_{Act(2,3)})$

$MinimalActivation(X_1, R_1) =$
$(\neg x_{1,a} \vee \neg((\neg r_{1,1} \vee ((false \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge$
$(\neg r_{1,2} \vee ((false \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge$
$(\neg x_{1,b} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (false \vee b_{Act_1})))\wedge$
$(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (false \vee b_{Act_1}))))\wedge$
$(\neg x_{1,c} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge$
$(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge$
$(\neg x_{1,d} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge$
$(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (false \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))\wedge$
$(\neg x_{1,-c} \vee \neg((\neg r_{1,1} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,b} \vee b_{Act_1})))\wedge$
$(\neg r_{1,2} \vee ((x_{1,a} \vee a_{Act_1}) \wedge (x_{1,d} \vee d_{Act_1}))) \wedge (\neg r_{1,3} \vee (x_{1,b} \vee b_{Act_1}))))$

$MinimalActivation(X_2, R_2) =$
$(\neg x_{2,a} \vee \neg((\neg r_{2,1} \vee ((false \vee a_{Act_2}) \wedge (x_{2,b} \vee b_{Act_2})))\wedge$
$(\neg r_{2,2} \vee ((false \vee a_{Act_2}) \wedge (x_{2,d} \vee d_{Act_2}))) \wedge (\neg r_{2,3} \vee (x_{2,b} \vee b_{Act_2}))))\wedge$
$(\neg x_{2,b} \vee \neg((\neg r_{2,1} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (false \vee b_{Act_2})))\wedge$
$(\neg r_{2,2} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (x_{2,d} \vee d_{Act_2}))) \wedge (\neg r_{2,3} \vee (false \vee b_{Act_2}))))\wedge$
$(\neg x_{2,c} \vee \neg((\neg r_{2,1} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (x_{2,b} \vee b_{Act_2})))\wedge$
$(\neg r_{2,2} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (x_{2,d} \vee d_{Act_2}))) \wedge (\neg r_{2,3} \vee (x_{2,b} \vee b_{Act_2}))))\wedge$
$(\neg x_{2,d} \vee \neg((\neg r_{2,1} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (x_{2,b} \vee b_{Act_2})))\wedge$
$(\neg r_{2,2} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (false \vee d_{Act_2}))) \wedge (\neg r_{2,3} \vee (x_{2,b} \vee b_{Act_2}))))\wedge$
$(\neg x_{2,-c} \vee \neg((\neg r_{2,1} \vee ((x_{2,a} \vee a_{Act_2}) \wedge (x_{2,b} \vee b_{Act_2})))\wedge$
$(\neg r_{2,2} \vee ((x_{1,a} \vee a_{Act_2}) \wedge (x_{2,d} \vee d_{Act_2}))) \wedge (\neg r_{2,3} \vee (x_{2,b} \vee b_{Act_2}))))$

$Consistency(X_1, R_1) =$
$(c_{Act_1} \rightarrow \neg(x_{1,-c} \vee -c_{Act_1})) \wedge (-c_{Act_1} \rightarrow \neg(x_{1,c} \vee c_{Act_1})) \wedge (d_{Act_1} \rightarrow \neg(x_{1,-d} \vee -d_{Act_1}))$
$Consistency(X_2, R_2) =$
$(c_{Act_2} \rightarrow \neg(x_{2,-c} \vee -c_{Act_2})) \wedge (-c_{Act_2} \rightarrow \neg(x_{2,c} \vee c_{Act_2})) \wedge (d_{Act_2} \rightarrow \neg(x_{2,-d} \vee -d_{Act_2}))$

$$
\begin{aligned}
&Inconsistency(X_2, R_1 \cup R_2) = \\
&(((x_{2,a} \vee a_{Act_2} \vee a_{Act_3}) \wedge (x_{2,b} \vee b_{Act_2} \vee b_{Act_3}) \wedge (r_{2,1} \vee r_{1,1})) \leftrightarrow c_{Act_{3,1}})\wedge \\
&(((x_{2,a} \vee a_{Act_2} \vee a_{Act_3}) \wedge (x_{2,d} \vee d_{Act_2} \vee d_{Act_3}) \wedge (r_{2,2} \vee r_{1,2})) \leftrightarrow -c_{Act_{3,2}})\wedge \\
&(((x_{2,b} \vee b_{Act_2} \vee b_{Act_3}) \wedge (r_{2,3} \vee r_{1,3})) \leftrightarrow d_{Act_{3,3}})\wedge \\
&(c_{Act_{3,1}} \rightarrow c_{Act_3}) \wedge (-d_{Act_{3,2}} \rightarrow -c_{Act_3}) \wedge (d_{Act_{3,3}} \rightarrow d_{Act_3})\wedge \\
&(c_{Act_3} \rightarrow c_{Act_{3,1}}) \wedge (-c_{Act_3} \rightarrow -d_{Act_{3,2}}) \wedge (d_{Act_3} \rightarrow d_{Act_{3,3}})\wedge \\
&((c_{Act_3} \wedge (x_{2,-c} \vee -c_{Act_3} \vee -c_{Act_2})) \vee ((-c_{Act_3} \wedge (x_{2,c} \vee c_{Act_3} \vee c_{Act_2}))\vee \\
&(d_{Act_3} \wedge (x_{2,-d} \vee -d_{Act_3} \vee -d_{Act_2})))
\end{aligned}
$$

This is what our final encoding for our sample rule base looks like. At a first glance, it appears like a really big formula for such a small rule base. However, if we consider that SAT solvers can solve formulas with millions of variables and clauses (Figure 11, Figure 12), this is actually pretty small in comparison. We can put the formula into a SAT solver and determine if it is Quasi-Inconsistent. In Figure 2 we can see the output of our encoding. It returns an issue, which is the same one that we already mentioned in subsection 3.1.

4 Implementation

In this chapter, we will look into how we can implement our findings from section 3 into an actual application. We will use Java as our programming language and rely heavily on the LogicNG library[1]. The source code is hosted on GitHub [2]. This repository also includes a "pom.xml" file, so that the project can be built with Apache Maven. For convenience's sake, a runnable *.jar file for the current version with all necessary libraries is also put into the project's main folder.

4.1 Program Overview

In this subsection, we will briefly explain how our program is structured. The program is started with the main method in the class *MainQiSAT*. As a first step, the supplied input arguments are read and processed by the class *ArgumentHandler*, where the validity of the parameters is checked and then written into our configuration object of type *QiSatConfiguration*. A detailed explanation of what parameters are possible, all of which are stored in the configuration object, can be found in subsection 4.4. Next up we will load up a parser object for the input. This can either be an instance of the class *DeclareModelParser* if the input file is in Comma Separated Value (CSV) format or an instance of *BusinessRuleFileParser* if it is a text rule file. Both of these implement the interface *InputFileParser* so we can easily exchange this for a new parser type if the need arises. The selected parser will then process the input and save it into our own data structure, which consists of (Java) records *Literal* and *Rule* as well as a *RuleBase* object

[1]https://logicng.org/
[2]https://github.com/tehmischi/qi-encoding-sat

which contains these records. Once the input is transformed and saved the next step is to generate the SAT encoding. This is done in multiple steps by different classes, that all implement the *SatEncoding* interface. These encoding classes execute the steps that are described in section 3. As previously mentioned to generate formulas in propositional logic we use the LogicNG library, which has very efficient implementations with factory methods to avoid unnecessary object generation as well as efficient CNF conversion. Another advantage is, that we can directly use this library to call a SAT solver, which is our next step after generating the encoding. There are two possible return values from the SAT solver: UNSAT or a satisfying assignment. With the return value UNSAT, we can immediately print out that the rule base is not Quasi-Inconsistent and exit the program. If a satisfying assignment is returned, this needs to be converted back into an output format that is easily understandable. As we have chosen the variable names in subsection 3.1 exactly for this purpose, this is easily achieved. An instance of the class *OutputStringFormatter* converts the assignment into a String that contains the issue in a recognizable form that can be printed out to the console. An example of this output can be seen in Figure 3. This is the standard program flow, if other parameters are specified this flow can obviously be deviated from, which will be explained further on in this chapter.

4.2 Input

The program reads a text file that has the following layout:

$$\begin{gathered} //Comments \\ l_0,\ l_1\ l_2 \dots l_n; \\ \dots \\ x_0,\ x_1\ x_2 \dots x_n; \end{gathered}$$

Rules are separated by Semicolons (for a better view it is recommended to use newlines, though this is not necessary). Inside a rule, the head of the rule comes first and is separated from the body by a comma. As the comma is used as a separator, variables with a comma are not allowed. Comments are escaped by a double slash (//) at the start of a line or by surrounding multiple lines with /* and */. This is the same comment format that is used by Java and many other programming languages.
We can see how our example rule base from section 3 could look like in Figure 1.

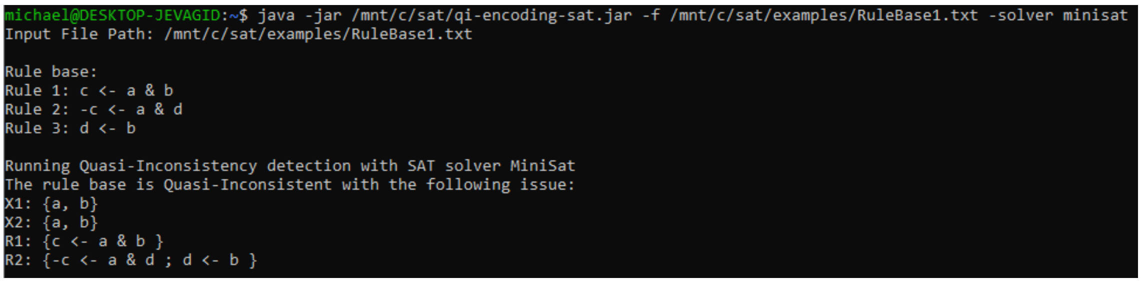
Alternatively, it can also read a CSV file that was generated by using MINERFul on real-world data as described in subsubsection 5.2.1.

4.3 Output

Depending on what the supplied input parameters were, the program will first print the data it will operate on. In all cases, this is the rule base file (or CSV file) that serves as our data base. If the option "-dimacs" is active it will also print out the output file path(see Figure 5). In all other cases, it will tell us what SAT solver is being used before

```
/*
Example rule base file for the rule base:
c <- a,b
not c <- a,d
d <- b
*/
c, a b;
-c, a d;
d, b;
```

Figure 1: Rule Base File for $\mathcal{B}_1$



```
michael@DESKTOP-JEVAGID:~$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/RuleBase1.txt -solver minisat
Input File Path: /mnt/c/sat/examples/RuleBase1.txt

Rule base:
Rule 1: c <- a & b
Rule 2: -c <- a & d
Rule 3: d <- b

Running Quasi-Inconsistency detection with SAT solver MiniSat
The rule base is Quasi-Inconsistent with the following issue:
X1: {a, b}
X2: {a, b}
R1: {c <- a & b }
R2: {-c <- a & d ; d <- b }
```

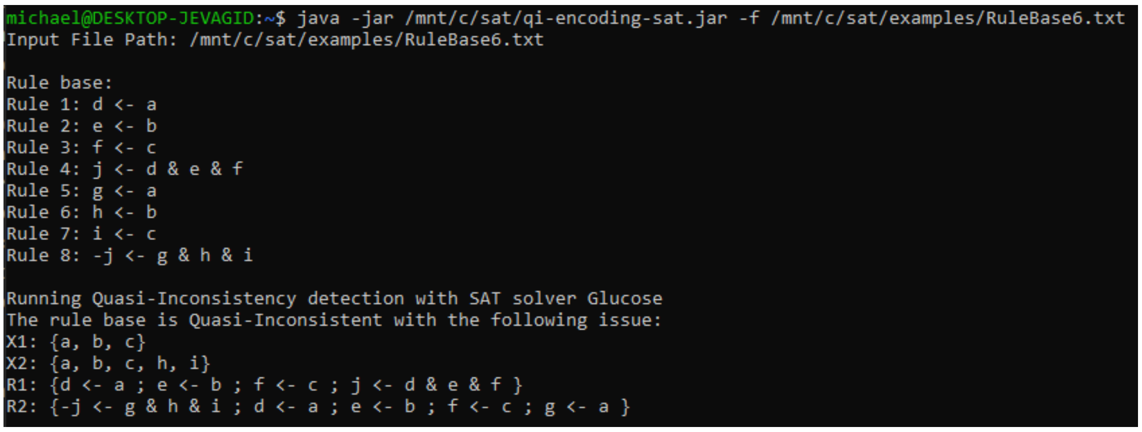
Figure 2: screenshot of the output for example rule base $\mathcal{B}_1$

running the inconsistency detection. The principal output (for all options but DIMACS) either returns an issue if the rule base is Quasi-Inconsistent or returns UNSAT if the rule base is not Quasi-Inconsistent. In Figure 2 we can see the output for our example rule base $\mathcal{B}_1$ that we used in section 3. A further example output for a slightly bigger rule base $\mathcal{B}_6$ can be seen in Figure 3.

4.4 Parameters

A brief overview of possible options can be found in Table 3. Parameters with a single dash "-" (-f, -dimacs, -solver) need to be specified with an additional option, whereas parameters with a double dash "--" (--debug, --timer, --benchmark) do not require further options. All "double dash" options are disabled by default and "single dash" options have a sane default value that they will fall back to. The most important parameter to give to our software is "-f *FileName*", where we need to specify the path to a rule base file as described in subsection 4.2 or to a CSV file generated by *MINERful*. The program will determine the type of file automatically by assuming that if the file has the ending *.csv it is a *MINERful* generated file and for all other endings it will assume a rule base file in text format. If this parameter is not specified or the given file path does not exist, it will default to "-f examples/RuleBase1.txt", which is a part of the GitHub Repository and contains our sample rule base $\mathcal{B}_1$ as shown in Figure 1.

With "-solver *SolverName*" we can choose which of the SAT solvers that are integrated



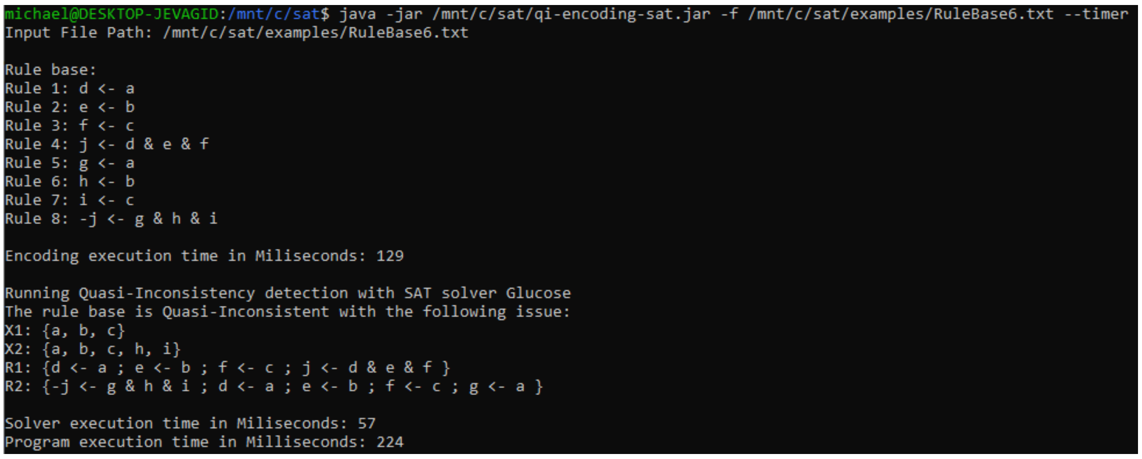
```
michael@DESKTOP-JEVAGID:~$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/RuleBase6.txt
Input File Path: /mnt/c/sat/examples/RuleBase6.txt

Rule base:
Rule 1: d <- a
Rule 2: e <- b
Rule 3: f <- c
Rule 4: j <- d & e & f
Rule 5: g <- a
Rule 6: h <- b
Rule 7: i <- c
Rule 8: -j <- g & h & i

Running Quasi-Inconsistency detection with SAT solver Glucose
The rule base is Quasi-Inconsistent with the following issue:
X1: {a, b, c}
X2: {a, b, c, h, i}
R1: {d <- a ; e <- b ; f <- c ; j <- d & e & f }
R2: {-j <- g & h & i ; d <- a ; e <- b ; f <- c ; g <- a }
```

Figure 3: screenshot of the output for example rule base $\mathcal{B}_6$

within *LogicNG* we want to use. The possible options for *SolverName* are “minisat”, “minicard” and “glucose”. If no solver is specified or *SolverName* does not match any of the options mentioned above the SAT solver *Glucose* is used by default.

If we want to test different solvers that are not integrated into *LogicNG* we also have the option to output our SAT encoding to a text file in DIMACS file format, by adding the parameter “-dimacs *FilePath*”. Of course, the user executing the program will have to have write access to the location specified, otherwise the software will throw an exception. DIMACS is the standard input for basically every SAT solver [BHM21]. This text file then can be used to call any preferred SAT solver, which might be useful to compare how different SAT solvers perform for our type of problem. When running with this option the program will not run any integrated SAT solver, but generate two output files: “*Filename*.cnf” which will contain the DIMACS encoding as well as “*Filename*.map” which includes the mapping from our encoding to the DIMACS encoding. We need this *.map file since in DIMACS format we only enumerate all variables starting with 1 and lose all further information (which in our case we need to show issues). Without this *.map file we would only be able to determine whether our rule base is Quasi-Inconsistent, but not show the issue(s). As mentioned before the “-dimacs” option will not run any SAT solvers by itself and thus supersedes or ignores all other commands but “-f *FileName*”. If the parameter “-dimacs” is run without any options it will default to placing the files “dimacs.cnf” and “dimacs.map” into the current working directory (if possible).

The option “--debug” enables additional output that can help us debug the program and detect problems. The standard output as described in subsection 4.3 either prints out an Issue to the console if the rule base is Quasi-Inconsistent or tells us that the rule base is not Quasi-Inconsistent. With debug mode enabled we get two major additional data points. Firstly it will print out the generated SAT encoding to the console before it is run through the solver. This can be helpful to be able to determine whether the implementation is working as intended and whether the conversion to SAT is correct.



```
michael@DESKTOP-JEVAGID:/mnt/c/sat$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/RuleBase6.txt --timer
Input File Path: /mnt/c/sat/examples/RuleBase6.txt

Rule base:
Rule 1: d <- a
Rule 2: e <- b
Rule 3: f <- c
Rule 4: j <- d & e & f
Rule 5: g <- a
Rule 6: h <- b
Rule 7: i <- c
Rule 8: -j <- g & h & i

Encoding execution time in Miliseconds: 129

Running Quasi-Inconsistency detection with SAT solver Glucose
The rule base is Quasi-Inconsistent with the following issue:
X1: {a, b, c}
X2: {a, b, c, h, i}
R1: {d <- a ; e <- b ; f <- c ; j <- d & e & f }
R2: {-j <- g & h & i ; d <- a ; e <- b ; f <- c ; g <- a }

Solver execution time in Miliseconds: 57
Program execution time in Milliseconds: 224
```

Figure 4: output for $\mathcal{B}_6$ with execution times

Secondly, if an issue is found it will print out the whole satisfying assignment of our SAT formula and not just the parts we need to display issues. Specifically, this means all additional generated variables, e.g. activation variables. An example output for this compared to the standard output can be seen in Figure 9.

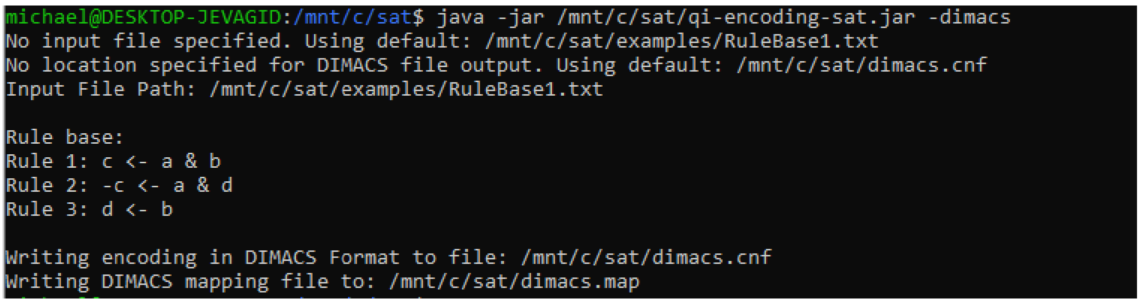
By adding "--timer" to our run parameters we will get different run time printouts in milliseconds to the console. We will be shown the overall program run time, the run time of the SAT solver (including the conversion to CNF) as well as the program execution time for the pre-processing tasks (reading the input, encoding the problem to SAT). This will be necessary to be able to judge how efficient our SAT encoding-based approach is going to be compared to other methods that will be discussed later on in section 5 and to get a general understanding of how fast it actually is.

The option "--benchmark" extends the timer option by already running the alternative approaches described in section 5 and printing out their respective run times. This way we can directly compare the different methods and evaluate how much their efficiency differs. Figure 7 and Figure 6 show how sample outputs of this would look.

None of these parameters are necessary to run the program. If wrong parameters or options are entered these will be just skipped, an error message will be printed out and the software will run with default options (see Figure 5 and Figure 8).

5 Evaluation

In this section, we will look into how our SAT implementation of Quasi-Inconsistency detection performs compared to a brute force method. As a data base, we will look at real-world data on the one hand, as well as "easy" artificial rule bases from various papers.
The benchmarks will be performed on a reasonably new Desktop PC with the following relevant specifications:



```
michael@DESKTOP-JEVAGID:/mnt/c/sat$ java -jar /mnt/c/sat/qi-encoding-sat.jar -dimacs
No input file specified. Using default: /mnt/c/sat/examples/RuleBase1.txt
No location specified for DIMACS file output. Using default: /mnt/c/sat/dimacs.cnf
Input File Path: /mnt/c/sat/examples/RuleBase1.txt

Rule base:
Rule 1: c <- a & b
Rule 2: -c <- a & d
Rule 3: d <- b

Writing encoding in DIMACS Format to file: /mnt/c/sat/dimacs.cnf
Writing DIMACS mapping file to: /mnt/c/sat/dimacs.map
```

Figure 5: -dimacs parameter output with default options

parameter	option	default value
-f *FileName*	Path to File	examples/Rulebase1.txt
-dimacs *FileName*	Path to File	*CurrentDirectory/dimacs*
-solver *SolverName*	glucose,minisat,minicard	glucose
--debug	*none*	off
--timer	*none*	off
--benchmark	*none*	off

Table 3: parameter options for the program

- CPU: AMD Ryzen 7 3700X 8-Core Processor 8 x 3,6 GHz (16 Threads)
- RAM: 32GB DDR4 2133 MHz
- Operating System: Windows 10 21H2 (OS Version: 10.0.19045.2311)

The SAT solver will only be able to run on one core of the system, whereas the brute force approach is going to be parallelized to run on all cores. We will use the LogicNG implementations of Minisat and Glucose as our SAT solvers. The program will be run for every rule base with the parameter *--benchmark* to get the run times of the different implementations as a console printout.

5.1 Brute Force Algorithm

We need another approach with which we can compare the performance results of our SAT-solving method. Therefore we are going to implement a very basic and easy brute force algorithm that checks for Quasi-Inconsistency. The basic approach for this algorithm will be to check all possible combinations of facts and rules whether they contain an issue. For this, we first analyze our rule base and extract all literals into a set that contains all possible literals. The power set of this set includes all possible sets of facts (X_1, X_2) that we have to check for issues. The same is obviously also true for all rules, so we need the power set of all rules that are part of the rule base. To get these power sets we use the efficient power set implementation of the *Google Guava Library*. [3]. We need to check each possible of these combinations if it is an issue. For this, we need our definition from subsection 2.3 again. The five requirements for an issue will be checked in the following way:

1. $X_1 \subseteq X_2$: For every literal in X_1 whether it is also contained in X_2, if we find one literal that is in X_1 but not in X_2 we can go to the next combination and if all literals are in X_2 we go to the next step.

2. X_1 ivates R_1: First we check if X_1 activates R_1. To do this we create copies of our sets X_1 and R_1. We run a loop over all rules in R_1 to check if any rule is activated by $X_1(copy)$. If a rule is activated we add the rule head to $X_1(copy)$ and remove this rule from $R_1(copy)$. The loop continues either until $R_1(copy)$ is the empty set in which case X_1 activates R_1 or until we do not add any more items to $X_1(copy)$ in a loop run, which means that X_1 does not activate R_1. We can also use this function to check for minimal activation. If we get the result that X_1 activates R_1 we check for every subset of X_1 with a cardinality that is exactly one less than X_1 if it activates R_1. If any of these sets activates R_1, then X_1 does not ivate R_1. If none of the sets activate R_1, we have proven that X_1 ivates R_1.

3. X_2 ivates R_2: We use the same method as described above for X_1 and R_1

4. R_1 is X_1 consistent and R_2 is X_2 consistent: We do this check by using the activation function described above. To check if R_1 is X_1 consistent we just need to check if $X_1(copy)$ is consistent in the last run of the loop. If $X_1(copy)$ is consistent to itself then so is R_1 wrt. X_1. The consistency check itself is done by checking that no literal has its opposite within the set.

5. $R_1 \cup R_2$ is X_2 inconsistent: For this, we also use the activation algorithm from above. Unlike with condition 4 however, we are now looking for inconsistency and not the absence thereof. We do this by calling the activation function with X_2 as the fact base and $R_1 \cup R_2$ as the rule base. We go through the activation loop until we can find no more possible activations and then check $X_2(copy)$ for inconsistencies. If we find an inconsistency then is requirement is satisfied otherwise not.

These points will be checked in order and if one of these requirements fails to be met we skip to the next combination that we need to check. In our implementation, the parallelized version of the iteration over all combinations is executed in the class *bruteForceLoop* in stacked for loops (1) while the checking will be done by the class *checkBruteForce*.

[3]https://github.com/google/guava

Algorithm 1 Brute Force Loop Algorithm

```
1: procedure BRUTEFORCELOOP(Rule Base)
2:     X <- PowerSet(allPossibleFacts(RuleBase))
3:     R <- PowerSet(allRules(RuleBase))
4:     for x1 : X do
5:         for x2 : X do
6:             for r1 : R do
7:                 for r2 : R do
8:                     checkBruteForce(x1,x2,r1,r2)
```

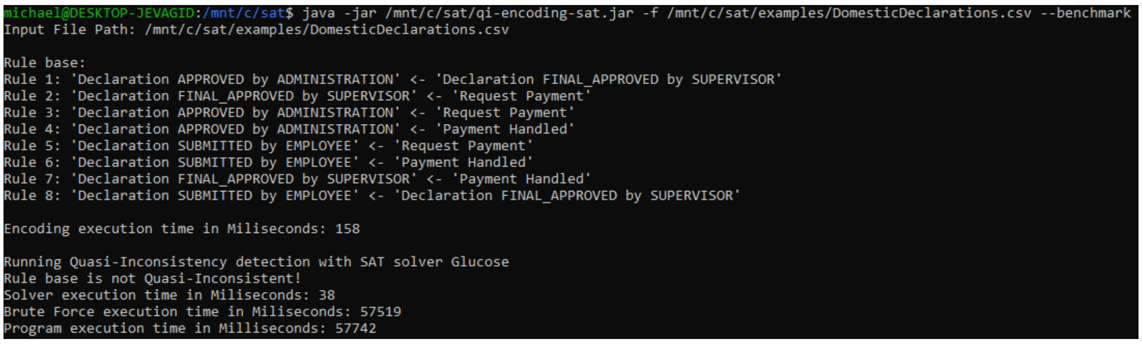
5.2 Data

We mentioned before that we will use two types of data that we will try to benchmark our application on: real-world data and “easy” artificial data from different papers. In this section, we are going to explain how we get to this data base. All examples that we will run the benchmark on are included in the Project's GitHub repository [4] in the folder *examples*. The artificial rule bases are named “RuleBaseX.txt” where X is the corresponding number in $\mathcal{B}_x$, whereas the real-world data is named after its source file from the BPI Challenge, e.g. “DomesticDeclarations.csv”.

5.2.1 Mining Real-World Data

Inspired by the process of using real-world data to analyze inconsistencies described in [CTD21], we will try to use similar transaction data sets to generate a business rule base, which we can later analyze. As our transaction input data, we will use different data sets from various years of the *Business Process Intelligence (BPI) Challenge*. Our main analysis will be regarding the BPI 2020 Challenge, which is based on travel and reimbursement logs from the Eindhoven University of Technology [vDon20]. Additionally, we will also briefly look at the data sets for the BPI Challenge 2013 [Ste14], BPI Challenge 2015 [vDon15] and BPI Challenge 2016 [DvD16].
We will mine the transaction data with the tool *MINERful* [CM15]. This tool will analyze the transaction from *.xes files that are part of the BPI challenges and gives us output rules for the formalism *Declare* [DM22]. This is a formalism, however, which is more complex than the logic programs we are looking at in our framework. Among other things Declare takes timing into account in most templates, which we cannot reasonably represent with our encoding. All possible rule types for Declare are shown in Figure 10. By looking at this table, we can see that we can only reasonably transfer a small subset of templates into our encoding. We have chosen to include the templates “Response” and “NotResponse” since they are the most straightforward but should still give us a reasonable rule base. A brief overview of how *MINERful* works can be found on its GitHub Wiki[5] with a more detailed explanation in the corresponding pa-

[4]https://github.com/tehmischi/qi-encoding-sat
[5]https://github.com/cdc08x/MINERful/wiki



```
michael@DESKTOP-JEVAGID:/mnt/c/sat$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/DomesticDeclarations.csv --benchmark
Input File Path: /mnt/c/sat/examples/DomesticDeclarations.csv

Rule base:
Rule 1: 'Declaration APPROVED by ADMINISTRATION' <- 'Declaration FINAL_APPROVED by SUPERVISOR'
Rule 2: 'Declaration FINAL_APPROVED by SUPERVISOR' <- 'Request Payment'
Rule 3: 'Declaration APPROVED by ADMINISTRATION' <- 'Request Payment'
Rule 4: 'Declaration APPROVED by ADMINISTRATION' <- 'Payment Handled'
Rule 5: 'Declaration SUBMITTED by EMPLOYEE' <- 'Request Payment'
Rule 6: 'Declaration SUBMITTED by EMPLOYEE' <- 'Payment Handled'
Rule 7: 'Declaration FINAL_APPROVED by SUPERVISOR' <- 'Payment Handled'
Rule 8: 'Declaration SUBMITTED by EMPLOYEE' <- 'Declaration FINAL_APPROVED by SUPERVISOR'

Encoding execution time in Miliseconds: 158

Running Quasi-Inconsistency detection with SAT solver Glucose
Rule base is not Quasi-Inconsistent!
Solver execution time in Miliseconds: 38
Brute Force execution time in Miliseconds: 57519
Program execution time in Milliseconds: 57742
```

Figure 6: BPI2020 - *DomesticDeclarations* output example

per [CM15]. As we do not want to go into much detail here, we basically have the three relevant parameters *Support*, *Confidence Level* and *Interest Factor*, that we need to specify when running *MINERful*. Summarized in a simplified manner this means that a combination of these values determines how likely it is that a rule is relevant for us and correct. The formal definitions and a more extensive explanation can be found in [CM15].

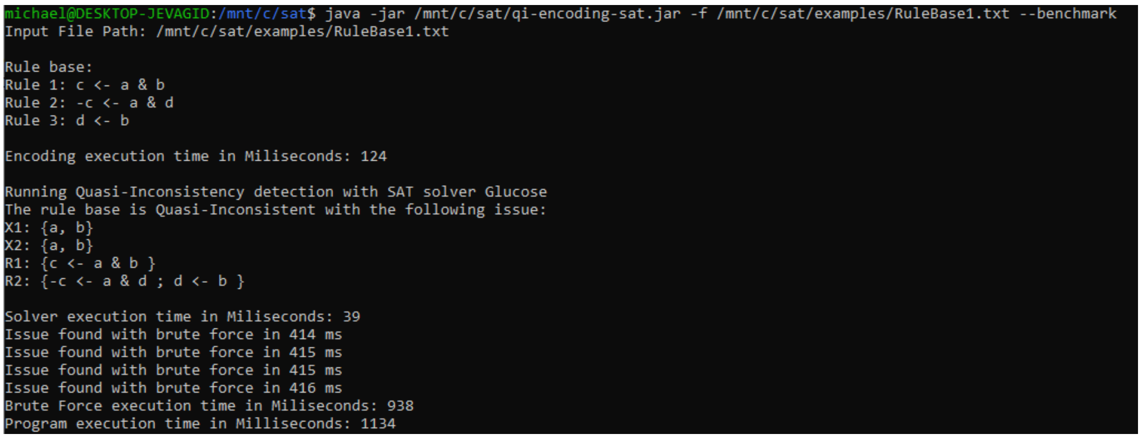
5.2.2 Artificial Data

We will use the following artificial rule bases:

$$
\begin{aligned}
\mathcal{B}_1 =& \{c \leftarrow a, b\ ;\ \neg c \leftarrow a, d\ ;\ d \leftarrow b\} \\
\mathcal{B}_2 =& \{b \leftarrow a\ ;\ c \leftarrow b\ ;\ \neg c \leftarrow a\} \\
\mathcal{B}_3 =& \{e \leftarrow a\ ;\ \neg e \leftarrow c\ ;\ c \leftarrow a\ ;\ e \leftarrow b\} \\
\mathcal{B}_4 =& \{c \leftarrow a\ ;\ \neg c \leftarrow a, b\} \\
\mathcal{B}_5 =& \{c \leftarrow a, f\ ;\ \neg c \leftarrow h, d\ ;\ d \leftarrow b\ ;\ f \leftarrow b\ ;\ h \leftarrow a\} \\
\mathcal{B}_6 =& \{d \leftarrow a\ ;\ e \leftarrow b\ ;\ f \leftarrow c\ ;\ j \leftarrow d, e, f\ ; \\
& g \leftarrow a\ ;\ h \leftarrow b\ ;\ i \leftarrow c\ ;\ \neg j \leftarrow g, h, i\}
\end{aligned}
$$

5.3 Results

First, we will look at the results for our artificial rule bases to get an initial impression. Then we will see how our approach can handle real-world data. The shown results in Table 4 were achieved with the *Glucose* as the default solver for our program. The other solvers (MiniSat and MiniCard) were even about 5-10 % faster for all rule bases on average. As can be seen in Table 4 the brute force approach runs into its limits even with relatively small rule bases like $\mathcal{B}_6$, where finding an issue already takes longer than 40 minutes compared to the 64ms of our SAT solving. The number of variations that have to be tried in the brute force increases exponentially with the input variables



```
michael@DESKTOP-JEVAGID:/mnt/c/sat$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/RuleBase1.txt --benchmark
Input File Path: /mnt/c/sat/examples/RuleBase1.txt

Rule base:
Rule 1: c <- a & b
Rule 2: -c <- a & d
Rule 3: d <- b

Encoding execution time in Miliseconds: 124

Running Quasi-Inconsistency detection with SAT solver Glucose
The rule base is Quasi-Inconsistent with the following issue:
X1: {a, b}
X2: {a, b}
R1: {c <- a & b }
R2: {-c <- a & d ; d <- b }

Solver execution time in Miliseconds: 39
Issue found with brute force in 414 ms
Issue found with brute force in 415 ms
Issue found with brute force in 415 ms
Issue found with brute force in 416 ms
Brute Force execution time in Miliseconds: 938
Program execution time in Milliseconds: 1134
```

Figure 7: Benchmark output for $\mathcal{B}_1$

Rule Base	Result	SAT	Brute Force	BF full
$\mathcal{B}_1$	Quasi-Inconsistent	42ms	442ms	996ms
$\mathcal{B}_2$	Quasi-Inconsistent	40ms	133ms	266ms
$\mathcal{B}_3$	Quasi-Inconsistent	41ms	694ms	2211ms
$\mathcal{B}_4$	Quasi-Inconsistent	39ms	77ms	139ms
$\mathcal{B}_5$	Quasi-Inconsistent	50ms	10435ms	76772ms
$\mathcal{B}_6$	Quasi-Inconsistent	64ms	2478851ms	> 2 days

Table 4: Results for artificial rule bases $\mathcal{B}_1$ - $\mathcal{B}_6$

Rule Base	#Rules	#Literals	Overall Combinations
$\mathcal{B}_1$	3	5	65,536
$\mathcal{B}_2$	3	4	16,384
$\mathcal{B}_3$	4	5	262,144
$\mathcal{B}_4$	2	4	4,096
$\mathcal{B}_5$	5	6	4,194,304
$\mathcal{B}_6$	8	11	274,877,906,944

Table 5: Possible Combinations for Brute Force

Rule Base	Result	SAT	Brute Force	BF full
$DomesticDeclarations$	Not Quasi-Inconsistent	42ms	60687ms	60687ms
$InternationalDeclarations$	Not Quasi-Inconsistent	51ms	> 1 day	> 1 day
$PermitLogs$	Not Quasi-Inconsistent	-	-	-
$PrepaidTravelCost$	Not Quasi-Inconsistent	41ms	37697ms	37697ms
$RequestForPayment$	Not Quasi-Inconsistent	40ms	1422ms	1422ms

Table 6: BPI 2020 - support 0.95, confidence 0.5, interest 0.125

and rules. As we have two power sets each for variables and rules which themselves have a cardinality of 2^n the number of possible combinations is:

$$\#Combinations = 4^{\#Rules} * 4^{\#Literals}$$

Table 5 shows the number of possible variations for our rule bases to get a better impression of how massive the growth actually is. As input size increases exponentially so does the required computation time, although slightly slower. However, this still makes the brute force approach unfeasible for almost any practical application. As a next step, we try out our benchmark with all data sets from the BPI 2020 Challenge. To mine these rules we used the options support 0.95, confidence 0.5 and interest 0.125 since these were the general recommended values from the authors. Unfortunately, we were not able to find Quasi-Inconsistencies for these data sets as can be seen in Table 6. The brute force approach for the data set *InternationalDeclarations* was canceled after a run time of about 24 hours. Roughly extrapolating from the results in Table 4 (where the growth in the run time of full brute force check increases almost completely as fast as exponential input size) and estimating this conservatively we expect it would take at least 2^{10} (exponential growth would be about 2^{16}) times longer than *Domestic Declarations*. This would amount to more than 40 days to finish. We have seen that we are already running into the limits of our brute force approach, but we still want to look at more data - this time just with our SAT implementation. As we want to challenge our program a bit more, we are now using the biggest data set available to us (BPI 2015) and mine it with lower thresholds to get even more rules. As we can see in Table 7 even

Rule Base	Result	Solving Time	#Rules
BPIC15_1	Quasi-Inconsistent	281ms	150
BPIC15_2	Quasi-Inconsistent	534ms	285
BPIC15_3	Quasi-Inconsistent	393ms	220
BPIC15_4	Quasi-Inconsistent	297ms	190
BPIC15_5	Quasi-Inconsistent	463ms	213

Table 7: BPI 2015 - support 0.5, confidence 0.05, interest 0.02

Rule Base	Result	SAT	Brute Force	BF full
$DomesticDeclarations$	Quasi-Inconsistent	50ms	6344ms	68312
$InternationalDeclarations$	Quasi-Inconsistent	63ms	-	> 1 day
$PermitLogs$	Not Quasi-Inconsistent	-	-	-
$PrepaidTravelCost$	Quasi-Inconsistent	41ms	484ms	36877ms
$RequestForPayment$	Quasi-Inconsistent	45ms	283ms	4740ms

Table 8: BPI 2020 - modified data

with 200 rules our program still finishes the check in a reasonable time frame. However, we still have not found any Quasi-Inconsistencies in our real-world data, which should really be there, as we have lots of rules and have chosen very low limits for rule inclusion. Looking at the base data (also located in the examples folder in the repository) we actually notice, that just with mining Declare BPI data with MINERful and Templates *Response* and *NotResponse* we will never get a Quasi-Inconsistency since either MINERful or the BPI data never returns *NotResponse* or negated rule heads at all. As it is a necessary requirement for Quasi-Inconsistency to have opposing rule heads, we will never be able to find any Quasi-Inconsistency this way! This is true for all other BPI data sets that were mentioned before as well, so will not look at them further, since they provide similar results.

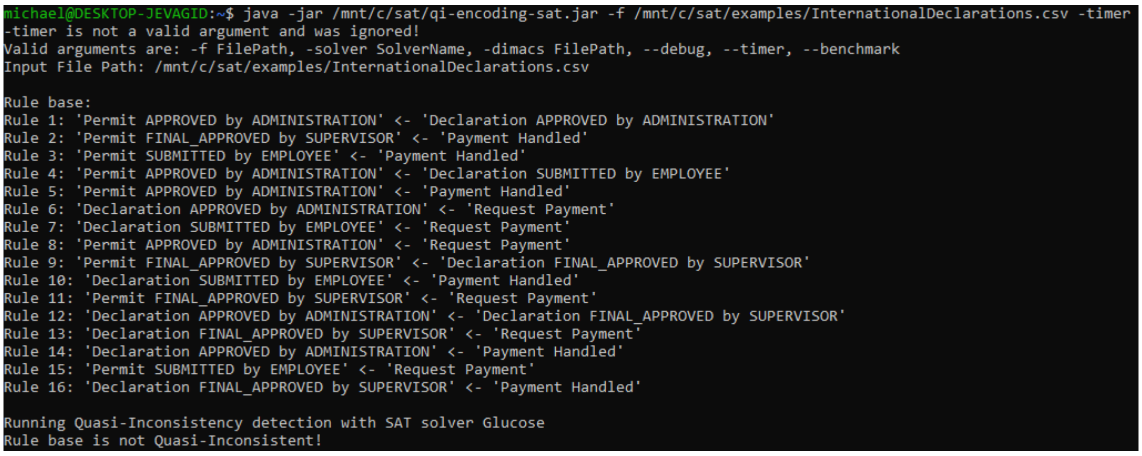
Due to these results and our realization that we will not find Quasi-Inconsistencies with these data sets and mining techniques, we tried to manipulate the data, so that we construct a (false) inconsistent rule base from our real data so that we can test our algorithm. So far with the real data, we have only seen how fast the program will determine that a rule base is consistent wrt. Quasi-Inconsistency. However, we also want to know how fast actual issues can be found. We use "wrong" manipulated data for this, as a more complex approach (or a semantic analysis of the rules) would be out of scope for this paper. To do this we randomized the handling of the Declare templates *Response* and *NotResponse* in our Declare parser. This means that for all rule heads it is now randomly chosen if they are negated or not. Like this we can at least get an idea, of how fast the SAT-solving techniques finds issues compared to the brute force method. The results of one run of these tests are on display in Table 8. Due to the randomness of the rule selection, these are obviously not exactly reproducible each time, but the overview just serves to give a general idea, while all of this should be taken with a grain of salt. As a final test of our SAT implementation, we used the largest single data set *BPIC15_2* with support 0.1, confidence 0.01 and interest 0.01 to get a rule base of 615 rules. Our algorithm was able to detect the consistency of this rule base in only 961ms.

5.4 Discussion

As we have seen in subsection 5.3 the idea to encode *DEC-QI* into SAT has been really quite successful. It delivers a significantly improved performance compared to traditional approaches and is able to solve the problem for rule bases that would be out of reach for standard approaches due to the exponential increase of input size and parameters. Our brute force approach ran into its limit really early even with very easy artificial rule bases. Unfortunately for the practical applications, we were only able to analyze a very limited amount of subset of templates for the formalism *DECLARE*. This led to the problems we described that we were not able to find any Quasi-Inconsistencies in our real-world data. Also, this type of problem made other, better approaches than brute force not meaningful to even try. For example, the improved graph-based algorithm in [Cor20] checks whether opposite literals exist in rule heads in its very first step, which is exactly the problem with our data, so it would immediately terminate in each case. Still, despite these problems, the results from just looking at the run times of the SAT solving part seem pretty impressive.

6 Outlook

In this thesis, we have established that a SAT-based approach to inconsistency detection is possible and at least a viable alternative to more traditional ways of inconsistency analysis. However, we were only able to test the implementation on very limited data sets that were not optimal for our comparison. Therefore there are still a lot of open questions that need to be answered. The first and most obvious step for future work to build on this paper in our opinion would be to try to extract more meaningful real-world data to benchmark our implementation against. A good comparison algorithm would be the already mentioned graph algorithm from [Cor20]. Multiple ways come to mind, how "better" data could be found. One way would certainly be to try to include the other non-temporal templates from Declare in the parser. Another idea could be to semantically analyze the data generated by MINERful since some inconsistencies could be found that way, e.g. by transforming facts that mention "DENIED" or "REJECTED" as found in BPI 2020 to $\neg$ "APPROVED". What could be also tried is just looking for other ways to parse this or other data to get a rule formalism other than Declare that is more similar to our logic program based rule format. Furthermore, it might be possible to include some of the easier temporal templates into our encoding by modifying it. However, at first glance, this looks much harder than just using different data. Once reasonably complex and large real data sets have been found one could also try to use



```
michael@DESKTOP-JEVAGID:~$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/InternationalDeclarations.csv -timer
-timer is not a valid argument and was ignored!
Valid arguments are: -f FilePath, -solver SolverName, -dimacs FilePath, --debug, --timer, --benchmark
Input File Path: /mnt/c/sat/examples/InternationalDeclarations.csv

Rule base:
Rule 1: 'Permit APPROVED by ADMINISTRATION' <- 'Declaration APPROVED by ADMINISTRATION'
Rule 2: 'Permit FINAL_APPROVED by SUPERVISOR' <- 'Payment Handled'
Rule 3: 'Permit SUBMITTED by EMPLOYEE' <- 'Payment Handled'
Rule 4: 'Permit APPROVED by ADMINISTRATION' <- 'Declaration SUBMITTED by EMPLOYEE'
Rule 5: 'Permit APPROVED by ADMINISTRATION' <- 'Payment Handled'
Rule 6: 'Declaration APPROVED by ADMINISTRATION' <- 'Request Payment'
Rule 7: 'Declaration SUBMITTED by EMPLOYEE' <- 'Request Payment'
Rule 8: 'Permit APPROVED by ADMINISTRATION' <- 'Request Payment'
Rule 9: 'Permit FINAL_APPROVED by SUPERVISOR' <- 'Declaration FINAL_APPROVED by SUPERVISOR'
Rule 10: 'Declaration SUBMITTED by EMPLOYEE' <- 'Payment Handled'
Rule 11: 'Permit FINAL_APPROVED by SUPERVISOR' <- 'Request Payment'
Rule 12: 'Declaration APPROVED by ADMINISTRATION' <- 'Declaration FINAL_APPROVED by SUPERVISOR'
Rule 13: 'Declaration FINAL_APPROVED by SUPERVISOR' <- 'Request Payment'
Rule 14: 'Declaration APPROVED by ADMINISTRATION' <- 'Payment Handled'
Rule 15: 'Permit SUBMITTED by EMPLOYEE' <- 'Request Payment'
Rule 16: 'Declaration FINAL_APPROVED by SUPERVISOR' <- 'Payment Handled'

Running Quasi-Inconsistency detection with SAT solver Glucose
Rule base is not Quasi-Inconsistent!
```

Figure 8: screenshot of the output of mined BPI2020 data

our program to benchmark different SAT solvers with this real-world data as most SAT benchmarks are synthetic ([BHM21]).

Main Sources

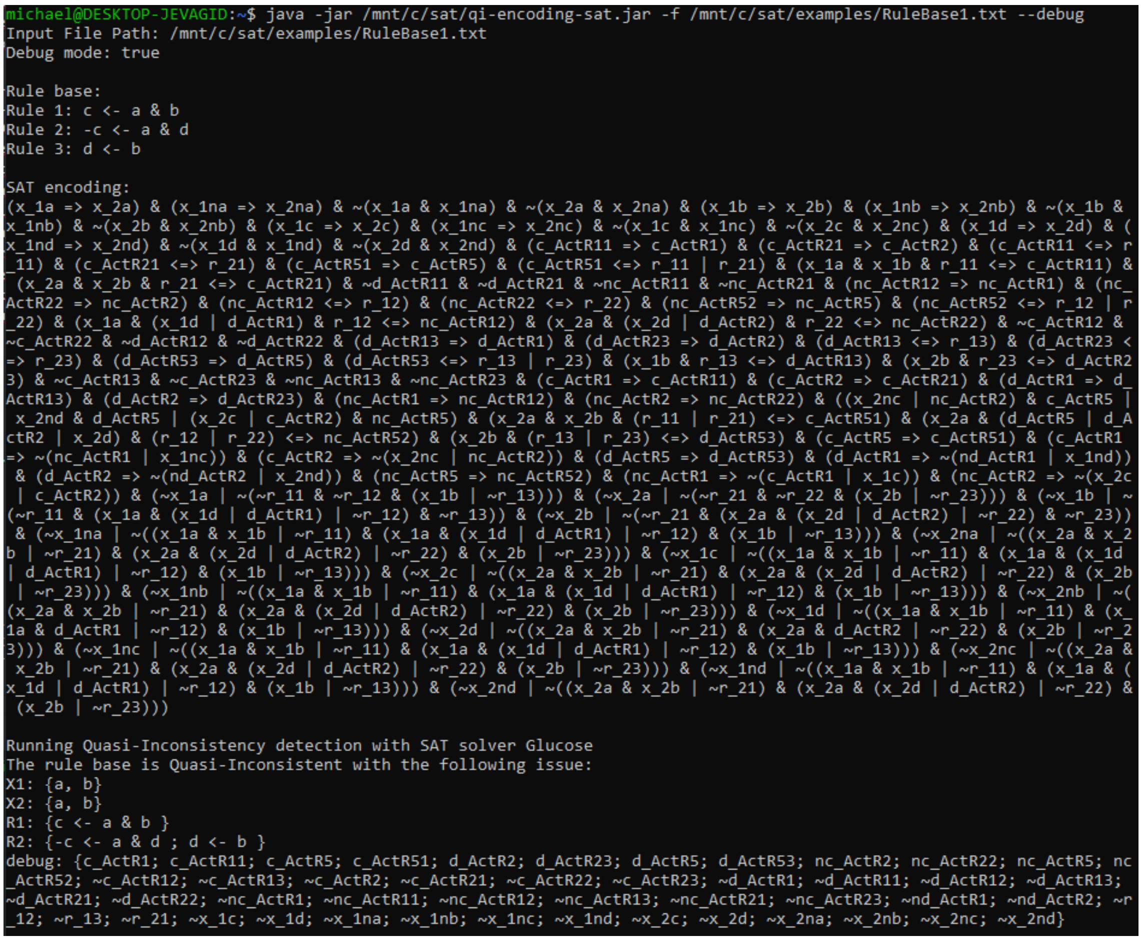
[BHM21] Armin Biere, Marijn Heule, and Hans van Maaren, eds. *Handbook of Satisfiability*. Second edition. Frontiers in Artificial Intelligence and Applications volume 336. Amsterdam ; Washington, DC: IOS Press, 2021. 1465 pp. ISBN: 978-1-64368-160-3.

[Bjö09] Magnus Björk. "Successful SAT Encoding Techniques". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.4 (July 1, 2009), pp. 189–201. ISSN: 15740617. DOI: 10.3233/SAT190085. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SAT190085 (visited on 08/08/2022).

[Cai+19] Jiatong Cai et al. "A New SAT Encoding Scheme for Exactly-one Constraints". In: *Journal of Physics: Conference Series* 1288.1 (Aug. 1, 2019), p. 012035. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/1288/1/012035. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1288/1/012035.

[CD18] Carl Corea and Patrick Delfmann. "Supporting Business Rule Management with Inconsistency Analysis". In: Sept. 9, 2018.

[CD19] Carl Corea and Patrick Delfmann. "Quasi-Inconsistency in Declarative Process Models". In: July 1, 2019, pp. 20–35. ISBN: 978-3-030-26642-4. DOI: 10.1007/978-3-030-26643-1_2.



```
michael@DESKTOP-JEVAGID:~$ java -jar /mnt/c/sat/qi-encoding-sat.jar -f /mnt/c/sat/examples/RuleBase1.txt --debug
Input File Path: /mnt/c/sat/examples/RuleBase1.txt
Debug mode: true

Rule base:
Rule 1: c <- a & b
Rule 2: -c <- a & d
Rule 3: d <- b

SAT encoding:
(x_1a => x_2a) & (x_1na => x_2na) & ~(x_1a & x_1na) & ~(x_2a & x_2na) & (x_1b => x_2b) & (x_1nb => x_2nb) & ~(x_1b &
x_1nb) & ~(x_2b & x_2nb) & (x_1c => x_2c) & (x_1nc => x_2nc) & ~(x_1c & x_1nc) & ~(x_2c & x_2nc) & (x_1d => x_2d) & (
x_1nd => x_2nd) & ~(x_1d & x_1nd) & ~(x_2d & x_2nd) & (c_ActR11 => c_ActR1) & (c_ActR21 => c_ActR2) & (c_ActR11 <=> r
_11) & (c_ActR21 <=> r_21) & (c_ActR51 => c_ActR5) & (c_ActR51 <=> r_11 | r_21) & (x_1a & x_1b & r_11 <=> c_ActR11) &
 (x_2a & x_2b & r_21 <=> c_ActR21) & ~d_ActR11 & ~d_ActR21 & ~nc_ActR11 & ~nc_ActR21 & (nc_ActR12 => nc_ActR1) & (nc_
ActR22 => nc_ActR2) & (nc_ActR12 <=> r_12) & (nc_ActR22 <=> r_22) & (nc_ActR52 => nc_ActR5) & (nc_ActR52 <=> r_12 | r
_22) & (x_1a & (x_1d | d_ActR1) & r_12 <=> nc_ActR12) & (x_2a & (x_2d | d_ActR2) & r_22 <=> nc_ActR22) & ~c_ActR12 &
~c_ActR22 & ~d_ActR12 & ~d_ActR22 & (d_ActR13 => d_ActR1) & (d_ActR23 => d_ActR2) & (d_ActR13 <=> r_13) & (d_ActR23 <
=> r_23) & (d_ActR53 => d_ActR5) & (d_ActR53 <=> r_13 | r_23) & (x_1b & r_13 <=> d_ActR13) & (x_2b & r_23 <=> d_ActR2
3) & ~c_ActR13 & ~c_ActR23 & ~nc_ActR13 & ~nc_ActR23 & (c_ActR1 => c_ActR11) & (c_ActR2 => c_ActR21) & (d_ActR1 => d_
ActR13) & (d_ActR2 => d_ActR23) & (nc_ActR1 => nc_ActR12) & (nc_ActR2 => nc_ActR22) & ((x_2nc | nc_ActR2) & c_ActR5 |
 x_2nd & d_ActR5 | (x_2c | c_ActR2) & nc_ActR5) & (x_2a & x_2b & (r_11 | r_21) <=> c_ActR51) & (x_2a & (d_ActR5 | d_A
ctR2 | x_2d) & (r_12 | r_22) <=> nc_ActR52) & (x_2b & (r_13 | r_23) <=> d_ActR53) & (c_ActR5 => c_ActR51) & (c_ActR1
=> ~(nc_ActR1 | x_1nc)) & (c_ActR2 => ~(x_2nc | nc_ActR2)) & (d_ActR5 => d_ActR53) & (d_ActR1 => ~(nd_ActR1 | x_1nd))
 & (d_ActR2 => ~(nd_ActR2 | x_2nd)) & (nc_ActR5 => nc_ActR52) & (nc_ActR1 => ~(c_ActR1 | x_1c)) & (nc_ActR2 => ~(x_2c
 | c_ActR2)) & (~x_1a | ~(~r_11 & ~r_12 & (x_1b | ~r_13))) & (~x_2a | ~(~r_21 & ~r_22 & (x_2b | ~r_23))) & (~x_1b | ~
(~r_11 & (x_1a & (x_1d | d_ActR1) | ~r_12) & ~r_13)) & (~x_2b | ~(~r_21 & (x_2a & (x_2d | d_ActR2) | ~r_22) & ~r_23))
 & (~x_1na | ~((x_1a & x_1b | ~r_11) & (x_1a & (x_1d | d_ActR1) | ~r_12) & (x_1b | ~r_13))) & (~x_2na | ~((x_2a & x_2
b | ~r_21) & (x_2a & (x_2d | d_ActR2) | ~r_22) & (x_2b | ~r_23))) & (~x_1c | ~((x_1a & x_1b | ~r_11) & (x_1a & (x_1d
| d_ActR1) | ~r_12) & (x_1b | ~r_13))) & (~x_2c | ~((x_2a & x_2b | ~r_21) & (x_2a & (x_2d | d_ActR2) | ~r_22) & (x_2b
 | ~r_23))) & (~x_1nb | ~((x_1a & x_1b | ~r_11) & (x_1a & (x_1d | d_ActR1) | ~r_12) & (x_1b | ~r_13))) & (~x_2nb | ~(
(x_2a & x_2b | ~r_21) & (x_2a & (x_2d | d_ActR2) | ~r_22) & (x_2b | ~r_23))) & (~x_1d | ~((x_1a & x_1b | ~r_11) & (x_
1a & d_ActR1 | ~r_12) & (x_1b | ~r_13))) & (~x_2d | ~((x_2a & x_2b | ~r_21) & (x_2a & d_ActR2 | ~r_22) & (x_2b | ~r_2
3))) & (~x_1nc | ~((x_1a & x_1b | ~r_11) & (x_1a & (x_1d | d_ActR1) | ~r_12) & (x_1b | ~r_13))) & (~x_2nc | ~((x_2a &
 x_2b | ~r_21) & (x_2a & (x_2d | d_ActR2) | ~r_22) & (x_2b | ~r_23))) & (~x_1nd | ~((x_1a & x_1b | ~r_11) & (x_1a & (
x_1d | d_ActR1) | ~r_12) & (x_1b | ~r_13))) & (~x_2nd | ~((x_2a & x_2b | ~r_21) & (x_2a & (x_2d | d_ActR2) | ~r_22) &
 (x_2b | ~r_23)))

Running Quasi-Inconsistency detection with SAT solver Glucose
The rule base is Quasi-Inconsistent with the following issue:
X1: {a, b}
X2: {a, b}
R1: {c <- a & b }
R2: {-c <- a & d ; d <- b }
debug: {c_ActR1; c_ActR11; c_ActR5; c_ActR51; d_ActR2; d_ActR23; d_ActR5; d_ActR53; nc_ActR2; nc_ActR22; nc_ActR5; nc
_ActR52; ~c_ActR12; ~c_ActR13; ~c_ActR2; ~c_ActR21; ~c_ActR22; ~c_ActR23; ~d_ActR1; ~d_ActR11; ~d_ActR12; ~d_ActR13;
~d_ActR21; ~d_ActR22; ~nc_ActR1; ~nc_ActR11; ~nc_ActR12; ~nc_ActR13; ~nc_ActR21; ~nc_ActR23; ~nd_ActR1; ~nd_ActR2; ~r
_12; ~r_13; ~r_21; ~x_1c; ~x_1d; ~x_1na; ~x_1nb; ~x_1nc; ~x_1nd; ~x_2c; ~x_2d; ~x_2na; ~x_2nb; ~x_2nc; ~x_2nd}
```

Figure 9: screenshot of a debug output

[CM15] Claudio Di Ciccio and Massimo Mecella. “On the Discovery of Declarative Control Flows for Artful Processes”. In: *ACM Transactions on Management Information Systems* 5.4 (Mar. 21, 2015), pp. 1–37. ISSN: 2158-656X, 2158-6578. DOI: 10.1145/2629447. URL: https://dl.acm.org/doi/10.1145/2629447 (visited on 11/20/2022).

[Cor20] Corea, Carl. “Handling Inconsistency in Business Rule Bases”. 2020. URL: https://kola.opus.hbz-nrw.de/opus45-kola/frontdoor/deliver/index/docId/2145/file/Thesis.pdf (visited on 08/09/2022).

[CT20] Carl Corea and Matthias Thimm. “On Quasi-Inconsistency and Its Complexity”. In: *Artificial Intelligence* 284 (July 1, 2020), p. 103276. ISSN: 0004-3702. DOI: 10.1016/j.artint.2020.103276. URL: https://www.sciencedirect.com/science/article/pii/S0004370220300357 (visited on 08/17/2022).

[CTD21] Carl Corea, Matthias Thimm, and Patrick Delfmann. “Measuring Inconsistency over Sequences of Business Rule Cases”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* 18.1 (Sept. 30, 2021), pp. 656–660. ISSN: 2334-1033. DOI: 10.24963/kr.2021/64. URL: https://proceedings.kr.org/2021/64/ (visited on 08/08/2022).

[Dei19] Matthias Robert Deisen. “(Quasi-)Inconsistency Library for Business Rule Management”. 2019.

[DM22] Claudio Di Ciccio and Marco Montali. “Declarative Process Specifications: Reasoning, Discovery, Monitoring”. In: *Process Mining Handbook*. Ed. by Wil M. P. van der Aalst and Josep Carmona. Vol. 448. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2022, pp. 108–152. ISBN: 978-3-031-08847-6 978-3-031-08848-3. DOI: 10.1007/978-3-031-08848-3_4. URL: https://link.springer.com/10.1007/978-3-031-08848-3_4 (visited on 11/15/2022).

[Gra06] Ian Graham. *Business Rules Management and Service Oriented Architecture: A Pattern Language*. Chichester, England ; Hoboken, NJ: John Wiley, 2006. 274 pp. ISBN: 978-0-470-02721-9.

[Ngu+20] Van-Hau Nguyen et al. “Empirical Study on SAT-Encodings of the At-Most-One Constraint”. In: *The 9th International Conference on Smart Media and Applications*. SMA 2020: The 9th International Conference on Smart Media and Applications. Jeju Republic of Korea: ACM, Sept. 17, 2020, pp. 470–475. ISBN: 978-1-4503-8925-9. DOI: 10.1145/3426020.3426170. URL: https://dl.acm.org/doi/10.1145/3426020.3426170.

[TW19] Matthias Thimm and Johannes P. Wallner. “On the Complexity of Inconsistency Measurement”. In: *Artificial Intelligence* 275 (Oct. 1, 2019), pp. 411–456. ISSN: 0004-3702. DOI: 10.1016/j.artint.2019.07.001. URL:

https://www.sciencedirect.com/science/article/pii/S0004370219301560 (visited on 08/17/2022).

[Zoe14] M.M Zoet. *Methods and Concepts for Business Rules Management*. Utrecht University, 2014. ISBN: 978-90-393-6130-6.

Secondary Sources

[Aud21] Gilles Audemard. “SAT Solver Essentials, SAT Modeling Introduction” (VTSA School - Liege). 2021.

[Bie18] Biere, Armin. “Encoding into SAT” (Manchester). Mar. 7, 2018. URL: http://fmv.jku.at/biere/talks/Biere-SATSMTAR18-talk.pdf (visited on 08/08/2022).

[Bie21] Biere, Armin. “A Personal History of Practical SAT Solving” (UCLA Berkeley). Mar. 24, 2021. URL: https://simons.berkeley.edu/talks/tbd-308 (visited on 08/27/2022).

[Bus03] Business Rules Group. *The Business Rules Manifesto*. 2003. URL: https://www.businessrulesgroup.org/brmanifesto.htm (visited on 08/28/2022).

[Coo71] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. New York, NY, USA: Association for Computing Machinery, May 3, 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047.

[DvD16] Marcus Dees and B.F. (Boudewijn) van Dongen. “BPI Challenge 2016”. Version 1. In: (Apr. 22, 2016). In collab. with UWV. DOI: 10.4121/UUID:360795C8-1DD6-4A5B-A443-185001076EAB. URL: https://data.4tu.nl/collections/_/5065538/1 (visited on 11/12/2022).

[Far19] Farzan, Azadeh. “CSC410 Tutorial: SAT for Problem Solving - Encoding Problems to SAT Problems” (University of Toronto). 2019.

[FBS19] Katalin Fazekas, Armin Biere, and Christoph Scholl. “Incremental Inprocessing in SAT Solving”. In: *Theory and Applications of Satisfiability Testing – SAT 2019*. Ed. by Mikoláš Janota and Inês Lynce. Vol. 11628. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 136–154. ISBN: 978-3-030-24257-2. DOI: 10.1007/978-3-030-24258-9_9. URL: http://link.springer.com/10.1007/978-3-030-24258-9_9 (visited on 08/27/2022).

[Fin15] Finke, Martin. “Equisatisfiable SAT Encodings of Arithmetical Operations”. HTWK Leipzig, 2015. URL: http://www.martin-finke.de/documents/Masterarbeit_bitblast_Finke.pdf (visited on 08/29/2022).

[Gen11] Ian Gent. “Encodings in SAT and Constraints” (University of St Andrews). 2011. URL: https://school.a4cp.org/summer2011/slides/Gent/SAT-CP-encodings.pdf (visited on 08/08/2022).

[Gup18] Gupta, Ashutosh. “Automated Reasoning 2018 Lecture 6: Encoding into SAT Problem” (IITB, India). 2018.

[GXD20] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. “Satune: Synthesizing Efficient SAT Encoders”. In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3428214. URL: https://dl.acm.org/doi/10.1145/3428214 (visited on 08/28/2022).

[IST13] Markus Iser, Carsten Sinz, and Mana Taghdiri. “Minimizing Models for Tseitin-Encoded SAT Instances”. In: *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*. SAT’13. Berlin, Heidelberg: Springer-Verlag, July 8, 2013, pp. 224–232. ISBN: 978-3-642-39070-8. DOI: 10.1007/978-3-642-39071-5_17. URL: https://doi.org/10.1007/978-3-642-39071-5_17 (visited on 08/29/2022).

[Kab20] Kabanets, Valentine. “SAT-Centered Complexity Theory” (UCLA Berkeley). Dec. 22, 2020. URL: https://simons.berkeley.edu/talks/sat-centered-complexity-theory (visited on 08/27/2022).

[Knu15] Donald Ervin Knuth. *Satisfiablility*. The Art of Computer Programming / Donald E. Knuth Volume 4, Fascicle 6. Boston Columbus Indianapolis: Addison-Wesley, 2015. 310 pp. ISBN: 978-0-13-439760-3.

[Lar+14] Frédéric Lardeux et al. “Set Constraint Model and Automated Encoding into SAT: Application to the Social Golfer Problem”. In: *Annals of Operations Research* 235 (June 27, 2014). DOI: 10.1007/s10479-015-1914-5.

[PS15] Tobias Philipp and Peter Steinke. “PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF”. In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 9–16. ISBN: 978-3-319-24318-4. DOI: 10.1007/978-3-319-24318-4_2.

[Ran19] Venkatesh-Prasad Ranganath. *SAT Encoding: Common Constraint Patterns And Their Encodings*. Medium. May 30, 2019. URL: https://rvprasad.medium.com/sat-encoding-common-constraint-patterns-and-their-encodings-1ade449709c1 (visited on 08/27/2022).

[Rod22] Rodriguez-Carbonell, Enric. “Encodings into SAT - Combinatorial Problem Solving (CPS)” (Universitat Politecnica de Catalunya). May 19, 2022. URL: https://www.cs.upc.edu/~erodri/webpage/cps/theory/sat/encodings/slides.pdf (visited on 08/08/2022).

[SB19] Sinz, Carsten and Balyo, Tomas. “Practical SAT Solving” (Karlsruhe Institute of Technology (KIT)). 2019. URL: https://baldur.iti.kit.edu/sat/ (visited on 08/27/2022).

[Ste14] Ward Steeman. "BPI Challenge 2013". Version 1. In: (Apr. 24, 2014). In collab. with Volvo IT. DOI: 10.4121/UUID:A7CE5C55-03A7-4583-B855-98B86E1A2B07. URL: https://data.4tu.nl/collections/_/5065448/1 (visited on 11/12/2022).

[Sum17] Alexander J Summers. "Encoding Problems to SAT" (ETH Zürich). 2017.

[vDon15] B.F. (Boudewijn) van Dongen. "BPI Challenge 2015". Version 1. In: (May 1, 2015). In collab. with Eindhoven University Of Technology. DOI: 10.4121/UUID:31A308EF-C844-48DA-948C-305D167A0EC1. URL: https://data.4tu.nl/collections/_/5065424/1 (visited on 11/12/2022).

[vDon20] Boudewijn van Dongen. "BPI Challenge 2020". Version 1. In: (Mar. 26, 2020). In collab. with Department Of Mathematics Eindhoven University Of Technology and Computer Science. DOI: 10.4121/UUID:52FB97D4-4588-43C9-9D04-3604D4613B51. URL: https://data.4tu.nl/collections/_/5065541/1 (visited on 11/12/2022).

[VG09] Miroslav N. Velev and Ping Gao. "Efficient SAT Techniques for Absolute Encoding of Permutation Problems: Application to Hamiltonian Cycles". In: *In: Proceedings Sara*. 2009.

[Vol] Matthias Volk. "Using SAT Solvers for Industrial Combinatorial Problems". In: (), p. 53.

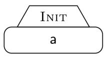
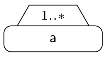
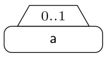
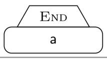
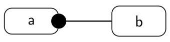
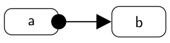
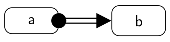
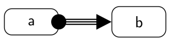
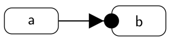
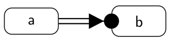
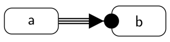
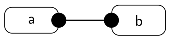
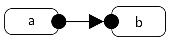
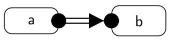
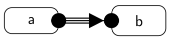
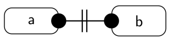
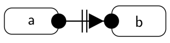
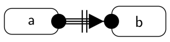
Constraint	**Explanation**	**Examples**				**Notation**
Existence constraints						
INIT(a)	a is the *first* to occur	✓⟨a, c, c⟩	✓⟨a, b, a, c⟩	×⟨c, c⟩	×⟨b, a, c⟩	
ATLEASTONE(a)	a occurs at least *once*	✓⟨b, c, a, c⟩	✓⟨b, c, a, a, c⟩	×⟨b, c, c⟩	×⟨c⟩	
ATMOSTONE(a)	a occurs at most *once*	✓⟨b, c, c⟩	✓⟨b, c, a, c⟩	×⟨b, c, a, a, c⟩	×⟨b, c, a, c, a, a⟩	
END(a)	a is the *last* to occur	✓⟨b, c, a⟩	✓⟨b, a, c, a⟩	×⟨b, c⟩	×⟨b, a, c⟩	
Relation constraints						
RESPONDEDEXISTENCE(a, b)	If a occurs in the trace, then b occurs as well	✓⟨b, c, a, a, c⟩	✓⟨b, c, c⟩	×⟨c, a, a, c⟩	×⟨a, c, c⟩	
RESPONSE(a, b)	If a occurs, then b occurs after a	✓⟨c, a, a, c, b⟩	✓⟨b, c, c⟩	×⟨c, a, a, c⟩	×⟨b, a, c, c⟩	
ALTERNATERESPONSE(a, b)	Each time a occurs, then b occurs afterwards, and no other a recurs in between	✓⟨c, a, c, b⟩	✓⟨a, b, c, a, c, b⟩	×⟨c, a, a, c, b⟩	×⟨b, a, c, a, c, b⟩	
CHAINRESPONSE(a, b)	Each time a occurs, then b occurs immediately afterwards	✓⟨c, a, b, b⟩	✓⟨a, b, c, a, b⟩	×⟨c, a, c, b⟩	×⟨b, c, a⟩	
PRECEDENCE(a, b)	b occurs only if preceded by a	✓⟨c, a, c, b, b⟩	✓⟨a, c, c⟩	×⟨c, c, b, b⟩	×⟨b, a, c, c⟩	
ALTERNATEPRECEDENCE(a, b)	Each time b occurs, it is preceded by a and no other b can recur in between	✓⟨c, a, c, b, a⟩	✓⟨a, b, c, a, a, c, b⟩	×⟨c, a, c, b, b, a⟩	×⟨a, b, b, a, b, c, b⟩	
CHAINPRECEDENCE(a, b)	Each time b occurs, then a occurs immediately beforehand	✓⟨a, b, c, a⟩	✓⟨a, b, a, a, b, c⟩	×⟨b, c, a⟩	×⟨b, a, a, c, b⟩	
Mutual relation constraints						
COEXISTENCE(a, b)	If b occurs, then a occurs, and vice versa	✓⟨c, a, c, b, b⟩	✓⟨b, c, c, a⟩	×⟨c, a, c⟩	×⟨b, c, c⟩	
SUCCESSION(a, b)	a occurs if and only if it is followed by b	✓⟨c, a, c, b, b⟩	✓⟨a, c, c, b⟩	×⟨b, a, c⟩	×⟨b, c, c, a⟩	
ALTERNATESUCCESSION(a, b)	a and b if and only if the latter follows the former, and they alternate each other in the trace	✓⟨c, a, c, b, a, b⟩	✓⟨a, b, c, a, b, c⟩	×⟨c, a, a, c, b, b⟩	×⟨b, a, c⟩	
CHAINSUCCESSION(a, b)	a and b occur if and only if the latter immediately follows the former	✓⟨c, a, b, a, b⟩	✓⟨c, c, c⟩	×⟨c, a, c, b⟩	×⟨c, b, a, c⟩	
Negative relation constraints						
NOTCOEXISTENCE(a, b)	a and b never occur together	✓⟨c, c, c, b, b, b⟩	✓⟨c, c, a, c⟩	×⟨a, c, c, b, b⟩	×⟨b, c, a, c⟩	
NOTSUCCESSION(a, b)	b cannot occur after a	✓⟨b, b, c, a, a⟩	✓⟨c, b, b, c, a⟩	×⟨a, a, c, b, b⟩	×⟨a, b, b⟩	
NOTCHAINSUCCESSION(a, b)	a and b cannot occur contiguously	✓⟨a, c, b, a, c, b⟩	✓⟨b, b, a, a⟩	×⟨a, b, c, a, b⟩	×⟨c, a, b, c⟩	

Figure 10: Declare Templates as shown in [DM22]

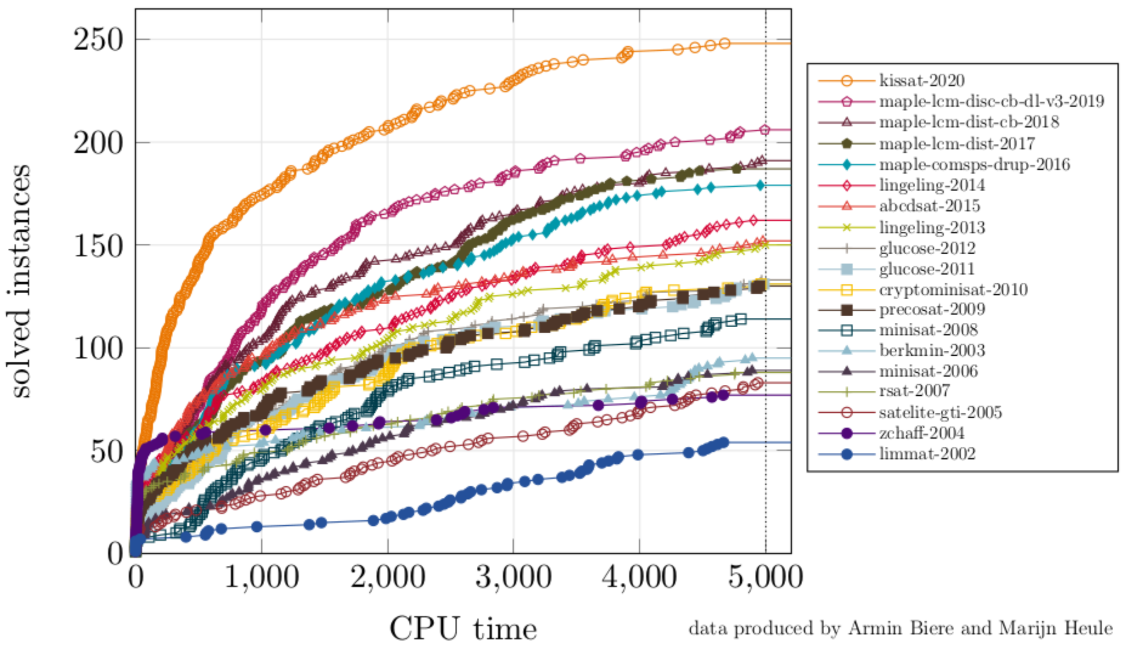


Figure 11: SAT solving competition winners comparison
https://twitter.com/ArminBiere/status/1288132283443142661/photo/1

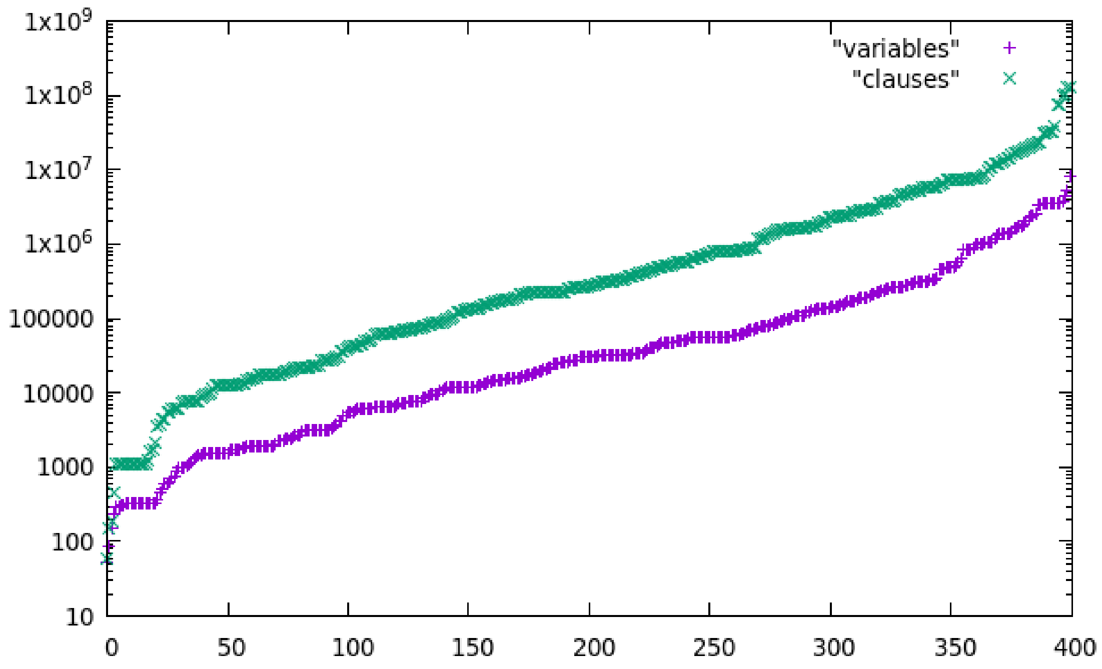


Figure 12: Distribution for Figure 11 instances
https://twitter.com/ArminBiere/status/1288552471141507073/photo/1