Imperial College London

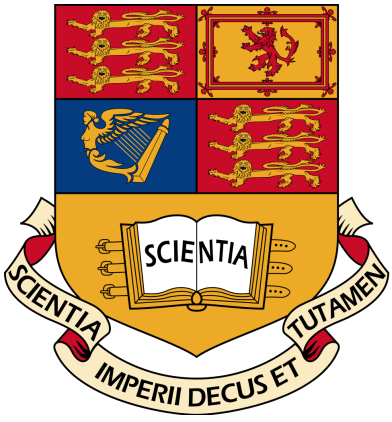
Individual Project Report

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 14, 2015

Abstract

Trying to verify if a workflow is satisfiable given that there are constraints and restrictions in place, and working out how and if a task can be executed are hard problems. As businesses and companies expand and have to become more compliant to rules and regulations, an interesting problem needs to be solved: can all the relevant tasks be executed without breaking these constraints? Otherwise if these constraints are violated, companies might incur a financial penalty.

We have tried to solve this problem using rules written in first order predicate logic to generalise the workflow and generate the relevant rules and axioms to test the satisfiability. We will then see if a set of users can be allocated to actually execute the relevant tasks.

We have developed an application which can handle an user input description of a workflow including its constraints. Then we must check whether the workflow is still satisfiable with the given constraints. Finally, we verify what we return back is complete and correct.

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

1.1 Project Aims

Over the years, enterprise systems [1], which are are tools that integrate different number of applications, tools and formats, have been growing as businesses expand. This means there is an increased need to support business processes, information flows and reporting as they become too large to manually handle. Business processes are processes that coordinate the workflow of tasks whose execution realises a business objective. Companies are looking to make sure that their systems are protected from fraud i.e. increasing cybersecurity, and follow compliance rules and regulations that are set by governing bodies. Otherwise, potential implications could mean financial loss or negative media attention.

Most companies today need to monitor these processes and keep logs of these workflows and their results to make sure that they are being compliant and that they are on record. There are many companies that already handle enterprise systems (SAP, IBM), especially in cybersecurity.

Many opportunities and threats occur within a business process if inconsistencies exist with their constraints and allocations. There are many stories in the news where businesses are constantly being attacked, where data and money can be stolen, or that businesses or individuals are fined huge sums of money for breaking compliance rules set by a particular governing or regulating body. A well known example of a compliance rule is the"Chinese Wall" [2] which is an information barrier that a business may have in order to prevent communication and exchanges of information. It may lead to a conflict of interest within an organisation, and therefore conflicts with a regulator.

For this reason, this project will focus on modelling these business processes, as it is very important that companies comply with rules and regulations to avoid financial penalties and risk damaging their reputation. A recent example of this is companies failing to to obey the compliance rules over social media [3]. Being online, it is hard to be compliant with rules from different countries without possibly breaking one. Therefore the constraints become larger and tighter within the process.

There are two types of analysis that businesses use to assess their workflows: dynamic and static analysis [4]. In this project we will be developing a tool which will implement static analysis. Static analysis is where we examine the workflow without ac-

tually executing the workflow first. However, with dynamic analysis, testing and running the workflow is performed during run time, as tasks are being executed. What is useful with the dynamic analysis technique is that we would potentially take log files of previous workflows and do audits on them to check for compliance and to make sure there are no defects or vulnerabilities which cannot be found during static analysis. These log files can become very large; over 50GB, but they need to be kept as some checks may need to be done to test their compliance to rules and regulations. We will mainly focus on static analysis because we want to explore all the possible execution paths, not just the execution path during run time. This is to ensure that there are no possible vulnerabilities to break the constraints.

The aim of the project will focus on how business processes coordinate the workflow of tasks whose execution relies on a business objective. The user should be able to input a workflow with some constraints. The application should then generate an output to say whether the workflow does not break any of the constraints and return the task executions. The user should specify these particular constraints as each workflow is different.

In this application, we have chosen to only cover a subset of compliance rules as there are so many and would be unrealistic to cover all the different types of rules. The description of the workflow is an using a basic language. We will show how to use this language in Appendix A.

1.2 Approach

At first, we decided that the application should be a basic language that should be able to describe a workflow along with its constraints rather than focusing on a user interface or a web application. We needed to make sure that the language would be easy to use and can effectively express a workflow.

For the back-end we could have implemented our own algorithm to solve these constraints over a workflow. But there are many solvers that are used for general constraint solving where the user inputs their own constraints on formulas and theories. Constraint solvers are easy to use to validate rules and constraints subject to a workflow. We need to be able to take the input and generate the correct code for the constraint solver to be able to solve a particular workflow. A reason why we have decided not to implement our own algorithm is that these constraint solvers are generally very efficient, and some scale very well with an increasing number of constraints. This is what we need for this project so we can focus on some of the rules and axioms that we need to produce for each workflow.

There are two stages to this application: to generate the code for the constraint solver so it is able to check if the workflow satisfies all the given constraints, and to verify that these results are correct, complete and consistent with the user input. If it is not complete, there must be something wrong with either the code that was generated and passed to the constraint solver, or that there is perhaps something wrong with the constraint solver itself that it cannot solve this problem.

The result that we receive back from the constraint solver must be easy to read. As the workflows get larger and there are more constraints, the user must need to know what was executed in the workflow, which user was allocated to carry out the execution and under what constraints.

Overall, this can become a very large problem especially for businesses with a lot of compliance rules and regulations, therefore that we hit the "P vs. NP" problem. The larger the workflow and the more rules in place, the harder it is to test if all the rules are obeyed. We also need to be able to certify that the workflow is consistent with the result returned from the constraint solver.

There has been a lot of previous work on *workflow satisfiability* [5] which studies the problem of determining whether there exists a workflow specification while simultaneously complying with constraints. We take a couple of examples from this paper to help build this application and assist us to construct a workflow and its constraints. However, most of the examples we use in this project and workflows to test are primarily invented for this project. Businesses are reluctant to share their workflows as they could reveal security vulnerabilities or are intellectual property of a company.

1.3 Accomplishments

The main accomplishments that we have achieved are:

1. A basic language to represent a workflow as well as the user constraints which can be translated into a language readable by our constraint solver of choice.
2. Create the relevant rules and axioms for each different constraint in the workflow.
3. Return to the user whether all the constraints in the workflow has been satisfied. Also, we must return to the user the results of these constraints and what has been executed within the workflow.
4. A method to generate the corresponding code from the user input to the constraint solver language, with all the correct and corresponding rules and axioms in place.
5. We need to verify the results returned by the constraint solver to make sure the it is correct and complete in respect to the user input. This ensures that the rules and axioms generated are the correct ones.

1.4 Report Structure

Chapter 2: This section describes the background knowledge and insight into what is needed to understand this report. We will explain in more detail what workflows are and how we can express them. We will also show some main concepts of the Z3 language that we will be using throughout this report as our choice of constraint solver.

Chapter 3: This section describes and explains some of the business rules we have implemented in this project. We also show how these can be expressed in the Z3 language.

Chapter 4: In this section we discuss the implementation of the application. This includes creating the language, lexer, parser and code generator. We will also discuss how we can take the users description of a workflow using the implemented language and how we give a useful output back to the user.

Chapter 5: In the evaluation chapter, we discuss whether the constraint solver of choice is useful in real world applications and if it can scale well since theses workflows can become huge including verification of the result. We will also be reviewing if we have achieved all our original objectives.

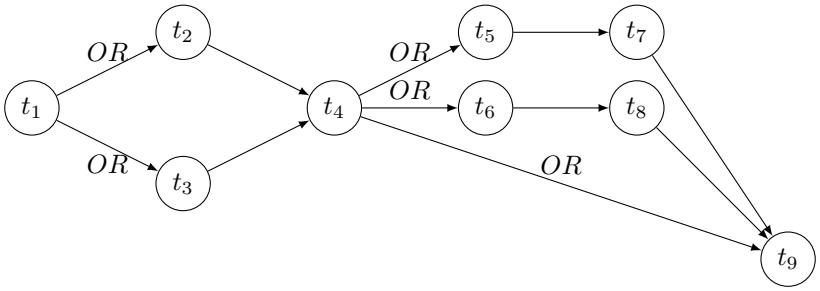
Chapter 6: Finally, the conclusion will analyse what we have actually achieved in this project as well as any interesting future work that can be accomplished on this project.

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 and output a certain goal. 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 and hard problem as more tasks are added to a workflow. Workflows are modelled as directed acyclic graphs [6], 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.



(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

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 employee depending on the two possible sizes of the order and are priced 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 the graph is forked and the constraints placed upon it. 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, a workflow is made up of tasks that need to be executed by users who are allocated to them. 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.

2.1.2 Users

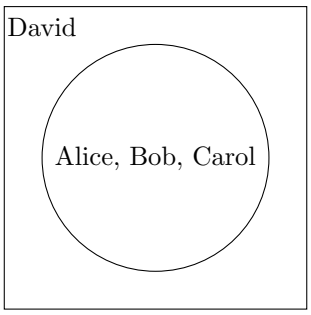
A user u belongs to the set of users who can be allocated to tasks in order to execute them. However, there are possible allocation constraints which refrain particular users from executing these tasks. If we have a universe of users, then we can consider two cases where S is a set:

- 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, so they are mapped to 1, but David has not and so he is mapped to 0.

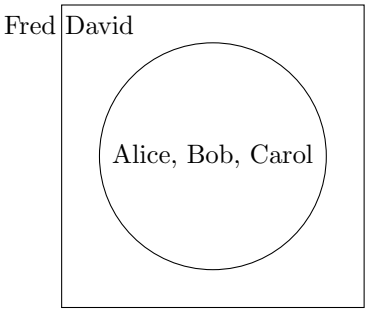
$$S \triangleq f : Universe \longrightarrow \{0, 1\}$$

- Or that was have an extra case where we are unsure whether they can be allocated, which is represented by ?. Looking at Figure 2.2b, we can see that Alice, Bob and Carol have been allocated in the universe of the workflow 1, David is mapped to 0 as he is not allocated, but Fred could be allocated into the workflow, so he is mapped to ?.

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



(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. 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. However, they 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) [7] 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) [8] , 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 [9] 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 contains 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 [10] - 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 [11] - 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 [12]- It is lacking a lot of features in Z3, especially the quantifiers.

2.4.1 Basics

Z3 SMT solver is a theorem prover which can check for the satisfiability of logical formulas over theories. Theories are a set of axioms, and axioms are a logical sentences with no free variables. In Z3, using declared variables are represented as existentials $\exists$.

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` All sentences are declared as assertions, regardless if they are existential or universally quantified.
- `(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 can be evaluated, and `unsat` if the theory is unsatisfiable and cannot be evaluated.
- `(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, they may never return a value or raise or throw exceptions, uninterpreted Z3 functions have no side effects since they are in classical first order logic and are total. A function that is total is defined for all input values for that function. Everything in Z3 is a function, including constants as they dont take in arguments.

- `(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 as uninterpreted functions only have two properties: its function name and its arity. 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

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

(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

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 21. Else, if it is neither the case, 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, real and boolean constants.

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

(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

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 or most recent push on the stack is removed from the stack, and the interpretation is reverted back to its previous state before the push.

In Figure 2.5, shown in lines 1-9, the theory is satisfied before the push. However, after the stack is pushed and the assertion $x < 2$ is added, this assertion violates the constraints already in the current frame which is $x > 2$, which was previously pushed onto the stack frame. Thus 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 at line 8.

This is useful for when we want to do some testing to see if some variables and constraints hold within a theory. We will be using these Z3 method calls for verifying the correctness and completeness of the model as well as some other tests.

```
1  (declare-const x Int)
2  (declare-const y Int)
3  (declare-fun f (Int) Int)
4  (assert (> x 2))
5  (assert (< y 2))
6  (assert (= (f x) (f y)))
7  (check-sat)
8  (get-model)
9  (push)
10 (assert (< x 2))
11 (check-sat)
12 (pop)
13 (check-sat)
14 (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

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. We can define out own sorts to define our own types.

- `(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" $\forall$. In Z3 SMT solver, $\forall$ is represented with the keyworkd `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:

- `(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, we were able to evaluate the theory and 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 as the theory was not able to be evaluated. 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.

2.4.7 Understanding Z3 Model Output

If the workflow is satisfiable, we can use `(get-model)` to return a Z3 interpretation that makes all the theories specified as true. However, as seen in the previous sections, these interpretations can be difficult to read. An example of a model that is returned is given

```
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

in Figure 2.7.

To interpret this model, we have an universe of tasks and an universe of users which the user has defined. The cardinality constraints for the universe of users is stating that a user can be any user in the defined elements of that universe and nothing else.

What we are more interested in is the function definitions at the bottom of the interpretation. We have mentioned previously that even constants are treated as functions that have no input. Therefore, all the constants that we define, in this case `Tasks` and `Users`. Their return type is the constant type. For example `(define-fun receive_order () Task Task!val!1)` is the defined task constant called receive_order. Z3 gives all its functions variable names. In this example `receive_order` has the result `Task!val!1`. So any use of `receive_order` will refer to it's result as `Task!val!1`.

In this example we have defined two functions, `alloc_user` adn `seniority`. Firstly, we will look at the function `alloc_user`. It is interpreted by mapping the parameter `x!1` which is type `Task` and return a user. We can map the variables in the function such as `Task!val!0` and `User!val!0` back to the results of the definitions of the constants `receive_order` and `bob` respectively. To examine the `seniority` function, Z3 has defined its own axillary functions, `seniority!16` and `k!14`. In the original function, it calls the function `seniority!16` and that function takes input of the outputs of the function `k!14` of each parameter of `seniority`. Looking at function `k!14`, it is a projection function. A projection function map the domain of the function to its relevant subset. So in this example, it is mapping the set of users defined to the set of users in the users universe [13].

```
;; 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)))
```

Figure 2.7: Z3 resulting model

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, follow regulations and ensure authorisation procedures are being adhered to. 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:

```
(declare-sort Task)
(declare-sort User)
(declare-const alice User)
(declare-const bob User)
(declare-const price_large_order Task)
(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 `receive_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)
```

Now we can allocate users to tasks. For example Alice is allocated the the checkout task:

```
(assert (= (alloc_user checkout) alice))
```

3.4 Separation of Duty

Separation of duties [14] is where users need to be different in order to complete a set of tasks. This is 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.

Formally, this is where a user who is allocated a task t, must also be a different user to a user who is allocated task t' if these tasks are bound or related. This is because knowledge from task t must not be used in order to execute task t'.

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, 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 this could lead to the creation of fake orders or authorise orders with incorrect prices which may have the intention of benefiting 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 to ensure the separation of duties.

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 in Figure 3.1b is as follows:

- Task `price_large_order` is allocated to user Alice. If we look at the function `price_large_order` which is given the variable name `Task!val!0`: `(define-fun price_large_order () Task Task!val!0)` and the function which is given to define the constant 'Alice' as `User!val!0`: `(define-fun alice () User User!val!0)`. We can map these variables to the function `alloc_user`. Examining `alloc_user` function, if `x!0` is equal to `Task!val!0` which is `price_large_order`, then the result should be `User!val!0` which is Alice in the universe.
- Task `price_small_order` is allocated to user Bob.
- Tasks `receive_order` and checkout are both allocated to user Carol.

```
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
       price_small_order))))
15 (assert (not (= (alloc_user price_large_order) (alloc_user checkout)
       )))
16 (assert (not (= (alloc_user checkout) (alloc_user price_small_order)
       )))
17 (assert (not (= (alloc_user price_large_order) (alloc_user
       receive_order))))
18 (assert (not (= (alloc_user receive_order) (alloc_user
       price_small_order))))
19
20 (assert (forall ((u User)) (or(= u alice)(= u bob)(= u carol))))
21
22 (check-sat)
23 (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 3.1: Separation of duty

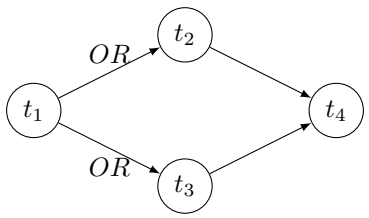


Figure 3.2: Part of the example workflow

3.5 Binding of Duty

Binding of duties [15] 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. This might be due to using knowledge from task t 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 16-17. 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.

```
(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 checkout) )))
(assert (not (= (alloc_user checkout) (alloc_user price_small_order) )))
(assert (= (alloc_user price_large_order) (alloc_user receive_order) ))
(assert (= (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 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 [5].

An example of seniority allocation is given in Figure 3.5:

- Whoever is allocated $t1$ must be the same user as whoever is allocated to task $t4$ as 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.
- 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 which are expressed in lines 22-25:

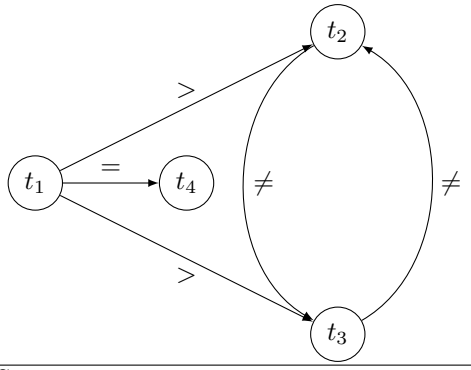
- Whoever is allocated receive_order has to be less senior than whoever is allocated checkout task.
- Also, price_large order and price_small_order have to have the same seniority as each other as they are both tasks that can be executed with the same skill.
- However, whoever is allocated receive_order must also be less senior than the pricing tasks.

```
(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.

So the worst time duration of the whole workflow would be the sum of the durations of all executed tasks.

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

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 task 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.

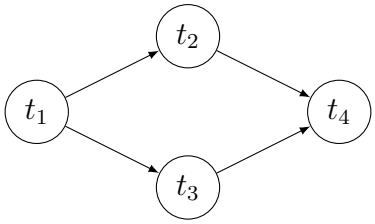
$$executed : Task \longrightarrow Bool$$

```
(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.

So to express executed tasks t:

```
(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:

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

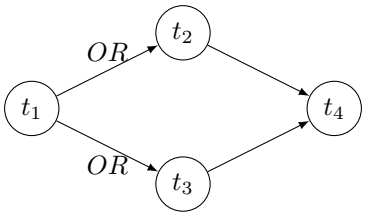


Figure 3.6: Over constrained workflow

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

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

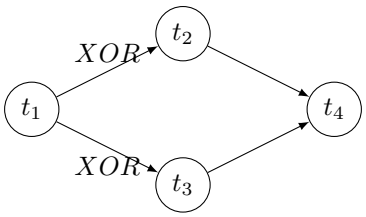


Figure 3.7: Over constrained workflow

3.10 Authorised User Allocation

In a business process, we can choose to authorise particular users to execute a task explicitly. This is similar to RBAC [16] 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. So either users Alice or Bob can be allocated the task of receiving the order to be executed and only them:

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

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 a workflow can be specified. The result that should be returned is whether the workflow is satisfiable or not, subject to all the constraints. Also, the model returned should be easy to read and understandable to the user. 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 the language Python [17] and the Z3 python module [18] as the back-end constraint solver.

The language is a simple language which can be read as a file or through command line prompt input. This is where the universe of tasks and users are defined. There are also predefined tokens which define certain constraints on a workflow as well as axioms and rules, both of which the user has specified and default ones that are for more general workflows. The application 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 the language as 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.

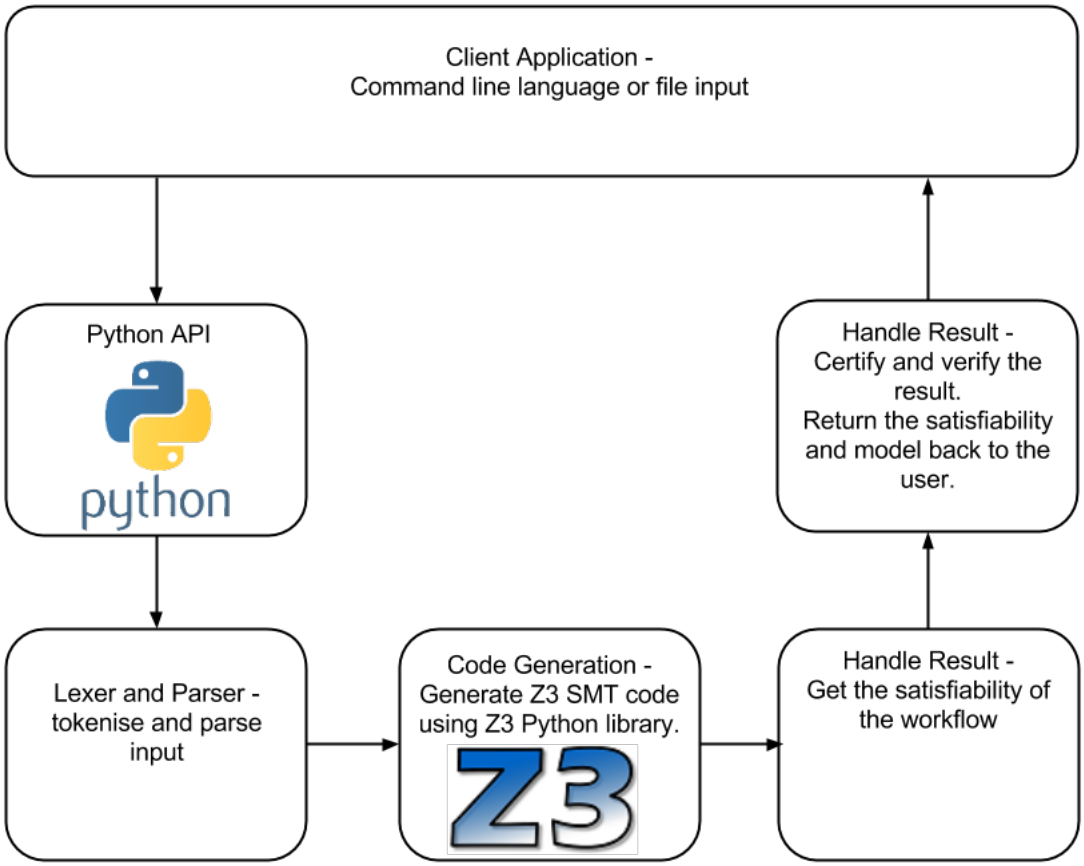


Figure 4.1: Application outline

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 users. 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 execution. Since the default is an and execution, we must specify whether it will be an Or or Xor execution. Each task execution is followed by a pair (t, [t_1,.., t_n]), where t is the task in the workflow where the forking occurs, the head of the fork. The tasks in the array [t_1,.., t_n] represent the possible tasks the execute after the head of the fork. 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'])
```

4.3.9 Verification

We enable the user to turn of verification if they are not interested in verifying that the model generated is complete with the user input.

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

4.4 Lexer and Parser

4.4.1 PLY

The interpreter is designed using PLY [19] (Python Lex-Yacc) which is a Python module. It is made up of two compiler construction tools: lex and yacc [20]. Yacc is "Yet another compiler compiler" and lex is the lexical analyser generator. The type of parsing used in PLY is LALR(1) [21] 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 and well suited for our basic language.

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

- Antlr [22] - 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 [23] - 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 allowed to use to specify rules. They are reserved for the language and the user cannot use them to define users or tasks. 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. 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
- `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 which uses regular expressions:

- `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` e.t.c. 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
```

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. The key is the task or user that we have applied a rule on, and the value is a list of tasks or users that are used within the rule.

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 the 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.

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

```
(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 junior user who senior to another user in the domain, so therefore they cannot be senior to themselves.

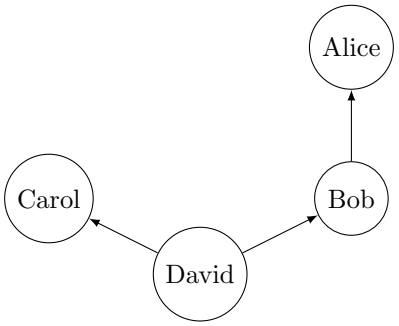
$$\forall u : User(\neg seniority(u, u))$$

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

We take note that the `seniority` function is not relational. For example, if we have:

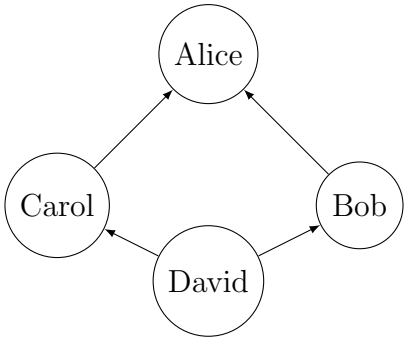
- $seniority(alice, bob)$
- $seniority(bob, david)$
- $seniority(carol, david)$

We have the seniority diagram as:



By not being relational, we cannot deduce that Bob and Carol are on the same level of seniority as each other since Alice is only senior to Bob and not senior to Carol. Therefore in our model, we express having the same seniority means that the user has to be the same user. Similarly to having different levels of seniority, it must just be the case that they are different users since `seniority` is not relational.

If we wanted to express that Carol and Bob were on the same seniority level, Alice would also need to be senior to Carol $seniority(alice, carol)$



4.5.3 Execution of Tasks in Workflows

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, $SoD(t_2, t_3)$ 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, $SoD(t_2, t_3)$ and $SoD(t_2, t_4)$ as t_2 was never executed. This means that we do not need to over constrain the workflow, making task allocation easier.

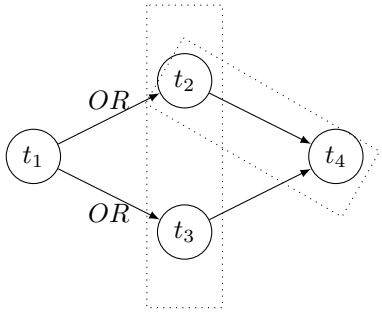


Figure 4.2: Over constrained workflow

We introduce an auxiliary user "bottom" who is 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. Therefore the user can never input a user in the universe called 'bottom'.

```
(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 not have any seniority relation to bottom, otherwise tasks could be allocated to bottom when they are executable:

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

```
(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 particular 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, that it is not allocated to the bottom user. This is due to the bottom user being a user who is allocated all the unallocated tasks.

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

```
(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 executed tasks are allocated to the bottom user. So now we must 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 executed and 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'))))
```

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 complete.

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, which is shown in Figure 4.3. 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 the

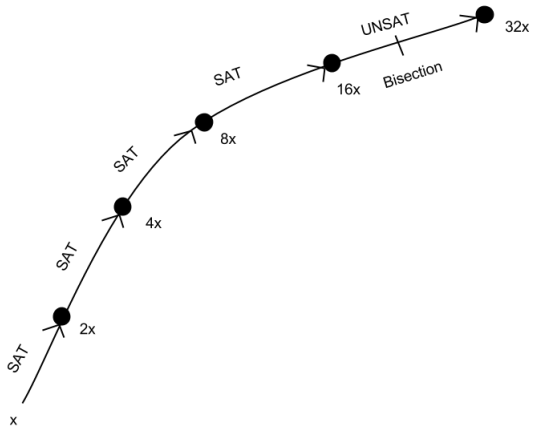


Figure 4.3: Unbound search

midpoint of the two bounds. The algorithm is shown in Figure 4.4

$$\frac{upperbound + lowerbound}{2}$$

```
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
```

Figure 4.4: Worst time completion

4.6 Validation and Completeness of Z3 Model

We need to be able to verify the completeness of the model with the actual input from the user. There could be the case the model returned from Z3 may be inconsistent even though the allocation returned back to the user may be correct. The application verifies four different aspects: separation of duty, binding of duty, seniority and lower bound worst time completion. We must check all combinations of user allocations, not including the bottom user as it is able to be allocated to all tasks, to check whether they are supposed to return satisfiable or unsatisfiable within the workflow.

4.6.1 Separation of Duty

To verify separation of duty of all tasks, we need to check that each pair of users that are allocated to the tasks that are under the constraint of separation of duty are not the same. We use the push-pop frame to test each allocation by pushing the stack and

popping the stack when we have received the result so we can continue to test other users.

In Figure 4.5, if the model itself is unsatisfied, then the verification fails and should return `false` as we cannot verify anything with an unsatisfied model. If it is satisfied, we can carry on the verification. We take each of the tasks in the separation of duty dictionary and we verify that each one is executed and that the users can be allocated the tasks. The result determines whether the model is consistent with the input. If the users are both the same and the result from `s.check()` is `sat` the verification should fail as the users who are allocated to the tasks should be different. Similarly, if the users are different and the result from the check is `unsat`, there must be some other constraint that is making this pair of users fail the verification. So we examine rules of seniority as a possibility of the failure to satisfy the allocation to the workflow.

```
1  def verify_result_sod(original, s, u):
2    verify_original = original[:]
3    verify = True
4    s.push()
5    res = s.check()
6    if res == unsat:
7      verify = False
8      return verify
9    for sod in dict_sod:
10     v = z3.parse_smt2_string(verify_original)
11     s.add(v)
12     s.push()
13     verify_original += "(assert (and (executed " + sod[0] +") (= (
           alloc_user " + sod[0] +")" + u[0] + ")))\n"
14     verify_original += "(assert (and (executed " + sod[1] +") (= (
           alloc_user " + sod[1] +")" + u[1] + ")))\n"
15     v = z3.parse_smt2_string(verify_original)
16     s.add(v)
17     if u[0] == u[1]:
18       if s.check() == sat:
19         verify = False
20         s.pop()
21       else:
22         s.pop()
23     elif u[0] != u[1]:
24       if s.check() == sat:
25         s.pop()
26       else:
27         s.pop()
28         self.verify_result_bod(original, s, u)
29   if not dict_sod:
30             verify = self.verify_result_bod(original, s, u)
31   return verify
```

Figure 4.5: Verification of separation of duty

4.6.2 Binding of Duty

Similarly to separation of duty, we must check the equality of the users that are allocated to the tasks that are listed in the binding of duty constraints.

In Figure 4.6, if the users are not the same and `s.check()` returns `unsat`, then it is breaking the binding of duty constraint and the verification should fail. However, if they are both the same user, then it should be `sat`. But if it is `unsat`, there may be another constraint on that pair of users being allocated those particular tasks. We check any seniority constraints to check the validity of the pair of users being allocated to the tasks.

```
1  def verify_result_bod(original, s, u):
2    verify_original = original[:]
3    verify = True
4    s.push()
5    for bod in dict_bod:
6      v = z3.parse_smt2_string(verify_original)
7      s.add(v)
8      s.push()
9      verify_original += "(push)\n"
10     verify_original += "(assert (and (executed " + bod[0] +") (= (
           alloc_user " + bod[0] +")" + u[0] + ")))\n"
11     verify_original += "(assert (and (executed " + bod[1] +") (= (
           alloc_user " + bod[1] +")" + u[1] + ")))\n"
12     v = z3.parse_smt2_string(verify_original)
13     s.add(v)
14     if u[0] != u[1]:
15       if s.check() == sat:
16                     verify = False
17     elif u[0] == u[1]:
18       if s.check() == unsat:
19         s.pop()
20         verify = self.verify_result_seniority(original, s, u)
21     s.pop()
22     verify_original += "(pop)\n"
23   if not dict_bod:
24   verify = self.verify_result_seniority(original, s, u)
25   return verify
```

Figure 4.6: Verification of binding of duty

4.6.3 Seniority

To check that all the users are valid with the seniority constraints, we need to check the different levels of security for each task with respect to other tasks.

We check every different constraint on the tasks as we have stored in our symbol table. For seniority constraints, we have a different dictionary for each type of security level. We test each level to make sure it complies with the user input and that it is not breaking any other constraints in the rule.

For two tasks to have equal levels of seniority they have to have the same user. So we do a similar check that we have done in separation of duty. Similarly with checking if two tasks having different levels of seniority, it is similar to the binding of duty verification.

When a task has to be allocated a user with greater seniority than another, we test if both tasks are executed with that particular allocation. If the result is `unsat` but the first user in the user pair is more senior which is compared to the seniority dictionary in the symbol table, then verification fails as it should be true. However, if the result from `s.check()` returns `sat` when the first user is less senior to the second in the user pair, verification should also fail.

When a task has to be allocated to a user with less seniority than another, we check that the task which needs to be allocated the less senior user is allocated the second user of the pair. If `s.check()` returns `unsat`, we check that the seniority is correct with the seniority input in the symbol table. If the less senior user has been allocated the task that required a less senior user but the result was `unsat` then verification should fail. If the result was `sat`, then we need to test to make sure that the user allocated to the less senior task is in fact less senior to the other user. If it is not, the verification should fail again.

```
def verify_result_seniority(original, s, u):
  verify = True
  verify_original = original[:]
  if dict_seniority:
    for t_key, t_value in dict_eq_tasks.iteritems():
      s.push()
      verify_original += "(assert (= " + u[0] + " " + u[1] + "))"
      v = z3.parse_smt2_string(verify_original)
      s.add(v)
      if s.check() == unsat:
        if u[0] == u[1]: verify = False
      else:
        if u[0] != u[1]: verify = False
      s.pop()
    for t_key, t_value in dict_gt_tasks.iteritems():
      for v in t_value:
        s.push()
        verify_original += "(assert (and (executed " + t_key + ") \
          "(executed " + v + ")" \
          "(= (alloc_user " + t_key +")" + u[0] + ")" \
          "(= (alloc_user " + v + ")" + u[1] + ")))"
        v = z3.parse_smt2_string(verify_original)
        s.add(v)
        if s.check() == unsat:
          for u_key, u_value in dict_seniority.iteritems():
            if u[0] == u_key and u[1] in u_value: verify = False
            else:
              for u_key, u_value in dict_seniority.iteritems():
                if u[0] == u_key and u[1] not in u_value:
                  verify = False
        s.pop()
    for t_key, t_value in dict_lt_tasks.iteritems():
      for v in t_value:
        s.push()
        verify_original += "(assert (and (executed " + t_key + ")" \
          "(executed " + v + ") " \
          "(= (alloc_user " + t_key +")" + u[1] + ")" \
          "(= (alloc_user " + v + ")" + u[0] + ")))"
        v = z3.parse_smt2_string(verify_original)
        s.add(v)
        if s.check() == unsat:
          for u_key, u_value in dict_seniority.iteritems():
            if u[0] == u_key and u[1] in u_value: verify = False
        else:
          for u_key, u_value in dict_seniority.iteritems():
            if u[0] == u_key and u[1] not in u_value: verify = False
        s.pop()
    if dict_neq_tasks:
      verify_original += "(assert(not(= " + u[0] +" "+ u[1] + ")))"
      v = z3.parse_smt2_string(verify_original)
      s.add(v)
      if s.check() == unsat:
        if u[0] != u[1]: verify = False
        else:
          if u[0] == u[1]: verify = False
      s.pop()
  return verify
```

4.6.4 Lower Bound Worst Time Completion

When computing the worst time completion, we need to make sure that the lower bound of the worst time completion cannot get any lower otherwise the average we take between the upper and lower bound could be worse.

```
s.push()
  duration_total = 0
  dur_tot = Real('duration_total')
  Task = DeclareSort('Task')
    for ms in m:
      if "executed" in str(ms) and "!" not in str(ms):
        for ts in tasks:
          t = Const(ts, Task)
          for mss in m:
            if "duration" in str(mss):
              duration_total = duration_total +
              m.eval(mss(t))
    s.add(dur_tot >= duration_total)
    if s.check() == unsat:
      return unsat
    s.pop()
```

Figure 4.7: Verifying worst time completion

To verify that there are no executions of tasks which fall below the lower bound and that the lower bound calculated is truly the lower bound we check the model returned using `s.model()`, shown in Figure 4.7. We check that if the task is executed, the sum of the duration time must be above the lower bound. If the total durations of all the executed tasks are less than the lower bound, then the lower bound we have calculated may not be correct and we should return `unsat`,otherwise we should return the worst time completion.

4.7 Handling The Result

Z3 has a API method call `model()` which returns a model back to the user. However, this model can be quite hard to read. With a huge workflow with many tasks and users, the universe it reports back will include all the constants and function mappings. In Figure 4.8a we can see the user input of a workflow and in Figure 4.8b the original Z3 model output.

```
Tasks: 'receive_order' -min_sec_lv:<['price_large_order', '
   price_small_order', 'checkout'] -duration:(50), '
   price_large_order' -min_sec_lv:>['receive_order'] -min_sec_lv:=['
   price_small_order'] -duration:(10), 'price_small_order' -
   min_sec_lv:>['receive_order'] -min_sec_lv:=['price_large_order']
   -duration:(100), 'checkout' -min_sec_lv:>['receive_order'] -
   duration:(10); Users: 'alice', 'bob', 'carol'; Seniority: ('alice
   ', 'bob'), ('alice', 'carol'), ('bob', 'carol');
```

(a) Command line workflow input

```
[receive_order = Task!val!2,
 completion_time = 170,
 price_large_order = Task!val!3,
 alice = User!val!3,
 elem!60 = Task!val!4,
 price_small_order = Task!val!0,
 bottom = User!val!0,
 checkout = Task!val!1,
 bob = User!val!1,
 carol = User!val!2,
 elem!908 = Task!val!2,
 before = [else -> False],
 executed = [else -> True],
 alloc_user = [else -> alloc_user!913(k!911(Var(0)))],
 seniority = [else ->
              seniority!914(k!910(Var(0)), k!910(Var(1)))],
 k!911 = [Task!val!1 -> Task!val!1,
          Task!val!2 -> Task!val!2,
          Task!val!3 -> Task!val!3,
          Task!val!4 -> Task!val!4,
          else -> Task!val!0],
 duration = [Task!val!0 -> 100,
             Task!val!2 -> 50,
             else -> 10],
 k!910 = [User!val!3 -> User!val!3,
          User!val!0 -> User!val!0,
          User!val!1 -> User!val!1,
          else -> User!val!2],
 alloc_user!913 = [Task!val!2 -> User!val!2,
                   else -> User!val!3],
 seniority!914 = [(User!val!1, User!val!2) -> True,
                  (User!val!3, User!val!1) -> True,
                  (User!val!3, User!val!2) -> True,
                  else -> False]]
```

(b) Z3 API model output

Figure 4.8: Z3 model result

4.7.1 Making It Human Readable

We need to make this output more human readable as it is difficult to map constants to functions, especially if the universe is much larger than the example given in Figure 4.8. We can automate this mapping as we finish the verification and give the output to the user.

As shown in Figure 4.10, we map the users and tasks to the corresponding Z3 declarations so it is easier to translate the model as Z3 uses their own declarations rather than the actual user and task names. We then loop through the model looking for each function and passing tasks or users to the relavent parameters of the function and use them to evaluate the function. If the function returns `True`, then we can append it to the final result dictionary.

Finally after going through the entire model, we piece together the final result with all the functions. Each function has their result translated back into the appropriate task and user names and combined to return one final output. As seen in Figure 4.9, we have a list of seniority pairs from `seniority` which is consistent with the user input. We also have the list of allocated users to tasks `alloc_user` of which those are executed. The list of executed tasks is also returned back to the user `executed_tasks`, as well as the worst time completion and the list of tasks which where given by the user in `before`.

```
worst time completion allocation:[(alice, checkout),
  (bob, price_small_order), (bob, price_large_order),
  (carol, receive_order)]
alloc_user:[(alice, checkout), (carol, receive_order),
  (alice, price_small_order), (alice, price_large_order)]
executed_tasks:[checkout, receive_order, price_small_order,
   price_large_order]
worst time completion:170.0
seniority:[('bob', 'carol'), ('alice', 'carol'), ('alice', 'bob')]
before:[]
```

Figure 4.9: Human readable output

```
def evaluate_final_model(model, total_worst_duration):
  model_user_map = { }
  model_task_map = { }
  model_result_map = { }
  Task = DeclareSort('Task')
  User = DeclareSort('User')
  for ms in model:
    if str(ms) in users:
      model_user_map[ms] = model[ms]
    if str(ms) in tasks:
      model_task_map[ms] = model[ms]
    if "before" in str(ms):
      before_task_list = []
      for t in itertools.product(tasks, tasks):
        t1 = Const(str(t[0]), Task)
        t2 = Const(str(t[1]), Task)
        before_tasks = model.eval(ms(t1, t2))
        if str(before_tasks) == "True":
          before_task_list.append(t)
      model_result_map["before"] = before_task_list
    if "alloc_user" in str(ms):
      model_list_list = []
      for t_key, t_value in model_task_map.iteritems():
        t = Const(str(t_key), Task)
        user_solution = model.eval(ms(t))
        for u_key, u_value in model_user_map.iteritems():
          if str(u_value) == str(user_solution):
            model_list_list.append((u_key, t_key))
        model_result_map["alloc_user"] = model_list
    if "executed" in str(ms):
      executed_task_list = []
      for t_key, t_value in model_task_map.iteritems():
        t = Const(str(t_key), Task)
        executed_task = model.eval(ms(t))
        if executed_task:
          executed_task_list.append(t_key)
          model_result_map["executed_tasks"] = executed_task_list
    if "seniority" in str(ms) and "!" not in str(ms):
      senior_users_list = []
      for u in itertools.product(users, users):
        u1 = Const(str(u[0]), User)
        u2 = Const(str(u[1]), User)
        senior_users = model.eval(ms(u1, u2))
        if str(senior_users) == "True":
          senior_users_list.append(u)
      model_result_map["seniority"] = senior_users_list
  model_result_map["worst time completion"] = round(
     total_worst_duration)
  model_map_str = ''.join("%s:%r\n" % (key,val) for (key,val) in
     model_result_map.iteritems())
  return model_map_str
```

Figure 4.10: Making the model more human readable

```
worst_time_completion_tasks = []
model_user_map = {}
  for ms in m:
    if str(ms) in users:
      model_user_map[ms] = m[ms]
    if "alloc_user" in str(ms) and "!" not in str(ms):
      for ts in tasks:
        t = Const(str(ts), Task)
        user_solution = m.eval(ms(t))
        for u_key, u_value in model_user_map.iteritems():
          if str(u_value) == str(user_solution):
             worst_time_completion_tasks.append((u_key, t))
    if "executed" in str(ms) and "!" not in str(ms):
      for ts in tasks:
        t = Const(ts, Task)
        for mss in m:
          if "duration" in str(mss):
            duration_total = duration_total + m.eval(mss(t))
p_c.worst_time_completion_list = worst_time_completion_tasks
```

Figure 4.11: Human readable worst time allocations

4.7.2 Including Worst Time Completion Task Allocation

What is interesting to businesses is calculating worst completion time of a workflow, so they can see how long executing a workflow can get. It is useful to give back to the user allocations to tasks lead up to this worst time completion.

When we calculate the worst time completion, we must parse the task allocations before we return from the method as this is all calculated in a push-pop frame. We parse the model similarly to how we parse the output to make it more human readable. We then take this rewritten allocation and append it to the result we return back to the user. This is shown in Figure 4.11

4.8 Unsatisfiable Core

In this report, we have talked about the satisfied core as we are interested in whether the workflow is satisfied or unsatisfied. However, it is interesting to see which assertions do fail if the workflow is unsatisfied. In Z3 we call the method `get-unsat-core`. We also need to set the SMT flag `(set-option :produce-unsat-cores true)` to make sure that we can report the unsat core. To label each constraint in SMT, we wrap each assertion with `!(constraint):named x` where x is the label. This is shown in Figure 4.12a.

The unsatisfied core reports which assertions are false within the given model. In order to state which assertions are being broken, we must label each one using `solver.assert_and_track(constraint, label)`. The label is the name we give to each assertion, so when we call `(solver.unsat_core())` we should get back an empty array `[]` if the theory is satisfied, or a list of labels of assertions that are false.

If our workflow is unsatisfiable, for example if we set the constraints as the task `checkout` has to be allocated a user who has less seniority than whomever is allocated the task `receive_order`. But we also have a constraint that whomever is allocated the task `receive_order` has to be less senior than the user chosen to be allocated the task `checkout`. As seen in Figure 4.13a, we can see that the violated constraints are the names listed which correspond back to the input:

- `(assert (! (=>(and (executed receive_order) (executed checkout)) (seniority (alloc_user receive_order) (alloc_user checkout))) :named lttask0))`
- `(assert (! (=>(and (executed checkout) (executed receive_order)) (seniority (alloc_user checkout) (alloc_user receive_order))) :named lttask1))`
- `(assert (! (executed receive_order) :named a3))`
- `(assert (! (executed checkout) :named a0))`

Which is consistent to our input.

```
(set-option :produce-unsat-cores true)
(declare-sort Task) (declare-sort User)
(declare-fun executed (Task) Bool)
(declare-fun before (Task Task) Bool)
(declare-fun seniority (User User) Bool)
(declare-const bottom User)
(declare-fun alloc_user (Task) User)
(declare-fun duration (Task) Real)
(assert (forall ((t1 Task) (t2 Task) (t3 Task))(=>(and(before t1 t2)
    (before t2 t3))(before t1 t3))))
(assert (forall ((u1 User) (u2 User) (u3 User)) (=> (and (seniority
   u1 u2) (seniority u2 u3)) (seniority u1 u3))))
(assert (forall ((u User))(=>(not(= u bottom))(and (not(seniority
   bottom u)) (not(seniority u bottom))))))
(assert (forall ((t Task))(not (before t t))))
(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 (forall ((t Task))(=> (executed t)(not(=(alloc_user t)
   bottom)))))
(assert (forall ((t Task))(=> (not(executed t))(=(alloc_user t)
   bottom))))
(assert (! (seniority bob carol) :named seniority0))
(assert (! (seniority alice carol) :named seniority1))
(assert (! (seniority alice bob) :named seniority2))
(assert (! (=>(and (executed receive_order) (executed checkout))(
   seniority (alloc_user receive_order) (alloc_user checkout))) :
   named lttask0))
(assert (! (=>(and (executed checkout) (executed receive_order))(
   seniority (alloc_user checkout) (alloc_user receive_order))) :
   named lttask1))
(assert (forall ((u User)) (or(= u carol)(= u bob)(= u alice)(= u
   bottom))))
(assert (! (not(= carol bob)) :named unique_users0))
(assert (! (not(= carol alice)) :named unique_users1))
(assert (! (not(= carol bottom)) :named unique_users2))
(assert (! (not(= bob carol)) :named unique_users3))
(assert (! (not(= bob alice)) :named unique_users4))
(assert (! (not(= bob bottom)) :named unique_users5))
(assert (! (not(= alice carol)) :named unique_users6))
(assert (! (not(= alice bob)) :named unique_users7))
(assert (! (not(= alice bottom)) :named unique_users8))
(assert (! (not(= bottom carol)) :named unique_users9))
(assert (! (not(= bottom bob)) :named unique_users10))
(assert (! (not(= bottom alice)) :named unique_users11))
(assert (! (executed checkout) :named a0))
(assert (! (executed price_small_order) :named a1))
(assert (! (executed price_large_order) :named a2))
(assert (! (executed receive_order) :named a3))
(check-sat) (get-unsat-core)
```

(a) Z3 unsatisfied core

```
unsat
(a3 a0 lttask0 lttask1)
```

(a) Z3 unsatisfied core result

Figure 4.13: Unsatisfied core result

Chapter 5

Evaluation

5.1 Results From Z3

As mentioned before, the reason why we chose Z3 SMT solver over other SMT solvers is that it is more scalable [24]. However, we need to check whether the back end of this application is scalable with Z3 as our constraint solver.

We have benchmarked our application by recording how long it takes to parse the result and get a model back from Z3 SMT solver and how long the verification takes. It is difficult to conduct this benchmark fairly as some constraints take longer to verify than others. In this test, we have a basic workflow of ten tasks and twenty users shown in the first input. We then continue to add more constraints on top of the workflow as inputs.

1. Tasks and Users:

```
Tasks: ‘t0’, ‘t1’, ‘t2’, ‘t3’, ‘t4’, ‘t5’, ‘t6’, ‘t7’,
    ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
```

   After calculating worst time completion and before verification of the model:
   00.137034s
   Returning the model after verification:
   01.720042s

2. With separation of duty:

```
Tasks: ‘t0’, ‘t1’, ‘t2’, ‘t3’, ‘t4’, ‘t5’, ‘t6’,
  ‘t7’, ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
```

After calculating worst time completion and before verification of the model:
00.069104s
Returning the model after verification:
25.576375s

3. With binding of duty

```
Tasks: ‘t0’, ‘t1’, ‘t2’, ‘t3’, ‘t4’, ‘t5’, ‘t6’,
  ‘t7’, ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
```

After calculating worst time completion and before verification of the model:
00.064547s
Returning the model after verification:
23.304050s

4. With seniority:

```
Tasks: ‘t0’, ‘t1’, ‘t2’, ‘t3’, ‘t4’, ‘t5’, ‘t6’,
  ‘t7’, ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
```

After calculating worst time completion and before verification of the model:
00.113158s
Returning the model after verification:
29.060795s

5. With seniority task constraints:

```
Tasks: ‘t0’ -min_sec_lv:=[‘t6’],
‘t1’ -min_sec_lv:>[‘t3’, ‘t2’],
‘t2’ -min_sec_lv:!=[‘t6’], ‘t3’, ‘t4’,
‘t5’ -min_sec_lv:<[‘t9’], ‘t6’ -min_sec_lv:<[‘t9’],
‘t7’, ‘t8’, ‘t9’;
```

```
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
```

After calculating worst time completion and before verification of the model:
00.100984s
Returning the model after verification:
39.805471s

6. With authorisation:

```
Tasks: ‘t0’ -min_sec_lv:=[‘t6’],
‘t1’ -min_sec_lv:>[‘t3’, ‘t2’],
‘t2’ -min_sec_lv:!=[‘t6’], ‘t3’, ‘t4’,
‘t5’ -min_sec_lv:<[‘t9’], ‘t6’ -min_sec_lv:<[‘t9’],
‘t7’, ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
Authorised: (‘t5’, [‘u6’, ‘u17’]),
  (‘t6’, [‘u6’, ‘u17’]);
```

After calculating worst time completion and before verification of the model:
00.107644s
Returning the model after verification:
91.723381

7. With execution:

```
Tasks: ‘t0’ -min_sec_lv:=[‘t6’],
‘t1’ -min_sec_lv:>[‘t3’, ‘t2’],
‘t2’ -min_sec_lv:!=[‘t6’], ‘t3’, ‘t4’,
‘t5’ -min_sec_lv:<[‘t9’], ‘t6’ -min_sec_lv:<[‘t9’],
‘t7’, ‘t8’, ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
```

```
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
Authorised: (‘t5’, [‘u6’, ‘u17’]),
  (‘t6’, [‘u6’, ‘u17’]);
Execution: Or(‘t0’, [‘t1’, ‘t2’]),
  Xor(‘t3’, [‘t4’, ‘t5’, ‘t6’]);
```

After calculating worst time completion and before verification of the model:
00.119209s
Returning the model after verification:
33.050251s

8. With duration:

```
Tasks: ‘t0’ -min_sec_lv:=[‘t6’] -duration:(10),
  ‘t1’ -min_sec_lv:>[‘t3’, ‘t2’] -duration:(90),
  ‘t2’ -min_sec_lv:!=[‘t6’], ‘t3’, ‘t4’,
  ‘t5’ -min_sec_lv:<[‘t9’], ‘t6’ -min_sec_lv:<[‘t9’],
  ‘t7’ -duration:(80), ‘t8’ -duration:(99), ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
Authorised: (‘t5’, [‘u6’, ‘u17’]),
  (‘t6’, [‘u6’, ‘u17’]);
Execution: Or(‘t0’, [‘t1’, ‘t2’]),
  Xor(‘t3’, [‘t4’, ‘t5’, ‘t6’]);
```

After calculating worst time completion and before verification of the model:
00.286741s
Returning the model after verification:
50.159936s

9. With before:

```
Tasks: ‘t0’ -min_sec_lv:=[‘t6’] -duration:(10),
  ‘t1’ -min_sec_lv:>[‘t3’, ‘t2’] -duration:(90),
  ‘t2’ -min_sec_lv:!=[‘t6’], ‘t3’, ‘t4’,
```

```
  ‘t5’ -min_sec_lv:<[‘t9’], ‘t6’ -min_sec_lv:<[‘t9’],
  ‘t7’ -duration:(80), ‘t8’ -duration:(99), ‘t9’;
Users: ‘u0’, ‘u1’, ‘u2’, ‘u3’, ‘u4’, ‘u5’, ‘u6’, ‘u7’,
  ‘u8’, ‘u9’, ‘u10’, ‘u11’, ‘u12’, ‘u13’, ‘u14’,
  ‘u15’, ‘u16’, ‘u17’, ‘u18’, ‘u19’;
SoD: (‘t0’, ‘t3’), (‘t1’, ‘t3’), (‘t2’, ‘t3’),
  (‘t7’, ‘t9’), (‘t8’, ‘t9’), (‘t1’, ‘t2’);
BoD: (‘t0’, ‘t1’), (‘t0’, ‘t4’), (‘t0’, ‘t5’),
  (‘t0’, ‘t6’), (‘t0’, ‘t7’), (‘t0’, ‘t8’);
Seniority: (‘u0’, ‘u1’), (‘u0’, ‘u2’), (‘u5’, ‘u2’),
   (‘u11’, ‘u0’), (‘u11’, ‘u5’), (‘u12’, ‘u11’);
Authorised: (‘t5’, [‘u6’, ‘u17’]),
  (‘t6’, [‘u6’, ‘u17’]);
Execution: Or(‘t0’, [‘t1’, ‘t2’]),
  Xor(‘t3’, [‘t4’, ‘t5’, ‘t6’]);
Before: (‘t0’, ‘t1’), (‘t0’, ‘t2’), (‘t1’, ‘t3’),
  (‘t2’, ‘t3’), (‘t3’, ‘t4’), (‘t3’, ‘t5’),
  (‘t3’, ‘t6’), (‘t4’, ‘t9’), (‘t5’, ‘t7’),
  (‘t6’, ‘t8’), (‘t7’, ‘t9’), (‘t8’, ‘t9’);
```

After calculating worst time completion and before verification of the model:
00.247599s
Returning the model after verification:
57.914044s

As we can see from the results from this benchmarks with twenty users and ten tasks. As we add more constraints before we do the verification on the model, it only takes Z3 about a second to generate each model which is represented by the line “Z3 SMT output” in Figure 5.2. This shows how scalable Z3 can be. As we add more constraints on the workflow, there is no substantial increase in time to compute. When we are adding more constraints, the time to solve the satisfiability of the workflow does not vary as much.

However, the verification takes a very long time. From the results of the “after verification” line in the graph, what is interesting is that when we add the `Authorised` constraint, it takes triple the time. This is probably due to checking every combination of product of users and seeing if it can be allocated to tasks `t5` and `t6` and only two users can ever be allocated to both of them. A possible reason why this trend doesn’t continue is that the next set of constraints that we add is the `Execution` rule. We do not necessarily need to execute task `t5` and `t6`. The result is satisfied in our verification and we do not need to check any further.

However, we have anticipated the increase of time with the increase of users, tasks and constraints in the workflow as we need to check all combinations of users. This becomes a hard problem. We check each test if they have been mentioned in the input. So as the number of constraints increase, the more tests we need to do to check the correctness of the model returned by Z3.

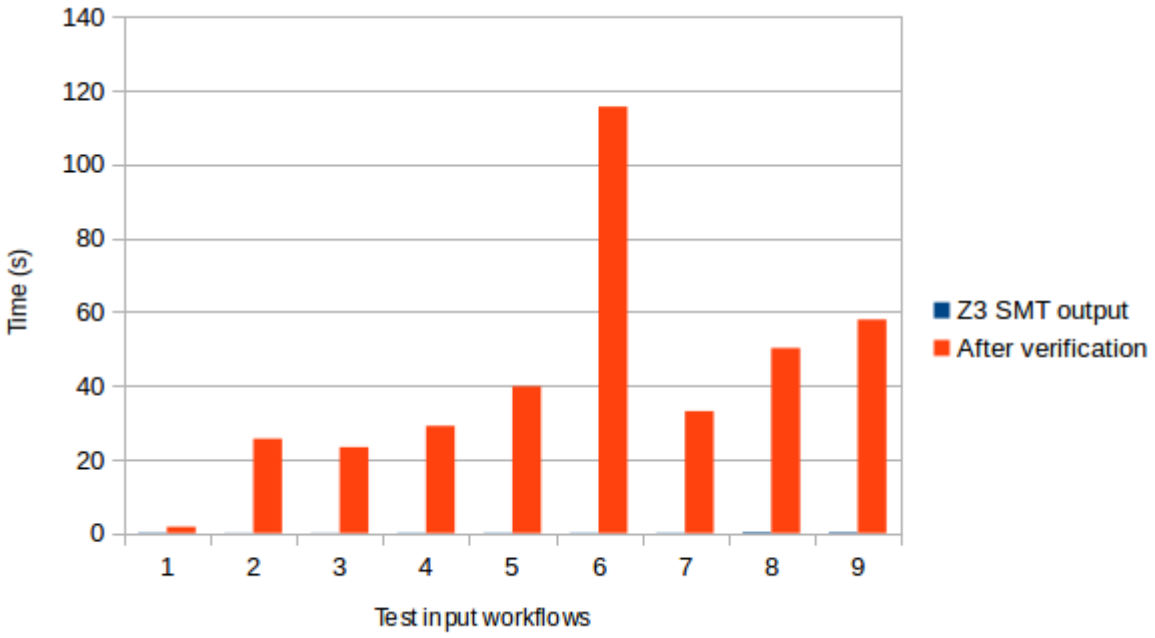


Figure 5.1: Bar chart representing the time taken to execute the tests specified. Each test is independent and takes the previous workflow and its constraints, with an additional constraint added. The blue bars represent the time taken to execute each workflow to return a satisfiable model. The red bars represent the time taken to execute the workflow but with the additional verification step.

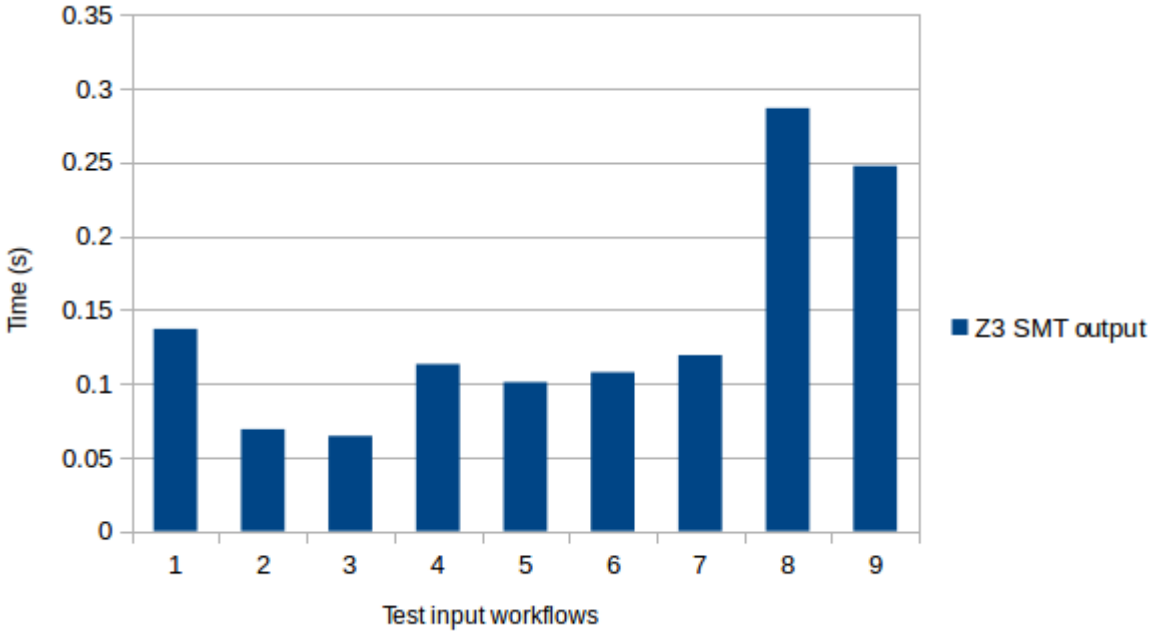


Figure 5.2: A clearer view of the times taken for each benchmark test to generate a Z3 model without verification.

5.2 Verification

Currently, the method of verification is quite slow as we have to check every combination of pairs of users in the universe besides `bottom` user. We are taking the product of each

user. At the moment, we cannot skip any combination of pairs as we need the ordering of the user pair for seniority. We interpret seniority as the first user is more senior than the second. However, we don't check every constraint if it has not been specified in the overall workflow.

If there are n users specified in the workflow, we have to check through a combination of $n \times n$ users.

Since verification takes a long time to compute as the workflow gets larger, we have implemented a feature where verification can be turned off. We have implemented this because some users may not be interested in the verification and might be more involved with the allocation of users to tasks and their executions.

5.3 Testing

It has been difficult to test this application on real industry case studies because businesses are not willing to share their secretive workflows with others and publish them online. So for this project, we have only done unit testing on some example workflows. This has enabled us to test the correctness of the logic rules and axioms within the code generated. It would be nice perhaps in the future to have some sample workflows or to do some hall way testing with some industry experts to see if this is application is what they want and is correct with much larger workflows.

Since we do not have realistic examples, we can turn to examples in papers. We have a look that an example based on seniority constraints, shown in Figure 5.3.We have the following input that describes the constraints:

```
Tasks: 'PrepC' -min_sec_lv:!=['PrepP']
  -min_sec_lv:<['AppC'],
  'PrepP' -min_sec_lv:!=['AppC'] -min_sec_lv:<['AppP'],
  'AppC' -min_sec_lv:!=['AppP'], 'AppP';
Users: 'u1', 'u2', 'u3';
Seniority: ('u1', 'u2'), ('u3', 'u1');
```

Before executing the application to solve this workflow, we anticipate that the allocation will be:
$alloc\_user(u1, PrepC), alloc\_user(u2, PrepP), alloc\_user(u3, AppC), alloc\_user(u1, AppP)$

We look at the result which is consistent with the prediction:

```
seniority:[('u3', 'u2'), ('u3', 'u1'), ('u1', 'u2')]
alloc_user:[(u2, PrepP), (u1, PrepC), (u3, AppC),
  (u1, AppP)]
executed_tasks:[PrepP, PrepC, AppC, AppP]
worst time completion:0.0
before:[]
```

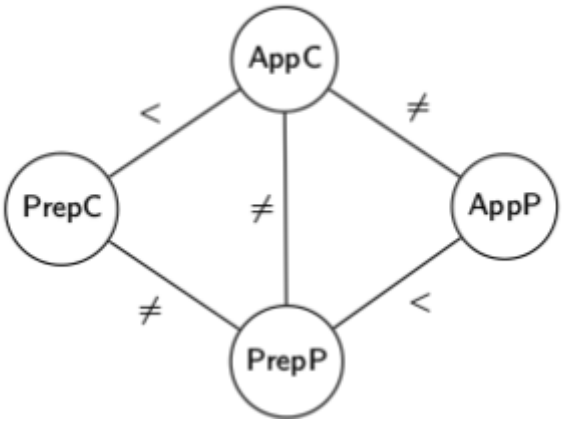


Figure 5.3: Seniority constraints on workflow

Chapter 6

Conclusion

The main aim of this project was to design a simple language in which important constraints on the execution of tasks within a business process can be specified, where our application would decide whether such a workflow is satisfiable subject to all constraints.

6.1 Achievements

As stated in the introduction, we will revisit our original accomplishments for this project.

1. A basic language to represent a workflow as well as the user constraints which can be translated into a language readable by our constraint solver of choice.
2. Create the relevant rules and axioms for each different constraint in the workflow.
3. Return to the user whether all the constraints in the workflow has been satisfied. Also, we must return to the user the results of these constraints and what has been executed within the workflow.
4. A method to generate the corresponding code from the user input to the constraint solver language with all the correct and corresponding rules and axioms in place.
5. We need to verify the results returned by the constraint solver to make sure the it is correct and complete in respect to the user input. This ensures that the rules and axioms generated are the correct ones.

We have created a language to describe a workflow and its constraints. However, we have not been able to do any user testing with industry experts who work in the field of business processes. Therefore, we are not sure if this language is descriptive enough but basic and easy to use as we have not gotten any feedback.

For the second point, we believe that we have generated relevant rules and axioms in Z3. Using unit testing, we have ensured that the result are what we expect them to be. As well as unit testing, we have looked at verifying the correctness of the resulting model that is returned from Z3 with the actual user input. This makes sure that even though what the model returns is correct, that all other possible allocations under the workflow constraints also hold.

We believe we have achieved point three as we are able to return whether the workflow is satisfiable under its constraints using Z3's `solver.check()` which returns the satisfiability of the model. Since Z3 also has a method to return the mode `solver.model()`, we are able to access the model to extract the universe and make it more human readable. Again, due to not being able to test it with industry experts, we are not too sure if this can be easily understandable to those working in business who design these workflows.

For the fourth point, we are able to generate relevant axioms for each workflow correctly and again, we can check that the model we build from code generation is consistent with the user input workflow and its constraints though verification.

Finally, we have implemented verification steps to check that the model is consistent with the user workflow. However it could probably be more efficient. Instead of looping through all possible combinations of users, some users might not need to be checked for some cases. For example, if two users are the same, there may not be a point in checking if they are able to be allocated to two tasks the are under the separation of duty constraint, but they must be checked with binding of duty and separation of duty due to the axiom that all users are unique. If we had more time, we would have implemented verification for checking the upper bound when computing the worst time completion as the worst time completion is calculated by taking the midpoint of the upper and lower bound. If the upper bound is incorrect, the worst time could also be incorrect.

6.2 Future Work

If this project were to be carried on, the most obvious implementation would be to extend the different types of constraints on the workflow. It would have been nice to include some automated constraints such as a "Chinese wall" configuration where the user would only need to specify which tasks and users are behind the wall and which are not.

Another constraint that would have been implemented if we had more time would be the number of times a task may be executed in a business.

Finally, we would have liked to have implemented concurrent or parallel tasks in the workflow to compute worst time completion. Currently, the implementation is done for sequential tasks for the forking. However, in business, if more than one path is executed, they would usually be done concurrently unless they are dependent on each other. It would have been interesting to see the worst time completion of parallel task execution.

In the implementation, we have discussed using `unsat-core` to report back to the user, a list of violated constraints. However, due to the current version of Z3 python library, we cannot currently implement this `unsat-core` feature as we are parsing direct SMT code rather than using the given python API language which is also supplied by Z3. Curently, it cannot handle labels on assertions and will skip these labels, giving us empty unsatisfied cores when we should be getting a non-empty unsatisfied core. If we had more time on this project, we would try to rewrite all the generated SMT code in the supplied

Z3 Python language. However, this can be difficult as some of these rules are generated dynamically, and constraints are not seen until run-time, making code generation harder.

Since this project represented workflows as acyclic directed graphs or flowcharts, an UI interface would have been helpful to the user as well as supplying a command line input and reading files. Below in Figure 6.1, we have designed a prototype for the UI. If we had more time, we would have implemented this as a web interface shown in Figure 6.1. The user would be able to create and drag task nodes and specify certain rules for each task, similar to the command line. The directional arrows represent the `before` function. The two circles represent `binding of duty (BoD)` and `separation of duty (SoD)`. They would also be able to specify users in the workflow domain, and also their seniority relationships to one another in the seniority section. An example is given `Alice, Bob` in the seniority window, that specifies that Alice is more senior than Bob.

6.3 Future Work for a Product

As the project stands, it is not ready to be used in industry as there are many other constraints that we have not explored that are currently conveyed in business processes. This application may have been aimed at organisations with moderately sized workflows. For example, we have defined a seniority rule where the user must input all the different users and who is senior from each other. Instead, we could have implemented roles that express different levels of seniority such as "associate" and "manager" implemented as `associate(User)` and `manager(User)` respectively. Then we could have underlying rules about which roles are senior to each other, therefore specifying seniority without the user needing to explicitly mentioning seniority. This would save a lot of time for the user when using this application.

Currently the verification process is too slow and needs to be optimised. If the workflows get too large, it could take minutes or more to evaluate the workflow.

Hopefully in a few years time and if work continued on this application to build more constraints, it could be used as a workflow constraint solver.

Figure 6.1: Application UI

Chapter 7

Appendix A: How to Use and Result Returned

As discussed, there are two ways of inputting a workflow: though command line argument and as a file. We have already discussed the language in Section 4.3.

When the user has an input, the list of tasks must be declared first, followed by the list of users. When these two are declared, the user can then choose to input constraints if they wish to. If no constraints are added, the application will simply try to check if it is satisfiable to allocate all tasks to users. IF constraints are considered, The constraints can be specified in any order.

Here is an example of input without any constraints considered:

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

This is an example of input with constraints considered:

```
Tasks: ‘receive_order’ -min_sec_lv:<[‘price_large_order’,
  ‘price_small_order’, ‘checkout’] -duration:(50),
  ‘price_large_order’ -min_sec_lv:>[‘receive_order’]
  -min_sec_lv:=[‘price_small_order’] -duration:(10),
  ‘price_small_order’ -min_sec_lv:>[‘receive_order’]
  -min_sec_lv:=[‘price_large_order’] -duration:(100),
  ‘checkout’ -min_sec_lv:>[‘receive_order’] -
  duration:(10);
Users: ‘alice’, ‘bob’, ‘carol’;
Seniority: (‘alice’, ‘bob’), (‘alice’, ‘carol’), (‘bob’,
   ‘carol’);
```

7.1 Command Line

To simply run from command line, run the script:
`$ > python workflow_parser.py`

And the following prompt will occur and is ready for a workflow input:
`busines_process >`

7.2 Read File

To read a file, run the script, followed by the file name.
`$ > python workflow_parser.py name\of\file`

7.3 Output To File

To output to a file, the user needs to pass two arguments when running the script. The first being the file to read, and the second is the file to output the result.
`$ > python workflow_parser.py name\of\input\file name\of\output\file`

Bibliography

[1] Billie Nordmeyer. *Three Different Types of Enterprise Systems*, (accessed June, 2015). http://smallbusiness.chron.com/three-different-types-enterprise-systems-73267.html.

[2] Lisa Smith. *The Chinese Wall Protects Against Conflicts Of Interest*, (accessed February, 2015). http://www.investopedia.com/articles/analyst/090501.asp#axzz1mIc41gWQ.

[3] Chloe Green. *Breaking the law: study shows that big business is failing to stay compliant over social media*, May 2015, (accessed June, 2015). http://www.information-age.com/it-management/risk-and-compliance/123459478/breaking-law-study-shows-big-business-failing-stay-compliant-over-social-media.

[4] Intel Inspector. *Dynamic Analysis vs. Static Analysis*, 2013, (accessed June, 2015). https://software.intel.com/sites/products/documentation/doclib/iss/2013/inspector/lin/ug_docs/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm.

[5] 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).

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

[7] 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).

[8] 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).

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

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

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

[12] 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).

[13] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. page 5. `http://research.microsoft.com/en-us/um/people/leonardo/files/ci.pdf` (accessed May, 2015).

[14] 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).

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

[16] 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).

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

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

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

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

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

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

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

[24] The SMT-LIB Initiative. *SMT-LIB The satisfiability Modulo Theories Library - Benchmarks*, (accessed April, 2015). `http://smtlib.cs.uiowa.edu/benchmarks.shtml`.