Company Management

- Formal Verification -

Project Report

Beyens Ziad
Kasmi Hamza
Lefevre Lucas
Nougba Hamza
Strebelle Cédric

ULB

Contents

1 Introduction **2**

2 UPPAAL **3**
- 2.1 Advantages and disadvantages . . . 3
- 2.2 Timed Automata . . . 3
- 2.3 UPPAAL . . . 4
  - 2.3.1 Modeling language . . . 4
  - 2.3.2 Query language . . . 5

3 Modeling **6**
- 3.1 Constants, Variables and Clocks . . . 6
- 3.2 Worker . . . 7
- 3.3 Company . . . 8
- 3.4 WorkerFee . . . 9
- 3.5 WorkerExpense . . . 10

4 Verification **11**
- 4.1 Computation tree logic . . . 11
- 4.2 Properties . . . 11
  - 4.2.1 Reachability . . . 11
  - 4.2.2 Safety . . . 12
  - 4.2.3 Liveness . . . 12
- 4.3 Correctness properties . . . 13
  - 4.3.1 Safety . . . 13
- 4.4 Analysis properties . . . 14
  - 4.4.1 Reachability . . . 14
  - 4.4.2 Safety . . . 15
  - 4.4.3 Liveness . . . 19

5 Conclusion **21**

Chapter 1

Introduction

The goal of this project is to model a company management with limited number of places, money and workers having a wallet and the possibility to go on holiday, unemployed or in burnout if they work too much.
First, the chosen tool UPPAAL will be discussed and the features used will be presented.
Then, the modeling will be explained by showing every part of the model.
After that, the different kind of properties used will be explained.
Next, correctness properties will be showed to ensure that the model is well designed.
Finally, analysis properties will be discussed, answering to different realistic problems that the workers and the company can ask.

Chapter 2

UPPAAL

2.1 Advantages and disadvantages

To model-check properties on our real-time system, the tool *UPPAAL* [1] has been used. This tool has been chosen because of its ability to verify systems that can be modeled as networks of timed automata.
Nevertheless, we encountered several problems because of the limitations of UPPAAL:

- Its query language is a simplified subset of CTL in contrast to SPIN that supports entirely CTL. However, SPIN is not suitable to our system as it can not model real-time system.
- Nesting of path formula in properties are not supported (e.g. `A[] E[] a`). In fact, the only one supported is `A[] a imply A<> b` (explained later) as it is widely used.
- Until and Next properties are not supported.
- To model-check liveness properties, the model needs to be limited in term of states. As bounded integers and clocks highly increase the number of states for each possible value, the model-checking exponentially blow-up by having more discret values. With high bounded integers, UPPAAL is more suitable for other kind of properties such as reachability.

2.2 Timed Automata

A **timed automata** is a finite-state machine extended with clocks variables, each evaluating to a real number.
It is defined as:

- L · **a finite set of locations**
  It is associated to the nodes of the model.

- l_0 · **an initial location**

- C · **a finite set of clocks**
  All the clocks progress synchronously.
- Σ · **an alphabet of actions**
- $E \subseteq L \times B(C) \times \Sigma \times 2^C \times L$ · **a finite set of transitions**
  B(C) is the set of conjunctions over conditions on clocks. Each element e is in the form (l, g, a, r, l'): an edge from nodes l to l' with optional guards on clocks, actions on e and resets of clocks.
- Inv · **a finite set of invariant on the clocks**
  An invariant is a condition that must be respected by one or more clock when arriving on a state, $L \to B(C)$

A **system** is a network of several timed automata (or process) in parallel.
A **state** of the whole system is defined by the locations of all automata, the clock constraints and the value of the bounded variables. Thus, the set of states is the set of all the combinations of these components.

2.3 UPPAAL

UPPAAL is a model-checker based on the theory of timed automata. Moreover, it has an extended modeling language allowing variables, bounded integers and urgency. Also, properties to be checked can be established by using UPPAAL query language that is a subset of CTL.

2.3.1 Modeling language

A **model** of the whole system can be created using the graphical editor.
The model can be divided into several **templates**, each one forming a different timed automaton defined with a set of parameters.
Then, these templates are used to declare one or more processes. Finally, a **process** can have local variables and can interact with other processes by global variables.

For our model, among other things, we use the following UPPAAL features:

- **Bounded integer variables** are declared as `int[min,max] name`, where min and max are the bounds. All the different values between the bounds are checked during the verification. Thus, for each integer variable, it is important to put low bounds to not have an exponential blow-up.

- **Urgent locations**, when the system state contains at least one urgent location, the time can not pass.
  On the Figure 2.1, `Break` is an urgent location.

- **Guard** on an edge is a condition that has to be respected to allow the system to take that transition.
  On the Figure 2.1, there is a guard `t==T_WORK` on the edge `Working->Break`.

- **Assignment** on an edge can update variables and clocks if the system pass through that transition.
  On the Figure 2.1, there is a reset of the clock `t=0` on the edge `Working->Break` and a boolean assignment `neverRework=true` on the edge `Break->Working`.

- **Invariant** on a location is a condition that has to be satisfied when the system contains this location. When it is not satisfied, the time can not pass.
  On the Figure 2.1, there is an invariant `t<=T_WORK` on the location `Working`. It means that the clocks will stop to increase when `t==T_WORK`.

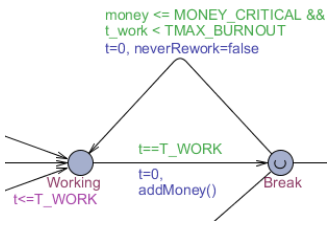

Figure 2.1 – Modeling language

2.3.2 Query language

UPPAAL can verify properties using a simplified version of CTL.
The query language consists of state formula and path formula. As already specified, a big inconvenient of UPPAAL is that it does not allow nesting of path formula.
State formula define individual states and does not always need to take into account the model. For example, `Company.Startup` checks if the process `Company` is on the location `Startup`.
Path formula quantify over traces of the model. This kind of formula will be explained later.

Chapter 3

Modeling

Using the modeling language of UPPAAL, a simplified version of a company and two workers has been modeled.
The basic concept is the following : each time a worker comes to work at the company, the company makes a profit and the worker fills his wallet. Initially, only one worker at a time can go to the company. The worker can go by several means of transport. The company has to pay regularly a fee for each worker but there is no fee when one is burned out, till he comes back home. Also, workers spend money to eat and live, this is represented by a regular loss of money in their wallet. Also, the company starts with a variable budget defining the fee paid for each worker and the amount of money generated and given to the workers when working. The company can upgrade into a big company allowing the two workers to work at the same time. Finally, if the company has no money at all, it can go to bankrupt.
The following sections will detail possible runs on each template of the model.

3.1 Constants, Variables and Clocks

The list of constants, variables and clocks are commented on the Figure 3.1.
Moreover, there is `addMoney()` that add money to the company and fill the wallet of the workers, and `giveMoney()` where the company regularly pays the fee for each worker. Also, simple observer booleans such as `neverRework` are used to check if a worker worked again after `Break`.
To simplify model-checking, the two workers share the same wallet. It can be seen as a man and his wife working in the same company. In fact, increasing the number of workers, edges or variables has been tried but it almost always implied an **exhausted memory** during the (too long) model-checking. The reason behind this is that increasing the number of discrete variables exponentially increases the number of state of the system.

```
clock t_global; //a global clock never reset
const int WORKERS = 2; //the number of workers process
typedef int[0, WORKERS-1] worker_t; //bounded integer (i.e. for id Worker)

const int MONEY_MAX = 5; //max money that the Company and the wallet can reach.
int[0,MONEY_MAX] money = 1; //initial money of the Company
const int MONEY_CRITICAL = 4; //threshold enabling the workers to rework after break

const int BUDGETS[3] = {1,2,3}; //one same budget chosen by the company
int[1,3] budget=1; //budget chosen in BUDGET, used when addMoney and giveMoney
//each time a worker works, addMoney, (2*budget) money to the company and
//add (budget) money to the wallet of the workers

int[0,MONEY_MAX] wallet = 2; //the common wallet of the workers
const int WALLET_CRITICAL = 3; //threshold enabling to go to holiday when at home
const int personExpense = 1; //common regular expenses of the workers (spendMoney)

int[1,2] counter_max = 1; //the number of worker that can work at the same time in the company. 2 if Big
int[0,2] counter = 0; //the number of worker working at the same time in the company

const int T_SPEND_MONEY = 8; //clock interval - workers spend common money
const int T_IDLE = 2;
const int TMAX_IDLE = 3; //clock limit - staying at Home or Holiday

const int EPS = 1; //epsilon clock
const int T_WALK=3; //min time to walk to Work
const int T_BUS=2; //min time to go by bus to Work
const int T_CAR=1; //min time to go by car to Work
const int T_SLEEP = 4; //min time of sleeping after work
const int TMAX_WALK=T_WALK+EPS;
const int TMAX_BUS=T_BUS+EPS;
const int TMAX_CAR=T_CAR+EPS;
const int TMAX_SLEEP = T_SLEEP+EPS;

const int T_WORK = 1; //time of working
const int T_WORK_CRITICAL = 3; //threshold to stop working to avoid burnout.
const int T_FEE=5; //clock interval - fee of each worker.
const int T_BURNOUT=4; //min time to go to Burnout
const int TMAX_BURNOUT = 8;
const int T_REST=5; //time to rest when in Burnout
```

Figure 3.1 – Constants, Variables and Clocks

3.2 Worker

There is two process of worker instantiated with a different id (0 and 1).
Workers have 2 local clocks: `t` reset on each timed transition and `t_work` reset each time the worker go `Outside`.

- Initially at `Home`, a worker can choose at any moment up to `TMAX_IDLE` to either go `Outside` if there is nobody at work (`counter<counter_max`), go on `Holiday` if he has enough money (`wallet>=WALLET_CRITICAL`), or be a lazy `Unemployed` if the company is full (`counter==counter_max`). It implies that at least one worker **will** go `Outside` if nobody is at work and the wallet is not filled enough.
- `Unemployed`, the worker can come back to `Home` only if there is place in the Company.
- On `Holiday`, the worker can come back to `Home` at any time before `TMAX_IDLE`
- `Outside`, a worker has to choose its mean of transport (round trip): by bus, by car or on foot. The traveling time is variable but the car is faster than the

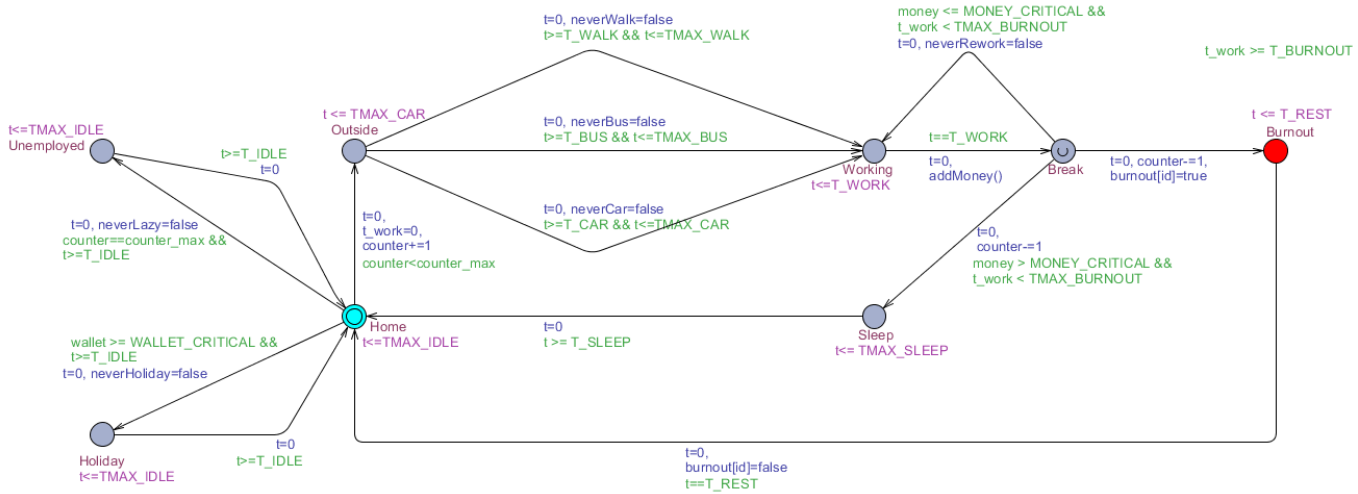

Figure 3.2 – Worker automata

bus which is itself faster than walking. It is interesting for the worker to know which one he can choose for his purpose (i.e. he can be against car pollution or too poor to buy a car).

- `Working`, a worker will work for a fixed amount of time (`T_WORK`) so it will add cash to the company and earn money. Then, he must take a `Break`.

- At the `Break`, he has to take a choice directly: go back home to `Sleep` if enough `money` in the company and not at maximal burnout time (`money>MONEY_CRITICAL and t_work<TMAX_BURNOUT`) or to rework for another period of time if the company needs more `money`.
  If a worker reworks too many times without going back `Home` (`t_work>=T_BURNOUT`), he may do a `Burnout`. He *will* do a `Burnout` if he reaches the maximum amount of time working supported by a human (`TMAX_BURNOUT`).
  If the company has only a small amount of money, it forbids the worker to go back to `Home` and forces him to work again. Of course, this might lead to a `Burnout`.

- In `Burnout`, the worker can not go to `Home` immediately, he must first rest for a long period of time (`T_REST`).

- In `Sleep`, the worker sleeps to recover from his work, during at most `TMAX_SLEEP`. Then he returns `Home`.

3.3 Company

There is one process of company instantiated.

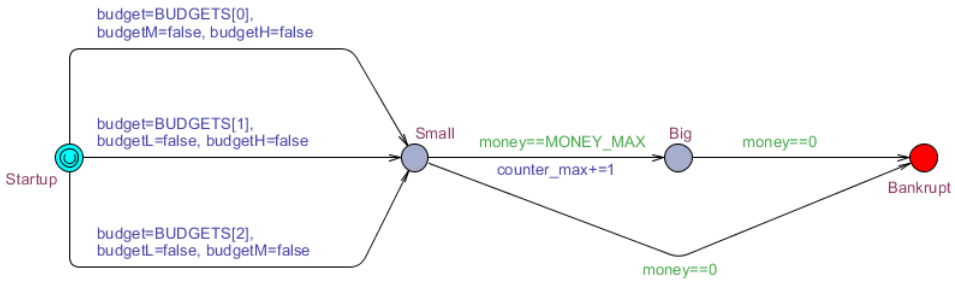

Figure 3.3 – Company automaton with initial choice of budget and possibility to increase the worker capacity.

- Initially a `Startup`, the company has to choose a low, medium or high budget (`budget=BUDGETS[0, 1 or 2]`). The higher the budget, the more it pays the workers, the higher is the gain of money each time they work and the higher is the fee related to each worker.
  It implies that with a high budget, the company loose its money faster and so, has more chance to go `Bankrupt`
- When `Small`, there is only one place available in the company. If the company is able to reach a certain amount of money, it can upgrade to a `Big` company, opening a second post to let the workers working at the same time.
- At any time (`Small` or `Big`), the company can go `Bankrupt` if it has no money left (`money==0`).

3.4 WorkerFee

There is two processes of WorkerFee instantiated with a different id (0 and 1).
WorkerFee have 1 local clocks: `s` reset on each timed transition.

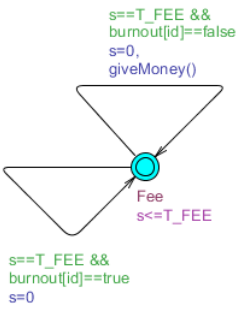

Figure 3.4 – WorkerFee automata - Regular operational costs of the company.

- Every `T_FEE`, the company pays a fee for each worker. No fee is paid when the workers are in `Burnout`.

3.5 WorkerExpense

There is one process of WorkerExpense instantiated.
WorkerExpense have 1 local clocks: `s` reset on each timed transition.

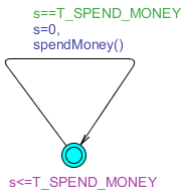

Figure 3.5 – WorkerExpense automata - Regular expenses of the workers.

- Every `T_SPEND_MONEY`, the workers spend their common money (`wallet`).

Chapter 4

Verification

Using the CTL query language of UPPAAL, a lot of properties have been defined to verify the model.
First, the different kind of properties used will be explained.
Then, **correctness** properties will be established to validate the basic behavior of the company model.
Finally, **analysis** properties will be discussed to, among other things, find the key values where the behavior of the model changes.

4.1 Computation tree logic

CTL uses the branching time semantics dealing with the execution tree. Unlike LTL, it considers all the possible futures, at any node.
CTL model checking is in polynomial time but remember that a state of a system is a combination of all the locations and the discrete variables. Thus, the number of combination increases really fast by extending the model.

4.2 Properties

There is three kinds of path formula that can be treated by UPPAAL. Remember that UPPAAL does not allow nesting of path formula.
Thereafter, ϕ represents a state formula.

4.2.1 Reachability

In the form of `E<>` ϕ, it checks that there exists a path where a state satisfying ϕ is reachable.

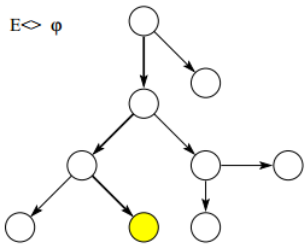

Figure 4.1 – Reachability Properties.

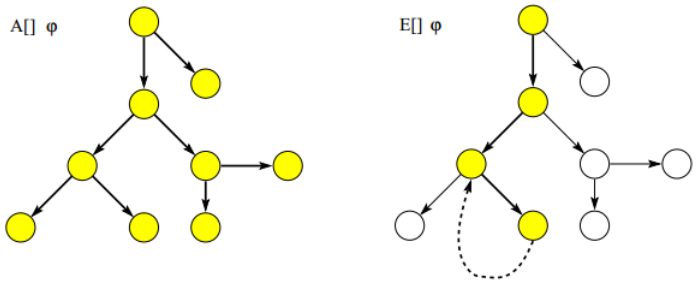

Figure 4.2 – Safety Properties.

4.2.2 Safety

In the form of `A[]` ϕ, it checks that for all paths, all the states satisfy ϕ.
Also in the form of `E[]` ϕ, it checks that there exists a path where all the states satisfy ϕ.
These forms are widely used to check that something bad will never happen.

4.2.3 Liveness

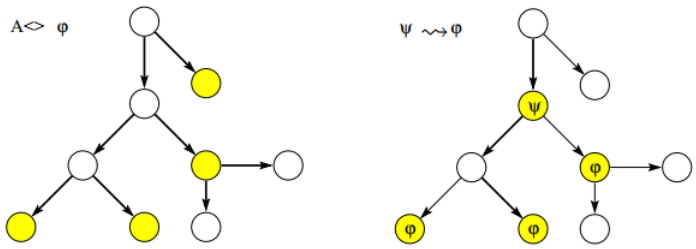

Figure 4.3 – Liveness Properties.

In the form of `A<>` ϕ, it checks that for all paths, eventually a state will satisfy ϕ.
A useful form, *leads to*, ϕ `-->` ψ, equivalent to `A[]` $(\phi \implies$ `A<>` $\psi)$ (note the

nesting path formula usable only in that form), checks that whenever ϕ is true, then eventually a state satisfying ψ will be reached.

4.3 Correctness properties

In the following, properties that ensures the correctness of the company model will be verified.
Most of them are trivial but in fact, they have been set up during the construction of the model to easily see what was wrong at each step of the modeling.

4.3.1 Safety

```
A[] not deadlock
```

True
There are no deadlocks in the system. At every state, with any value for the variables, there is always a way to take an other transition. Deadlocks can have happen when a process is on a state with an invariant. As a reminder, an invariant does not allow the clock to pass if it is not satisfied. Then, the process should be able to take a transition to avoid the deadlock. Thus, if the process can not ever take the transition because of the guards not satisfied, this would be a deadlock.
During the construction of the model, this property was very useful to verify that invariants and guards are well designed.

```
A[] (Worker(0).Working && Worker(1).Working) imply counter_max>=2
```

True
This property means that if the two workers are `Working` at the same time, it implies necessarily that the company has two posts (i.e. it has been upgraded to a `Big` company). It also means that the 2 workers are not able to work at the same time if the company is `Small` (`counter_max=1`). The latter can be also checks by the following formula.

```
A[] Company.Small imply not (Worker(0).Working && Worker(1).Working)
```

True
The workers are never able to work at the same time if the company is `Small`.

```
A[] Worker(0).Burnout imply Worker(0).t_work >= T_BURNOUT
```

True
This checks that if a worker is burned out, it is because he has previously worked too long (more than `T_BURNOUT`). It is trivially satisfied because a worker can only take the transition to the `Burnout` state if the guard `t_work >= TMAX_BURNOUT > T_BURNOUT` is true.

4.4 Analysis properties

From now on, considering that the company model is correct, the most interesting properties will be set up.
These properties will let the company and the workers analyzing the key values where the properties hold on.
Indeed, different values may lead to different results in some case, and this is particularly useful to know them when a choice needs to be done by the company or by the workers.
Also, these values can be needed for optimization problems (i.e. minimal cost).
The following analysis will be grouped into several set of similar properties.
For some interesting properties, a simulation trace will be provided to prove the result.

4.4.1 Reachability

(1.1) `E<> budgetL && t_global < 6 && Company.Big && wallet==MONEY_MAX`

True
This formula checks that it is possible that the workers and the company reach the maximum amount of money and with the company getting `Big` in less than a certain amount of time[1] (`t_global < 6`) when the company has a low budget.

(1.2) `E<> budgetL && t_global < 5 && Company.Big && wallet==MONEY_MAX`

False
Same as above except the time value. This time it is not possible to the maximum amount of money in that time, because the workers cannot work fast enough to gather this money. From this formula and the previous one, we can deduce that the fastest way to have `money==MONEY_MAX` will take between 12 and 13 units of time.

(1.3) `E<> budgetL and Company.Big`

True
This is a simple property. The company can be `Big`[2] with a low budget. But see property (1.4).

(1.4) `E<> budgetL and Company.Big and neverRework`

False
This shows that the company can not be `Big` with a low budget and with the workers always going to `Sleep` after the first `Break`.

[1] `t_global` is a global clock never reset
[2] the company can be `Big` if `money==MONEY_MAX`

(1.5) `E<> budgetM and Company.Big and neverRework`

True
Same with a medium budget. So it would be more profitable to the workers that the company chooses this budget so they don't work too much and don't go easily `Burnout`. It is true with a high budget too.

4.4.2 Safety

(2.1) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0`

True
From now on, an analysis of the `Worker.Burnout, Company.Bankrupt` starts.
This property[3] checks whether there is a path such that workers are never in burnout and the company can never go bankrupt.
This analysis is a nice way to find a strategy where both parts (workers and company) are happy.

(2.2) `A[] money > 0`

False
This property checks whether for any path, the company can never be bankrupted.
A simulation trace justifying this property is given in the Figure 4.4. `Bankrupt` has been reached because `Worker(1)` went to `Holiday` instead of going to work.
Note that nobody is burned out.

(2.3) `A[] not exists(i:worker_t) Worker(i).Burnout`

False
This property checks whether for any path, workers are never in burnout.
A simulation trace justifying this property is given in the Figure 4.5. `Worker(1).Burnout` has been reached because the company can force the workers to work again regardless of their `t_work` if the company is nearly bankrupted.

(2.4) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetH`

True
Same than (2.1) but it checks whether it holds when the company has a high budget.
It implies that the workers need to work a lot to avoid bankrupt because the company looses faster its money, and so the workers can be burned out more easily.
As this is true, it is trivially true for the low budget and the medium one.
Therefore, the company can choose any budget to avoid bankrupt and workers burnout.

[3] `exists(i:worker_t) Worker(i).Burnout` checks whether there is at least one worker (with id 0 or 1) that is in `Burnout`

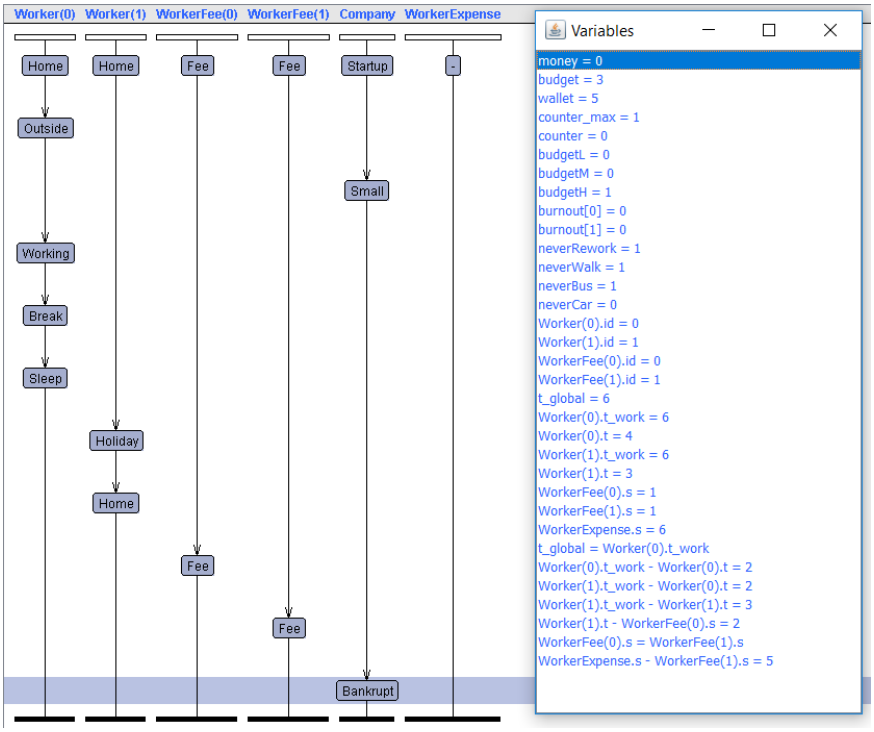

Figure 4.4 – Simulation trace (2.2) reaching `Company.Bankrupt`

(2.5) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetH and neverRework`

False

Same than (2.4) but it checks whether it holds when the workers never work again just after `Working`.
As the fee are high, the workers must to rework sometimes to avoid burnout and bankruptcy.

(2.6) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetM and neverRework`

True

Same than (2.5) but it checks whether it holds when the budget is medium (or low).
This shows that workers can always go to `Sleep` at the first `Break`.

(2.7) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetH and neverCar`

False

Same than (2.4) but it checks whether it holds when the worker never takes the car

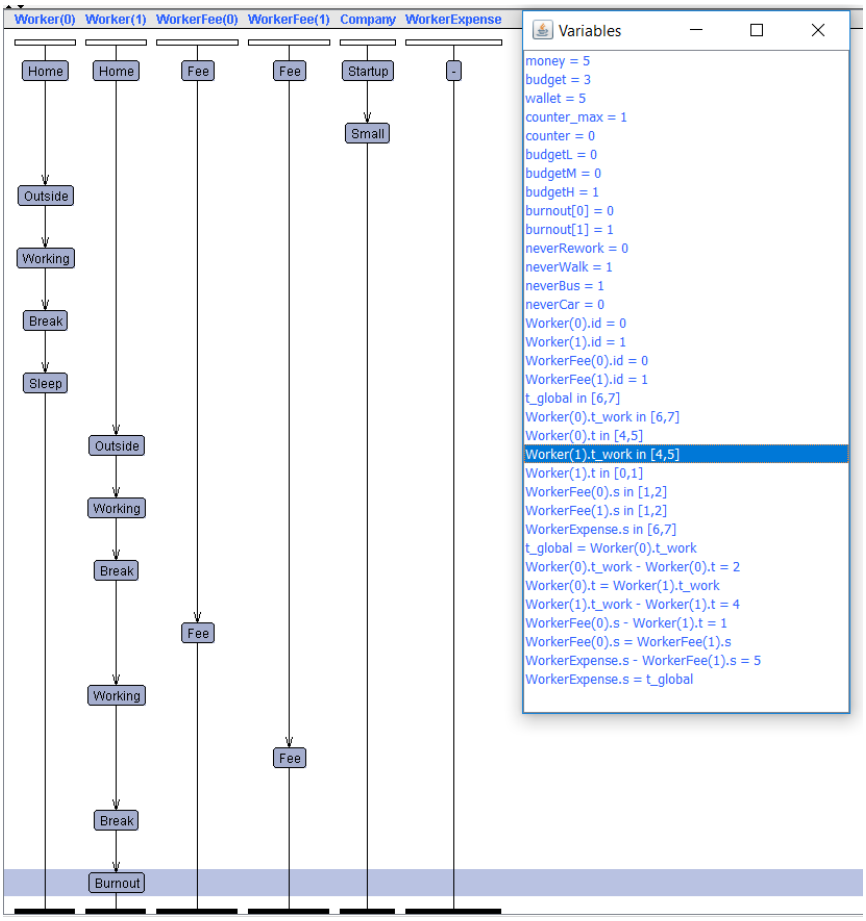

Figure 4.5 – Simulation trace (2.3) reaching `Worker(1).Burnout`

(that is faster than the bus and walking).
As it is true without the `neverCar` condition, this shows that workers must take the car to satisfy the other conditions.

(2.8) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetM and neverCar`

True
Same than (2.7) but it checks whether it holds when the company takes a medium budget (or low).
As it is true, this shows that workers can at least take the bus to satisfy the other conditions. But this does not prove that the workers can avoid the bus and always walk. For that, see property (2.9).

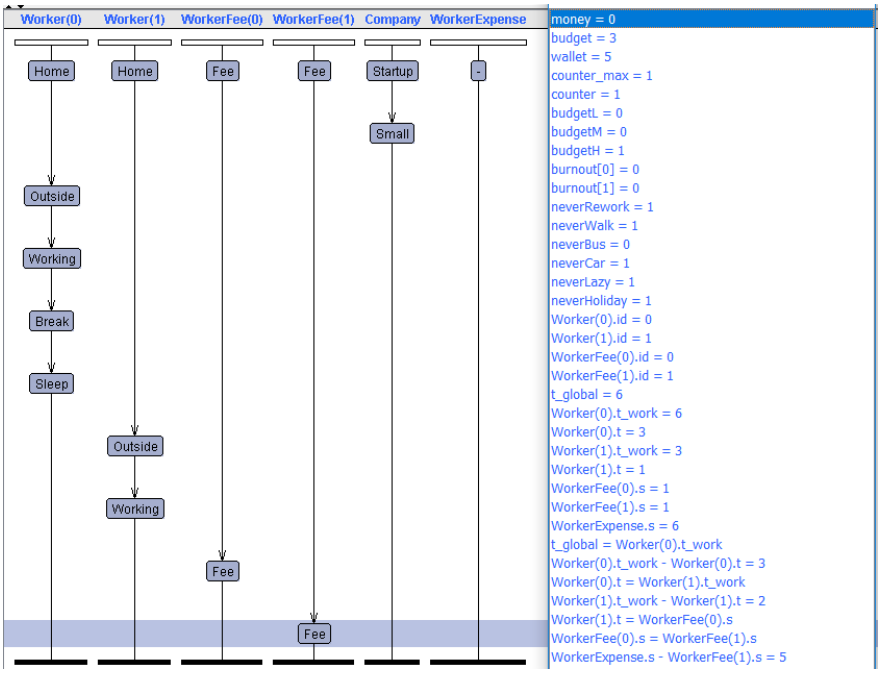

Figure 4.6 – Simulation trace (2.7) reaching `money=0`

(2.9) `E[] (not exists(i:worker_t) Worker(i).Burnout) && money > 0 && budgetM and neverCar and neverBus`

False

Same than (2.8) but it checks whether it holds when the worker never takes neither the car nor the bus.
This shows that workers can not avoid bus or car to satisfy the other conditions.
This is interesting for the thrifty worker because he does not need to buy a car but he should take a bus subscription.
If `budgetM` is replaced by `budgetL`, it is also false.

(2.10) `A[] wallet > 0`

True

From now on, an analysis of the `wallet` starts.
This property checks whether for any path, workers wallet is never empty.
This is interesting because it is showing that the model has also been designed for this purpose.
It could not have been the case if what they earn after working was not sufficient for their lifestyle and their spending.

(2.11) `E[] wallet < MONEY_MAX`

True
This formula means that there is a way such that the workers are never reaching the maximum amount of money. This can happen if they work only when it is really necessary to fill up their wallet.

(2.12) `E[] money < MONEY_MAX and wallet < MONEY_MAX`

True
Similarly to the formula above, this shows that there exist at least one path such that neither the workers nor the company reach the maximum amount of money. Of course, it is not surprising that both conditions can be satisfied simultaneously because the workers and the company earn money at the same moment.

(2.13) `E[] money < MONEY_MAX and wallet < MONEY_MAX`

True
Similarly to the formula above, this shows that there exist at least one path such that neither the workers nor the company reach the maximum amount of money. Of course, it is not surprising that both conditions can be satisfied simultaneously because the workers and the company earn money at the same moment.

(2.14) `E[] (not exists(i:worker_t) Worker(i).Burnout) and money>0 and neverCar and neverHoliday and neverLazy`

True
This shows that there is a path where workers can never go on holiday et unemployed even if they never use the car.

(2.15) `E[] (not exists(i:worker_t) Worker(i).Burnout) and money>0 and neverBus and neverCar and neverHoliday and neverLazy`

False
This shows that workers must go on holiday or unemployed at least one time if they never take the bus or the car. This is interesting in the way that walking is so slow that the other worker have the time to do something else.

4.4.3 Liveness

(3.1) `A<> (exists(i:worker_t) Worker(i).Working)`

True
This property checks whether for all paths, there is at least one worker going to work. This is true because at `Home`, one has to go `Outside` if their wallet is low (no `Holiday`)

and if nobody is working (no `Unemployed`).
With the same reasoning, this is also true for `Outside, Break`.

(3.2) `A<> (exists(i:worker_t) Worker(i).Sleep)`

False
This property checks whether for all paths, there is at least one worker going to `Sleep`.
This is false because workers can go always go `Burnout` if they always work too much.
With the same reasoning, this is also false for `Burnout, Unemployed, Holiday`.

(3.3) `Worker(0).Home and Worker(0).t==TMAX_IDLE and counter==counter_max --> Worker(0).Holiday or Worker(0).Unemployed`

False
This property is false because there is always a possibility that the company go `Big`.
Afterwards, both workers can always go `Outside` when at `Home`.

(3.4) `Worker(0).Break and Worker(0).t_work == TMAX_BURNOUT --> Worker(0).Burnout`

True
This is true because once a worker is in Break and `t_work==TMAX_BURNOUT`, he will eventually burn out.

Chapter 5

Conclusion

The modeling part took time before finding a good one, from a parking management to an company management.
The first idea was a parking with four places with detectors, with an entry and an exit way (both with detectors too at each end). This was abandoned because it was too much deterministic and thus not interesting to verify.
Then, a company management has been added to the parking. As the parking was still not relevant, only the company management has been retained.
Thereafter, the model was lacking again of non-determinism. This issue was resolved by adding more transitions and interval clock conditions so the process can take the transition at several moment, generating more paths with different results.
An other difficulty encountered was the adjustment of the values to not make the verification too much long with UPPAAL. Even if UPPAAL give the possibility to program any model, it's not to forgot that it is converted in a network of timed automaton with easily a huge number of possible states. Because of this performance constraint, the discrete values used have been minimized such that the simplified model is still relevant.
After several attempts and a big rework of the model, everything is working well and the verification of properties does not take more than 5 minutes.
Also, it was problematic that only a simplified version of CTL was usable. The SPIN tool would have been chosen if the model was not real-time. However, the verification of the properties with its analysis was very interesting because it gives realistic solution to several (decision) problems that can ask the workers and the company.

Bibliography

[1] Behrmann Gerd et al. *Uppaal Tiga User-manual*. `http://people.cs.aau.dk/~adavid/tiga/manual.pdf`.