Imperial College London

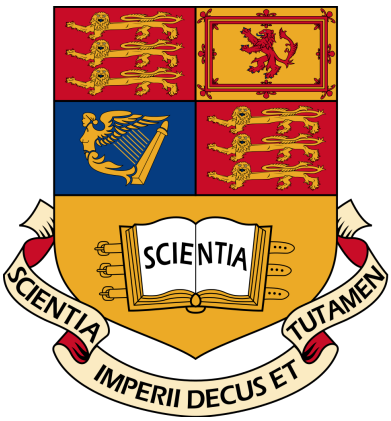
Individual Project Report

Department of Computing

Business Management Processes: Verifying their Compliance with Security and Business Rules

Author:
Joanna Diep

Supervisor:
Professor. Michael Huth
Second Marker:
Dr. Anandha Gopalan



June 5, 2015

Abstract

TODO

Acknowledgements

I would like to whole heartedly thank my supervisor Professor. Michael Huth for his continuous and invaluable advice, feedback and support throughout the course of this project. As well as giving time for meetings to discuss ideas.
I would like to thank Dr. Anandha Gopalan, the second marker for their feedback and suggestions to greatly improve my report.
Finally, I would like to thank my family and friends for their love, support and who have had to put up with me throughout this project and throughout my time at Imperial College.

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Workflows

Business management processes can be represented as a workflow or flowchart of tasks, where each of these tasks produce a certain output for the next task to be realised. These tasks are assigned in the workflow as a sequence, where in order to perform the next task, the current task must be completed. Therefore, in order to start a task, it must meet all the constraints within workflow. However, in a large business, making sure that all the constraints are met in a workflow can become a large problem as the more tasks are added to a workflow. Workflows are modeled as directed acyclic graphs [1], where there are no directed cycles. The vertices are tasks in the workflow and the directed edges connecting to each task vertex is the execution order.

An example of a business workflow is given in Figure 2.1a with nine tasks that need to be allocated:

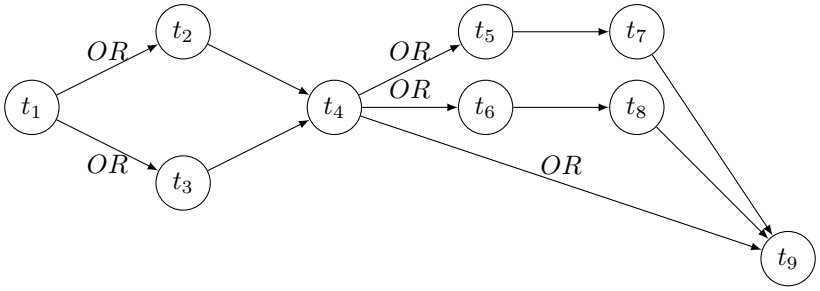
- A user in the business receives the order from a customer.
- They then pass it onto another user depending on the two possible sizes of the order and are given different prices accordingly.
- Someone then needs to approve and authorise the price for checkout.
- Then a discount may be provided depending on the current total cost of the order.
- Finally, the sale is approved and the new total is returned back to the customer.

However, the order of of execution in Figure 2.1a is affected by how to graph is forked. These forks are represented as constraints or rules within a business which may prevent tasks being executed such as a government restriction on business logic. The fork at task t_1 is an OR-fork showing that either task t_2 or task t_3 can be executed. Depending on whether t_2 or t_3 can be executed, determines whether task t_4 can then be executed afterwards and affects the whole execution of the workflow.

2.1.1 Tasks

In business management processes, users have to be allocated tasks for an execution to occur. These tasks are represented as vertices in the graph as t_n. The tasks in the example workflow given in Figure 2.1a are listed in Table 2.1b.

If users cannot be allocated to these tasks, then there is no way in which the task can be executed. Therefore it is not executed and the workflow may become unsatisfiable.



(a) Ordered business workflow with nine tasks

Task Number	Task
t_1	Receive order from customer
t_2	Give total of large sale
t_3	Give total of small sale
t_4	Approve and authorise checkout
t_5	Give 10% discount
t_6	Give 20% discount
t_7	Give new total of sale
t_8	Give new total of sale
t_9	Approve and return new total

(b) Table of tasks

Figure 2.1: Business management process workflow

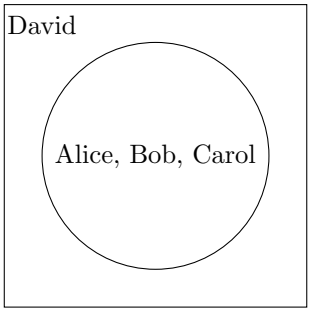
2.1.2 Users

A user u belongs to the set of users who can be allocated to tasks in order to execute them. However, there are possible allocation constraints which refrain particular users from executing these tasks. If we have a universe of users, then we can consider two cases where S is a set:
Either the users in the set can be allocated in the universe of the workflow or not. In the example in Figure 2.2a, we can interoperate that in the set if users, Alice, Bob and Carol have been allocated, but Fred has not.

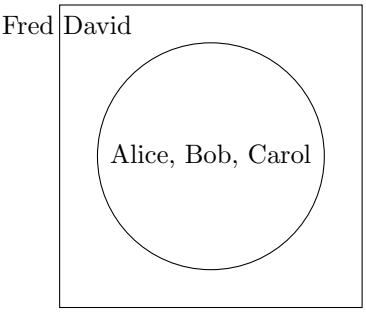
$$S \triangleq f : Universe \longrightarrow \{0, 1\}$$

Or that was have an extra case where we are unsure whether they can be allocated, which is represented by the ?. Looking at Figure 2.2b, we can see that Alice, Bob and Carol have been allocated in the universe of the workflow, but Fred could be allocated into the workflow.

$$S \triangleq f : Universe \longrightarrow \{0, 1, ?\}$$



(a) Set of definite users



(b) Set of users with unknown allocation

Figure 2.2: Set of users

2.2 Business and Security Rules

Business and security rules are used to prevent fraud and follow business compliance rules. For example, in some cases, different users are needed to execute a certain set of tasks to perhaps prevent fraud or erroneous activities in a workflow. In Figure 2.1 some constraints about which users in the business can execute these tasks are added below:

- Whomever is allocated task t_1 receives the order from customers and pass them onto the relevant user, but cannot be allocated and execute t_2 and t_3, but can be allocated other tasks within the workflow besides t_2 and t_3.
- Whomever is allocated task t_2 gives a total of a large sale cannot be the same user that is allocated t_3 who gives a total of a small sale. Therefore whomever is allocated t_3 cannot be allocated to t_2.

With the additional constraints to the workflow it may be satisfiable given that there are enough users. But there may be other constraints that can make this workflow unsatisfiable. For example, if there are not enough users to be allocated to ensure that some of the tasks do not have the same user. So a valid workflow is a satisfied workflow if there can be users allocated tasks in the workflow that do not break the constraints within the given model.

2.3 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) [2] check the satisfiability of logical formulas over given theories. It helps to determine whether there is a solution in a formula which expresses a constraint. It is one of the fundamental problems in the area of computer science to check boolean satisfiability over logical domains and the completeness and incompleteness of logical theories and complexity theory.

SMT is similar to Boolean or Propositional Satisfiability Problem (SAT) [3] , where the problem is to determine if there exists a determination that satisfies the boolean formula. But SAT ranges only over binary predicates which are predicates that only take in two arguments. Whereas SMT covers non-binary predicates with types and sorts. This project will focus on SMT rather than SAT as we can use non-binary predicates provided by SMT to define and solve some of these constraints.

2.4 Z3

Z3 [4] is a SMT solver developed by Microsoft Research. It is used to integrate several decision procedures and verify the satisfiability of logical formulas over given theories. The theories our case, is the workflow model. There are many features of Z3 which will come in useful including:

- Uninterpreted functions - A theory that has an empty set of sentences. An example of this can be an axiom, where the satisfiability of the axiom depends on whether the uninterpreted function can be evaluated to true.
- Linear arithmetic
- Bitvectors, arrays, datatypes
- Quantifiers
- Satisfiability core
- Returns a model

There were many other SMT solvers we considered but did not include certain built in theories and features such as:

- Yices [5] - It almost has all features of Z3 but doesnt have quantifiers, which is needed to define general rules to satisfy a formula in the domain.
- CVC4 [6] - It has similar features to Z3 including quantifiers that are not included in Yices. But typically with CVC4, it is not very scalable. It is intended to run with small finite models, but realistically, business processes can be huge within large organisations.
- MathSAT 5 [7]- It is lacking a lot of features in Z3, especially the quantifiers.

2.4.1 Basics

A simple example is illustrated in Figure 2.3 shows how some simple first order predicate logic in Figure 2.3a can be expressed in Z3 SMT solver in Figure 2.3b.

- To define constants in Z3, in this case x and y, we can declare them as constants using the keyword `(declare-const x Int)` where x is the name of the constant and the type of x is an Integer.
- Z3 uses assertions to add constraints to the solver as a keyword `assert`.
- `(check-sat)` (line 5) is a call to Z3 to check the satisfiability of the theory. It returns `sat` if the theory is satisfiable and `unsat` if the theory is unsatisfiable.
- `(get-model)` (line 6) is a call to Z3 to return an interpretation of the theory which makes all the formulas defined in the Z3 stack true. If the theory returns `unsat`, no model is able to be retrieved.

In the example below, $x > 10$ is given as an assertion `(assert(> x 10))`, and $y \times 10 \geq x$ is given the assertion `(assert (>= (* y 10) x))`. When this is run in Z3, a result is returned using `(check-sat)`. It returns as `sat` which means that this theory is satisfiable.
What we can also see, is that Z3 gives back an appropriate model as a result that satisfies these constraints with $x = 11$ and $y = 2$. This is true as if we put $x = 11$ and $y = 2$ back into the constraints, $11 > 10$ and $20 \geq 11$.

2.4.2 Functions

Z3 also has uninterpreted functions. Unlike most programming languages where functions have side effects, may never return a value or raise or throw exceptions, Z3 functions have no side effects since they are in classical first order logic and are total. Everything in Z3 is a function, including constants as they dont take in arguments.

$$x > 10$$
$$y \times 10 \geq x$$

(a) Simple predicate logic using logic symbols

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (assert (> x 10))
4 (assert (>= (* y 10) x))
5 (check-sat)
6 (get-model)
```

(b) Simple predicate logic in Z3

```
sat
(model
  (define-fun y () Int
    2)
  (define-fun x () Int
    11)
)
```

(c) Z3 Result

Figure 2.3: Simple predicate logic

- `(declare-fun f (Int) Int)` - We declare a function f which takes in and integer as its parameter and returns an integer

In Figure 2.4, a function `f` has been declared which takes an integer as input, and returns an integer. Since this is an uninterpreted function, Z3 does not know what this function does. But we can add some constraints, so when we apply the function to the integer, it ensures that the interpretation is consistent within the theory and constraints.

In Figure 2.4b, there are two assertions `(assert (= (f x) x))` which represents $f(x) = x$ and `(assert (> (f y) (f x))` as $f(y) > f(x)$. The result that Z3 returns in Figure 2.4c has still kept the values of `x` and `y` as it is the same as Figure 2.3. But for function f, it takes in an integer as we have specified in our function declaration as `(x!1 Int)` which means that the first variable has a type Int (integer). It returns an integer which is consistent and the type integer is interpreted.

Looking at the model returned in Figure 2.4c, we can see that `x` and `y` are both interpreted as a function as well as `f`. It interprets `f` to take in an integer, the `ite` stands for "if-then-else". So we can read the definition of `f` as if x!1 is equal to 20, then return 20, else if x!1 is equal to 2, then return 2 else, return 21. Else, return 20. So for the case that `x` is put into the function `f`, then $x!1 = 20$, then the value of the function is 20, else if `y` is put into the function, then $x!1 = 2$, then the value of the function is 21. If any other input is put in, then it will return 20.

Z3 also includes built in arithmetic functions such as $=, -, +, \times, div, mod, \geq, \leq, >, <, not$ that support integer and real constants.

2.4.3 Stack

Z3 has a stack implementation, where constraints and formulas can be pushed onto and popped off the stack using the commands `(push)` and `(pop)` which pushes and pops constraints off the stack respectively. These commands can be used to check the satisfiability of some rules or definitions. When the solver stack is pushed, the state of the solver is saved. When the stack is popped, any rules and assertions declared between that pop and the corresponding push on the stack is removed from the stack, and the

$$x > 10$$
$$y \times 10 \geq x$$
$$f(x) = x$$
$$f(y) > f(x)$$

(a) Predicate logic with functions

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (assert (> x 10))
4 (assert (>= (* y 10) x))
5 (declare-fun f (Int) Int)
6 (assert (= (f x) x))
7 (assert (> (f y) (f x)))
8 (check-sat)
9 (get-model)
```

(b) Z3 with functions

```
sat
(model
  (define-fun y () Int
    2)
  (define-fun x () Int
    20)
  (define-fun f ((x!1 Int)) Int
    (ite (= x!1 20) 20
    (ite (= x!1 2) 21
      20)))
)
```

(c) Z3 Result with functions

Figure 2.4: Predicate Logic with Functions

interpretation is reverted back to its previous state before the push.

In Figure 2.6, the theory is satisfied before the push. However, when the stack is pushed and an assertion is added which violates the constraints already in the current frame which is $x < 2$, but $x > 2$ has been pushed onto the stack frame previously, the model becomes unsatisfied. Since the constraint $x < 2$ was between a push-pop frame, it can be popped off the stack and the model is returned back to its previous state on the stack.

2.4.4 Sorts

When a constant is defined, they are declared as a type which is a sort in Z3. For example, integers, reals and booleans are declared, they are a pre-defined sort in Z3.

- `(define-sort t1 Task)` - The command defines a new symbol with the type Task.

2.4.5 Quantifiers

One of the reasons why we chose Z3 as the back end constraint solver was because it is able to have quantifiable logic such as the universal quantifier which is interpreted as "for all" $\forall$. The universal quantifier asserts that all predicates within the scope of the quantifier must be true of every value of the predicate. In Z3, they are represented as:

```
(declare-const x Int)
(declare-const y Int)
(declare-fun f (Int) Int)
(assert (> x 2))
(assert (< y 2))
(assert (= (f x) (f y)))
(check-sat)
(get-model)
(push)
(assert (< x 2))
(check-sat)
(pop)
(check-sat)
(get-model)
```

(a) Z3 with stack

```
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    3)
  (define-fun f ((x!1 Int)) Int
    (ite (= x!1 3) 1
    (ite (= x!1 0) 1
      1)))
)
unsat
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    3)
  (define-fun f ((x!1 Int)) Int
    (ite (= x!1 3) 1
    (ite (= x!1 0) 1
      1)))
)
```

(b) Z3 Result with stack

Figure 2.5: Predicate Logic with stack

- `(assert (forall ((x Int)) (x > 0)))` which in first order predicate logic is $\forall x : (x > 0)$. So for all integers, they must be greater than zero.

2.4.6 Satisfiability and Validity

Validity

A formula f is valid if f always evaluates to true for any assignment to an appropriate value.

Satisfiability

A formula f is satisfiable if there is some assignment to an appropriate value to the function where f evaluates to true.

As we mentioned previously, Z3 SMT Solver gives back the satisfiability of the interpretation. It has three states when the `(check-sat)` command is called:

- `sat` - Satisfied model, a model can be returned. We give an example of a satisfied formula in Figure 2.6a.
- `unsat` - Unsatisfied model - a model cannot be returned. We give an example of an unsatisfied formula in Figure 2.6b.
- `unknown` - When Z3 does not know whether a formula is satisfiable or not. Z3 cannot evaluate the theory.

What is good about whether a formula is satisfiable is that it is about finding a solution under a set of constraints.

```
1 (declare-const x Int)
2 (assert (> x 10))
3 (assert (< x 100))
4 (check-sat)
5 (get-model)
```

(a) Z3 with satisfied core

```
1 (declare-const a Int)
2 (assert (> a 10))
3 (assert (< a 10))
4 (check-sat)
5 (get-model)
```

(b) Z3 with unsatisfied core

Figure 2.6: Predicate Logic with stack

Chapter 3

Theory of Business Rules

3.1 Business Management Workflows

There are many rules and restrictions within business management workflows in order to prevent fraud, and a matter of authorisation and following regulation. Each rule is different, and therefore will have different axioms that follow. We will talk about the rules that have been implemented in the application.

3.2 Tasks and Users

Firstly, tasks and users need to be defined in the application in order to define the basic workflow domain. These are defined in Z3 as sorts where they define the types task and user respectively:

- `(declare-sort Task)`
- `(declare-sort User)`

Then each task and user in the domain are able to be defined after the sort has been defined:

```
1 (declare-sort Task)
2 (declare-sort User)
3 (declare-const alice User)
4 (declare-const bob User)
5 (declare-const price_large_order Task)
6 (declare-const price_small_order Task)
```

We have now declared a user called “alice” and another user “bob” in the user domain, as well as defining two tasks in the task domain “receive_large_order” and “recieve_small_order”.

3.3 User Allocation

The most basic constraint in a workflow is that the tasks can only be executed if they have been allocated a user. Obviously, if there was not a user in the domain able to be allocated to a task, no one is able to execute and complete the task for the workflow to progress.

$$alloc\_user : Task \rightarrow User$$

In Z3, we define user allocation as a function, which takes in a task, and returns a user:

- `(declare-fun alloc_user (Task) User)`

3.4 Separation of Duty

Separation of duties [8] is where users need to be different in order to complete a set of tasks. This is implemented usually implemented in order to avoid conflicts of interests that may cause fraud by an individual or break some rules within a business. This restricts and reduces powers of individuals within a business where there could be a chance of collusion happening.

This is where a user who is allocated a task t, must also be a different user to who is allocated task t if these tasks are bound as knowledge from task t must not be used in order to execute task t. It may be that separation of duties cannot use their knowledge to participate in certain tasks.

This is easy to spot within a workflow diagram giving in Figure 3.2. We give a scenario that there are three users in the domain: Alice, Bob and Carol. The separation of duties are as follows: the user who is allocated the task of receiving the order and authorising the payment at checkout and cannot not handle any form of pricing the order. Whereas whichever user handles pricing of large orders cannot be the same user that handles pricing of small orders. The same users are not authorised to be allocated the tasks of creating and authorising orders, otherwise they could create fake orders or authorise orders with incorrect prices which may benefit themselves. It is also the case that large orders and small order prices should not be affected by each other, otherwise they could affect the final price or cause some sort of price fixing.

The main rules we will focus on Figure 3.1a is between lines 14-18. As seen, we must give a rule that whichever user is allocated a certain task cannot be the same (not equal) as which ever user is allocated the other task which is defined in the separation of duty.

The resulting model in Figure 3.1b shows that this is in fact satisfiable under the given separation of duty constraints and the model is consistent with the description.

$$alloc\_user(t) \neq alloc\_user(t')$$

The allocation is as follows:

- Task price_large_order is allocated to user Alice.
- Task price_small_order is allocated to user Bob.
- Tasks receive_order and checkout are both allocated to user Carol

```
(declare-sort Task)
(declare-sort User)

(declare-fun alloc_user (Task) User)

(declare-const alice User)
(declare-const bob User)
(declare-const carol User)
(declare-const receive_order Task)
(declare-const price_large_order Task)
(declare-const price_small_order Task)
(declare-const checkout Task)

(assert (not (= (alloc_user price_large_order) (alloc_user
    price_small_order))))
(assert (not (= (alloc_user price_large_order) (alloc_user checkout)
    )))
(assert (not (= (alloc_user checkout) (alloc_user price_small_order)
    )))
(assert (not (= (alloc_user price_large_order) (alloc_user
    receive_order))))
(assert (not (= (alloc_user receive_order) (alloc_user
    price_small_order))))

(assert (forall ((u User)) (or(= u alice)(= u bob)(= u carol))))

(check-sat)
(get-model)
```

(a) Z3 Separation of Duty

```
sat
(model
  ;; universe for Task:
  ;;   Task!val!2 Task!val!3 Task!val!0 Task!val!1
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun Task!val!2 () Task)
  (declare-fun Task!val!3 () Task)
  (declare------
  ;; universe for User:
  ;;   User!val!2 User!val!0 User!val!1
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun User!val!2 () User)
  (declare-fun User!val!0 () User)
  (declare-fun User!val!1 () User)
  ;; cardinality constraint:
  (forall ((x User)) (or (= x User!val!2) (= x User!val!0) (= x User
     !val!1)))
  ;; -----------
  (define-fun price_small_order () Task
    Task!val!1)
  (define-fun checkout () Task
    Task!val!2)
  (define-fun receive_order () Task
    Task!val!3)
  (define-fun bob () User
    User!val!1)
  (define-fun carol () User
    User!val!2)
  (define-fun price_large_order () Task
    Task!val!0)
  (define-fun alice () User
    User!val!0)
  (define-fun alloc_user ((x!1 Task)) User
    (ite (= x!1 Task!val!0) User!val!0
    (ite (= x!1 Task!val!1) User!val!1
      User!val!2)))
)
```

(b) Z3 result for separation of duty

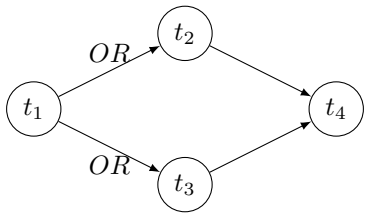


Figure 3.2: Part of the example workflow

3.5 Binding of Duty

Binding of duties [9] is where a user who is allocated a task t, must also be the same user who is allocated task t if these tasks are bound as knowledge from task t must be used in order to execute task t. It may be that binding of duties is needed as the user who is allocated these tasks have the required knowledge in order to execute both.

$$alloc\_user(t) = alloc\_user(t')$$

Given Figure 3.2, who ever is allocated the task of receiving an order has to price the order, regardless of whether it is a large or small order as they may be authorised to do so.

As seen in Figure 3.3a, the rule is specified between lines 14-17. Whichever user is allocated task price_large_order has to be the same (equal to) whichever user is allocated task price_small_order. This is matches the definition of binding of duties.

Looking at the result produced in Figure 3.4b, this is satisfiable for the given constraints and is consistent to the user input and is complete:

- Task checkout is allocated to user Alice
- All other tasks `(receive_order, price_large_order, price_small_order)` are all allocated to user Carol. Which matches the users need for binding of duties.

```
1  (declare-sort Task)
2  (declare-sort User)
3
4  (declare-fun alloc_user (Task) User)
5
6  (declare-const alice User)
7  (declare-const bob User)
8  (declare-const carol User)
9  (declare-const receive_order Task)
10 (declare-const price_large_order Task)
11 (declare-const price_small_order Task)
12 (declare-const checkout Task)
13
14 (assert (not (= (alloc_user price_large_order) (alloc_user checkout)
       )))
15 (assert (not (= (alloc_user checkout) (alloc_user price_small_order)
       )))
16 (assert (= (alloc_user price_large_order) (alloc_user receive_order)
       ))
17 (assert (= (alloc_user receive_order) (alloc_user price_small_order)
       ))
18
19 (assert (forall ((u User)) (or(= u alice)(= u bob)(= u carol))))
20
21 (check-sat)
22 (get-model)
```

(a) Z3 Binding of Duty

```
sat
(model
  ;; universe for Task:
  ;;   Task!val!2 Task!val!3 Task!val!0 Task!val!1
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun Task!val!2 () Task)
  (declare-fun Task!val!3 () Task)
  (declare-fun Task!val!0 () Task)
  (declare-fun Task!val!1 () Task)
  ;; cardinality constraint:
  (forall ((x Task))
          (or (= x Task!val!2)
              (= x Task!val!3)
              (= x Task!val!0)
              (= x Task!val!1)))
  ;; -----------
  ;; universe for User:
  ;;   User!val!0 User!val!1
  ;; -----------
  ;; definitions for universe elements:
  (declare-fun User!val!0 () User)
  (declare-fun User!val!1 () User)
  ;; cardinality constraint:
  (forall ((x User)) (or (= x User!val!0) (= x User!val!1)))
  ;; -----------
  (define-fun price_small_order () Task
    Task!val!2)
  (define-fun checkout () Task
    Task!val!1)
  (define-fun receive_order () Task
    Task!val!3)
  (define-fun carol () User
    User!val!1)
  (define-fun bob () User
    User!val!0)
  (define-fun price_large_order () Task
    Task!val!0)
  (define-fun alice () User
    User!val!0)
  (define-fun alloc_user ((x!1 Task)) User
    (ite (= x!1 Task!val!1) User!val!0
      User!val!1))
)
```

(b) Z3 result for binding of duty

3.6 Seniority

In most businesses and corporations, there are different levels of seniority based on their positions in the company. Different seniority levels allow users to execute different tasks. For example, a confidential task maybe worked on by a less senior member of a department, but have to be authorised by a more senior member as they may have more experience or the appropriate training [10].

An example of seniority allocation is given in Figure 3.5. As we can see, whoever is allocated $t1$ must be the same user as whoever is allocated to task $t4$ as they will have the same level of seniority, the same user will have the same seniority. From a business prospective, it does not make sense that the allocation has to have different users of the same seniority as the trust and experience levels are the same. Also in the figure, whoever is assigned to $t1$ has to be more senior than whoever is assigned to $t2$ and $t3$. This also means that the users assigned to $t2$ and $t3$ have to be less senior than the user allocated to $t1$. The tasks $t2$ and $t3$ cannot have users of the same seniority, so therefore the users allocated to those tasks must not be the same.

We can declare a new function for seniority which takes two users as parameters and returns a boolean whether the users are senior to each other:

$$seniority : User \times User \rightarrow Bool$$

- `(declare-fun seniority (User User) Bool)`

Given the example below, `u` is senior to `u'`.

- `(assert (seniority (u u'))`

The different types of seniority explained above are:

- t has to be allocated a user that is more senior than the user allocated for t' ($t > t'$): `(assert (seniority (alloc_user t) (alloc_user t')))`
- t has to be allocated a user that is less senior than the user allocated for t' ($t < t'$): `(assert (seniority (alloc_user t') (alloc_user t)))`
- t has to be allocated a user that is the same seniority as t ($t = t'$): `(assert (= (alloc_user t) (alloc_user t')))`
- t has to be allocated a user that is not the same seniority as t' ($t! = t'$): `(assert (not (= (alloc_user t) (alloc_user t'))))`

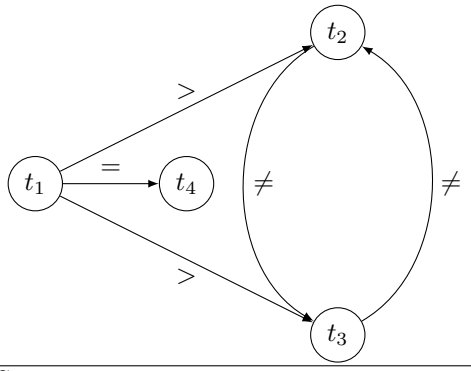
We give a scenario in Figure 3.4a where whoever is allocated receive_order has to be less senior than whoever is allocated checkout task. Also, price_large order and price_small_order have to have the same seniority as each other as they are both tasks that can be executed with the same skill. However, whoever is allocated receive_order must also be less senior than the pricing tasks. This can be expressed in Z3 as in lines 22-26:

```
(declare-sort Task)
(declare-sort User)

(declare-fun seniority (User User) Bool)

(declare-fun alloc_user (Task) User)
(declare-fun duration (Task) Real)

(assert (forall ((u User))(not (seniority u u))))

(declare-const carol User)
(declare-const bob User)
(declare-const alice User)
(declare-const checkout Task)
(declare-const price_small_order Task)
(declare-const price_large_order Task)
(declare-const receive_order Task)

(assert (seniority bob carol))
(assert (seniority alice bob))
(assert (seniority alice carol))
(assert (seniority (alloc_user checkout) (alloc_user receive_order))
   )
(assert (seniority (alloc_user price_small_order) (alloc_user
   receive_order)))
(assert (seniority (alloc_user price_large_order) (alloc_user
   receive_order)))
(assert (seniority (alloc_user checkout) (alloc_user receive_order))
   )
(assert (forall ((u User)) (or(= u carol)(= u bob)(= u alice))))

(check-sat)
(get-model)
```

(a) Z3 Seniority

```
sat
(model
  ;; universe for User:
  ;;   User!val!2 User!val!1 User!val!0
  ;; ----------
  ;; definitions for universe elements:
  (declare-fun User!val!2 () User)
  (declare-fun User!val!1 () User)
  (declare-fun User!val!0 () User)
  ;; cardinality constraint:
  (forall ((x User)) (or (= x User!val!2) (= x User!val!1) (= x User!val!0)))
  ;; ----------
  ;; universe for Task:
  ;;   Task!val!0 Task!val!1 Task!val!2 Task!val!3
  ;; ----------
  ;; definitions for universe elements:
  (declare-fun Task!val!0 () Task)
  (declare-fun Task!val!1 () Task)
  (declare-fun Task!val!2 () Task)
  (declare-fun Task!val!3 () Task)
  ;; cardinality constraint:
  (forall ((x Task))
          (or (= x Task!val!0)
              (= x Task!val!1)
              (= x Task!val!2)
              (= x Task!val!3)))
  ;; ----------
  (define-fun receive_order () Task
    Task!val!1)
  (define-fun checkout () Task
    Task!val!0)
  (define-fun carol () User
    User!val!1)
  (define-fun bob () User
    User!val!0)
  (define-fun price_small_order () Task
    Task!val!2)
  (define-fun price_large_order () Task
    Task!val!3)
  (define-fun alice () User
    User!val!2)
  (define-fun seniority!16 ((x!1 User) (x!2 User)) Bool
    (ite (and (= x!1 User!val!1) (= x!2 User!val!1)) false
    (ite (and (= x!1 User!val!0) (= x!2 User!val!0)) false
    (ite (and (= x!1 User!val!2) (= x!2 User!val!2)) false
      true))))
  (define-fun k!14 ((x!1 User)) User
    (ite (= x!1 User!val!1) User!val!1
    (ite (= x!1 User!val!0) User!val!0
      User!val!2)))
  (define-fun seniority ((x!1 User) (x!2 User)) Bool
    (seniority!16 (k!14 x!1) (k!14 x!2)))
  (define-fun alloc_user ((x!1 Task)) User
    (ite (= x!1 Task!val!0) User!val!0
    (ite (= x!1 Task!val!1) User!val!1
      User!val!2)))
)
```

(b) Z3 result for binding of duty



Symbol	Seniority
$=$	Users can be allocated this task with the same rank
$\neq$	Different ranked users
$>$	$t_x > t_y$ User who takes on t_x is more senior than t_y
$<$	$t_x < t_y$ User who takes on t_x is less senior than t_y

Figure 3.5: Seniority Relationships with tasks

3.7 Worst Time Completion

In business, it is useful to find the worst time completion in a workflow as some task executions are not helping to optimise the total workflow duration. The aim is to then find the worst time to completion after setting rules and constraints. Users usually have finite time to execute a task and in business, there are deadline to be met. This can be established if tasks have an estimate worst time duration, the longest time it could take to execute this task.

$$duration : Task \rightarrow Real$$

We will define a function for the duration of a task and each task will have a duration listed:

- `(declare-fun duration (Task) Real)`
- `(assert (= (duration t) 60))`

Task t has a worst time duration of sixty minutes to execute. Since in Z3, they do not have a time unit, the user can decide what the real value can be, i.e. minutes, hours.

3.8 Temporal Order of Task Execution

Temporal order of task execution is when tasks need to be executed in a certain order. We can easily represent this in the workflow as a directed graph, where the edges are directed to tasks (vertices) which is to be executed next. This rule is important to model in business management processes because some tasks cannot be executed before another and some tasks may be dependant on another to be executed.

In Figure 3.2, it has directional arrows between vertices to represent the temporal order of task execution. For example, we cannot price orders if we have not received orders. Similarly, we cannot proceed to checkout if no pricing has been handled on the orders. We can model temporal order of task execution in Z3 where task t is executed before t' as:

- `(declare-fun before (Task Task) Bool)`
- `(assert (before t t'))`

3.9 Task Execution (and, or, exclusive-or)

Another constraint we are going to explore is which tasks are actually executed. In some workflows some tasks are not necessarily needed to be executed. Given in Figure 3.2 we can create a fork at task receive_order and determine which pricing task is next to be executed. We can have three different types of forking:

- And - All tasks in the forking have to be executed, we cannot choose which tasks to execute.
- Or - One or more tasks in the forking have to be executed, we cannot choose to execute none.
- Exclusive or - One and only one tasks can be executed.

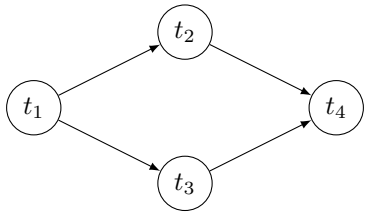
We can define a function `executed` to express which tasks are actually being executed. The function takes a task as a parameter and returns a boolean, whether the task has been executed or not.

$$executed : Task \longrightarrow Bool$$

```
1 (declare-fun executed (Task) Bool)
```

We assume that if the input has not explicitly stated that there is an or fork or an exclusive-or fork, that they are all and-forks. All tasks must be allocated and executed. So in the example workflow below, all tasks t_1, t_2, t_3, t_4 must be allocated and executed.
To express all executed tasks:

```
1 (assert (executed t))
```



To express an or fork, one or more tasks can be allocated and executed. In the example bewlow, at task t_1 either t_2 or t_3 or both can be executed and therefore allocated:

```
1 (assert (or (and (executed t1) (executed t2))(and (executed t1) (
    executed t3))))
```

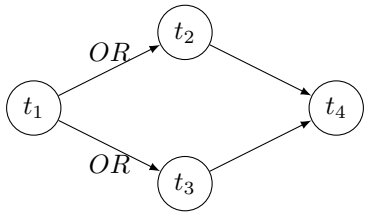


Figure 3.6: Over constrained workflow

To express an exclusive-or fork, similar to the or fork, but only t_2 or t_3 can be executed, but not both.

```
1 (assert (xor (and (executed t1) (executed t2))(and (executed t1) (
    executed t3))))
```

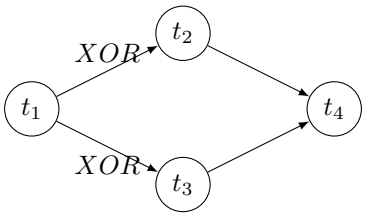


Figure 3.7: Over constrained workflow

3.10 Authorised User Allocation

In a business process, we can choose the authorise particular users to execute a task explicitly. This is similar to RBAC [11] where a system restricts access to authorised users in a domain. Authorisation is usually assigned to them by their roles. We have not implemented roles in the set of workflow constraints, but rather just users who are only allowed by input. If there are more than one users authorised to execute a task, then we can use Z3 to choose which user to execute the task:

- `(assert (or (=(alloc_user receive_order) alice) (= (alloc_user receive_order) bob)))`

So either users Alice or Bob can be allocated the task of receiving the order to be executed and only them.

Chapter 4

Implementation

4.1 Project Focus

The main aim of this project is to design a simple language in which constraints on execution of tasks within can be specified. The result that should be returned is whether the workflow is satisfiable. Also, the model returned should be easy to read and understandable to the user. The project should implement an application which would decide whether a workflow is satisfiable or not, subject to all the constraints. We also need to verify that the model returned to the user is complete and consistent to the user input.

The first discussion was whether to implement an algorithm that would solve these constraints or to use a back-end constraint solver. The benefit of using a back-end constraint solver like Z3 is that these algorithms are very efficient, perhaps more efficient than if it was implemented from scratch in this project. The application is built using Python [12] and the Z3 python module [13].

The language is a simple command line language, where a domain of tasks and users are defined. There are also predefined tokens which define certain constraints on a workflow as well as predefined axioms and rules. The application the interprets the workflow described in the language and checks if the workflow is satisfiable and gives back a suitable model.

4.2 Application Outline

The basic outline of this application is given in Figure 4.1.

- Firstly the user inputs in the command line either a file containing syntax for a workflow or command line input of a workflow.
- Then the application, which is written in Python interoperates the input.
- The lexer tokenises the input and the parser parses the input.
- After parsing, the Z3 SMT code is generated.
- The result is handled to check whether the workflow is satisfiable given the constraints.
- The resulting model is verified and certified to make sure that the resulting model is consistent with the user input.
- Depending on whether the workflow was satisfiable and whether it was verified determines if a model will be returned to the user.

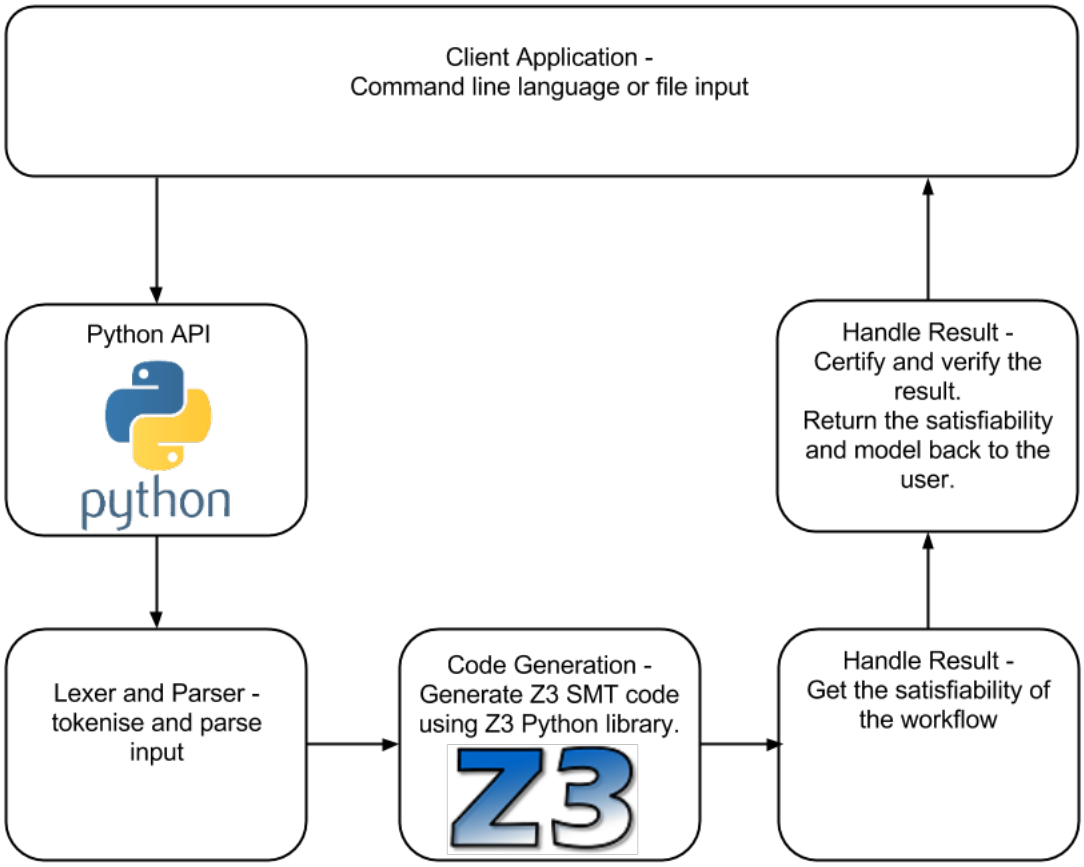


Figure 4.1: Application outline

4.3 The Language

Our requirements given was to design a language in which important constraints on the execution of tasks within a business process can be specified. It should be:

- easy to express workflows.
- include temporal execution of tasks.
- include separation of duty.
- include binding of duty.
- include constraints on allocation of tasks to users in the workflow.
- include seniority.
- include which users are authorised to execute which tasks.
- include constraints to control the execution of the workflow.

4.3.1 Tasks and Users

We have defined a basic language to fulfil these requirements. At first, the user must declare tasks and users within the domain of the workflow. In this example, the user has defined a list of tasks and users that are declared using `Tasks:` and `Users:` respectively. Following their declarations, is the list of tasks and users which are expressed using quotation marks and separated by commas.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
```

4.3.2 Separation of Duty

We can then express different constraints on the workflow. To express separation of duty, we can use a keyword `SoD:` as the separation of duty declaration for all pairs of tasks. Each task expressed in each pair has to have different user. For the given example, whomever is allocated `receive_order` must be different to the user who is allocated `checkout`.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
SoD: (’receive_order’, ’checkout’);
```

4.3.3 Binding of Duty

The constraint on binding of duty can be similarly expressed like separation of duty. The keyword for binding of duty is `BoD:` which delcares the list of pairs of tasks which must have the same user. In this example, whomever is allocated `price_large_order` must be the same user as the user who is allocated `price_small_order`.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
BoD: (’price_large_order’, ’price_small_order’);
```

4.3.4 Seniority

There are many rules expressed using seniority. To express levels of seniority, we can again, give a list of pairs of users in this binary relation. The first user of their pair is more senior than the second user. We can use the keyword `Seniority:` to declare pairs of users who are senior to each other. In this example, we can see that Alice is more senior than Bob and Carol, and Bob is more senior than Carol.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
Seniority: (‘alice’, ‘bob’), (‘alice’, ‘carol’), (‘bob’, ‘carol’);
```

To use the seniority rules we have just specified, we can now set tasks to have relative seniority to other tasks within the workflow. These are given as single task options with `-min_sec_lv:` as an option flag, followed by a list of tasks.

```
Tasks: ‘receive_order’ -min_sec_lv:<[‘price_large_order’, ‘
   price_small_order’, ‘checkout’], ‘price_large_order’ -min_sec_lv
   :>[‘receive_order’] -min_sec_lv:=[‘price_small_order’], ‘
   price_small_order’ -min_sec_lv:>[‘receive_order’] -min_sec_lv:=[‘
   price_large_order’], ‘checkout’ -min_sec_lv:>[‘receive_order’];
Users: ‘alice’, ‘bob’, ‘carol’;
Seniority: (‘alice’, ‘bob’), (‘alice’, ‘carol’), (‘bob’, ‘carol’);
```

We can specify that tasks can be:

- equal seniority to another task.
  `‘price_large_order’ -min_sec_lv:=[‘price_small_order’]`, so the task `price_large_order` has to have a user that is equal seniority to whoever is allocated task `price_small_order`. In our case, this means that they are the same user. If we specified a constraint that they have to be different users, in our case, this would be a contradiction and the workflow would not be satisfied.

- greater seniority to another task.
  `‘checkout’ -min_sec_lv:>[‘receive_order’];`, so the task `checkout` has a minimum security level that it must be allocated a user that has a greater seniority than the user who is allocated the task `receive_order`.

- less seniority to another task.
  `‘receive_order’ -min_sec_lv:<[‘price_large_order’, ‘price_small_order’, ‘checkout’]`, the task `receive_order` has to be allocated a user that is less senior than the user who is allocated the task `price_small_order`.

- not equal seniority to another task.
  `‘receive_order’ -min_sec_lv:!=[‘price_large_order’, ‘price_small_order’, ‘checkout’]`, the task `receive_order` must not be the same seniority as the user allocated the task `price_small_order`. In our case, they would have to be separate users, so having a binding of duty rule specifying that `receive_order` must be the same user as `price_small_order` produces a contradiction similarly to equal seniority.

4.3.5 Worst Time to Completion

We can express worst time completion by giving each task a duration time. This again is given as an option flag `-duration:` followed by the duration of the task. We can see in the example below that the task `receive_order` has a duration of 50. This could be any unit of time and the user must account for this by making all their duration times the same unit.

```
Tasks: ‘receive_order’ -duration:(50), , ‘price_large_order’ -
   duration:(60), ‘price_small_order’ -duration:(100), ‘checkout’ -
   duration:(10);
Users: ‘alice’, ‘bob’, ‘carol’;
Seniority: (‘alice’, ‘bob’), (‘alice’, ‘carol’), (‘bob’, ‘carol’);
```

4.3.6 Temporal Order of Execution

For expressing temporal order of execution, we can give a rule `Before:` which declares a list of pairs of tasks that have a before relationship. The first task of this pair is before the second task. In this example, the task `price_large_order` is executed before task `checkout`.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
Before: (’price_large_order’, ’checkout’);
```

4.3.7 Task Execution

There are three different task executions: and, or and exclusive-or. We can specify each task execution rule with a rule `Execution:` with the following task executions. Each task execution is followed by a pair `(t, [`t_1`,.., `t_n`])`. In this example, we have specified an `Or` execution at the task `receive_order`. Then, depending on any other constraints on the workflow, it can either execute `price_large_order` or `price_small_order` or both.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
Execution: Or(‘receive_order’, [‘price_large_order’, ‘
   price_small_order’]);
```

4.3.8 Authorised Users

We can give a list of users that are authorised to execute certain tasks and only those users. To declare authorised users to a task we can use the keyword `Authorised:`, followed by a pair, where the first of the pair is the task that is to be allocated, and the second is a list of users that are authorised to be allocated to that specific task. In this example, the only users that can be allocated the task `price_large_order` is `alice` and `bob`, but not `carol` as she has not been listed as an authorised user.

```
Tasks: ‘receive_order’, ‘price_large_order’, ‘price_small_order’,
‘checkout’;
Users: ‘alice’, ‘bob’, ‘carol’;
Authorised: (‘price_large_order’, [‘alice’, ‘bob’])
```

4.4 Lexer and Parser

4.4.1 PLY

The interpreter is designed using PLY [14] (Python Lex-Yacc) which is a python module which is a parsing tool written in Python. It is made up of two compiler construction tools: lex and yacc [15]. Yacc is sayYet another compiler compiler and lex is the lexical analyser generator. The type of parsing used in PLY is LALR(1) [16] Look-Ahead Left-Right parser.

We chose to use PLY rather than any other resources for tokenising, lexing and parsing because the library we are using for Z3 is a module for Python. This meant we had to choose a lexer and parser that is written in Python. PLY is a pure Python implementation of lex and yacc and uses a lot of python features that make it easy to use.

We considered other lexers and parsers, but unfortunately, their specifications did not match with what we needed for this project:

- Antlr [17] - It has a good lexer and parser with regular expressions. It generates a parsing tree visitor which is useful for code generation. However, the language that we have developed in this project is quite basic and does not need such complicated tools.
- Pyparsing [18] - It has an alternative approach to creating and executing simple grammars. It is different to traditional lex and yacc approach and does not use regular expressions. Therefore it may have been too heavy weight for this project. The main aim of this project is to focus on simple grammars with regular expressions.

4.4.2 Lexer

The lexer uses the PLY lexer module. It takes in a list of tokens and reserved words that are exclusive to the language which the user is not allowed to use. We will later use these tokens and reserved words to apply parsing rules to them. Reserved words are keywords that are used in the syntax and the tokens are symbols. It can also detect illegal syntax.
We have a list of reserved words for the language:

- `Tasks` - Keyword to start the list of tasks
- `Users` - Keyword to start the list of users
- `Before` - Keyword to start the list of pairs of tasks temporal execution
- `SoD` - Keyword to start the list of pairs of tasks that are bound to separation of duty
- `Seniority` - Keyword to start the list of pairs of users that are senior to each other
- `BoD` - Keyword to start the list of pairs of tasks that are bound to binding of duty
- `users` - Keyword option for all users
- `allocate` - Keyword to request of an allocation
- `min_sec_lv` - Keyword option for giving a minimum security level seniority for a task
- `Or` - Keyword that decides the execution of a task by using an or-fork
- `Xor` - Keyword that decided the execution of a task by using a xor-fork
- `Execution` - Keyword option for the execution of tasks
- `Authorised` - Keyword option for the list of users who are authorised to be allocated to a task
- `duration` - Keyword option to state the duration of a task

We also have a list of tokens:

- `Colon` - :
- `Option` - -
- `Comma` - ,
- `LParen` - (
- `RParen` -)
- `End` - ;
- `Eq` - =
- `Lt` - <
- `Gt` - >
- `Neq` - !=
- `LSqParen` - [
- `RSqParen` -]
- `Number` - (decimal) [0-9]+[\.[0.9]+]?
- `Node` - "(\\ "|[^"])*"

4.4.3 Parser

As we have mentioned in this section earlier, PLY uses LALR(1) parsing. LALR(1) parsing is where a text is parsed according to a set of rules specified by a formal grammar.
We give an example of LALR(1) parsing:

```
begin : TASKS COLON task_node USERS COLON user_node
      | TASKS COLON task_node USERS COLON user_node rules
```

The grammar symbols (terminals) are represented using capital letters such as TASKS COLON, USERS etc. The grammar rules or identifiers (non-terminals) are expressed in lower case such as task_node, user_node, rules. They are made up of other non-terminals and terminals.

The semantic behaviour of a language is defined by syntax directed translation, where there is a rule and action for each grammar symbol. We can give an example of the language:

```
Tasks: 'receive_order', 'price_large_order', 'price_small_order',
'checkout'; Users: 'alice', 'bob', 'carol';
```

The keywords `Tasks`, `Users`, `:`, `;` and `,` are all terminals and are shifted when parsing as they do not have any grammar rules. However, the non-terminals such as `'receive_order'` and `'alice'` have grammar rules associated to them and are reduced to the rules `task_node` and `user_node` respectively. In the example, we have given a list of tasks, which are separated by a comma, `'receive_order'` is reduced to a `NODE` terminal, the comma `,` is then reduced to the comma, which matched the second rule in `task_node`. The next part of the rule is `task_node` which recurses the `task_node` rule and again, reduces the next task node `'price_large_order'` and so on. When we reach the last task node `'checkout'`, we reach the first rule `'end'`.The parser then goes to the end rule and reduces the `END` terminal as `;`. We can see that we have reached the end of the `task_node` rule and is finally reduced to a `task_node` non-terminal. A similar process is done when we reach the non-terminal `user_node`. When the list of users is reduced to a `user_node`, the rule is then reduced to the non-terminal `begin`. The whole input has been parsed as the first rule in `begin`.

```
task_node : NODE end
          | NODE COMMA task_node
          | NODE variable_task_option
user_node : NODE end
          | NODE COMMA user_node
          | NODE user_option
          | NODE end_rule
end : END
    | END begin
```

PLY does some error handling for the lexer and parser

As the input is being parsed, reduced and shifted, these are being added to a symbol table ready to generate the correct code. The symbol table is made up of dictionaries, where each entry has a key and value.

4.5 Code Generation

We need to be able to generate the correct code so that we can parse it to Z3 correctly to get the satisfiability of the workflow as well as the model if applicable. Not only must we generate the correct code, but we have to have to correct axioms for each of the rules we have specified in the language that we do not need the user to know about or explicitly input.

The code is generated in such a way to make sure the ordering complies with Z3. We must declare all sorts before using them to define functions and constants in the theory. So in this implementation, it is best to declare all the sorts first, `Tasks` and `Users`. This gets rid of any unexpected sorts that Z3 might not recognise and return an error. We can then safely declare functions and constants with the correct types and finally any rules and axioms that are in that push-pop frame.

4.5.1 Before Axioms

Since a workflow is an acyclic and directed graph, tasks cannot have any cycles. This is an applicable axiom as a task that has been executed cannot be executed again in our case. This makes defining workflows a lot easier.

$$\forall t : Tasks(\neg before(t, t))$$

```
(assert (forall ((t Task))
(not (before t t))))
```

The `before` function is also transitive.

$$\forall t, t', t'' : Task(before(t, t') \wedge before(t', t'')) \longrightarrow before(t, t'')$$

```
(assert (forall ((t1 Task) (t2 Task) (t3 Task))
(=> (and (before t1 t2) (before t2 t3))
(before t1 t3))))
```

4.5.2 Seniority Axioms

The `seniority` function is transitive as it is a hierarchy.

$$\forall u, u', u'' : User(seniority(u, u') \wedge seniority(u', u'')) \longrightarrow seniority(u, u'')$$

```
(assert (forall ((u1 User) (u2 User) (u3 User))
(=> (and (seniority u1 u2) (seniority u2 u3))
(seniority u1 u3))))
```

This function is also acyclic because in a business, there cannot be a user who is both senior and junior to another user in the domain, so therefore they cannot be senior to themselves.

$$\forall u : User(\neg seniority(u, u))$$

```
(assert (forall ((u User))
(not (seniority u u))))
```

4.5.3 Execution of Tasks in Worflows

In the workflow, the user can define different forms of forking: and, or and exclusive-or. This means that there are different possibilities of tasks which are actually allocated and therefore executed in the workflow. For example, if we have a separation of duty within a workflow and there is an or fork before this constraint, we might not need to actually execute those tasks under that constraint. An example of this workflow is given in Figure 4.2. The dotted lines show the tasks that are constrained under separation of duty, so $SoD(t_2, t_3)$ and $SoD(t_2, t_4)$. So if task t_3 is chosen to execute as task t_2 cannot be allocated or would be unsatisfied, then we do not need to worry about the separation of duty constraints in place as those tasks under it are never executed. This means that we do not need to over constrain the workflow, making task allocation easier.

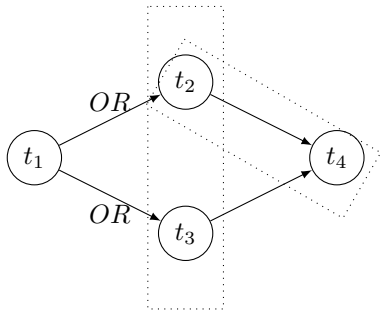


Figure 4.2: Over constrained workflow

We introduce an auxiliary user "bottom" who a user and is allocated tasks which cannot be executed. This is an in-system user who has been predeclared without the user explicitly listing it.

```
(declare-const bottom User)
```

We must ensure that all the input users are different to bottom, otherwise bottom could be allocated to tasks which actual users in the domain could be allocated to. We need to keep this separate. So we make sure that all users in the domain must be more senior than bottom:

$$\forall u : User((u \neq bottom) \longrightarrow ((\neg seniority(bottom, u) \wedge (\neg seniority(u, bottom))))$$

```
(assert (forall ((u User))
=>(not(= u bottom))
(and (not(seniority bottom u)) (not(seniority u bottom))))))
```

But `bottom` must be able to be allocated to other tasks in the case that the task is not executable:

$$alloc\_user(t) = bottom$$

```
(assert(=(alloc_user t) bottom))
```

Firstly, we have to make sure that if a task is executed, the it is not allocated to the bottom user since the bottom user is there as a user who is allocated all the unallocated tasks with the user domain.

$$\forall t : Task(executed(t) \longrightarrow (alloc\_user(t) \neq bottom))$$

```
(assert (forall ((t Task))
(=> (executed t)
(not(=(alloc_user t) bottom)))))
```

We have to also make sure that all the not executed tasks are allocated to the bottom user.

$$\forall t : Task(\neg executed(t) \longrightarrow (alloc\_user(t) = bottom))$$

```
(assert (forall ((t Task))
(=> (not(executed t))
(=(alloc_user t) bottom)))
```

With these axioms in place, we can make sure that only the executed tasks are allocated to the domain of given users and the unexecuted tasks are allocated to the bottom user. So now, we wrap all our previous axioms and rules with the executed axiom.

For the seniority rules and axioms, we must wrap them with the execution rule so that we know which tasks are allocatable to the domain of users, and which tasks are not.
For two tasks to be executed by users of the same seniority (the same user):

$$(executed(t) \wedge executed(t')) \longrightarrow (alloc\_user(t) = alloc\_user(t'))$$

```
(assert
(=>(and (executed t) (executed t'))
(=(alloc_user t) (alloc_user t'))))
```

Similarly to two tasks having to have the same level of seniority. When two tasks have different seniority, they have to have different users:

$$(executed(t) \wedge executed(t') \longrightarrow (alloc\_user(t) \neq alloc\_user(t')))$$

```
(assert
(=>(and (executed t) (executed t'))
(not(=(alloc_user t) (alloc_user t')))))
```

Greater and less than seniority are shown below respectively where `(assert (seniority t t'))`

$$(executed(t) \wedge executed(t')) \longrightarrow seniority(alloc\_user(t), alloc\_user(t')$$

```
(assert
(=> (and (executed t) (executed t'))
(seniority (alloc_user t) (alloc_user t'))))
```

$$(executed(t) \wedge executed(t')) \longrightarrow seniority(alloc\_user(t'), alloc\_user(t)$$

```
(assert
(=>(and (executed t') (executed t))
(seniority (alloc_user t') (alloc_user t))))
```

We also need to wrap separation of duty and binding of duty around an executed rule which are shown below respectively.

$$(executed(t) \wedge executed(t')) \longrightarrow (alloc\_user(t) \neq alloc\_user(t'))$$

```
(assert
(=> (and (executed t) (executed t'))
(not (=(alloc_user t) (alloc_user t')))))
```

$$(executed(t) \wedge executed(t')) \longrightarrow (alloc\_user(t) = alloc\_user(t'))$$

```
(assert
(=> (and (executed t) (executed t'))
(=(alloc_user t) (alloc_user t'))))
```

4.5.4 Unique Users

We have to ensure that all users are unique from each other. Z3 may not be able to execute some of the axioms and it may not be able to tell users apart and therefore not be able to allocate users to tasks. We have provided two axioms to solve this issue: to explicitly declare that each user is unique to each other and that they are the only users in the set, and therefore the only users in the universe.

To declare that all users are unique to each other, where $u, u' \in Users$:

$$u \neq u'$$

```
(assert (not(= u u')))
```

To declare that the users listed are the only users in the universe including the `bottom` user, where $u, u', u'' \in Users$:

$$\forall u, u' : User((u = u') \vee (u = u''))$$

```
(assert (forall ((u User))
(or (= u u') (= u u''))))
```

4.5.5 Worst Time Completion

Another feature we included in this tool is to generate the worst time completion of a workflow. We need to generate this at the end after we check if the workflow is satisfiable. If it's unsatisfiable, then there is no need to check the worst time to completion as the workflow is not completed.

At first, we need to declare a constant that can represent the worst time in the workflow. The constant is a `Real` as the duration could possibly be an integer or a decimal.

```
(declare-const completion_time Real)
```

To obtain the completion time, we need to only sum the duration of all the tasks that have been executed in the workflow. To do this in Z3, we need to check each of the tasks in the workflow to see if they have been executed. If they have been executed, then they should be summed, otherwise, they should not be in the sum. To achieve this, we need to use an "if-then-else" (`ite`). If they are executed, then sum up the tasks' duration time, otherwise it's duration time should be zero as a task that has not been executed has no duration time in the satisfied workflow.

$$completion\ time = \sum_{executed(t)=true} duration(t)$$

```
(assert (= completion_time
(+ (ite (executed t) (duration t) 0)...
   (ite (executed tn (duration tn) 0)))))"
```

Now that we have a completion time, we can write an algorithm that will do an unbounded search on the worst time completion. We pass the completion time through as a parameter in Z3 with `delta` which is the limit where the algorithm will stop the bisection. To find the upper bound time, we keep checking that the completion time can be greater than double the models completion time. When we reach this point, we know that there is no longer a time where it could be worse and therefore we have reached the upper bound completion time.

Once we have the upper bound and the lower bound completion times we can begin the bisection. If the completion time is greater than the bisection of the upper and lower bounds, then the lower bound completion time should be set as the value of the bisection. However, if it's unsatisfiable, it means there is no completion time greater than the value of the bisection, so we must search the lower half of the bisection, setting the upper bound to the value of the bisection. We recurse though this bisection until we reach the bijection limit `delta`. We have then found the worst time completion as $\frac{upperbound+lowerbound}{2}$

```
def worst_time_completion(x, delta, s):
        res = s.check()
        if res == unsat:
            return unsat
        else:
            m = s.model()
        # Finding the upper bound time
        x_s = Real(x)
        # Unbounded search
        while res == sat:
            s.push()
            s.add(x_s > 2*m[x_s])
            res = s.check()
            if res == sat:
                m = s.model()
            s.pop()
        # Bisection
        v = m[x_s]
        v = float(v.as_decimal(10)[:])
        max_time = 2*v
        min_time = v
        while (max_time-min_time) > delta:
            s.push()
            s.add((((max_time - min_time)/2)+min_time) <= x_s)
            res = s.check()
            if res == sat:
                min_time = (((max_time-min_time)/2)+min_time)
            else:
                max_time = (((max_time-min_time)/2)+min_time)
            s.pop()
        y = (max_time+min_time)/2
        return y
```

4.6 Handling The Result

4.6.1 Making It Human Readable

4.7 Validation and Completness of Z3 Model

Chapter 5

Evaluation

Chapter 6

Conclusion

Chapter 7

Appendix A: How to Use and Result Returned

Bibliography

[1] Mark Senn. *Acyclic Digraph*, (accessed February, 2015). http://mathworld.wolfram.com/AcyclicDigraph.html.

[2] Leonardo de Moura and Nikolaj Bjrner. Satisfiability modulo theories: An appetizer. pages 1–8, 2009. http://research.microsoft.com/en-us/um/people/leonardo/sbmf09.pdf (accessed Feburary, 2015).

[3] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selmanr. Satisfiability solvers. *Handbook of Knowledge Representation*, pages 91–92, 2008. http://www.cs.cornell.edu/gomes/papers/satsolvers-kr-handbook.pdf (accessed Feburary, 2015).

[4] Microsoft Research. *Z3 Codeplex*, 2014. http://z3.codeplex.com/, (accessed February, 2015).

[5] SRI International's Computer Science Laboratory. *The Yices SMT Solver*, 2015. http://yices.csl.sri.com/, (accessed May, 2015).

[6] *CVC4 - The SMT Solver*, 2015. http://cvc4.cs.nyu.edu/web/, (accessed May, 2015).

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013. http://mathsat.fbk.eu/index.html, (accessed May, 2015).

[8] John Gregg, Michael Nam, Stephen Northcutt and Mason Pokladnik. *Separation of Duties in Information Technology*, 2015. http://www.sans.edu/research/security-laboratory/article/it-separation-duties, (accessed Feburary, 2015).

[9] Sigrid Schefer, Mark Strembeck, Jan Mendling. Checking satisfiability aspects of binding constraints in a business process context. pages 465–466, 2012. (accessed Feburary, 2015).

[10] Jason Crampton, Michael Huth, Jim Huan-Pu Kuo. Authorized workflow schemas: deciding realizability through ltl(f) model checking. pages 1–3, 2013. (accessed November, 2015).

[11] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. pages 1–2, 1992. http://research.microsoft.com/en-us/um/people/leonardo/sbmf09.pdf (accessed May, 2015).

[12] Python. *Python*, 2015. https://www.python.org/, (accessed December, 2014).

[13] Microsoft Research. *Z3*. http://research.microsoft.com/en-us/um/redmond/projects/z3/z3.html, (accessed December, 2014).

[14] PLY. *PLY (Python Lex-Yacc)*. http://www.dabeaz.com/ply/, (accessed May, 2014).

[15] David M. Beazley. *PLY (Python Lex-Yacc)*, 2009. http://web.cs.dal.ca/~sjackson/lalr1.html, (accessed May, 2014).

[16] Stephen Jackson. *A Tutorial Explaining LALR(1) Parsing*. http://www.dabeaz.com/ply/ply.html, (accessed May, 2014).

[17] Terence Parr. *Antlr4 - Python Target*, 2014. https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Python+Target, (accessed Feb, 2014).

[18] Pyparsing. *Pyparsing*, 2015. https://pyparsing.wikispaces.com/, (accessed March, 2014).