Automated Reasoning and Learning for Automated Payroll Management

Sebastijan Dumančić, [1] **Wannes Meert,** [1] **Stijn Goethals,** [2] **Tim Stuyckens,** [2] **Jelle Huygen,** [2] **Koen Denies** [2]

[1] KU Leuven, Belgium
[2] Teal Partners, Belgium
{sebsatijan.dumancic,wannes.meert}@kuleuven.be, {stijn,tim,jelle,koen}@tealpartners.com

Abstract

While payroll management is a crucial aspect of any business venture, anticipating the future financial impact of changes to the payroll policy is a challenging task due to the complexity of tax legislation. The goal of this work is to automatically explore potential payroll policies and find the optimal set of policies that satisfies the user's needs. To achieve this goal, we overcome two major challenges. First, we translate the tax legislative knowledge into a formal representation flexible enough to support a variety of scenarios in payroll calculations. Second, the legal knowledge is further compiled into a set of constraints from which a constraint solver can find the optimal policy. Furthermore, payroll computation is performed on an individual basis which might be expensive for companies with a large number of employees. To make the optimisation more efficient, we integrate it with a machine learning model that learns from the previous optimisation runs and speeds up the optimisation engine. The results of this work have been deployed by a social insurance fund.

Introduction

Payroll management is a crucial aspect of any business venture. While performing the tax calculation given all necessary inputs is straightforward, anticipating the impact of changes in the inputs is a challenging task due to the complexity of the legislation. Evaluating changes, however, is necessary to find an optimal payroll policy (e.g., allowing a car or paying part of a salary in the shares of the company). The goal of this work is to automatically explore potential policies and find a set of optimal policies.

To achieve this goal, two challenges need to be solved. The first challenge is to translate legal knowledge into a formal representation flexible enough to support and simulate a variety of scenarios that can occur in payroll calculations. The advantage of working with tax legislation is that it is precisely defined and deterministic whereas many other areas of legislation are vague and open-textured (Prakken 2017). The major challenge in this regard is that the exact calculation is not known upfront and thus a single algorithm or a computation graph cannot be used. An alternative, and more general, approach is to encode the knowledge in a formal declarative language from which computation graph can be generated and customised on demand. The benefit of the single knowledge base supporting all tasks is that there is no additional maintenance cost and lower risk that errors are introduced.

The second challenge is to systematically explore various policies. Simple intuition is often misleading – increasing the gross wage by a small amount could change the taxation grade of an employee and result in a smaller net wage – and all possible scenarios should be tried and checked. This requires an intractable amount of time if many choices are available and is thus not a valid approach. As a consequence, exploring various policies is often a manual ad-hoc exercise based on limited intuitions. For a certain set of tasks, automated approaches are available but they are limited to tasks that can be expressed as linear programs (LP) or mixed-integer programs (MIP). But many payroll tasks are non-linear and current software packages for payroll management, to the best of our knowledge, have limited support for finding optimal values in this setting (e.g., TurboTax).

In this work we focus on any financial decision that can be expressed as an optimisation tasks of the following form:

Optimise an output quantity given a number of open input variables, while respecting the financial computation rules.

For instance, a self-employed person might be interested in the minimally required number of working days and the desired rate to achieve an income above a certain amount. Important to note is that the exact optimisation task is not known upfront and depends on the question that a customer asks. Therefore, the task cannot be coded upfront and we need to support all tasks with various input-output relations.

In this work, we describe a general methodology for solving the described optimisation tasks based on the state-of-the-art constraint solving technology, a field of artificial intelligence (De Moura and Bjørner 2008). The methodology is integrated into VIREN[1], a platform which helps companies to manage the complexity related to payroll regulation and enables users to generate, simulate and compare different earning scenarios in real-time. The results of this work have been deployed successfully for almost one year by Xerius, a social insurance fund, and Teal Partners. The knowledge for this particular use case has been supplied by SD Worx, one of the main payroll service providers in Belgium, and the

[1]https://www.viren.be/

results have been verified by an independent and licensed accountant.

Constraint solving is a natural fit for this problem for three reasons: (1) the rules present in the legislation can be **translated into a set of constraints**; (2) all **supported tasks can be automatically encoded** as constraints with objective functions, dynamically adapting to various input and output combinations; and (3) it **guarantees** that found solutions satisfy the constraints. There are, however, several requirements that are not trivial such as optimising over chained, numerical, non-linear formulas. Although in theory a hard problem, we found that in practice the problems can be tackled by applying certain transformations and a customised optimisation algorithm.

The **first contribution** of this work is a compilation procedure of the payroll knowledge, specified within VIREN, to a formal representation that allows finding optimal payroll policies automatically. This includes the tasks where the outputs depend non-linearly on the input variables, a case that is unsupported in other software packages. The **second contribution** is the integration of a machine learning model which assist the optimisation engine such that it finds new solutions faster. In practice, the optimisation task has to be solved for (tens of) thousands of employees; this puts a large burden on the system if the task itself is expensive. However, while most people are different, many people are similar to at least a few other people; this observation can be used to *guess* an initial solution that can then be further refined by the optimization algorithm. The **third contribution** is that we have deployed this method and validated it in a real-world setting.

The presented methodology is not limited only to payroll management problems. Any financial task formulated as an optimisation problem that has to adhere to constraints, legal or any kind, can be tackled in the same way. Payroll legislation is particularly appealing because the constraints and computational rules are precise and explicit.

Problem Statement

The VIREN calculation engine (Goethals 2019) is a platform for executing a high load of complex calculations with a very low response time. The VIREN engine solves problems of the following form

Given a domain model consisting of

- input parameters $\mathcal{I}$
- output parameters $\mathcal{O}$
- a set of computational rules relating $\mathcal{I}$ to $\mathcal{O}$
- an optional value assignment A to $\mathcal{I}' \subset \mathcal{I}$

Find an assignment $\mathcal{A}$ to the input parameters $\mathcal{I} \setminus \mathcal{I}'$ such that output values $o \subseteq \mathcal{O}$ are optimised.

The domain model describes legislative knowledge and customers' interests. The inputs and outputs for the general domain model are specified in advance. However, when applied to a specific case, some of the input and outputs have fixed values; which ones is not known in advance.

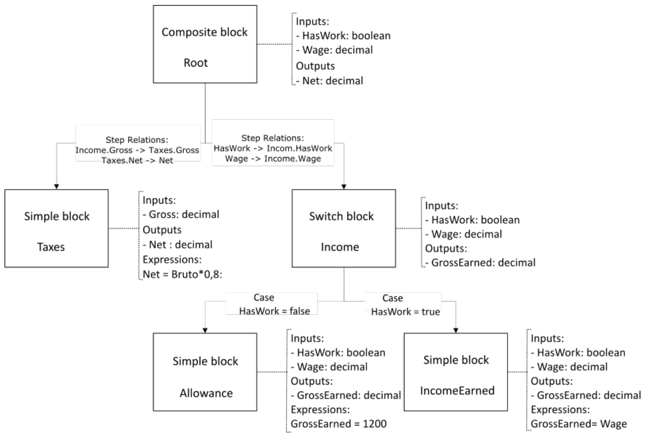


Figure 1: Example of an FML model

Financial Modelling

The tool used for describing legislative knowledge and defining customers' interests in this work is the VIREN Financial Modelling Language (FML), but the compilation can be easily adapted to other languages. FML is a mathematical expression language with, at its core, rules that associate formulas and variables. The underlying idea is similar to formulas in a spreadsheet. The rules support standard mathematical calculation over boolean values, integers and decimals, extended with functionality for database lookup, control flow, (business) date calculation and user-defined functions (Figure 2 gives the expression grammar for FML).

One of the reasons why payroll optimisation is a challenging computational problem is that payroll scenarios not only involve calculations with different variable types but also require additional domain-specific types of variables. On top of the standard unit types, such as Boolean, decimal, datetime and strings, the VIREN platform allows users to specify *complex* types representing the composition of several unit types, as well as a *table* type which is a complex type used to implement a database functionality.

FML organises rules and computation in blocks, where each block has clearly defined input and output parameters and rules. The VIREN platform introduces three types of blocks: *simple* blocks that perform the computation, *composite* blocks that combine results of multiple simple blocks, and *switch* blocks that choose which block to execute given a condition.

Figure 1 illustrates five blocks in FML that computes the (simplified) net income earned by a person. The created model has two input variable: *HasWork* and *Income*. The first quantity the model calculates is the *gross income*: for employed persons that equals their wage, while for the unemployed people that equals the minimal wage (€ 1200 in this case). To calculate the *tax* on the *gross income*, the block *Taxes* applies a flat rate of 20% to the *gross income*. The result is linked back to the root block, providing the end-user with the result of the computation.

```
Model      → Block | Block, Block
Block      → Simple | Composite | Switch
Simple     →  Expr
Composite  → Compose(Block, Block, Relations)
Switch     → ite(Condition, Block, Block)
Expr       → Expr | Expr, Expr | Condition |
               Const | Var | !Expr | −Expr |
               Complex | Expr BinaryOp Expr |
               ite(Condition, Expr, Expr)
Condition  → Expr Compare Expr |
               Condition AND Condition |
               Condition OR Condition
Compare    → == | > | ≥ | < | ≤ |  !=
BinaryOp   → + | − | / | % |  × | Bitwise
Bitwise    → & | || | << | >>
Complex    → Database | Date | Arithmetic
Database   → Lookup | LookupMany
Date       → AddDays | AddMonths | AddYears |
               FirstOfMonth | LastOfMonth | Date |
               Day | Month | Year | DaysInYear |
               DaysInMonth | DateInYear |
               BusinessDaysInMonth | ExactYears |
               BusinessDaysUntil | YearsBetween |
               MonthsBetween | DaysBetween
Arithmetic → Count | Sum | Min | Max | Average |
               Floor | Ceiling | Pow | RoundDown
               FloorToNearestMultiple | Round
```

Figure 2: An expression grammar of the Financial modelling language. Arguments of the functions are left out for brevity.

Lessons Learned

Solving constraint satisfaction problems (CSPs) is fundamental to computer science. In short, CSPs considers problems of finding a solution to a set of constraints imposing the conditions that the variables must satisfy. The solution to the constraint satisfaction problem (CSP) is the value assignments to variables that satisfy all constraints. Most often many solutions exist, but not all are equally desirable. To indicate the preference for a particular solution, an objective function can be imposed.

Here we briefly summarise the main insights gained through this project, before proceeding with compilation procedure which is the main contribution of the work. We split the insights into two categories: choosing the right constraint solving paradigm and model transformations.

Constraint Solving Paradigms

With CSPs being fundamental to computer science, various paradigms for solving CSPs have emerged over the past decades. These include Boolean satisfiability (SAT) (Biere et al. 2009), Constraint programming (CP) (Rossi, van Beek, and Walsh 2006) and Satisfiability modulo theory (SMT) (De Moura and Bjørner 2008). SAT does not fit our case as it only supports reasoning with Boolean variables. In essence, CP and SMT paradigms are equivalent: both are a generalisation of the SAT problem in which decision variables can take values from any countable or finite domain, including Boolean, integer and decimal values. Both paradigms support rich types of constraints including arithmetic (e.g., `incomeNet > 50000`), logical (e.g., `!nocar || !typea`), complex conditioning (e.g., `if ... then ... else ...`) and much more. We have found SMT solvers to be best suited for the kind of tasks occurring in payroll management. In the rest of this section, we will briefly outline the reasons why it is so.

The first reason concerns **search procedures**. The main difference between SMT and CP are the underlying search procedures. SMT solvers operate as a collection of specialised solvers that are smartly orchestrated together; that is, Boolean constraints are handled through a SAT solver while the numeric constraints are handled with the specialised numeric solver. Consequently, they can efficiently search infinite real domains. The search procedures within CP are *monolithic* – regardless of whether the variables are Boolean or integer, the search procedure is identical.

The second reason concerns the **support for decimal numbers**. Payroll management scenarios involve numerical computation with decimal numbers and the precision of the calculation is often defined by law. Consequently, performing any kind of approximations is unacceptable. We have found that SMT solvers have substantially better support for decimal numbers compared to other solvers (e.g., MiniZinc (Nethercote et al. 2007)). The main challenge are equality constraints, such as

$$incomeGross \equiv dailyWage \times workingDays$$

which states the variable `incomeGross` needs to have exactly the same value as the product of variables `dailyWage` and `workingDays`. Such constraints are necessary to encode the computational rules regarding payroll management. In contrast to SMT solvers which incorporate specialised solvers for constraints involving real numbers, other solvers from the CP family encounter rounding errors when the number of computational rules is large; this, consequently, prevents them from finding a satisfiable solution.

The third reason concerns the type of constraints various solvers expect. Existing constraint solvers typically expect *linear constraints* such as

$$yearlyWage \equiv 12 \times monthlyWage + benefits$$

i.e., two decision variables are never multiplied with each other. The problems we encountered are, however, mostly non-linear – constraints that have multiple (two or more) decision variables are generated. In contrast, SMT solvers have a specialised solver for non-linear problems (Jovanovic 2017). Though solving non-linear constraints is undecidable in theory, many practical non-linear problems can be solved with SMT solvers, including the typical programs in VIREN.

Model Transformations

The state of the art SMT solver, such as Z3 (De Moura and Bjørner 2008), are mature and performant technology. However, we have found that certain model transformations improve performance significantly.

Domain experts typically write general domain models that account for many input variables. However, when the general model is applied to a specific case, many of the input variables have concrete fixed values. As we explain later (Section), propagating this information through the model can significantly reduce the number of constraints.

Financial calculations introduce variables with custom domains, e.g., employment benefits come with predefined categorical choices. Though such data types can usually be emulated with Boolean or integer data types, we have found out that introducing custom data types avoids several unwanted properties (Section). Moreover, custom types make it possible to perform complex calculation efficiently: date calculations are ubiquitous in financial domains (Section), but none of the SMT solver support such calculation inherently.

Date calculations typically cannot be performed analytically, as they require looping over days and years. However, SMT solvers do not support loops and unrolling loops naively would make the corresponding CSP significantly more complex to solve. Thus, we have found it important to develop analytical solutions for the quantities that require looping calculation (Section).

The FML Compiler

The Financial Modelling Language (FML) represents knowledge as a set of *financial modelling rules* ($\mathrm{FM_R}$). Unfortunately, these cannot be used directly by an SMT solver. A number of inference and transformation steps are required to translate the procedural computation rules to *financial modelling constraints* ($\mathrm{FM_C}$). The constraints in $\mathrm{FM_C}$ will be expressed in the standardised SMT-LIB format, which all SMT solvers know how to read. The tool presented in this work presents a compiler that automates this process. The concept is generic and can generate constraints that are compatible with most constraint solvers.

Once the VIREN model is translated to the task-specific constraints, the model is passed to an optimizer based on Z3 (De Moura and Bjørner 2008), a state-of-the-art SMT solver. The answer(s) produced by Z3 are processed by the VIREN cloud-based computation engine and shown visually in the user interface.

An example of the VIREN user interface for knowledge modellers is shown in Figure 4. The left pane shows the blocks in the model and how they are connected, the middle part shows the selected block's computation rules in FML, the right pane shows the block's inputs and outputs. In the remainder of this section, we provide more details on these components in the following sections.

Blocks

FML groups computation rules in *blocks* that together compute one concept. A block encapsulates a number of computations from its environment, and it is defined by (1) input variables, (2) output variables, and (3) rules connecting the input to output variables. This block-wise structure introduces a namespace behaviour which has to be considered as different blocks can contain variables with the same name and be reused across different regulations. Output variables are connected to input variables by constraints making them equal to each other, or by replacing the input values by the connected output values. Additionally, Z3 requires a topological ordering based on the block dependency graph because variables need to be defined before being used. See Figure 3 for an example.

```
FMR:
BasicFlatRateProfessionalCosts = AnnualWageCompanyDirector +
                                 TotalBenefitsVAA + CarVAA
PercentageFlatRate = lookup(BV_FlatRateProfessionalCosts,
                            'Percentage', TypeComputation)
MaximumFlatRate = lookup(BV_FlatRateProfessionalCosts,
                         'Maximum', TypeComputation)
FlatRate = min(BasicFlatRateProfessionalCosts *
               PercentageFlatRate, MaximumFlatRate
Basic = BasicFlatRateProfessionalCosts - FlatRate

FMC:
(declare-const BSC_PercentageFlatRate Real)
(declare-const BSC_MaximumFlatRate Real)
(declare-const BSC_Basic Real)
(declare-const BSC_BasicFlatRateProfessionalCosts Real)
(declare-const BSC_FlatRate Real)
(assert (= BSC_BasicFlatRateProfessionalCosts
  (+ (+ BSC_AnnualWageCompanyDirector 252.0) 2200.0)))
(define-fun min_Real ((x Real)(y Real)) Real
  (ite (< x y) x y))
(assert (= BSC_FlatRate
  (min_Real BSC_MaximumFlatRate
           (* BSC_BasicFlatRateProfessionalCosts
              BSC_PercentageFlatRate))))
(assert (= BSC_Basic
  (- BSC_BasicFlatRateProfessionalCosts BSC_FlatRate)))
```

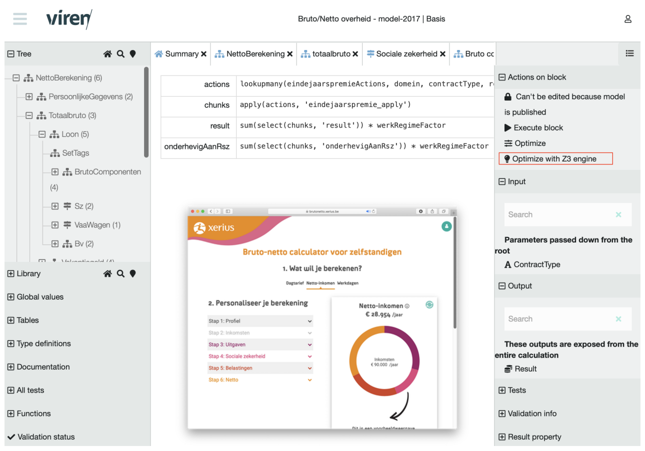
Figure 3: Example block BasicSocialContributions (BSC), both the FML version (top) and the FMC version (bottom).

Data Types for Financial Modelling

The FML data types need to be matched to the data types available in Z3. In case a variable in FML is not typed, it is inferred recursively from the rule what the expected type is.

Fixed-point Numbers FML uses fixed-point arithmetic as most financial software does. In Z3 this is translated into real numbers. For most use cases this representation is sufficient but a post-processing validation step is performed to guarantee correctness.

Categorical Choices Certain variables can only have values from a predefined set of choices. This is often the case with benefits that are offered with the employment contracts; for instance, a person might get a company car (of a certain type) or not. To model such categorical choices, we leverage Z3's ability to specify custom data types. Note that the custom types could be equally realized with boolean variables indicating individual choices. However, custom types make the entire process more interpretable and avoid undesired edge cases. For instance, mutual exclusivity of the values should be explicitly modelled in the boolean encoding, whereas custom types assume it as a default behaviour.



actions	`lookupmany(eindejaarspremieActions, domein, contractType, r`
chunks	`apply(actions, 'eindejaarspremie_apply')`
result	`sum(select(chunks, 'result')) * werkRegimeFactor`
onderhevigAanRsz	`sum(select(chunks, 'onderhevigAanRsz')) * werkRegimeFactor`

Figure 4: The VIREN UI (large) and the end-user UI (small, blurred for double-blind review)

Dates In financial calculations, many computations involve date manipulation: salaries are paid on the last day of a month and certain calculations rely on knowing the exact number of days between the two given dates. It is therefore of utmost importance to manipulate the dates correctly. Unfortunately, Z3 has no built-in date operations and supporting them is non-trivial.

To support date calculations, we have opted to use the ordinal date representation[2] allowing us to map dates to a tuple of integers. Ordinal date representation represents a date as a tuple `(YYYY,DDD)` where `YYYY` indicates the year and `DDD` indicates the index of the day within the year. The advantage of this representation is that crucial calculations required by VIREN can be performed analytically.

The first type of date calculation we explain in detail is to map a given date to a date corresponding to the first day of the same month. For instance, April 14 2017 would be mapped to April 1 2017. To do so, we use the ordinal date representation to calculate the day of the month of the given date as

$$(\texttt{DDD} \bmod 30) + i - \left\lfloor 0.6\left(\left(\left\lfloor \frac{\texttt{DDD}}{30} \right\rfloor + 1\right) + 1\right)\right\rfloor$$

where i is 3 for normal and 2 for leap years.

The second type of date computation is calculating the time passed between the two dates. This value takes the form of a real number where the integer value represents the number of years passed and the decimal value represents the proportion of the days passed within the last year of the difference (i.e., 365 days minus the number of days needed to make the two dates exactly N years apart). This calculation is difficult to obtain analytically unless ordinal date representation is used.

A more straightforward date representation would be *the number of days since January 1 year 0*, which is often a standard. But performing the required calculations with this representation cannot be achieved analytically anymore: it requires looping over the years and days. As Z3 does not support loops, they would have to be unrolled which would make the target CSP significantly more complicated.

[2]NIST. 1988. Representation for Calendar Date and Ordi- nal Date for Information Interchange. Federal Information Processing Standards Publication 4-1

Rules

Every single financial modelling rule (i.e., a formula) needs to be translated into a form that is compatible with Z3.

Mathematical Operations All operations on numbers (e.g., multiplication, min) have a direct mapping to Z3 operations (Barrett, Fontaine, and Tinelli 2017).

Domains One of the key concepts of constraint satisfaction is the bound inference – techniques actively try to reduce the domains of variables to sensible regions during the search. For instance, while we would represent a year with an integer number, not all integer numbers qualify as a valid solution: a person typically cannot be 300 years old. As the size of domains substantially impacts the complexity of the search, supplying sensible bounds upfront makes the search faster. If domains are known from FML, they are incorporated as constraints.

Tables FML supports the representation of tables in which values can be looked up. Given the input and output variables of a lookup operation, a table can be represented by a sequence of if-then-else statements.

```
FMR:
lookup(NoticePeriod, 'p2', Anc)
```

```
FMC:
(define-fun lookup_p2_from_noticeperiod ((Anc Real)) Real
(ite (and (<= 0.0 Anc) (< Anc 0.25)) 2.0
(ite (and (<= 0.25 Anc) (< Anc 0.5)) 4.0
...
```

Tables are losslessly compressed by applying the C4.5 decision tree algorithm where the input values are the features and the output value is the target variable (Quinlan 1993).

Control Flow If-statements are available in both FML and Z3. A difference is that FML allows different assignments to the same variable in both branches of an if-else-statement. When expressing this as a constraint, a variable can only be made equivalent to one statement and thus requires the variable equivalence on the outer level. Therefore, variables are moved from within the if-statement to the outside of it.

```
FMR:
if not carclass == "None": carVAA = carVAAfromtable
else: carVAA = 0.0
```

```
FMC:
(assert (= carVAA
(ite (not (= carclass "None") carVAAfromtable 0.0))))
```

Custom Functions FML allows users to define custom functions. Fortunately, Z3's language supports such constructs which allow for a direct mapping of the functions defined in FML to the SMT-LIB specification. Date functionality described in Section make use of this functionality.

Propagating Information

The models defined in VIREN represent general rules of computation for a certain domain/problem, with many input variables. However, when applying the general model to a specific case (e.g., a particular employee), some of the input variables become known and cannot be changed during the optimisation (e.g., birth date and the number of children). Consequently, their values can be *propagated* through the computational rules rendering many intermediate calculations to a fixed value and, by doing that, reducing the number of variables in $\mathrm{FM_C}$. The VIREN platform does this automatically – this reduces the number of constraints by a factor of 10 consequently allowing Z3 to find a solution faster.

Note that Z3 has built-in simplification methods, but these do not apply to tables. In many VIREN use cases, the tables are one of the bottlenecks, sometimes having up to several tens of thousands of rows. For example, official tables expressing the tax rate per sector. This is information you cannot change and it is thus not useful to encode the entire table as constraints. Instead, we automatically select the relevant tuple from the table by propagating values throughout the entire program.

Optimisation

Given a set of constraints that represent the knowledge about tax calculation, we want to optimise a given target quantity. The Z3 solver, however, is only focused on finding whether a solution exists, and what that solution is. In this section, we introduce the optimisation strategy that we developed on top of the Z3 solver.[3]

Our optimisation strategy is designed to exploit two main insights we obtained. First, we know the types of queries that will be asked and all can be mapped to finding the minimal value of a variable given ranges on a number of other variables. Second, Z3 is based on a conflict-driven approach and thus, on average, fast at finding whether a situation is unsatisfiable. Thus even if finding a globally optimal solution is slow, finding a solution close to the global optimal might be fast in practice and sufficient.

Minimisation Algorithm

In the VIREN tool, we implemented an algorithm, VMinimise, that searches for the minimal value of a target variable given a number of input variables (see Algorithm 1). Given a threshold, VMinimise essentially adds an inequality constraint between the target variable and a threshold to the already existing set of constraints and verifies whether this new program is satisfiable. Next, a variation of bisection is used with a few changes to speed up the search. If the program is unsatisfiable we know there is no such solution and the lowest possible value for the target variable will be higher than the threshold. We can set the new threshold to a point between the previous threshold and the lower bound. In case it is satisfiable and the returned solution found a value for the target variable that is lower than any previous solutions, we have thus found a new lowest value. We can set the new threshold to a point between the newfound value, thus not the previous threshold, and the upper bound. Using the found value for the target variable is correct because the constraint we added states that the found value is smaller or equal to the threshold. This process can be repeated until a certain precision or a maximal number of iterations is reached.

```
Algorithm: VMinimize
input  : Target variable $T$, bounds $T_{min}$ and $T_{max}$, tolerance $\epsilon$, max
         number of iterations $i_{max}$, $FM_C$
output: Best model (i.e., values) for $FM_C$
$\sigma \leftarrow 0.4$;                                  // where to split
$bestmodel \leftarrow none$;
$model \leftarrow$ Z3($T_{min} \leq T \leq T_{min} + \epsilon \cup FM_C$);
if $model.sat = true$ then return $model$;
$\tau_u \leftarrow model.T$;          // lowest UNSAT threshold for $T$
$model \leftarrow$ Z3($T \leq T_{max} \cup FM_C$);
if $model.sat = false$ then return $none$;
$\tau_s \leftarrow model.T$;          // highest SAT threshold for $T$
$\tau \leftarrow (1-\sigma)\tau_u + \sigma\tau_s$;                 // threshold for $T$
$i \leftarrow 0$;
while $|\tau - \tau_s| > \epsilon \wedge i < i_{max}$ do
    $model \leftarrow$Z3($T \leq \tau \cup FM_C$);
    if $model.sat = true$ then
        $bestmodel \leftarrow model$;
        $\tau_s \leftarrow model.T$;          // use found value of T
    else
        $\tau_u \leftarrow \tau$;
    $\tau \leftarrow (1-\sigma)\tau_u + \sigma\tau_s$;
    $i \leftarrow i + 1$;
return $bestmodel$
```

Algorithm 1: The VIREN minimisation algorithm. A variation on the bisection search algorithm that introduces a skew to split in unequal parts to benefit from Z3's conflict driven approach that is called in each iteration.

Because Z3 is conflict-driven, iterations in the optimisation algorithm that conclude unsatisfiability are executed faster. But only iteratively increasing the threshold to reach a satisfiable program might take many steps. We thus opt for a bisection algorithm with a slight bias towards checking too low thresholds by not splitting the search space into two equal parts using a $0.5/0.5$ split but skewed to $0.4/0.6$ split. This offers a good trade-off because the maximum number of iterations is bounded by a slightly larger number than equal splits, $\frac{\log(T_{max}-T_{min})-\log(\epsilon)}{\log(1/(1-\sigma))}$, but less expensive iterations. On average, the optimal value for skewness σ is $\frac{cost(unsat)}{cost(sat)+cost(unsat)}$, assuming $cost(unsat) \leq cost(sat)$. We can estimate this to be $\sigma = 0.4$ for the use case in this paper.

This minimisation algorithm is sufficient to support the three different tasks of interest. First, minimisation is the native use of this algorithm. Second, maximisation is achieved by introducing a new auxiliary variable that is the negation of the original variable:

```
(assert (= TargetVar (- TargetVarOrig)))
```

[3]Support for optimisation is available in Z3 but it only supports linear problems.

```
Algorithm: VirenMinimiseMachineLearning
input  : Target variable $T$, tolerance $\epsilon$, max number
         of iterations $i_{max}$, initial guess $T_i$, $FM_C$
output: Best model (i.e., values) for $FM_C$
$T_{init} \leftarrow$ ModelPrediction($FM_C$);      // Learned Model
$\tau_u \leftarrow none$; $\tau_s \leftarrow none$; $\tau \leftarrow T_{init}$; $i \leftarrow 0$;
while $\tau_s$ is none do
    $model \leftarrow$ Z3($T \leq \tau \cup FM_C$);
    if model.sat then $\tau_s \leftarrow model.T$ ;
    else $\tau_u \leftarrow \tau$; $\tau \leftarrow \tau + \epsilon^{i+1}$ ;
    $i \leftarrow i + 1$;
$\tau \leftarrow T_{init}$; $i \leftarrow 0$;
while $\tau_u$ is none do
    $model \leftarrow$ Z3($T \leq \tau \cup FM_C$);
    if model.sat then $\tau \leftarrow \tau - \epsilon^{i+1}$ ;
    else $\tau_u \leftarrow \tau$ ;
    $i \leftarrow i + 1$;
return VirenMinimize($T$, $\tau_u$, $\tau_s$, $\epsilon$, $i_{max}$, $FM_C$)
```

Algorithm 2: The Viren minimisation algorithm. A variation on the bisection search algorithm that introduces a skew to split in unequal parts to benefit from Z3's conflict-driven approach that is called in each iteration.

And third, the closest value to a given target value can be found by introducing a new auxiliary variable that is the difference between the target value and the target variable:

```
(assert (= TargetVar (abs (- TargetVarOrig TargetVal))))
```

Pareto-front Algorithm

Often, there is no one optimal set of input values that achieves the best solution but a set of different combinations of input values that lead to the same optimal result. In such a case, a Pareto front is interesting because it expresses the trade-off between the different combinations of input values (Mas-Colell, Whinston, and Green 1995). This allows the user to express a preference for one of these combinations even though it is not explicitly encoded in the constraints. In some cases, the user might not be aware of the impact of this trade-off and wants to explore the Pareto front.

To search for a Pareto frontier, which is the set of Pareto-optimal options, one collects all possible solutions and then selects those solutions that are Pareto optimal with respect to the other solutions. Concretely, a solution t^* is (strongly) Pareto-optimal if no other solution t' weakly Pareto-dominates t^*. An instance t^* weakly Pareto dominates t' if $\forall i \leq n : utilty_i(t^*) \geq utility_i(t')$ and $\exists i \leq n : utilty_i(t^*) > utility_i(t')$. Here we take the utility function to be the identity relation but it can be used to assign different weights to the variables.

Selecting the Pareto optimal solutions afterwards is a naive and costly approach. Fortunately, this can be made more efficient by including the Z3 solver in the search and incrementally adding constraints that exclude solutions that we know cannot be Pareto optimal anymore. Thus avoiding regions of the solution space that are not interesting as soon as possible (see Algorithm 3).

```
Algorithm: VParetoFront
input  : Target variable $T$, target value $\tau_t$, input variables $\mathbf{I}$, accuracy $\delta$,
         max iterations $i_{max}$, and $FM_C$
output: Pareto models for $\mathbf{I}$
$\mathbf{\Pi} \leftarrow \{\}$;            // Potential Pareto front models
$\mathbf{C} \leftarrow \{\}$;         // Constraints to exclude non Pareto
$model \leftarrow$ Z3($T - \tau_t \leq \delta \cup FM_C$);
while model.sat = true do
    $\mathbf{\Pi} \leftarrow \mathbf{\Pi} \cup model$;
    // Exclude all solutions that cannot be Pareto optimal
    foreach $I \in \mathbf{I}$ do
        $\mathbf{C} \leftarrow \mathbf{C} \cup I \geq model.I$;
    $model \leftarrow$ Z3($T - \tau_t \leq \delta \cup \mathbf{C} \cup FM_C$);
// Exclude all not Pareto optimal solutions
$\mathbf{\Pi} \leftarrow \{\pi \in \mathbf{\Pi} \mid \forall \pi' \in \mathbf{\Pi} \setminus \pi \forall I \in \mathbf{I} : \pi.I \leq \pi'.I \wedge \exists I \in \mathbf{I} : \pi.I > \pi'.I\}$;
return $\Pi$
```

Algorithm 3: The VIREN Pareto front algorithm. Constraints are incrementally added to use the Z3 solver to find Pareto optimal solutions more efficiently.

Improving Optimisation Speed with Machine Learning

The payroll optimisation task is performed on an individual employee basis. This can be an expensive task for a company of many thousands of employees. To overcome this issue, we can improve the efficiency of the optimisation step by using the solutions from previous calculations to provide a better initial point for the optimisation procedure. Though very few employees would share the identical characteristics, many employees would be similar which could give us a good estimate of the range for the optimal target value.

Remembering the optimal solutions for all previously solved cases is not a feasible strategy. Therefore, we will use machine learning to discover the relationship between the subset of input values of the individuals and the optimal payroll target and use the predicted value as an initial value during optimisation. Individual optimisation instances will represent the examples for the machine learning problem. Each instance is described with a fixed set of parameters, divided into the input and decision parameters, and the optimal target value obtained by performing the optimisation. The input parameters are provided by the user ahead of time, and the goal of the optimisation is to find the assignment to the decision parameters such that the target value is optimal.

We will focus on predicting the optimal target value rather than the exact values of decision parameters for two main reasons. Firstly, the relationships between the parameters (both input and decision) are dictated by complex constraints that are very difficult to discover with machine learning models. Secondly, our system needs to guarantee that the given solution obeys all constraints specified by law. Predictions of machine learning models would likely violate the

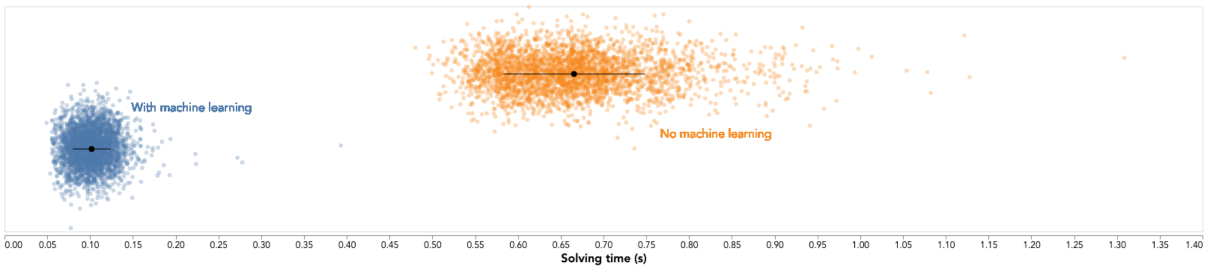


Figure 5: Using machine learning to initialise the optimisation procedure significantly improves the performance of the constraint solver. The solver is able to find the optimal solution up to $4\times$ faster.

specified constraints if applied to the remaining parameters. Therefore, we use the predicted value as an *initial estimation* (instead of determining it through constraint solving) and modify it to fit the law requirements. This procedure is detailed in Figure 2.

Note that we cannot use machine learning to provide the final output directly. A reasoning procedure is always required to guarantee that all constraints are met and the solution is legal.

Experiments and Results

Our system **always finds the optimal solution** up to a certain precision. Therefore, the purpose of these experiments is to (i) evaluate how long the system takes to find the optimal solution, and (ii) whether machine learning can improve the performance of the constraints optimisation procedure in finding the optimal solution.

We extract 3016 real-world payroll calculation cases come from the gross-net calculator by Xerius[4], executed on the VIREN platform. All cases are the same type of optimisation: how many days should a self-employed person work and at what rate to maximise net income. The (propagated) models have 937 constraints on average, with the maximum being 963. First, we perform the optimisation step for each use-case in order to obtain the optimal target value. Next, we repeat the computations but guide the optimisation using the prediction given by a Random Forest model (Breiman 2001) that is trained on a separate set of optimisation results (using Scikit-learn).

We follow the standard 10-fold cross-validation procedure while training a model. For each fold, we perform inner 10-fold cross-validation in order to select the best hyperparameters of the model. We then use the selected hyperparameters to train the model of the training data of the particular split. To evaluate the benefit of a machine learning model, we use the predictions on the test set as the initial values for the optimisation

The results (Figure 5) demonstrates that our system is able to find the optimal solution faster when initialised with the predictions of the predictive models. The improvements are substantial as the solver is able to find the optimal solution 4-5 times faster than the solver with no initial value provided. Moreover, the variance in the runtimes is likewise substantially reduced.

Though the reduction in absolute time for an individual problem might not look impressive, from 0.7s to 0.1s, the constraint problem in practice needs to be solved once for each employee which means that the reductions accumulate. For instance, for a company with a 10000 employees, solving the constraint problem without an ML component would take 116 min while relying on an ML estimate reduces the solving time to 16 min. However, for a company providing this service to other companies, this would constitute even larger savings.

Related Work

The application presented in this work is unique in that it uses a formal representation of the payroll legislation to derive the constraints relevant for the task. The encoded legislation is identical to the knowledge used for the actual payroll computations. Creating and maintaining such a knowledge base for payroll legislation is a non-trivial effort and is, to our knowledge, not yet used in related work. There are some suggestions for other domains of legislation such as case law (Bench-Capon et al. 1987), or more recently traffic law where Prakken (2017) analyses the difficulties of representing law. Alternatively, general formal representations are suggested (Kowalski 1995), tools are introduced for reasoning (Prakken 2013), consolidation (Arnold-Moore and Sacks-Davis 1991) and compliance (Giblin et al. 2005) for legislation that can be expressed by Boolean atoms. However, none of these previous works offer a functional solution for tax law with its many numerical constraints.

Many analytical tasks in the financial industry, on the other hand, require a certain form of constraint satisfaction on numerical constraints. However, the focus tends to be on verification or on modelling market behaviour, not legislation. For instance, the Imandra platform[5] offers solutions for designing and analysing financial trading venues (Passmore and Ignatovich 2017) based on constraint satisfaction technology. Jin, Tsang, and Li (2009) demonstrates how many central economic problems, such as bargaining, financial investment and supply-chain management can be treated as optimisation problems with certain constraints associated. Nikolov, Nikolov, and Antonov (2013) focusses on

[4]https://brutonetto.xerius.be/nl/netto-inkomen/eenmanszaak

[5]https://www.imandra.ai/imarkets

operations with financial instruments that model dynamic financial markets. Back and Back (1995) uses tax constraints to verify and guide planning but uses hardcoded parts of legislation as an expert system. None of these works have access to the complete formally represented tax legislation nor do they offer optimisation services respecting the constraints imposed by legislation. Moreover, the work by Jin, Tsang, and Li (2009) considered only two types of constraints, whereas VIREN platform offers a generic language for expressing a wide set of constraints.

Deployment

The VIREN calculation engine (Goethals 2019) is a software platform for executing a high load of complex calculations with a very low response time. While the industry average for a complex wage on a mainframe is 56 calculations per second, the VIREN engine is capable of performing 10,000 calculations per second. Additionally, it is designed such that analysts and lawyers can enter and review knowledge directly without the need of programmers. For simplifying development we created the Viren.Net package which contains everything needed for the communication with the Viren engine.

The Viren engine is already used in a wide variety of applications and has already executed more than 6 568 819 calculations[6].

Conclusions and Future Work

In this work we presented a compilation procedure to translate legislative knowledge about payroll to a formal representation that allows for finding optimal policies automatically. The method is made faster by integrating a machine learning model that learns how to assist the optimisation engine based on previous results. Additionally, this solution has been deployed and validated in a real-world setting.

Acknowledgements

This research was supported by VLAIO O&O innovation project "Platform for complex computations" (HBC.2017-0123). The authors would like to thank Joris Valkenborgh (SD Worx) to recognise this opportunity and help initiating the project; and Stefano Teso for his insights about Z3.

References

Arnold-Moore, T.; and Sacks-Davis, R. 1991. Databases of legislation: The problems of consolidations. In *International Conference on Research and Development in Information Retrieval (SIGIR)*.

Back, B.; and Back, R. 1995. Financial statement planning in the presence of tax constraints. *European journal of operational research* 85(1): 66–81.

Barrett, C.; Fontaine, P.; and Tinelli, C. 2017. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[6]as of July 16 2020

Bench-Capon, T. J.; Robinson, G. O.; Routen, T. W.; and Sergot, M. J. 1987. Logic programming for large scale applications in law: A formalisation of supplementary benefit legislation. In *Proceedings of the 1st international conference on Artificial intelligence and law*, 190–198.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. ISBN 978-1-58603-929-5.

Breiman, L. 2001. Random Forests. *Mach. Learn.* 45(1): 5–32.

De Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 337–340. Springer-Verlag.

Giblin, C.; Liu, A. Y.; Muller, S.; Pfitzmann, B.; and Zhou, X. 2005. Regulations Expressed As Logical Models (REALM). In *JURIX*, 37–48. Citeseer.

Goethals, S. 2019. *Viren Documentation*. Teal Partners, Van de Wervestraat 20 bus 206, 2060 Antwerp, Belgium, 1 edition.

Jin, N.; Tsang, E.; and Li, J. 2009. A constraint-guided method with evolutionary algorithms for economic problems. *Applied Soft Computing* 9(3): 924 – 935. ISSN 1568-4946.

Jovanovic, D. 2017. Solving Nonlinear Integer Arithmetic with MCSAT. In *VMCAI*.

Kowalski, R. A. 1995. Legislation as logic programs. In *Informatics and the Foundations of Legal Reasoning*, 325–356. Springer.

Mas-Colell, A.; Whinston, M. D.; and Green, J. R. 1995. *Equilibrium and its Basic Welfare Properties, Microeconomic Theory*. Oxford University Press.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, 529–543. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-540-74969-1. URL http://dl.acm.org/citation.cfm?id=1771668.1771709.

Nikolov, S.; Nikolov, V.; and Antonov, A. 2013. A Constraint-Based Approach for Analysing Financial Market Operations. In *Proceedings of the 14th International Conference on Computer Systems and Technologies*, CompSysTech '13, 231–238.

Passmore, G. O.; and Ignatovich, D. 2017. Formal Verification of Financial Algorithms. In de Moura, L., ed., *Automated Deduction – CADE 26*, 26–41. Cham: Springer International Publishing. ISBN 978-3-319-63046-5.

Prakken, H. 2013. *Logical tools for modelling legal argument: a study of defeasible reasoning in law*, volume 32. Springer Science & Business Media.

Prakken, H. 2017. On the problem of making autonomous vehicles conform to traffic law. *Artificial Intelligence and Law* 25(3): 341–363.

Quinlan, J. R. 1993. C4.5: programs for machine learning. *The Morgan Kaufmann Series in Machine Learning* .

Rossi, F.; van Beek, P.; and Walsh, T. 2006. *Handbook of Constraint Programming*. USA: Elsevier Science Inc. ISBN 9780080463803.